A Robust Adaptive Metric for Deadline Assignment in Heterogeneous Distributed Real-Time Systems

Jan Jonsson

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

Abstract

In a real-time system, tasks are constrained by global end-to-end (E-T-E) deadlines. In order to cater for high task schedulability, these deadlines must be distributed over component tasks in an intelligent way. In this paper, we present an improved version of the slicing technique and extend it to heterogeneous distributed hard real-time systems. The salient feature of the new technique is that it utilizes adaptive metrics for assigning local task deadlines. Using experimental results we show that the new technique exhibits superior performance with respect to the success ratio of a heuristic scheduling algorithm. For smaller systems, the new adaptive metric outperforms a previously-proposed adaptive metric by 300%, and existing non-adaptive metrics by more than an order of magnitude. In addition, the new technique is shown to be extremely robust for various system configurations.

1 Introduction

In a distributed real-time computing system, applications are decomposed into tasks, which are then assigned to processors according to a task-assignment technique based on, for example, clustering [1], list scheduling [2], or a branch-and-bound strategy [3, 4]. Task assignments are governed by *locality constraints* that are either *strict* (the assignment of a task is known beforehand) or *relaxed* (more than one assignment alternative exist for each task). While there are some well-known solutions to the task-assignment problem, an important remaining problem is *deadline distribution*. To guarantee the functionality of a real-time system, an application is constrained to start its execution and complete within a given time span called the *end-to-end deadline*. The application has usually been logically decomposed into a set of sequential and/or parallel tasks, 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 E-T-E deadline must be distributed over the component tasks.

Many researchers have addressed the deadline-distribution problem [5, 6, 7, 8, 9, 10], all under a common assumption that task assignments are known *a priori*, that is, strict locality constraints. In many real-time systems, however, only a small number of task assignments are governed by strict locality constraints, typically 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. This means that *a priori* information regarding task execution times and intertask communication cost will not be available. For a homogeneous system and negligible intertask communication cost this poses no problem since all processors are interchangeable. For a heterogeneous system, however, tasks may have different execution times on different processors, and hence the deadline-distribution problem under relaxed locality constraints is much harder to solve. Furthermore, task-assignment techniques require information about individual task deadlines for scheduling purposes. Deadline distribution using conventional techniques, on the other hand, can only be performed if the task assignment is completely known. Thus, there exists a circular dependency between the deadline-distribution and task-assignment problems which makes the combined deadline-distribution and task-assignment problem even harder to solve. The task-assignment problem is, in the general case, an NP-complete problem [11], and good solutions to the combined deadline-distribution and task-assignment problem must, therefore, be found through the use of sub-optimal heuristic techniques.

In this paper, we present a heuristic technique for deadline distribution in a heterogeneous system under relaxed locality constraints. Our technique is based on the *slicing technique* [5, 12] where the E-T-E deadline for each sequence of tasks in the application is decomposed into a set of non-overlapping task execution windows called *slices*. The distinguishing feature of the slicing technique is that the slices for sequential tasks are non-overlapping. This allows for a divide-and-conquer approach to solving the problem that first divides the overall problem into smaller problems that are solved locally and then combined to obtain a global solution. This can drastically reduce the computational complexity of the problem. More importantly, however, non-overlapping slices guarantees that each task will finish its execution before the arrival

time of its successor task. This has the following important implications.

I1. The scheduling of sequential tasks on two different processors can be performed independently of each other. This is useful in heterogeneous systems where different scheduling strategies are employed in different processors. It also allows for parallel scheduling of sequential tasks, a feature that can aid in increasing the scheduling performance for systems with on-line scheduling [13].

I2. The release jitter of each task as caused by precedence constraints can be eliminated. As discussed in, for example, [14, 15], uncontrolled release jitter can negatively affect the schedulability of real-time applications.

The deadline-distribution problem is addressed in the context of distributed hard real-time systems. In such systems, task assignment and scheduling are usually assumed to be performed pre-run-time 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. The applications of interest in this paper are those that consist of sequential–parallel precedence-constrained tasks with individual arrival times and deadlines. Our scheduling objective is to maximize the *success ratio*, that is, the ratio of the number of successfully scheduled task sets to the total number of considered task sets. We consider a system with a non-preemptive, time-driven, task dispatching strategy.

We demonstrate the salient features of our technique by means of an extensive experimental evaluation using randomly-generated application task sets and a multiprocessor system of varying size. In particular, we study the impact of variations in architecture and application properties on the success ratio for a baseline deadline-driven task assignment and scheduling algorithm. The properties under investigation are system size, tightness of E-T-E deadlines, and task execution time distribution. The first property is interesting because schedulability could depend on how well the inherent application parallelism can be exploited on the available processors, the second because it directly controls the amount of slack time available for distribution, and the third because, with a non-preemptive dispatching strategy, the schedulability is directly affected by the execution time distribution.

Our main contributions in this paper are:

C1. We apply the slicing technique to a heterogeneous system with relaxed locality constraints. This is an extension of the work recently reported in [12] where only homogeneous systems were considered. Other known deadline-distribution techniques all make the common assumption that task assignment is already known, and thereby avoid the many difficulties associated with scheduling on heterogeneous systems.

C2. We propose an improvement of the work in [12]. We show that the new technique exhibits very good performance with respect to the success ratio. For smaller systems, the performance increase is more than 300% over the technique in [12], and an order of magnitude over the technique in [5]. In fact, for systems with near-uniform task execution times, the increase in performance over *both* of these existing techniques is as high as an order of magnitude. Moreover, the new technique is shown to exhibit extremely robust performance over a wide range of architecture and application scenarios.

The rest of the paper is organized as follows: Section 2 describes work related to ours. Section 3 describes the assumed system models. 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. Section 7 discusses complementary results and possible future work. Finally, Section 8 summarizes the results in this paper.

2 Related work

The slicing technique proposed by Di Natale and Stankovic in [5] assigns slices, execution windows, to tasks 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 basic metrics (previously proposed in [9, 10]) were used for evaluating paths in the task graph: one assigns a task deadline based on its execution time, and the other assigns a task deadline based on the number of tasks 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.

In a recent paper [12], we proposed a set of adaptive metrics for the slicing technique, suitable for homogeneous systems with relaxed locality constraints. The proposed metrics were shown to outperform the original metrics in [5] in situations where application parallelism cannot be fully exploited on the system. In [12], the slicing technique was evaluated with respect to the maximum task lateness, because the application E-T-E deadlines were assumed to be loose enough to guarantee a near 100% success ratio.

Several deadline-distribution techniques have been proposed where task assignment is assumed to be known beforehand. In [6], 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 [7] presented a technique for scheduling a system of flow-shop tasks. Local deadlines are assigned by distributing E-T-E deadlines evenly over tasks. For this method, the simplifying assumption is made that execution times are either identical for all tasks or identical for all tasks assigned to the same processor. Saksena and

Hong [8] proposed a deadline-distribution technique based on a critical scaling factor that is applied to the task execution times. The E-T-E 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 tasks schedulable. The local deadline assignment is then chosen to maximize the largest value of the scaling factor. The techniques in [6, 7, 8] all assume that the application consist of purely sequential tasks and that task assignment is known beforehand.

Kao and Garcia-Molina presented multiple strategies for distributing E-T-E deadlines over sequential [9] and sequential–parallel [10] tasks. However, these strategies are only aimed at, and evaluated in the context of, soft real-time systems with complete *a priori* knowledge of task–to–processor assignment.

3 System model

3.1 Architecture Model

We consider a multiprocessor architecture with a set $\mathbf{P} = \{p_q : 1 \leq q \leq m\}$ of schedulable processors and an interconnection network with a set of communication links. The processors are heterogeneous in the sense that they have different hardware configurations in terms of processing speed, instruction pipeline, and cache/primary memory resources. This means that the execution time for a task may differ depending on which processor it will execute on. Therefore, we introduce a set $\mathbf{E} = \{e_k : 1 \leq k \leq m_e\}$ of *processor classes*, and then associate to each processor p_q a processor class $e(p_q) \in \mathbf{E}$ that determines the actual hardware configuration. Given a set of processor classes, the processors can be classified as being either *identical*, *uniform*, or *unrelated* [16]. For identical processors, the hardware configuration is the same for all processors and each task takes the same amount of time to execute on any processor. For uniform processors, a task's execution time is the product of a basic execution time and a scaling factor associated with each processor. For unrelated processors, a task's execution time on one processor is not necessarily related to its execution time on some other processor.

The interconnection network is an arbitrary topology that may include dedicated as well as shared links. The architecture is assumed to support *asynchronous* communication in the network, that is, communication is allowed to take place concurrently[1] with processor computation. The communication between two tasks residing on the same processor is done via accessing shared memory, and its cost is assumed to be negligible. The communication cost between two tasks on different processors is expressed as the product of the length of a message and a nominal communication delay. The nominal delay is an upper-bounded and predictable worst-case communication delay that reflects the scheduling strategy of the underlying interconnection network.

We will assume that the system maintains a global *system time* that is discrete and represented by *time units* indexed by the natural numbers, that is, $t \in \mathbf{N}$. Without loss of generality, we assume that task activities begin and end at time units, and that application timing parameters are expressed as a multiple of time units. A consistent view of system time is maintained in each processor by means of a system-wide clock synchronization mechanism. We assume that an exact clock synchronization is maintained in the system. In addition, we assume that processors and communication links are free from faults during the lifetime of the system.

[1] Using, for example, a communication co-processor.

3.2 Application model

We consider a real-time application that consists of a set $\mathbf{T} = \{\tau_i : 1 \leq i \leq n\}$ of tasks. Each task $\tau_i \in \mathbf{T}$ is characterized by a 4-tuple $\langle c_i, \phi_i, d_i, T_i \rangle$, which we will refer to as the *static task parameters*. The *worst-case execution time* c_i is an array of upper bounds on the execution times of the task for each processor class. The worst-case execution time (WCET) for τ_i on a processor of class e_k is $c_i[e_k]$. We make the assumption that the worst-case execution time includes various architecture overhead such as the cost for cache memory misses, pipeline hazards, and context switches. Whenever it is clear to which processor τ_i is assigned, we write c_i. In the case when task assignments are not yet known, we use an *estimated WCET*, $\bar{c}_i$. Suitable strategies for estimating $\bar{c}_i$ will be evaluated later in this paper. The *phasing*, ϕ_i, is the earliest time at which the first invocation of the task will occur, measured relative to some fixed origin of time. The *relative deadline*, d_i, is the amount of time within which the task must complete its execution, once it has been invoked. The *period*, T_i, is the time interval between two consecutive invocations of the task.

A periodic task τ_i gives rise to an infinite sequence of invocations, and we denote the k^{th} invocation of the task by τ_i^k, where $k \in \mathbf{Z}^+$. The dynamic behavior of τ_i^k is characterized by the *dynamic task parameters* (a_i^k, D_i^k), where the *absolute arrival time*, $a_i^k = \phi_i + T_i(k-1)$, is the earliest time at which τ_i^k is allowed to begin execution, and the *absolute deadline*, $D_i^k = a_i^k + d_i$, is the latest time at which τ_i^k must finish its execution. When it is clear from the context what invocation of τ_i is being referred to, we will drop the use of invocation superscripts and write (a_i, D_i).

Precedence constraints between tasks are represented by an irreflexive partial order $\prec$ over $\mathbf{T}$. If task τ_j cannot begin its execution until task τ_i has completed its execution, we write $\tau_i \prec \tau_j$. In this case τ_i is said to be a *predecessor* of τ_j, and, conversely, τ_j a *successor* of τ_i. In addition, whenever $\tau_i \prec \tau_j$ and the condition $\neg(\exists \tau_k : (\tau_i \prec \tau_k) \wedge (\tau_k \prec \tau_j))$ holds, we write $\tau_i \prec\cdot \tau_j$. In this case, τ_i is said to be a *immediate predecessor* of task τ_j and τ_j a *immediate successor* of τ_i. A task which has no predecessors is called an *input task* and a task which has no successors is called an *output task*.

A *task chain* is defined as a task followed by a series of immediate successors. The *length* of a task chain is the sum of the estimated worst-case execution times of all tasks in the chain. The *static level*, $SL(\tau_i)$, for task τ_i is the length of the longest task chain that starts with τ_i and ends with an output task. A *path* is a task chain that begins with an input task and

ends with an output task. A *critical path* is one for which a given performance measure assumes a minimal or maximal value.

Communication of data in the application can be embedded in the precedence constraints between tasks, and implement general communication primitives such as SEND–RECEIVE–REPLY and QUERY–RESPONSE [3]. The amount of data sent between task τ_i and task τ_j is denoted by a message size $m_{i,j}$. The worst-case communication cost for sending a message from one task to another depends on factors such as task assignment, message size, communication medium bandwidth, and message dispatching strategy employed in the system. We assume that the cost for packetizing and depacketizing messages is constant and included in the worst-case execution times of the communicating tasks.

Rather than illustrating the computational and communication demands of all tasks in the task set, and the precedence constraints among them, in terms of parameter tuples and a partial order, we use a directed acyclic graph $\mathbf{G} = (\mathbf{N}, \mathbf{A})$ called a *task graph*. $\mathbf{N}$ is a set of nodes representing the tasks in set $\mathbf{T}$. $\mathbf{A}$ is a set of directed arcs representing the precedence constraints between the tasks in $\mathbf{T}$, that is, if $\tau_i \prec\cdot\, \tau_j$ then $(\tau_i, \tau_j) \in \mathbf{A}$. Each node in $\mathbf{N}$ is annotated with a non-negative weight representing the computational demands of the corresponding task. For those arcs in $\mathbf{A}$ that represent communication channels, a non-negative weight is used represent the message size.

3.3 Multiprocessor Scheduling

A *time-driven, non-preemptive multiprocessor schedule* for a task set $\mathbf{T}$ and a multiprocessor architecture $\mathbf{P}$ is the mapping of each task $\tau_i \in \mathbf{T}$ to a *start time*, s_i, and a processor, $p(\tau_i) \in \mathbf{P}$. The task is then scheduled to run without preemption on processor $p(\tau_i)$ in the time interval $[s_i, f_i]$, with its *finish time* being $f_i = s_i + c_i$. The time interval $[a_i, D_i]$, denoted by w_i, is called the *execution window* of τ_i. For periodic tasks, the static task parameters are assumed to satisfy $d_i \leq T_i$, that is, the execution windows of two invocations of the same task cannot overlap in time. Furthermore, the execution time, c_i, cannot exceed the length, $|w_i|$, of the execution window. Note that we assume a *static assignment* strategy, which implies that the execution of all invocations of τ_i is performed on the same processor $p(\tau_i)$.

To handle a periodic task system, we need only analyze the task behavior within a specific period that will repeat itself throughout the lifetime of the system. This period, P, is called the *planning cycle* of the task set. For a set of tasks with identical arrival times, the planning cycle can be found as follows. Without loss of generality, assume that $\forall \tau_i \in \mathbf{T} : a_i = 0$. We then choose $P = [0, L)$, where L, the length of the planning cycle, is defined as the least common multiple of $\{T_i : \tau_i \in \mathbf{T}\}$. Within P, τ_i will be invoked L/T_i times. For a set of tasks with arbitrary arrival times, we find the planning cycle as follows. Without loss of generality, assume that $\min\{a_i : \tau_i \in \mathbf{T}\} = 0$. Furthermore, define $a = \max\{a_i : \tau_i \in \mathbf{T}\}$. We then choose $P = [0, a + 2L)$.

4 Deadline Distribution

4.1 Problem statement

The problem addressed in this paper is the following. We assume that the application is represented by a task graph $\mathbf{G} = (\mathbf{N}, \mathbf{A})$ with $n = |\mathbf{N}|$ tasks. Given an E-T-E deadline, D_α, and a corresponding input–output task pair $(\tau_{\alpha_1}, \tau_{\alpha_2})$, the *deadline distribution problem* is to partition (distribute) D_α into an arrival time, a_i, and a relative deadline, d_i, for a task τ_i in the task graph in such a way that the path constraint

$$\sum_{\tau_i \in \Phi} d_i \leq D_\alpha \tag{1}$$

is satisfied for every path Φ between τ_{α_1} and τ_{α_2}.

4.2 Quality assessment

A solution to the deadline-distribution problem cannot be accepted simply because it satisfies the path constraint defined above. One must also consider the practical issue of schedulability: the relative deadline of a task must be derived in such a way that the task is likely to be feasibly scheduled. If one deadline-distribution strategy is able to feasibly schedule more task sets than another strategy, we can clearly consider the first strategy superior to the other. Therefore, our primary performance measure for a deadline distribution strategy is the *success ratio*. If a deadline distribution strategy is able to find feasible schedules for x of the considered y task sets, its success ratio is said to be (x/y).

When the E-T-E deadlines are loose enough to guarantee a near 100% success ratio, a secondary performance measure should be used to assess the quality of different deadline-distribution strategies. Two important measures that are often used for this purpose are the *minimum laxity* and the *maximum lateness* taken over all tasks in the system. The laxity, $X_i = d_i - \bar{c}_i$, is the maximum amount of time that the execution of task τ_i can be delayed in its execution window without it missing its absolute deadline. The laxity is determined before the task is scheduled and is thus an indicator of how much contention for the processors the task can withstand during scheduling. The lateness, $L_i = f_i - D_i$, is the amount by which task τ_i misses its deadline, that is, a non-positive quantity for a valid schedule. The lateness is determined after the tasks have been scheduled and is thus an indicator of the quality of the schedule. The maximum lateness refers to the lateness of only one task – the one with its lateness closest to 0 – and is thus an indicator of "how far" from infeasibility the schedule is and how much additional background workload the schedule can handle.

4.3 The slicing technique

We solve the deadline-distribution problem with an improved version of the *slicing technique* proposed in [5]. A fundamental concept in the slicing technique is that of *critical path*. A critical path in a task graph is one for which a given performance measure assumes a minimum or maximum value. Correct identification of a critical path is crucial for the quality of the deadline distribution and the system's

schedulability. A *critical path metric* is used to assess the criticalness of each path in the task graph.

When a critical path has been identified, the E-T-E deadline is distributed over the tasks in the critical path. This is done by assigning *slices*, non-overlapping execution windows, of the E-T-E deadline to each of the tasks in the critical path. The slices are derived governed by the constraint that the arrival time of a task must be equal to the absolute deadline of its predecessor in the critical path. Note that, whereas the execution time windows of tasks in the same path cannot overlap, the execution windows of tasks in different paths may overlap and thus are subject to contention for available processors during scheduling. Therefore, the size of each task slice must be derived in such a way that the task can be scheduled to meet its timing constraints even in the presence of other, overlapping, task slices or task release jitter as caused by interprocessor communication delays. This can be achieved by assigning an ample laxity to each task.

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 [5]. When the assignment is not entirely fixed, however, finding the best critical path is no longer an easy task. The reason is that it is not yet known what pairs of tasks will be afflicted with interprocessor communication overhead. Therefore, the deadline-distribution algorithm must rely on the prediction of the "possibly best" critical path.

In [12] we presented an adaptive slicing technique that was able to make better predictions on the critical path under relaxed locality constraints than did the original technique in [5]. The predictions in [12] were based on two observations. First, we found that in systems where the application parallelism exceeds the number of processors available, it is a good strategy to assign longer slices to tasks whose execution times exceed a certain threshold. The rationale for this is that tasks with longer execution times are likely to be most vulnerable in the case of high resource contention. Second, we found that it is a good strategy to assume that there will be no communication cost between tasks whose processor assignments are not known. The intuition behind this is that many task assignment and scheduling algorithms tend to cluster tasks that communicate heavily [1]. Furthermore, in many distributed real-time applications the communication volume is quite low [17], and therefore any communication cost will be relatively small compared to the task execution times. In [12], we also showed that, even in the presence of significant communication cost, it is better to assume no communication cost as this will yield the largest amount of overall laxity to distribute over the application tasks.

4.4 Deadline-distribution algorithm

A pseudo-code form of the algorithm for distributing E-T-E deadlines over the tasks according to the slicing technique is given in Figure 1. The algorithm takes a task graph as the input and produces an annotated task graph containing information about task arrival times and relative deadlines. The various steps in the algorithm are described below in detail.

Algorithm SLICING:

1. initialize set Π with all tasks in the task graph;
2. **while** { $\Pi \neq \emptyset$ } **loop**
3. find a critical path Φ in Π that minimizes metric R;
4. distribute the E-T-E deadline of Φ by assigning arrival times and deadlines to the tasks in Φ;
5. **for** { each task τ in Φ } **loop**
6. **for** { each immediate predecessor τ_p of τ } **loop**
7. assign an E-T-E deadline to τ_p that is equal to the arrival time of τ;
8. **end loop**;
9. **for** { each immediate successor τ_s of τ } **loop**
10. assign an arrival time to τ_s that is equal to the absolute deadline of τ;
11. **end loop**;
12. **end loop**;
13. remove all tasks in Φ from Π;
14. **end loop**;

Figure 1: The deadline-assignment algorithm.

Initialize task set (Step 1): Assume that all input/output tasks have already been assigned appropriate arrival times and E-T-E deadlines, respectively, according to the temporal requirements of the application. All tasks in the task graph are then inserted into a task set Π that represents all tasks not yet assigned arrival times and deadlines. The time complexity of this step is $O(|\mathbf{N}|)$. This does not include the time for initializing data structures used by the critical path metric R.

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 tasks in Π. The critical path is identified using the critical path metric, R. The time complexity of a breadth-first traversal of the task graph is $O(|\mathbf{N}| + |\mathbf{A}|)$. Therefore, for any acyclic task graph, the total time complexity of this step, taken over all iterations of the main loop, is $O(|\mathbf{N}|^2)$, not counting the time needed for evaluating R.

Distribute the E-T-E deadline (Step 4): The E-T-E deadline D_Φ of the critical path Φ found in Step 3 is distributed over the tasks in Φ. The deadline distribution is governed by the constraint that the arrival time of a task must be equal to the absolute deadline of its immediate predecessor in Φ. Thus, all tasks in the path will be assigned slices of the E-T-E deadline. The total time complexity of this step is $O(|\mathbf{N}|)$.

Attach the remaining tasks (Steps 5 – 12): The tasks in Φ now constitute a "spine" to which the remaining tasks must attach, that is, adapt their arrival times and absolute deadlines. Therefore, the arrival time for each task not in Φ is set to the latest absolute deadline of any immediate predecessor task in Φ. Similarly, the absolute deadline for each task not in Φ is set to the earliest arrival time of any immediate successor task in Φ. The total time complexity of this step is $O(|\mathbf{N}| + |\mathbf{A}|)$.

Remove critical-path tasks (Step 13): The tasks in Φ are removed from Π to mark that they have been assigned arrival times and deadlines. The arrival times and deadlines of tasks not in Φ now constitute new E-T-E deadlines in Π. The total

time complexity of this step is $O(|\mathbf{N}|)$.

Repeat until no tasks remain (Steps 2 and 14): The main loop in the algorithm is repeated until no tasks are left in Π.

Adding the time complexities of the steps described above, the worst-case time complexity of the algorithm is $O(n^2)$, not counting the time needed for evaluating R.

4.5 Critical path metrics

In [5], Di Natale and Stankovic introduced two metrics for the slicing technique. The first metric, the *normalized laxity ratio* (*NORM*), is the ratio of the overall laxity to the sum of the execution times of all tasks in a path Φ. With this metric, laxity is assigned in proportion to task execution time. The metric value, R_{NORM}, and the relative deadline, d_i, for each task τ_i are consequently defined as

$$R_{\mathrm{NORM}} = (D_\Phi - \sum_{\tau_i \in \Phi} \bar{c}_i) / \sum_{\tau_i \in \Phi} \bar{c}_i \quad (2)$$

$$d_i = \bar{c}_i(1 + R_{\mathrm{NORM}}) \quad (3)$$

The second metric in [5], the *pure laxity ratio* (*PURE*), is the ratio of the overall laxity to the number of tasks, n_Φ, in a path Φ. With this metric, all tasks are assigned an equal share of laxity. The metric value, R_{PURE}, and the relative deadline, d_i, for task τ_i are thus

$$R_{\mathrm{PURE}} = (D_\Phi - \sum_{\tau_i \in \Phi} \bar{c}_i) / n_\Phi \quad (4)$$

$$d_i = \bar{c}_i + R_{\mathrm{PURE}} \quad (5)$$

In [12], we showed that none of the NORM or PURE metrics are suitable for systems with relaxed locality constraints owing to their lack of information on contention over resources. As a remedy, we therefore introduced two concepts as an aid in improving the performance of the slicing technique, namely, *execution time threshold* and *virtual execution time*. The execution time threshold is a mechanism for guaranteeing that only certain tasks are allotted extra laxities. By using the execution time threshold to filter out tasks with sufficiently large execution times, we managed to improve the performance in those situations where task graph parallelism cannot be fully exploited. The purpose of a virtual execution time is to make a task appear computationally more consuming than it actually is. By assigning virtual execution times that are larger than the real execution times, the deadline-distribution algorithm will allocate more laxity to those tasks for which the real execution time is equal to or above the given execution time threshold.

On the basis of these two concepts, we introduced the *globally adaptive laxity ratio* (*ADAPT-G*). This metric is similar to the PURE metric but has a virtual execution time of $\hat{c}_i$ instead of the estimated execution time, $\bar{c}_i$. The ADAPT-G metric is adaptive in the sense that the amount of assigned surplus is not fixed but will adapt itself to the degree of task graph parallelism that can be exploited on the system. The virtual execution time for task τ_i is defined as

$$\hat{c}_i = \begin{cases} \bar{c}_i & \text{if } \bar{c}_i < c_{thres} \\ \bar{c}_i(1 + k_G \xi / m) & \text{if } \bar{c}_i \geq c_{thres} \end{cases} \quad (6)$$

where k_G is the *global adaptivity factor*, ξ is the *average task graph parallelism*, and m is the number of processors in the system. The expression $k_G \xi / m$ constitutes a *surplus factor* that defines the amount of laxity by which the real execution time should be increased for those tasks whose execution times exceed c_{thres}. The average task graph parallelism, ξ, is defined as the application workload divided by the length of the longest path in $\mathbf{T}$, that is

$$\xi = \sum_{\tau_i \in \mathbf{T}} \bar{c}_i / \max\{SL(\tau_j) : \tau_j \in \mathbf{T}\} \quad (7)$$

Our main contribution in this paper is to propose the *locally adaptive laxity ratio* (*ADAPT-L*) metric, which is an improvement of the ADAPT-G metric in the sense that ADAPT-L is able to identify the available parallelism that affects only a certain task. For ADAPT-L, the virtual execution time is defined as:

$$\hat{c}_i = \begin{cases} \bar{c}_i & \text{if } \bar{c}_i < c_{thres} \\ \bar{c}_i(1 + k_L |\Psi_i| / m) & \text{if } \bar{c}_i \geq c_{thres} \end{cases} \quad (8)$$

where k_L is the *local adaptivity factor*, Ψ_i is the *parallel set* of τ_i, and m is the number of processors in the system. The parallel set Ψ_i is the set of tasks that are potential candidates for executing in parallel with τ_i, that is, those tasks that are neither predecessors nor successors of τ_i. ADAPT-L introduces the extra complexity to the deadline-distribution algorithm that a parallel set must be calculated for each task. The parallel set Ψ_i for task τ_i can easily be found by first constructing the transitive closure $\mathbf{G}^*$ for the task graph $\mathbf{G}$, and then including in Ψ_i only those tasks that are neither reachable from τ_i, nor can reach τ_i. The construction of the transitive closure $\mathbf{G}^*$ can be done during the initialization phase (Step 1) of the deadline assignment algorithm and takes time $O(n^3)$ [18].

5 Experimental setup

5.1 System architecture

We have used an experimental platform based on a heterogeneous multiprocessor architecture with a shared bus interconnection network. The system size ranges from two to eight processors. The number of processor classes in the system, $m_e = |\mathbf{E}|$, was randomly chosen to be one, two, or three. The class, $e(p_q)$, of each processor p_q in the system was randomly chosen from the set, $\mathbf{E}$, of generated processor classes. 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.

5.2 Workload

In all experiments[2], a set of 1024 task graphs was generated using a random task graph generator. Each task graph contained between 40 and 60 tasks. Task execution times were chosen at random assuming a uniform distribution with a mean execution time, c_{mean}, of 20 time units. The execution time for each task deviated by at most $\pm 25\%$ from the mean execution time c_{mean} for different processor classes. To mimic the situation that a task requires special hardware resources for its execution, we also deemed a task inappropriate for execution on a particular processor class with a probability of 5%.

To mimic different application scenarios, task execution times were chosen based on the *execution time distribution* (*ETD*), the maximum deviation (in percent) of a task's execution time from the mean execution time c_{mean}. Thus, for a given ETD, the task execution times were chosen at random to be in the range $[c_{mean}(1 - \mathrm{ETD}), c_{mean}(1 + \mathrm{ETD})]$. To evaluate the performance of the metrics under variably tight timing constraints, an E-T-E deadline was chosen for each input–output task pair based on the *overall laxity ratio* (*OLR*), the ratio of the E-T-E deadline to the average accumulated task graph workload. The precedence constraints in the task graph were also randomly generated. The number of successors/predecessors of each task was chosen at random to be in the range of one to three, and the depth of the task graph was chosen at random to be between eight and 12 levels.The number of data items in each message passed between a pair of tasks was chosen in such a way that the *communication–to–computation cost ratio* (*CCR*) of the average message communication cost to the average task execution time corresponded to 0.1.

5.3 Estimation of WCET

We investigate three different WCET estimation strategies. For the *WCET-AVG* strategy, $\bar{c}_i$ is calculated as the average of all valid execution times, taken over all processor classes:

$$\bar{c}_i = \sum_{e_k \in \mathbf{E}} c_i[e_k] / |\mathbf{E}| \tag{9}$$

For the *WCET-MAX* strategy, $\bar{c}_i$ is set to the maximum of all valid execution times, that is

$$\bar{c}_i = \max_{e_k \in \mathbf{E}} c_i[e_k] \tag{10}$$

Finally, for the *WCET-MIN* strategy, $\bar{c}_i$ is set to the minimum of all valid execution times:

$$\bar{c}_i = \min_{e_k \in \mathbf{E}} c_i[e_k] \tag{11}$$

5.4 Task assignment and scheduling

We used a baseline task assignment and scheduling strategy based on a list scheduling version of the earliest-deadline-first (EDF) algorithm. For each scheduling step, our EDF algorithm selected one task (the one with the closest absolute deadline) from all ready tasks. The selected task was then scheduled on the available processor that yielded the earliest start time, taking into account possible communication cost and the arrival time constraints of the task. The set of ready tasks was updated with the immediate successors of the scheduled task. The complexity of this algorithm is $O(n^2 * m)$ for a system with n tasks and m processors.

[2] All modeling and simulations in the experiments were performed within the GAST [19] evaluation framework.

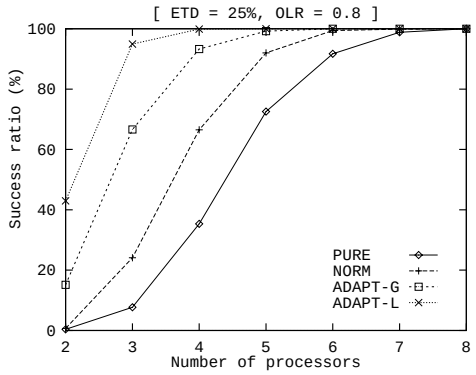

Figure 2: Success ratio as a function of system size.

6 Experimental evaluation

This section evaluates the performance (success ratio) of the metrics presented in Section 4.5. The experimental evaluation was made using the deadline-distribution algorithm in Figure 1 and the experimental setup in Section 5. Unless otherwise noted, all simulations were performed assuming a scenario with $\mathrm{ETD} = 25\%$ and $\mathrm{OLR} = 0.8$. The following default parameter values were used for the adaptive metrics during the experiments: $c_{thres} = 1.0 * c_{mean}$, $k_G = 1.5$, $k_L = 0.2$. The estimated WCET for a task was calculated according to the WCET-AVG strategy.

6.1 Effect of system size on performance

Since one of our objectives is to find metrics that perform well for high processor contention, it is interesting to study how the slicing metrics will behave for different system sizes. By varying the number of processors, we can identify the capability of each metric to exploit the available application parallelism on the platform architecture. Intuitively, all metrics should perform worse for small system sizes than for a larger system. For a small system, that is, where the application parallelism exceeds the number of available processors, the contention between tasks over a few available processors forces multiple tasks to be scheduled within overlapping execution windows on the same processor. As the system size increases, more parallelism in the task graph can be exploited, which will decrease the contention over processors. Consequently, the success ratio will increase with increasing system size until a point at which all generated task graphs are successfully scheduled. This behavior is duly corroborated by the plots in Figure 2.

With a deadline-driven scheduling strategy, tasks with longer execution times will be the most vulnerable to processor contention as shorter tasks are more likely to have shorter

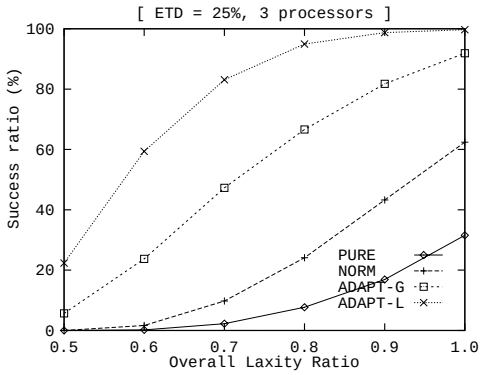

Figure 3: Success ratio as a function of OLR.

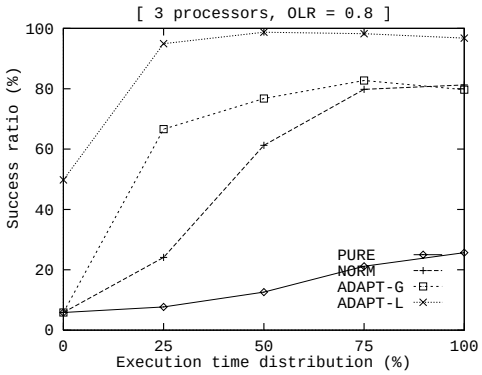

Figure 4: Success ratio as a function of ETD.

deadlines. As a consequence, longer tasks should be assigned more laxity to compensate for this imbalance. The figure shows how the PURE metric, with its equal-share distribution strategy, exhibits the worst performance among all metrics. For example, only 35% of all scheduling attempts succeed on a system with four processors when PURE is used. The performance of the NORM metric is significantly better, mainly owing to its strategy to assign deadlines in proportion to task execution times. This approach gives tasks with longer execution times more laxity and hence a better chance to be scheduled feasibly. For the four-processor system, more than 65% of all scheduling attempts succeed using NORM.

The figure clearly illustrates that the non-adaptive metrics (PURE and NORM) do not perform well for small systems. For example, the fraction of successful scheduling attempts does not exceed 30% for a system with three processors for any non-adaptive metric. The explanation for the poor performance is that these metrics do not account for the resource contention that occurs when the application parallelism exceeds the number of available processors. Hence, many task deadlines will be too short to guarantee a feasible schedule. This shortcoming is partly remedied with the ADAPT-G metric. By assigning task laxities based on the knowledge that application parallelism may not be fully exploited on the system, it is reasonable to believe that a higher performance can be attained with ADAPT-G. As the plots indicate, this extra intelligence of ADAPT-G gives a significant performance increase for small systems. For example, more than 60% of all scheduling attempts succeed for the three-processor system when ADAPT-G is used. Note that this is more than twice the performance of the best non-adaptive metric.

The ADAPT-L metric exhibits the best performance in this experiment. For example, the plots indicate that around 95% of all scheduling attempts succeed for the three-processor system when the ADAPT-L metric is used. This is a three-fold improvement as compared to the non-adaptive metrics. The improvement is even more stunning for a two-processor system: the number of successful scheduling attempts is more than an order of magnitude higher for ADAPT-L than for the non-adaptive metrics, and four times higher than for the ADAPT-G metric. The superior performance associated with ADAPT-L can be attributed to the detailed knowledge it possesses regarding the contention situation for individual tasks. This is in contrast to ADAPT-G, where the potential contention for each task is derived using the same constant value, namely the average task graph parallelism. However, it must be recalled that the superior performance of ADAPT-L comes at the price of a higher time complexity.

6.2 Effect of OLR on performance

Another goal with a deadline-distribution strategy is that it should use metrics that perform well for varying tightness of the E-T-E deadlines. In particular, by varying the value of OLR when there is significant processor contention, we can identify the capability of each metric to exploit the available surplus time and to distribute it wisely. The plots in Figure 3 show some salient results for a system with three processors.

Again, we observe the same relative performance of the metrics. Here, too, ADAPT-L outperforms the other metrics by a significant amount. For example, the performance gain as compared to the non-adaptive metrics is nearly an order of magnitude for tight deadlines. The ADAPT-G metric also performs consistently better than the non-adaptive metrics. For tight deadlines, the plots indicate a three-fold increase in performance for ADAPT-G over the non-adaptive metrics.

6.3 Effect of ETD on performance

So far, our adaptive metrics have been shown to outperform their non-adaptive counterparts under all system sizes and all degrees of deadline tightness. In this section, we will identify the robustness of each metric under different application scenarios with varying execution time distributions. The plots in Figure 4 illustrate the success ratio as a function of ETD assuming a fixed system size and a fixed OLR. We show here the results for $\mathrm{OLR} = 0.8$ and three processors while varying ETD from 0% to 100% in steps of 25%.

The observed trends in this experiment are quite similar to the ones demonstrated in the previous experiments, with one notable exception. As the plots clearly indicate, the performance of the NORM metric is not as consistent as in the previous studies. Instead of consistently being inferior to ADAPT-G, the performance of NORM will increase past ADAPT-G as ETD gets large enough. This behavior can be attributed to the proportional-share distribution strategy of NORM. With an increasing proportion of short tasks, there will be more tasks with shorter deadlines competing for the

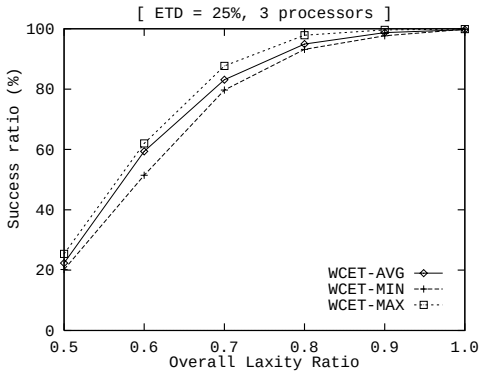

Figure 5: Success ratio for ADAPT-L as a function of OLR for different WCET estimation strategies.

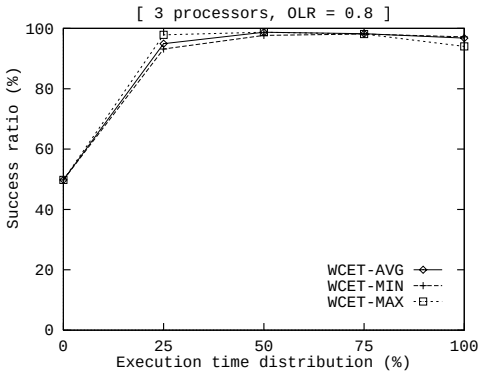

Figure 6: Success ratio for ADAPT-L as a function of ETD for different WCET estimation strategies.

processors. NORM is able to compensate long tasks with a larger amount of laxity to handle the contention. Because it is based on the PURE metric, ADAPT-G cannot handle this situation with the same amount of flexibility.

Note that, when $\mathrm{ETD} = 0\%$, the PURE, NORM, and ADAPT-G metrics all converge to the same success ratio (6%). This is because all tasks have identical estimated execution times $\bar{c}_i$. It is easy to see that, with identical execution times, the deadline for each task τ_i in a critical path Φ will be $d_i = D_\Phi / n_\Phi$, regardless of what metric is used. In the case of ADAPT-G, the virtual execution times $\hat{c}_i$ are also identical since, for $\mathrm{ETD} = 0\%$, the original execution times of all tasks are equal to or higher than the execution time threshold c_{thres} and the surplus factors (k_S and k_G, respectively) are constant. In the case of ADAPT-L, however, the virtual execution time of each task is defined by the size of its parallel set and may thus differ between tasks. This property of ADAPT-L gives it an order-of-magnitude performance increase for systems with uniform task execution times.

Also, note the anomalous behavior of ADAPT-G and ADAPT-L as ETD exceeds 50%. As shown in [20], the performance of the adaptive metrics is most vulnerable to changes in ETD. For the chosen values of k_G and k_L, the performance of both ADAPT-G and ADAPT-L will drop slightly for applications with a large ETD.

6.4 Effect of WCET estimation strategy on performance

Since we are assuming a heterogeneous distributed system, it is interesting to investigate how the strategy for calculating $\bar{c}_i$, the estimated WCET of τ_i, affects the schedule quality. In this section, we investigate the three WCET estimation strategies presented in Section 5.3.

The plots in Figure 5 illustrate the success ratio for ADAPT-L as a function of WCET estimation strategy assuming a three-processor system. As can be seen in the plots, the overall best performance is attained when the pessimistic WCET-MAX strategy is used. By accounting for the worst-case situation that can occur during scheduling, this strategy outperforms WCET-AVG by approximately 5%. The worst performance is attained with WCET-MIN owing to its overly optimistic estimations regarding the final task assignment. As can be seen in the plots, this strategy performs approximately 5% worse than WCET-AVG.

It should be noted that WCET-MAX is not as robust for all application scenarios. Figure 6 illustrates the success ratio as a function of ETD assuming a fixed OLR. As ETD increases past 75%, the performance of WCET-MAX becomes worse than that of the other strategies because the larger presence of tasks with long execution times causes too much overall laxity to be consumed from task with short execution times.

The results from this study indicate that, for systems with uniform or near-uniform task execution times, the WCET-MAX strategy is the best choice. For systems with a large distribution of task execution times, on the other hand, the WCET-AVG strategy is the preferred choice.

7 Discussion

7.1 Finding the best adaptive parameters

To be able to apply the adaptive metrics (ADAPT-G and ADAPT-L) in practice, it is important to find what the value of the adaptivity factors k_G and k_L must be for a given application. In the general case, one must be aware of the fact that there exists no overall best value for any factor. However, we believe that the values used in our experiments are also useful for many other applications. Although the adaptivity factors of ADAPT-G and ADAPT-L are clearly sensitive to the actual application parallelism, the adaptive metrics will still outperform the non-adaptive counterparts as long as the parallelism deviates within reasonable bounds from the chosen values.

7.2 Complexity issues

Whether to use ADAPT-G or ADAPT-L depends to a large extent on the nature of the target system. For example, in a system that is scheduled off-line, the cost of the deadline-distribution algorithm are often not of major concern and hence the $O(n^3)$ algorithm with ADAPT-L could be used. For a system with on-line scheduling, on the other hand, tasks typically arrive dynamically and hence the scheduling complexity is a major source of consideration. Here, the $O(n^2)$ algorithm with ADAPT-G may be a better choice. Another property of the system that will affect the choice of metric is the task assignment and scheduling strategy used. If an

$O(n^2)$ polynomial-time scheduling algorithm is used, applying ADAPT-G adds little to the total complexity. However, if a branch-and-bound algorithm is used, its high complexity would make ADAPT-L a viable alternative as its complexity would be negligible in comparison.

7.3 Future work

We believe that the techniques presented here can be applied not only to computational resources such as processors but also to general resources including shared data structures. Future work would include evaluation of techniques that can also take such general resource requirements into account.

Although the slicing technique has been evaluated under a time-driven non-preemptive task dispatching policy in this paper, it is not restricted to that policy as we showed by implications I1 and I2 in Section 1. Future work would therefore encompass exploring the performance of the new metrics under various task assignment and scheduling policies.

8 Conclusions

Distribution of E-T-E deadlines over tasks 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 tasks are not pre-assigned to particular processors. In this paper, we have proposed a new adaptive metric that will significantly improve the performance of our original adaptive slicing technique [12]. The results of an extensive simulation study show that the new metric outperforms all existing metrics over a wide range of system configurations. In particular, we find that, for small systems, the new metric outperforms non-adaptive metrics by as much as an order of magnitude. Furthermore, it outperforms the previously-proposed adaptive metric with a three-fold increase in performance.

References

[1] 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.

[2] F. Wang, K. Ramamritham, and J. A. Stankovic, "Bounds on the Performance of Heuristic Algorithms for Multiprocessor Scheduling of Hard Real-Time Tasks," *Proc. of the IEEE Real-Time Systems Symposium*, Phoenix, Arizona, Dec. 2–4, 1992, pp. 136–145.

[3] 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.

[4] C.-J. Hou and K. G. Shin, "Allocation of Periodic Task Modules with Precedence and Deadline Constraints in Distributed Real-Time Systems," *Proc. of the IEEE Real-Time Systems Symposium*, Phoenix, Arizona, Dec. 2–4, 1992, pp. 146–155.

[5] 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.

[6] 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.

[7] 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.

[8] M. Saksena and S. Hong, "Resource Conscious Design of Distributed Real-Time Systems: An End-to-End Approach," *Proc. of the IEEE Int'l Conf. on Engineering of Complex Computer Systems*, Montreal, Canada, Oct. 21–25, 1996, pp. 306–313.

[9] 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.

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

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

[12] J. Jonsson and K. G. Shin, "Deadline Assignment in Distributed Hard Real-Time Systems with Relaxed Locality Constraints," *Proc. of the IEEE Int'l Conf. on Distributed Computing Systems*, Baltimore, Maryland, May 27–30, 1997, pp. 432–440.

[13] S. Cheng, J. A. Stankovic, and K. Ramamritham, "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems," *Proc. of the IEEE Real-Time Systems Symposium*, New Orleans, Louisiana, Dec. 2–4, 1986, pp. 166–174.

[14] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, Sept. 1993.

[15] M. Di Natale and J. A. Stankovic, "Applicability of Simulated Annealing Methods to Real-Time Scheduling and Jitter Control," *Proc. of the IEEE Real-Time Systems Symposium*, Pisa, Italy, Dec. 5–7, 1995, pp. 190–199.

[16] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," *Annals of Discrete Mathematics*, P. L. Hammer, E. L. Johnson, and B. H. Korte, Eds., vol. 5: Discrete Optimization II, pp. 287–326. North-Holland, Amsterdam, 1979.

[17] R. S. Raji, "Smart Networks for Control," *IEEE Spectrum*, pp. 49–55, June 1994.

[18] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, 1990.

[19] J. Jonsson, "GAST: A Flexible and Extensible Tool for Evaluating Multiprocessor Assignment and Scheduling Techniques," *Proc. of the Int'l Conf. on Parallel Processing*, Minneapolis, Minnesota, Aug. 10–14, 1998, pp. 441–450.

[20] J. Jonsson, *The Impact of Application and Architecture Properties on Real-Time Multiprocessor Scheduling*, Ph.D. thesis, School of Electrical and Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, Sept. 1997.