J.King Saud Univ., Vol.9, Computer & Information Sci., pp. 57-93 (A.H. 1417/1997)

Prospects for Predictable Dynamic Scheduling in RTDOS

Bradley R. Swim, Murat Tayli and Mohamed Benmaiza
Computer Science Department, College of Computer & Information Sciences
King Saud University, Riyadh, Saudi Arabia

(Received,10 April 1995; accepted for publication, 09 October 1995)

Abstract. Future dynamic distributed hard real-time systems may control unpredictable environments and will need operating systems that can handle unknown and changing task populations. In this extreme case, not only is task scheduling totally dynamic, but the system's topology and architecture must adapt to unforeseen configurations. This paper addresses the difficult problem of dynamic task scheduling in a Real Time Distributed Operating System (RTDOS). RTDOS is unique because it possesses the potential to map, at execution time, a flexible topology of networked nodes onto a network of tasks. Attempts will be made to characterize the predictable adaptability of the scheduler so as to relax the pre-run-time scheduling requirements for an RTDOS application. A scheduler architecture and dynamic deadline guarantee scheme are presented along with some experimental results.

1. Introduction

Hard real-time distributed systems can be classified into 4 major clusters with respect to their environment and behavior (Fig. 1). Even though many hard real-time systems of today are considered to be static within a predictable environment (cluster A) [1], current and future systems are seen to be more dynamic and controlling *unpredictable environments* (cluster C). The environment's flexibility requires new task types to be injected into the system even during production mode (e.g. space exploration, battle management, undersea exploration) [2].

Today's research and development in hard distributed real-time systems focuses on dynamic systems controlling predictable environments (cluster B). These systems assume that the controlled environment is predictable to the extent that all application task types are known. From the task scheduling perspective, not all task instantiations are scheduled statically. New task requests, from predefined task types, can be scheduled dynamically within known limits. The dynamic aspect of these systems is confined to task creation and scheduling.

The scheduling complexity follows an exponential path (Fig. 1) between static predictable systems and dynamic unpredictable systems (clusters A, B, C). It is obvious that task scheduling is complicated significantly if the operating system handles unknown and changing task populations. In this extreme case, not only is task scheduling totally dynamic, but the system's topology and architecture must adapt to unforeseen configurations.

Given the complexity of task scheduling for cluster C, many assumptions must be simplified in order to build an efficient dynamic task scheduler. Currently, our Real Time Distributed Operating System (RTDOS) [3] resides in cluster B, but we are investigating the extent to which we can push RTDOS to control an unpredictable environment. RTDOS is projected to be able to adapt itself to unforeseen configurations during production mode operations.

This paper describes current research underway to develop a predictable dynamic scheduler for RTDOS. The existing platform currently supports predictable communication between tasks distributed within the network [4], and synchronized clocks [5]. Because we believe that a key solution to the dynamic scheduling problem is the timely availability of resources, namely the CPU, we have implemented a Task Manager along with a medium-level CPU scheduler on top of an existing hardware low-level scheduler. The medium-level CPU scheduler enforces task deadlines even under transient overloads and improves CPU utilization significantly (discussed below in section 3.5). Building upon this predictable platform, the predictability of our scheduling and time-dependent resource management approaches will be analyzed in order to formalize the behavior and limitations of our solution.

The paper is organized as follows. The next section highlights existing scheduling approaches with a focus on dynamic scheduling. Section 3 explains the architectural model and design of the RTDOS scheduler along with some experimental results from an RTDOS scheduler prototype.

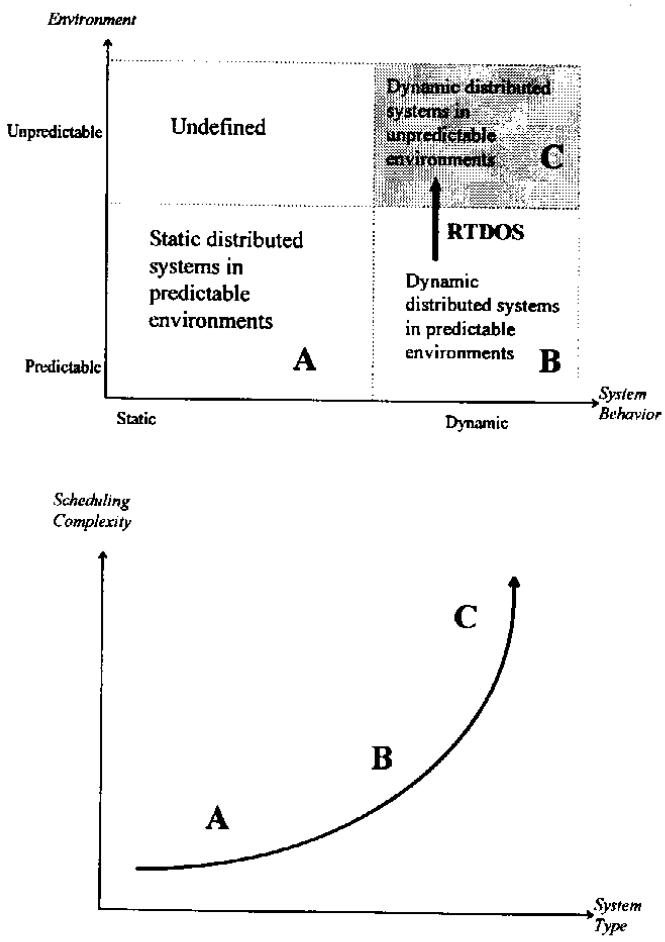

Fig. 1. Hard real-time distributed system classification.

2. Dynamic Scheduling

In hard real-time systems, deadline task scheduling is crucial because missed deadlines have severe consequences that compromise safety, security, and cost constraints. Tasks can be characterized by their:

1. Periodicity: tasks may be classified as periodic or aperiodic.
2. Deadlines: along a spectrum from hard to soft to none. Hard deadlines, if missed, imply catastrophic system failure. Less hard deadlines, if violated, compromise system integrity and utility, yet may still permit the system to operate within defined safety limits. Some tasks are background and have no deadlines.

3. Precedence: the application may define a partial execution order for a set of tasks because of logical relationships between tasks, communication constraints, or resource availability.
4. Resources requirements: CPU, I/O devices, communication, memory, etc.

Static systems are those where complete knowledge of all of the above task characteristics is known *a priori*. In such systems, static scheduling can determine a feasible schedule *off-line* for all possible task requests. The bulk of static scheduling approaches limit their task sets to have periodic tasks where the task request times and other characteristics such as deadline, computation time, and precedence are known. If aperiodic tasks with deadlines exist, then they are mapped into equivalent periodic tasks. Optimal scheduling algorithms, such as the one described in [6], can derive a feasible schedule on multiple processors for all known periodic tasks considering their request time, computation, deadline, as well as their precedence and exclusion relations. The schedule is implemented at run-time by a deterministic task dispatcher, often based on task priorities.

Dynamic systems do not require *a priori* knowledge of all tasks and their characteristics. Yet, *critical* periodic tasks are scheduled off-line so that their resources can be preallocated (if a critical task fails, the system fails). For the remainder of tasks with *soft* deadlines, a scheduler can determine *on-line* (dynamically) if a task instantiation can meet its deadline without violating other task deadlines in the running system [7, 8, 9, 10, 11]. Heuristics are used in all cases to speed up the otherwise lengthy and costly process of determining a feasible schedule, in theory NP-complete. Because these algorithms are distributed, they apply dynamic load distribution to globally search the network for a suitable remote host for the incoming task. Attempting near-optimal scheduling of all aperiodic tasks at runtime can be complex and consumes precious time. Some authors have shown that heuristics can achieve satisfactory results in such algorithms [9].

3. RTDOS Predictable Dynamic Scheduling

In RTDOS, critical and all known periodic tasks can be scheduled pre-run-time. Critical tasks are assumed to never exceed estimated resource requirements, while periodic tasks can be defined with average-case resource estimates. All pre-allocated tasks can have resource and precedence constraints (e.g. I/O, memory, IPC, order of execution and data availability). Off-line scheduling algorithms consider fine-grained task execution

ordering when determining a feasible schedule (unlike on-line techniques). This will yield a verifiable feasible schedule that guarantees satisfaction of timing, resource, and precedence constraints for many types of tasks (most of them having hard deadlines).

There remains the need to schedule on-line unknown periodic, aperiodic and background tasks with deadlines and resource constraints. Because dynamic scheduling may be the only alternative in unpredictable environments, the current research aims to permit as many tasks as possible to be scheduled on-line using the novel technique proposed in section 3.2. Before discussing the RTDOS scheduler, a brief overview of the RTDOS system architecture is presented.

3.1 RTDOS architecture overview

RTDOS is an operating system running on a multi-loop topology of transputers [12]. Although transputers are utilized in the current development platform, the architecture of RTDOS does not depend on that particular processor. RTDOS implements CSP [13] as the basic communication model for application level processes. Figure 2 illustrates the topology of the transputer network assumed for RTDOS.

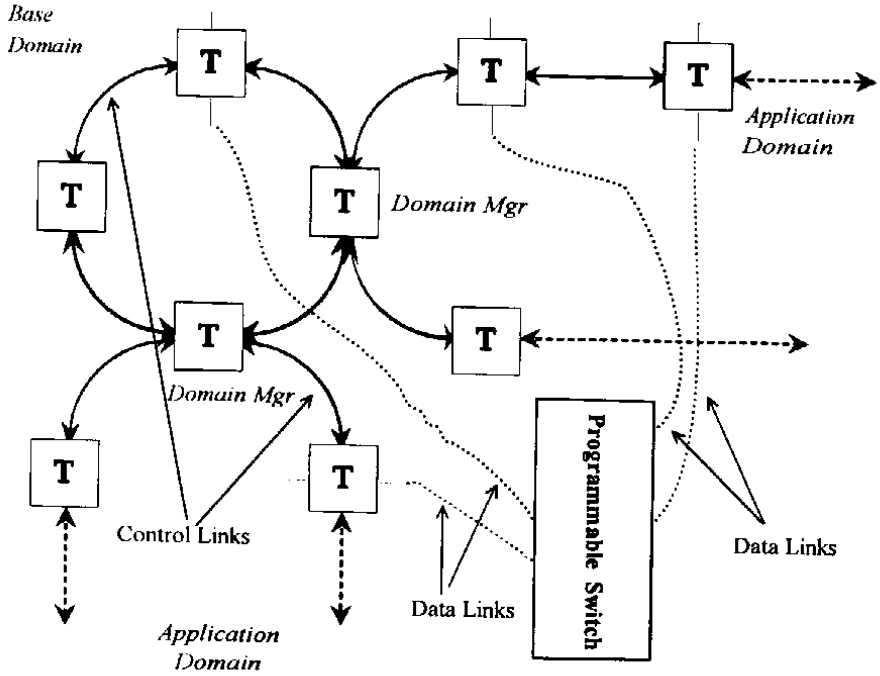

Fig. 2. RTDOS network topology and communication architecture.

In summary, an RTDOS network is a set of intersecting loops of transputers. The RTDOS topology as a minimum has at its core a Base Domain. At each transputer of this loop, it is possible to attach additional loops scaled to meet application demands. These loops are intended as Application Domains. The transputers connecting the main system loop with each application/server domain are called Domain Managers. These Domain Managers are reserved for system functions (e.g. monitoring, scheduling, etc.). Domain nodes are linked through bi-directional links to form the *control loop* . The control loop is dedicated for a datagram-based service between RTDOS kernels replicated on each transputer. Transputer Links not used by the control loop are intended for application specific data traffic between nodes. These data links can be all wired to a Programmable Switch or to each other. At run time, the links are allocated between communicating tasks on disjoint transputers. The application can choose to either allocate the links in a permanent way (due to application timing constraints) or temporarily for the duration of one communication transaction. Moreover, the network topology can be dynamically reconfigured if the control loops are formed via a programmable link switch. RTDOS's predictable circuit-switched point-to-point Inter Process Communication (IPC) facility is described in more detail in [4,14].

The transputer architecture [12] presents unique opportunities and challenges when compared to more traditional architectures. Each transputer has at least 4 high-speed serial links that can be connected with other transputers or a link switch to form a network. The hardware supports many threads on one CPU (they are called processes, but in practice they share the same address space) and dispatches them according to 2 priorities - High and Low. This dispatching (and forced timeslicing) is performed when a Low-Priority thread blocks itself (e.g. on I/O), or executes one of many designated descheduling instructions (e.g. timer, jump/loop, process start/end, etc.). Context switching of threads is very fast - in the order of less than 1 microsecond. On the other hand, the transputer does not have a memory management unit. This limitation of no virtual memory renders program code relocation and process migration from one processor to another completely impractical.

The RTDOS topology and architecture can be exploited to provide alternate approaches to time-dependent resource management.

3.2 RTDOS scheduler architecture

The architecture of RTDOS enables a scaleable approach to processing power because a given application can be mapped to one or more domains (leading to a given application architecture). Moreover, after the initial application architecture has been

determined, RTDOS can quickly and dynamically adapt the configuration of a domain to possess more or less processing and communication power (more or less nodes - within limits). Therefore, it is possible to dynamically scale and reconfigure the network to meet unforeseen resource needs of an application in a time dependent fashion. The adaptive hybrid dynamic scheduling of RTDOS is illustrated in Fig. 3.

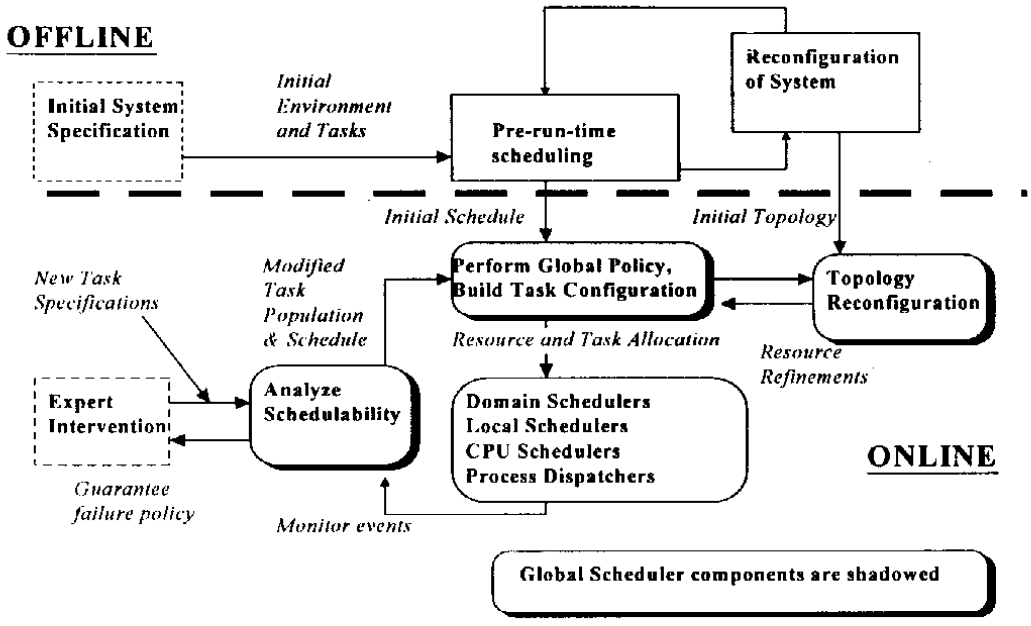

Fig. 3. The adaptive hybrid dynamic scheduling of RTDOS.

Conventional distributed scheduling attempts to map a network of tasks upon a fixed topology of networked processors. Adaptation of current systems frequently employ pools of slack resources (e.g. processor pool) to support the changing demands of the application. Nonetheless,the topology of the distributed system may remain unchanged with respect to non-uniform IPC costs. For example, if a pool of spare nodes were located on a disjoint segment or ring apart from the overloaded nodes, then IPC costs to/from the pool are different from IPC costs among the overloaded nodes. On the contrary, RTDOS has the potential to map, at execution time, a flexible topology of networked nodes onto a network of tasks. That is, a domain can be extended by "switching" in a new processor (perhaps from a pool) into the loop; or a domain can be split into 2 smaller loops. In any case, after the domain has been reconfigured, the IPC costs remain uniform and predictable. In this way, the network topology changes at run-time to absorb the demanded task network configuration.

Off-line, the application tasks' requirements are specified using a declarative language including all necessary task attributes such as deadline, period, computation time, resource requirements, precedence constraints, etc. Static analysis of the specification can determine an initial feasible periodic schedule and domain topology where resource, deadline, and precedence constraints are guaranteed. The static scheduler allocates resources for known periodic tasks as well as periodic servers for those aperiodic tasks with large laxities and high probabilities of occurrence. It also determines an initial upper bound of pre-loaded task clones (duplicates) needed to meet the resource and deadline requirements of the known dynamically scheduled tasks (e.g.known aperiodic tasks with small laxities or low probabilities of occurrence). Nodes containing a task clone are capable of hosting that dynamically scheduled task from birth to death. Static scheduling produces an initial topology, task configuration and schedule for known tasks.

RTDOS uses a 5-layered on-line scheduler model (Fig. 4). Level 5 is the Global Scheduler which loads tasks in the initial schedule onto the network as well as the current topology configuration. It is responsible to analyze the schedulability of the current system (based on events generated by Level 4), to implement guarantee failure policy based on instructions from an outside "expert", and to reconfigure the domains if necessary according to current policy in order to better fit the task topology. Any topology or policy changes are forwarded to the concerned Domain Schedulers so that their configurations can be updated. It also sets policy for task-forwarding by Domain Schedulers. That is, if a Domain Scheduler cannot guarantee a deadline, it forwards the task to another domain in hopes that the deadline can still be guaranteed.

Level 4 is the Domain Scheduler resident in every domain (Fig. 2), one per application domain. This scheduler accepts task instantiation requests and attempts to guarantee the deadlines of those tasks given total resource availability (e.g. CPU, memory, I/O devices, transputer Links, etc.). Task requests can be local (intra domain) or global (inter domain forwarding of tasks). All resources of a domain are managed and scheduled by the Domain Scheduler. It uses a set of resource "planes"(described in section 3.4) that keeps track of each resource's allocation with respect to the deadline and computation requirements of the requesting tasks. If it cannot guarantee a task, it attempts to forward (according to the global policy) the task request to another domain for servicing. It also generates scheduling events to the fifth layer so that schedulability can be analyzed.

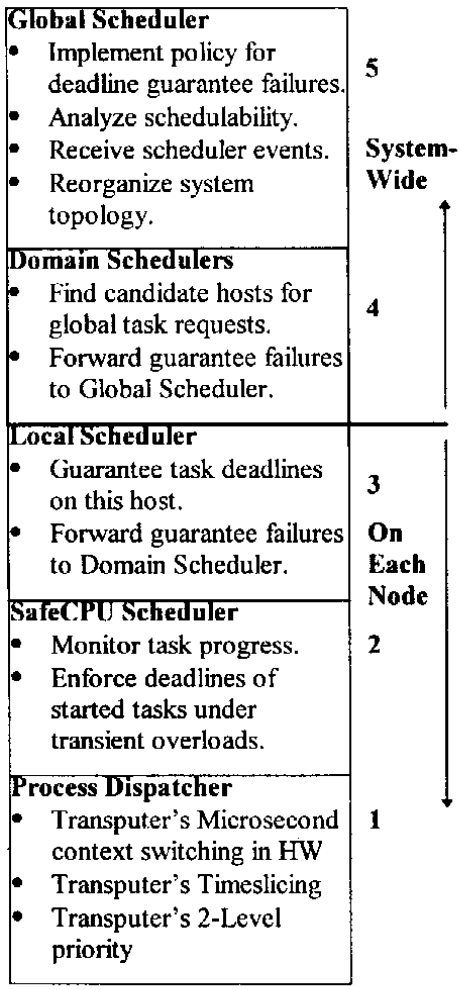

Fig. 4. RTDOS's on-line 5-layer scheduler model.

Level 3 is the Local Scheduler resident on each host. All new task requests are initially submitted to Local Scheduler, and if it cannot guarantee the task then the task is forwarded to the Domain Scheduler. If the task is schedulable, then the task is started as soon as possible.

Level 2 is the medium-level CPU scheduler: *Safe* CPU described in section 3.5. SafeCPU resides on every node in RTDOS and enforces, if necessary, task deadlines for all started tasks on its node. It improves CPU utilization and ensures that all previously guaranteed tasks meet their deadlines even under transient overloads. If any deadlines must be missed, they will be missed in an order that reflects their importance (i.e. well-behaved tasks will finish and meet their deadlines before overloaded tasks).

Level 1 is the transputer's hardware scheduler. This performs microsecond level context switching between processes on a single processor.

Level 4 and 5 work together to map the network topology to the changing application. Run-time monitors record significant task events (e.g. guarantee, failure, overrun of allotted time, etc.) whereby schedulability statistics can be produced. If time remains for an unguaranteed task, it is forwarded to a remote host for servicing. Concurrently, schedulability analysis might be able to determine a more optimal schedule and topology for the current environment. The system could then adapt by extending or collapsing its domain structures subject to expert advise. Similarly, if new tasks, previously unknown to the system, are injected into the system then the same schedulability analysis can be performed (under expert control) yielding a modified schedule and topology (see Fig. 3).

Currently, our research focus is on the dynamic scheduler and its predictability. Utilizing RTDOS's predictable IPC and synchronized clocks, the above scheduling/monitoring policies and mechanisms are being developed and formalized on the current development platform. We presume the existence of a reconfiguration and static schedule generation facility (the off-line components of Fig. 3). We also presume that future research will address the on-line schedulability and topology analysis and reconfiguration tools. These tools are important components of the system because they improve the system's reliability. When a task fails to be guaranteed, a series of network reconfigurations could ensue enabling the system to prevent future failures.

3.3 Task model

This section lays the theoretical framework for RTDOS's task model. The model assumes that tasks follow CSP semantics precisely and use RTDOS IPC exclusively for communication. This precludes shared memory models of communication reducing problems associated with exclusion constraints found in other scheduling algorithms. Additionally, tasks are scheduled by the Local Scheduler in a nonpreemptible fashion. See Fig. 5 for an illustration of the model's concepts.

The RTDOS task model supercedes other models [6,15,16] inasmuch as tasks with resource constraints, scheduled both statically and dynamically, are supported. Because the ultimate goal of RTDOS is dynamic adaptability, the static aspects of our approach are only a step towards a completely dynamic model.

- An Application A = $\{TG_1,\ldots,TG_m\}$ is a finite set of m Task Groups utilizing a

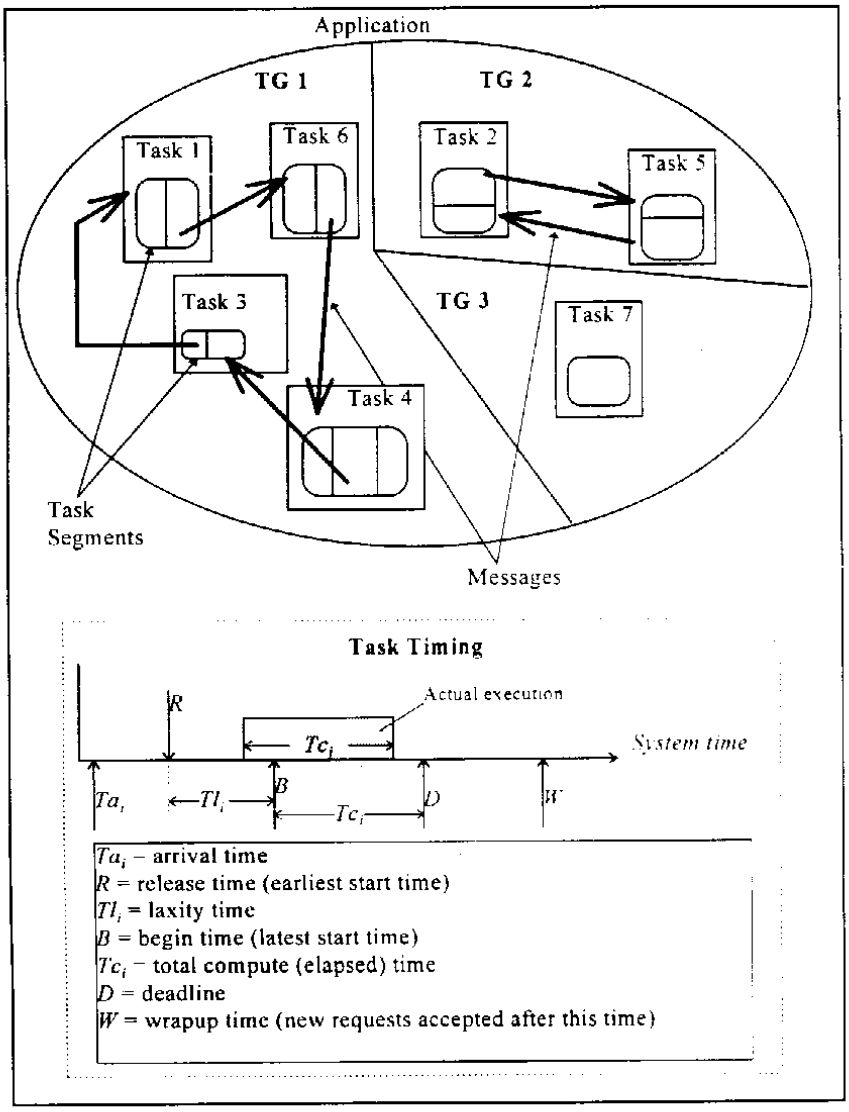

Fig. 5. An application composed of communicating task groups, and tasks with associated task timing.

distributed system consisting of N nodes. Task Groups are user defined in order to specify intertask dependencies, i.e. precedence constraints among cooperating *dynamic* tasks. Precedence constraints for *static* tasks are specified using task segments (discussed below). Task Group specifications are also utilized during dynamic scheduling in order to optimize the placement of tasks on processor nodes.

- A Task Group $TG_i = \{T_1,\ldots,T_t\}$ is a finite set of t Tasks distributed over one or more nodes.

- A Task T_i is the basic execution unit and has 2 attributes: a set of s resources needed by the task{$R^1_i,\ldots,R^s_i$ } and a finite sequence of k execution segments T_i [1],...,T_i [k]. Resources are non shared entities such as CPU, memory, I/O devices and communication facilities. Execution segments are ordered such that T_i [1] is the first segment and T_i [k] is the last segment. Fine grained precedence constraints for static tasks are defined as ordered pairs of task segments (e.g. (T_1 [3],T_4 [1]) implies that the third segment of task T_1 must complete its execution before the first segment of T_4 begins). Task segments are the smallest granule utilized when specifying precedence relations. Hence, they are visible only to the static scheduler. Such precedence constraints must be specified for the static scheduling algorithm so that proper order is maintained in the derivation of a feasible schedule. More investigation is needed to find the best method of expressing task segments at the application level and how to dispatch them at runtime. *Dynamic* tasks are defined with $k=1$ segment. The entire task is the smallest granule of distribution possible within the network.
- Each segment T_i [j] is characterized by{T_i [j]^r, T_i [j]^b, T_i [j]^c, T_i [j]^d} where T_i [j]^r = release time of segment j, T_i [j]^b = latest begin time of segment j, T_i [j]^c = elapsed compute time of segment j, and T_i [j]^d = deadline of segment j .

Given a particular task, T $\in TG_i$, the following time relationships hold for both periodic and aperiodic tasks:

- Ta is the arrival, or instantiation real time of task T.
- Tr = T[1]^r is the user defined release time, or earliest time that the task's first segment may begin execution. Release times for other segments, if not user defined, are defined as T[i +1]^r = T[i]^d+1, i.e. the release time for segment i +1 is immediately after segment i 's deadline (assuming discrect time).
- Tb = T[1]^b ≤ Td -Tc is the begin time, or the latest time that the task's first segment may begin execution. Otherwise, the task's deadline will be compromised. This is usually computed by the scheduler, but can also be user defined. Begin times for all other segments, if not user defined, are computed as T[i]^b ≤ T[i]^d-T[i]C, i.e. the begin time for segment i is no later than the elapsed time necessary to execute segment i .
- Tl = Tb -Tr is the laxity time between the task's earliest start time and latest

start time. This measure, initially defined by the scheduler, quantifies the urgency of the task with respect to its deadline. This value changes dynamically, after the task is started, to be T*d* -T*c* (the deadline minus the remaining compute time needed for this task).

- T*c* is the user defined compute time needed by this task. It is the sum of the estimated/calculated execution elapsed times for all segments; i.e. $Tc = \sum_i T[i]^c$. After the task starts execution, T*c* represents the remaining compute time required. T*c* should include all processing, I/O, IPC, and synchronization time. Note that among all timing measures, only the compute time T*c* and laxity time T*l* constraints are tallies of elapsed time. The other task timing measures can be non-negative offsets from the arrival time T*a* .
- T*d* is the user defined deadline by which all segments of task T must complete. In addition, the relations $\forall i$, $T[i]^d < T[i+1]^d$ and $Tr+Tc \le (T[k]^d) \le Tb + Tc = Td$ must hold (T[*k*] being the last segment). That is, all segment deadlines must be ordered such that a segment's deadline is strictly less than its successor segment's deadline, if it exists. Also, the deadline for the last segment must lie between the earliest time at which the task can terminate and the latest time at which the task can terminate which is the task's deadline.
- T*w* is the user defined wrap-up, or recovery time after the task's deadline before which a subsequent request for task T cannot be serviced. After T*w*, the task is free to be reinstantiated. This parameter can be used to specify recovery times for aperiodic tasks as well as to formulate the period for periodic tasks.

In real time, if $R = Ta + Tr$, $B = Ta + Tb$, $D = Ta + Td$, and $W = Ta + Tw$, then initially the relation $Ta \le R \le R + Tl = B < B + Tc = D \le W$ holds. This relationship is also shown in Figure 5.

Any task (either statically or dynamically scheduled) is capable of missing its deadline by exceeding its allotted compute time T*c* . RTDOS's SafeCPU scheduler degrades gracefully and does not allow these excessive tasks to corrupt the deadlines of other tasks. Nonetheless, when either the deadline or T*c* has been exceeded, a system-defined or user-defined exception handler will be invoked. This feedback mechanism allows an application to terminate "faulty"tasks and facilitates dynamic monitoring, debugging, system reconfiguration and scheduling.

3.4 RTDOS deadline guarantee policy

Dertouzos and Mok [16] showed that an optimal dynamic scheduler for 2 or more processors is impossible to build. However, they did prove necessary and sufficient conditions required to ensure that a task set is conflict free at any point in time (i.e. all tasks will meet their deadlines) without *a priori* knowledge of the task start times. They demonstrated that both the *Earliest Deadline* and *Least Laxity* algorithms can implement these conditions. Their analysis considered the CPU as the only resource and that tasks were preemptible.

RTDOS dynamic scheduling considers resource constraints including CPU, memory, I/O devices, and communication resources. The scheduler receives requests to start new tasks or task groups. As assumed in [6], tasks with precedence constraints form a task group and each task in the group is given the same arrival time T*a*, the same deadline T*d*, and the same wrap-up time T*w* since these tasks "are most likely to be constrained to occur in the same time period" [6, p.151]. In contrast, the group does not have a summed compute time T*c* since tasks conceptually receive processing power in parallel on disjoint hosts. In the following discussion task and task group are interchangeable.

Extending the scheduling game model given in [16], when the task t arrives, it can be characterized by the ordered pair $t = (Tl_t, Tc_t)$ (laxity and compute). If we assume that the task's arrival time $Ta_t = Tr_t$ (the task's release time) then, by definition, $Tc_t + Tl_t = Td_t$ (the task's deadline). The ordered pair $t = (Tl_t, Tc_t)$ can be plotted on a Cartesian Coordinate system (LxC) for a particular resource R_i (Fig. 6). For example, if a task with a deadline of 3 arrives, then it can be plotted anywhere on the line C = 3-L. This line and the line $L = Td = 3$ separate the plane into 3 regions Rg_1, Rg_2, Rg_3. Region Rg_1 contains tasks whose deadlines are earlier than $Td = 3$ and they must receive the resource within the next 3 time units. Region Rg_2 contains tasks whose deadlines are later than Td but they must receive the resource for a time period proportional to their distance from the line L=3. Region Rg_3 contains tasks that do not need attention within the next 3 time units. As shown in Fig. 6, task (1,1) has a deadline of 2 and needs 1 unit of computation time. Task (2,2) has a deadline of 4 and needs 2 units of computation time, however, it must receive at least 1 unit of computation time within the next 3 time units. Task (4,1) has a deadline of 5 and needs 1 unit of computation time to complete, however, it needs no attention within the next 3 time units.

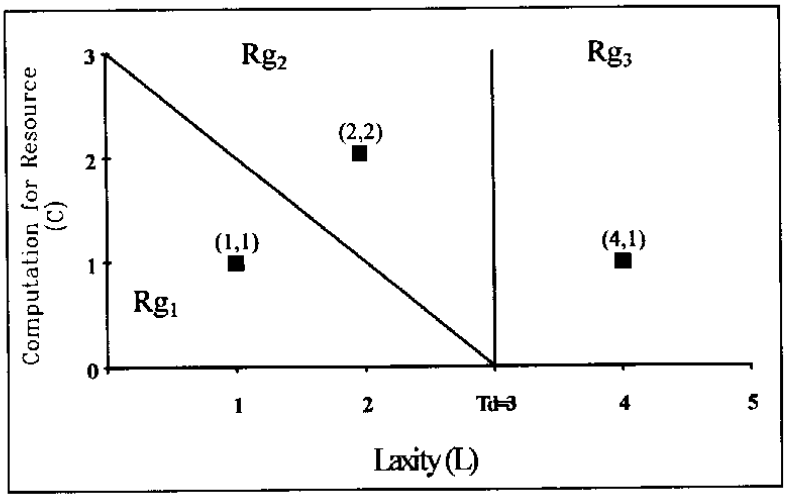

Fig. 6. Utilization forecast plane for resource R_i.

Using the above defined regions, the dynamic Local Scheduler can determine if there exists enough spare resource utilization time upon the arrival of a new task. If

$$0 \leq \sum_{Rg_1} C_i + \sum_{Rg_2} (Td_t - L_i) \leq Tl_t$$

then the new task can be scheduled. That is, if the sum of computation times for all tasks in region Rg_1 plus the sum of all required computation time needed in the next Td_t units by tasks in region Rg_2 is less than or equal to the laxity of the new task t, then the new task can be scheduled. Considering the above example scenario, region Rg_1 has task (1,1) and region Rg_2 has task (2,2). Therefore, in the next 3 time units, the resource is already committed for 2 time units. If the new task has a deadline of 3, then it can only be scheduled to utilize the resource if its computation time $Tc \leq 1 \Rightarrow Tl \geq 2$ possibly corresponding to the task t = (2,1).

When scheduling the next task to run, the scheduler chooses the task with the *least laxity and/or the earliest deadline*. In the above example, the scheduler looks ahead for the next 3 time units (Td). If the new task's computation Tc = 1, then it is schedulable and it is plotted at t = (2,1). The next task to be scheduled is (1,1) since it has least laxity. Afterwards, task (2,2) and the new task (2,1) both have least laxity. Since the new task (2,1) has a earliest deadline, it will be scheduled second. Next, task (2,2) is scheduled.

Because tasks are nonpreemptible from the Local Scheduler's perspective and all resources must be in place before the task is scheduled, the dynamic scheduling algorithm can model a (LxC) plane for every schedulable resource - CPU, memory, I/O devices, transputer links, and Connection Service capacity (the Connection Service establishes point-to-point connections for IPC and has a limited capacity of connections per second). Replicated resources (e.g. identical CPUs, memory segments, transputer links, maximum connections per second) can be represented as heaps. A resource plane will then be dynamically allocated as a subset of the heap. All resource planes in use are conceptually updated every time unit. In practice, this can occur less frequently such as at every scheduling point. The "cube" of (LxC) resource planes allows the dynamic scheduler to maintain proper control and allocation of all resources in its domain loop.

3.5 RTDOS deadline monitoring

In most cases, a task's required compute time (Tc) is derived by statistical analysis. Deadline scheduling policies based on stochastic computation times perform well in average but fail in the presence of transient overloads, causing already "guaranteed" tasks to miss their deadline. Moreover, a task missing its deadline may result in a domino effect, causing other tasks to miss their deadline. And these deadlines are not missed in an order that reflects their importance [17]. This may have adverse effects on the system's behavior as critical tasks begin to miss their deadlines. For example, the Rate Monotonic algorithm [18], with transient overloads, causes the processes with the longest periods to miss their deadlines. And according to Burns, most scheduling schemes (including Earliest Deadline) cause aperiodic deadlines to be missed not in an order that reflects their importance. Therefore, any deadline scheduling strategy *without proper CPU scheduling policies to handle transient overloads* is subject to failure even though the deadline was previously "guaranteed". This is true for both static and dynamic deadline scheduling strategies.

This section presents an approach using medium-term scheduling policies to cope with the problem of transient overload in dynamic real-time environments. As our approach also considers computation times as stochastic, we do not claim to solve the problem in all the cases, but experimental results showed that the proposed approach performs better than the existing ones, and produces higher CPU utilization ratios.

The RTDOS solution to this problem is a medium-term CPU scheduler operating on top of the transputer hardware scheduler. The medium-term scheduler, running at every node, monitors periodically the laxity and deadline of executing tasks. It reschedules the active task list according to Least Laxity/Earliest Deadline policy.

This activity results in the suspension (and therefore the CPU preemption) of tasks with large laxity to the benefit of tasks with smaller laxities. As a result the proposed schema always takes the system from a "safe state" (all the tasks are schedulable) to another safe state. Note that RTDOS approach departs from existing approaches [19, 16] that establishes fixed schedules. It periodically establishes new feasible execution schedules depending on task laxities and deadlines. This preemptive strategy deals efficiently with transient overloads, and increases CPU utilization.

3.5.1 The SafeCPU algorithm

The following is assumed by the SafeCPU algorithm:

a) Tasks are classified as critical, essential, or background (the task clout). Tasks may be periodic or aperiodic.

b) Time constraints for an RTDOS task T are expressed by several parameters including Tr = task release time (earliest start time), Tb = task latest begin time, Tl = task laxity time, Tc = task's remaining elapsed (compute) time to finish, and Td = task deadline in real-time. Task laxity (Tl) is defined as Td-Tc (spare time between the current clock and the task's deadline considering how much compute time remains).

c) Critical tasks never exhaust their specified compute requirement (the initial Tc). However, maximum Tc values of essential tasks are determined stochastically and may not represent the worst-case. It is possible for an essential task to exceed its initial Tc. This applies to both periodic and aperiodic tasks. In this case, the task can be called a *gluttonous* task. If a task does not exceed its initial Tc then it is called *temperate* (well-behaved).

d) A task is monitored by SafeCPU if and only if its deadline has been guaranteed either off-line, or on-line by the Local Scheduler. Off-line scheduling preallocates all necessary resources for known tasks (processor time, memory, communication load, I/O devices, etc.) considering their time and fine-grained precedence constraints. The Local Scheduler dynamically performs the same function on-line, but precedence constraints of dynamically scheduled tasks are handled by giving the constrained tasks a common group release time and deadline. The RTDOS kernel processes are not monitored by SafeCPU.

e) When a new task is guaranteed, it may begin execution anytime between Tr and Tb. The task can straightway be launched at Tr, or as soon as all required resources are available. This implies that as new tasks are guaranteed, they are dispatched as soon as possible.

The SafeCPU algorithm (Fig.7) periodically wakes up to ensure that the CPU is not overburdened with active tasks causing a deadline to be compromised. If a deadline is in jeopardy, then SafeCPU will "freeze" (suspend) sufficient tasks in order to meet the most urgent deadline(s). Conversely, it will also "thaw" (resume) frozen tasks when the system is again in a safe state. It thus monitors the sets of active and frozen tasks maximizing CPU utilization and ensuring that no deadline is compromised. SafeCPU's period is a crucial tuning parameter for the application system. We plan to build tools that will aid the application developer in determining a proper period given various task population characteristics such as CPU vs. non CPU behavior, shortest and longest application task times, deadline laxity profiles, etc.

```
SafeCPU algorithm
Every τ milliseconds do
    Recalculate the Laxity for all tasks in Q(A) and Q(F)
    for all tasks in Q(A) do  -- merge Q(A) into Q(F)
        EnQ( DeQ(Q(A)), Q(F) )
    end for
    if Q(F) ≠∅ then -- some tasks asking for CPU "loans"
        SlackCPUTime = Tl (laxity) of first task in the Q(F)
        Approve (thaw if needed) the first task in the Q(F)
        EnQ( DeQ(Q(F)), Q(A) )
        for the remaining tasks in the Q(F) do
            TimeRemain = remaining compute time Tc of first task in Q(F)
            if TimeRemain ≠ 0 and TimeRemain ≤ SlackCPUTime
                                and SlackCPUTime ≠ 0 then
                SlackCPUTime = SlackCPUTime – TimeRemain;
                Approve (thaw if needed) this task
                EnQ( DeQ(Q(F)), Q(A) )
            else
                SlackCPUTime = 0;
                exit for loop
            endif
        endfor
        Freeze all remaining tasks in Q(F)
    endif
end do
```

Fig. 7. SafeCPU algorithm.

SafeCPU utilizes two Queues, ReadyQ and FrozenQ and each queue is ordered by task clout so that critical tasks are serviced before essential tasks, which in turn receive attention before background tasks. Furthermore, each task clout is ordered according to the scheduling policy: Least Laxity and/or Earliest Deadline (LL/ED) [16]. In order to describe SafeCPU's queue structure more formally,

1. Let $C = \{T_i : T_i$ is a critical task$\}$, $E = \{T_i : T_i$ is an essential task$\}$, $B = \{T_i : T_i$ is a background task$\}$.
2. Define the queue of critical tasks ordered by LL/ED as $QC = T_1, T_2, \ldots, T_n$ where $T_i \in C$ and T_i has least laxity and/or earlier deadline than T_{i+1} (i.e., $Tl_i < Tl_i + 1$ or $(Tl_i = Tl_i + 1$ and $Td_i \leq Td_i + 1)$). QC can be $\varnothing$. Form similar queues QE for E and QB for B. Also, QC, QE, QB contains all released unterminated tasks presently in the system.
3. Let Q (A) be an ordered queue of active tasks defined initially as QC, QE, QB (active tasks ordered by clout). Let Q (F) be an ordered queue of frozen tasks defined similarly to Q(A) but initially $\varnothing$. It is noteworthy that task clout overrules scheduling policy LL/ED. That is, it is possible for a critical task to have a greater laxity or later deadline than an essential or background task. Nonetheless, the critical task will receive allocation of CPU resources before the essential or background task. In other words, essential or background tasks will never be able to cause a critical task to miss its deadline.
4. Define EnQ (T,Q) as inserting the given task T into the given queue Q preserving the queue's properties based on the task's clout, laxity and deadline. Similarly, define DeQ (Q) as removing and returning the task at the head of the given queue Q. Q may either be Q (A) or Q (F).

Active tasks are those tasks that can use processing resources. Frozen tasks are denied the CPU resource in the interest of preserving a safe state for the most important tasks: those with higher clout, or least laxity and earliest deadline within the same clout.

SafeCPU is a periodic system task whose period may be application system dependent and can change dynamically, if necessary. Because real-time deadlines are "moving targets", laxities are always changing based on how much CPU time each task has received and the distance between the current clock and the deadline. Therefore, laxities for all tasks in both queues must be recalculated for each iteration of SafeCPU (see the following example). After laxities are adjusted, Q (A) can be merged into Q (F). Conceptually, tasks in Q (F) receive no CPU time and are thus in the correct order with

respect to themselves. However, tasks in Q (A) receive CPU time and thus have new laxities. Merging Q (A) into Q (F) places all tasks into proper order based on the scheduling policy. SafeCPU can then begin to "loan out" the CPU to the tasks at the head of Q(F).The first task is always approved, and subsequent tasks are approved *as long as sufficient CPU time remains to allocate the entire task to its completion* . Critical tasks are approved before essential tasks, which are approved before background tasks. Approving a task means changing its state back to Active and, if it is suspended, the process is placed back onto the transputer's hardware dispatch chain. Freezing a task means changing its state to "frozen". The task will suspend itself by executing an instruction that removes the process from the hardware dispatch chain. It can only be "thawed" by another iteration of SafeCPU.

For example, assume that essential tasks 1, 2, 3, 4 are created and their deadlines are guaranteed on a particular CPU by the Local Scheduler. Task 1 has a deadline = 20 and requires 5 compute units. Recall that compute time Tc = CPU + I/O (elapsed time). Task 2 has a deadline =17 and requires 9 compute units. Task 3 has a deadline =10 and requires 5 compute units. Task 4 has a deadline = 15 and requires 1 compute unit. All 4 tasks are started and become active at the same time. Figure 8 shows the initial queue structure at time $t=0$ and latest begin times for each task. The timing diagram is not a feasible schedule since there is only one CPU per node. Task 3 has least laxity out of the four tasks. SafeCPU has not yet been invoked.

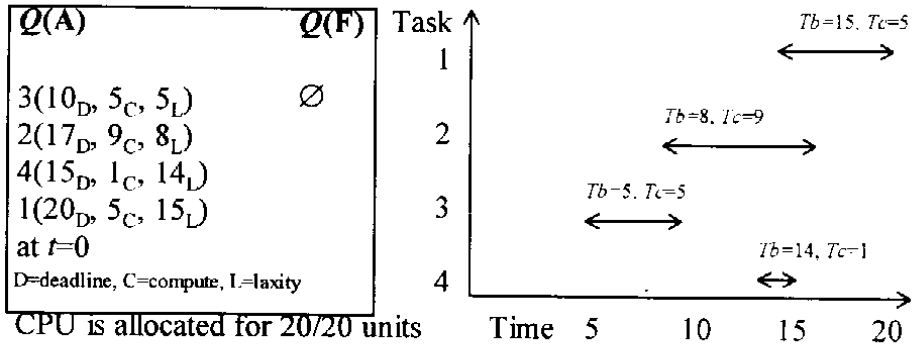

Fig. 8. Example of initial queue state and task compute times.

Assume that SafeCPU's period is 3 time units. Figure 9 shows the sequence of Queue changes until all tasks finish. For time 1..3, assume task 3 receives 2 units, task 2 receives 1 unit. On the first iteration of SafeCPU, the laxities of all tasks are recomputed based on the current clock and remaining compute time. The tasks are then

merged into Q (F) based on LL/ED. Task 4's deadline is 3 units closer, but it did not receive any compute time. Thus, it's laxity has dropped from 14 to 11. SafeCPU begins to loan the CPU to the tasks. Task 3 is approved and SlackCPUTime (in the algorithm, Fig. 7) is set to 4 which is the laxity of task 3. SafeCPU tries to give the SlackCPUTime to task 2, but task 2 requires 8 time units. Therefore, it cannot be approved. Thus only task 3 is active in Q (A) while tasks 2, 4, 1 are frozen in Q (F). Note that even though Task 4 could be approved (since it only needs 1 time unit), SafeCPU does not violate the scheduling policy: LL/ED because deadlines are top priority. For time 4..6, task 3 receives exclusive compute time and terminates at t=6 (its deadline was 10). On the second iteration of SafeCPU at t=6, the laxities are recomputed although tasks 2, 4, 1 are properly ordered in Q (F). Task 2 is approved. SlackCPUTime = 3; the laxity of task 2. Task 4 needs 1 compute time unit and so SlackCPUTime is decremented to 2. Task 1 cannot be approved since SlackCPUTime<5. Thus tasks 2 and 4 are approved and active, while task 1 remains frozen. At t=9, task 4 completes (its deadline was 15) and tasks 2 and 1 compete for CPU loans. Task 2 is approved and task 1 remains frozen. Task 2 still requires exclusive use of the CPU at t=12, until it completes at t=15 (its deadline was 17). At this point SafeCPU can approve task 1 and it finally terminates at t=20, just in time.

	Merged Q(F)	**SlackCPUtime**	**New Q(A) Active**	**New Q(F) Frozen**
t=3	$3(7_D, 3_C, 4_L)$ $2(14_D, 8_C, 6_L)$ $4(12_D, 1_C, 11_L)$ $1(17_D, 5_C, 12_L)$ task 3 gets 2, task 2 gets 1	4 =4−8=0 0 0	$3(7_D, 3_C, 4_L)$	$2(14_D, 8_C, 6_L)$ $4(12_D, 1_C, 11_L)$ $1(17_D, 5_C, 12_L)$
t=6	>>$3(4_D, 0_C, 4_L)$<< $2(11_D, 8_C, 3_L)$ $4(9_D, 1_C, 8_L)$ $1(14_D, 5_C, 9_L)$ task 3 gets 3 and exits	3 =3−1=2 =2−5=0	$2(11_D, 8_C, 3_L)$ $4(9_D, 1_C, 8_L)$	$1(14_D, 5_C, 9_L)$
t=9	$2(8_D, 6_C, 2_L)$ >>$4(6_D, 0_C, 6_L)$<< $1(11_D, 5_C, 6_L)$ task 4 gets 1 and exits, task 2 gets 2	2 =2−5=0	$2(8_D, 6_C, 2_L)$	$1(11_D, 5_C, 6_L)$
t=15	>>$2(2_D, 0_C, 2_L)$<< $1(5_D, 5_C, 0_L)$ task 2 gets 3 at t=12 and 3 at t=15 and finally exits task 1 finally exits at t=20	0	$1(5_D, 5_C, 0_L)$	∅

Fig. 9. Queue changes by SafeCPU.

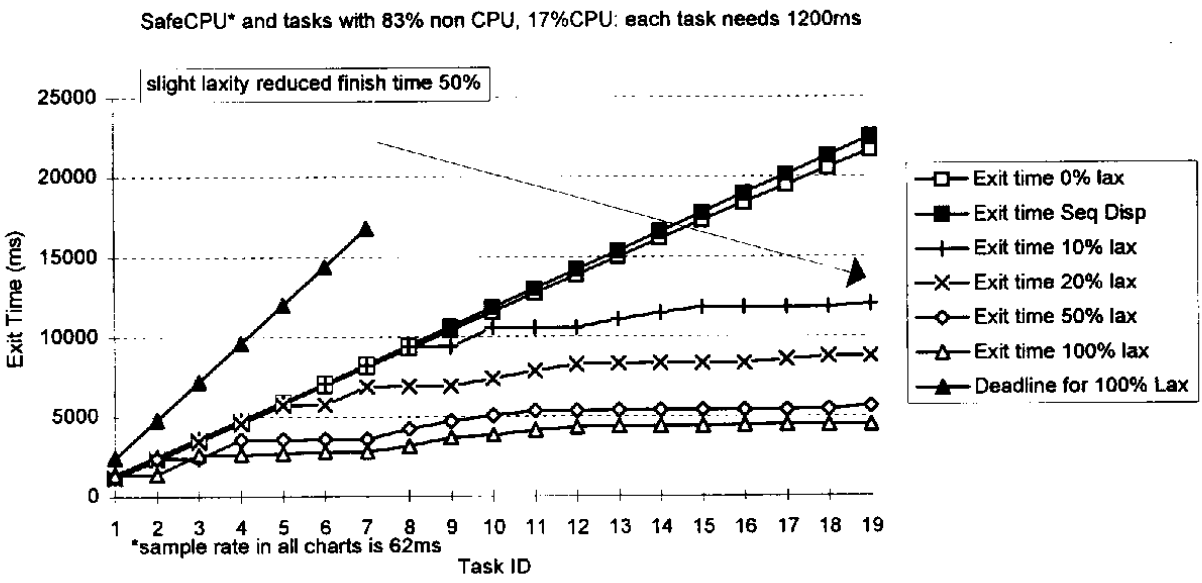

Fig. 10. Non CPU tasks with various laxities.

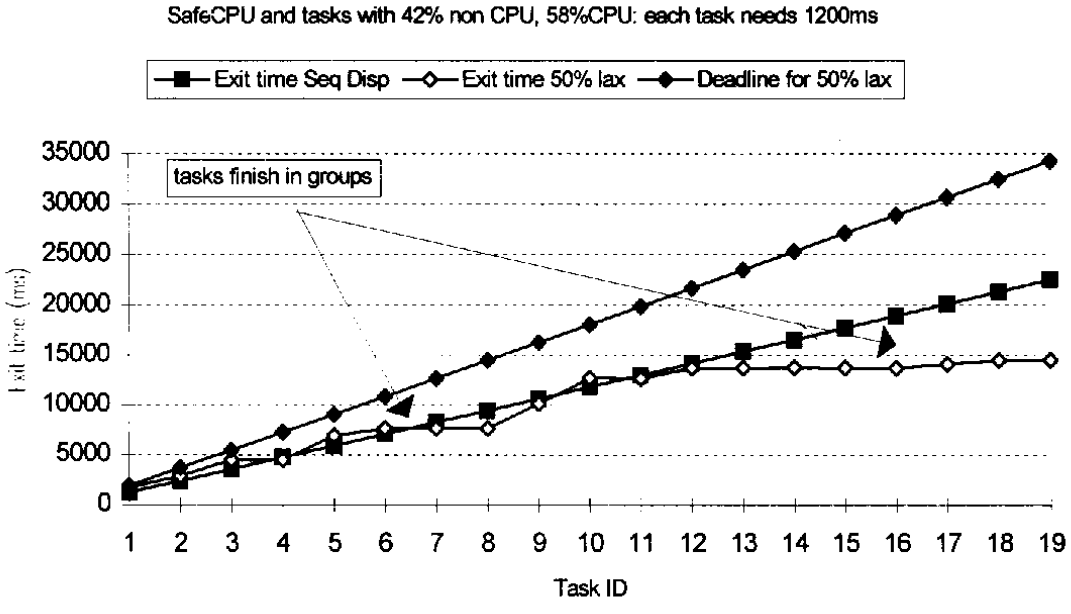

Fig. 11. CPU-type tasks with 50% laxity.

task 1 begins at time t=0, then the tightest deadlines under 0% laxity is $Td_1 = 100, Td_2 = 300, Td_3 = 350$.

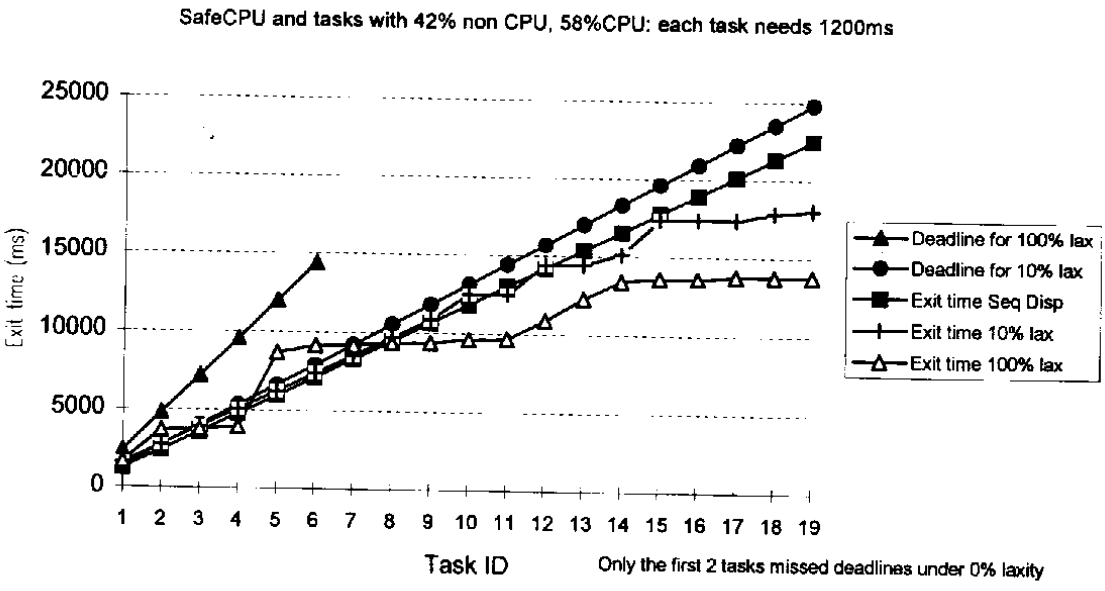

Fig. 12. CPU-type tasks with 10% and 100% laxity.

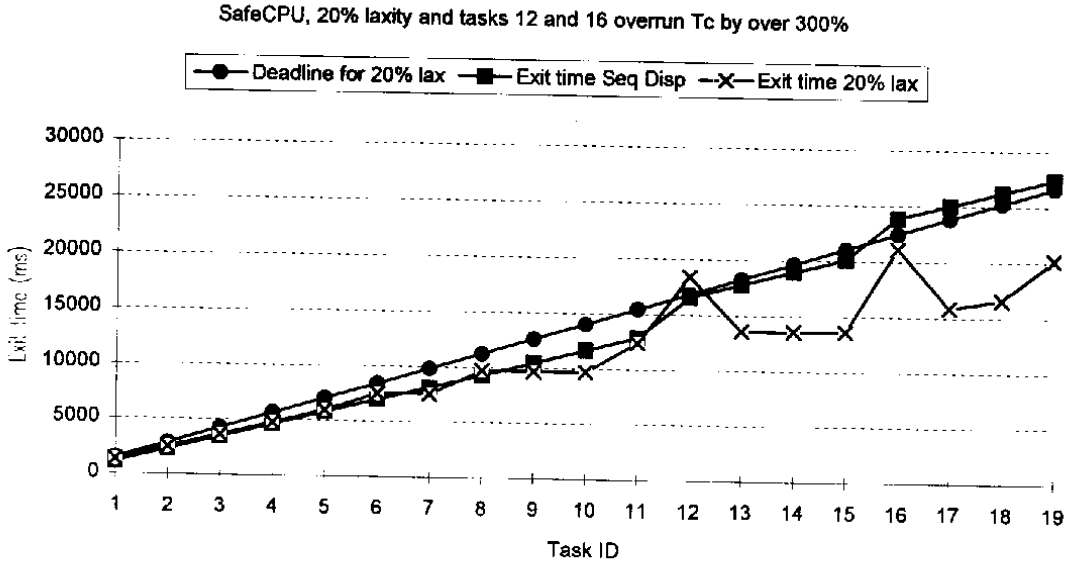

Fig. 13. Two tasks overrun Tc by 300%, uniform laxity of 20%.

Sequential dispatching is the simple comparison to SafeCPU. Sequential dispatching starts each task (after its release time) when all required resources are ready. No other task is started until the current task completes. The next task is dispatched in like fashion.

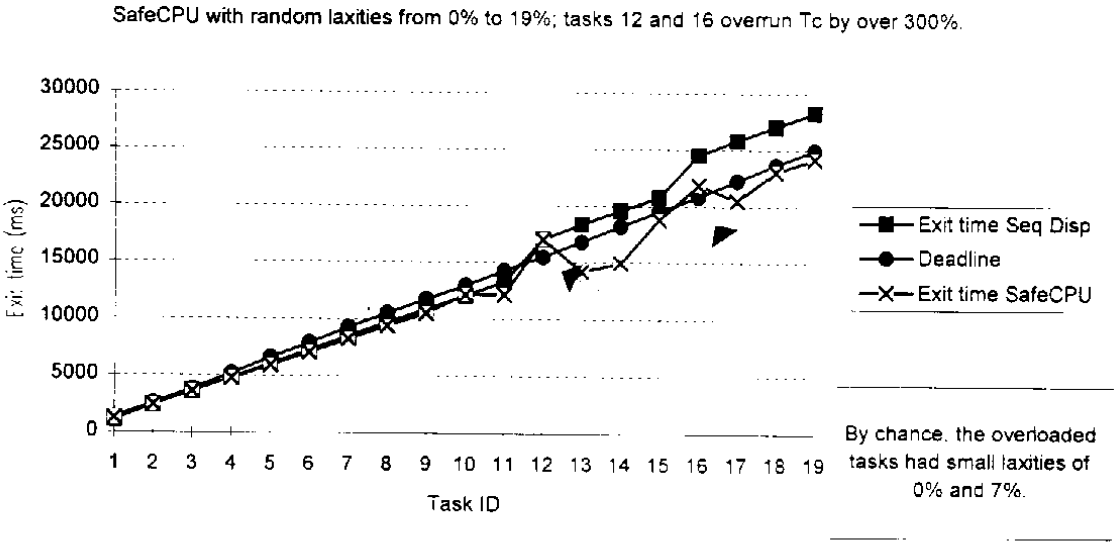

Fig. 14. Transient overload with random laxities.

3.5.2.1 Normal load

In order to test SafeCPU under normal conditions, each task consumed as much compute time as specified by its Tc. Figure 10 illustrates that given a task population which is non CPU intensive (83% non CPU, 17% CPU), SafeCPU performs quite well. Even with 0% laxity, SafeCPU performs slightly better than sequential dispatching because very small levels of concurrency occur (due to the transputer's efficient process dispatcher). With the slightest bit of laxity (10%), SafeCPU allowed several tasks to switch between the CPU and still ensured that tasks 1...8 met their deadlines by close margins. By this time, however, the other tasks had received enough CPU time (tasks 9...19) so that they all soon finished well before their deadlines (in about half the time as sequential dispatch). Higher laxities produced better results. Please note that not all deadlines are shown because they would skew the y-axis scaling. For reference, only part of the "Deadline for 100% laxity" line is shown. The line for 0% laxity deadlines is not shown, but it follows the "Exit time Seq Disp" line; other laxity deadlines are between these two extremes.

Tasks can be more CPU intensive. Fig. 11 and Fig. 12 illustrate SafeCPU's behavior with tasks that are 58% CPU and 42% non CPU. 50% laxity is shown in Fig. 11 and the extreme laxities 10% and 100% are shown in Fig. 12. Only part of the 100% laxity deadline is shown so as to not skew the y-axis scaling thus rendering the other plots unreadable. Again, all tasks were injected into the system at the same time and all tasks were started immediately after their creation. As expected, both charts show

that with greater than 0% laxity, SafeCPU loans available CPU time out to collections of tasks according to the scheduling policy. This explains why the sample tasks (with equal compute times) finish in stair-step fashion. 100% laxity clearly shows this behaviour where task 1 finishes by the deadline, yet sufficient CPU time was given to tasks 2,3,4 so that they soon finished next. Similarly, tasks 5 through 11 were approved collectively so that they all finished about the same time. And the same holds true for the remaining tasks.

The Exit time for 0% laxity is not shown for clarity since it is very close to the "Exit time Seq Disp" line. All tasks under 0% laxity finished their deadlines except the first two tasks. This can be explained that as all 19 tasks were thrust into the system at the beginning, it took at least one iteration of SafeCPU to freeze all tasks but the first one. And perhaps SafeCPU's period was not sufficiently frequent to thaw task 2 in time for it to meet its deadline. But more likely, crucial CPU time was consumed by all tasks at their initial burst into the system before the latter tasks (3..19) could be frozen. This caused task 1 and 2 to miss their deadlines. This is the extreme initial case and demonstrates that SafeCPU behaves poorly if a too many tasks are injected into the system and the initial tasks have 0% laxity. Further experiments are needed to determine the relationship between SafeCPU's period and the maximum rate at which new tasks all with 0% laxity can be started at the same time.

3.5.2.2 Transient overload

To experience transient overload, we caused 2 of the tasks to exceed their Tc considerably. That is, the normal 17 tasks still had profiles of 42% non CPU and 58% CPU, whereas 2 tasks (task Ids 12 and 16) became excessively CPU intensive and overran their *Tc* by over 300% (from 1200 ms to 3700 ms) making their profiles 14% non CPU, and 86% CPU. All tasks had deadlines with 20% laxity and all tasks specified $Tc = 1200$. As Fig. 13 illustrates, sequential dispatching had enough cumulative laxity built up in reserve to survive the first overrun (task 12), but the second overrun (task 16) missed its deadline and caused all remaining tasks (17,18,19) to also miss their deadlines. SafeCPU, however, only missed the first deadline. All other deadlines were met.

SafeCPU's behaviour is deterministic during transient overloads, as described below. When a task overruns, its remaining compute time essentially becomes negative. SafeCPU treats all negative values as zero and thus the task's laxity (deadline-remaining compute time) becomes greater than any other active task with the same deadline (since these non overrun tasks have a non zero remaining compute time). This has the effect of

demoting the overrun task to receive less CPU than other normal tasks *with the same deadline*. As time elapses, the deadline gets closer and closer causing the laxity to decrease enabling the overrun task to compete with other tasks who have later deadlines. However, immediately after the overrun task's deadline has past, SafeCPU computes the task's laxity as zero and thus the SlackCPUTime in Fig. 7 is zero causing all tasks to freeze except the overrun task (unless a task of higher clout exists). If all active tasks are of the same clout, this has the effect of promoting this *overrun and overdue* task to receive exclusive use of the CPU time until it completes. This is demonstrated in Fig. 13 since task 12 soon finishes after its deadline (exit time 17000ms). Tasks 16 and 19 are the only remaining tasks in the system after exit time 17000ms and they finish after task 12 (task 13, 14, 15, 17, 18 all finish before exit time 17000ms). In most cases, such behavior is unacceptable and the application should take necessary steps to kill the overdue and overrun task.

It is important to note that SafeCPU's queue structure keeps critical tasks before essential tasks, which are before background tasks. Because SafeCPU's overhead is low, it can be used to manage several critical tasks in the CPU at the same time, instead of the classical approach of exclusive CPU use for each critical task. We also assume that critical tasks will never overrun. Even if a task's laxity is zero (the smallest possible value), it will only be serviced if there are no other tasks present with higher clout. This means that even if during a transient overload, an essential task overruns and misses its deadline, and if the laxities are very tight (e.g. 0%), the overrun and overdue task will *not* cause critical tasks to miss their deadlines.

In order to simulate non deterministic task deadline behaviour, task laxities (and hence their deadlines) were drawn randomly from a uniform distribution ranging from 0% to 19%. The tasks were still composed of 42% non CPU, 58% CPU and all *Tc*'s remained defined as 1200 ms. The same two tasks (IDs 12 and 16) experienced over 300% *Tc* overrun. Sequential dispatching had worse (unpredictable) results, as shown in Figure 14. The first overrun caused itself and (this time) all 7 remaining tasks to miss their deadlines. It is important to remember that these deadlines were previously "guaranteed". But, SafeCPU had predictable results. Only the overran tasks missed their deadlines. All other tasks met their deadlines. The overran tasks missed their deadlines because, by chance, they received extremely low laxities: 0% and 7%. Thus, in a dynamic deadline environment, SafeCPU successfully monitors task progress predictably ensuring that tasks meet their deadlines. It also degrades gracefully during transient overloads.

3.5.3 Real time boiler control

In order to fully test SafeCPU, a real time steam boiler control application was designed and developed. A boiler simulator was constructed that modeled a certain volume (V) of water, of a certain temperature (T), and under a certain pressure (P). Inputs into the boiler were water flow and heat. Boiler output was steam. Inputs and output were controlled by valves that increased or decreased the flow (water, heat, steam). The purpose of the control application was to automatically maintain given pressure and water level parameters so that the boiler did not explode. Steam demand (output) could fluctuate and thus created unpredictable events at which the application corrected the inputs (more or less heat and water) in order to maintain steady pressure and water level. Process control applications such as this commonly utilize a Proportional Integral Derivative (PID) algorithm which compares past and present readings against elapsed time. The simple PID algorithm $k\ (V_{n-1} - V_n) + k(V_{goal} - V_n)$ [20] was used in this experiment to control the water level. Please note that V_{n-1} is the previous Volume reading, V_n is the present Volume reading, V_{goal} is the setpoint Volume parameter that must be maintained, and k is an experimental constant related to the control substance and the time between measurements. A similar algorithm was used to stabilize the Pressure (P).

The application's design (Fig. 15) consisted of 4 major tasks: Controller, Monitor, Correction, and Display along with the boiler simulator. The Controller was a periodic critical task that received commands from an operator (to change steam output

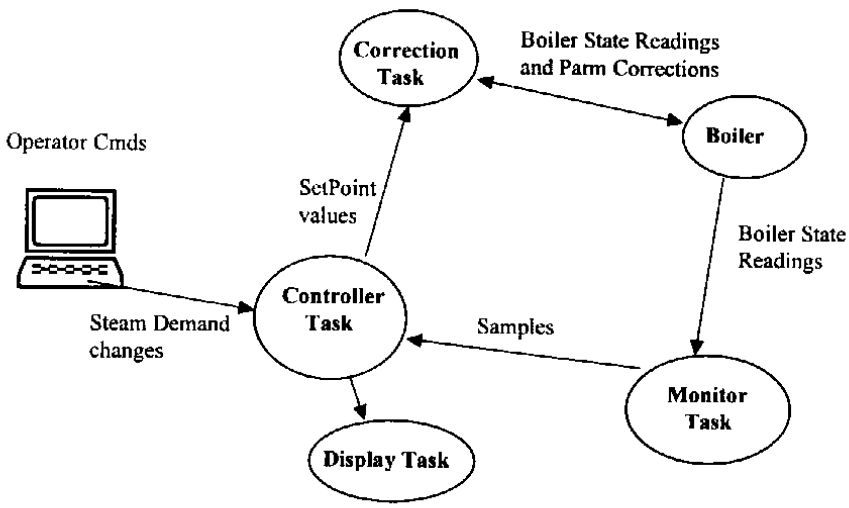

Fig. 15. Boiler control communication graph.

settings along with the required water level (V) and internal boiler pressure (P)). It also received Monitor samples and compared them with the required parameter settings. If V or P changed, the Controller launched the Correction Task to bring the boiler back to equilibrium. It also made sample data available to the display task. The Monitor task was a critical periodic task which never terminated. Its period was 800ms and it had a compute time Tc =10ms and a deadline Td =50ms after the start of each period. The Corrective task was an essential periodic task that was scheduled and launched at unpredictable times. It had a lifetime of 110000ms, its period was 200ms, its compute time Tc =60ms, and its deadline Td =100ms after the start of each period. It terminated when V and P stabilized around their setpoints. The display task was a background periodic task that displayed all parameters on the console.

The above application executed comfortably on 1 host (Fig. 16) where P = 10

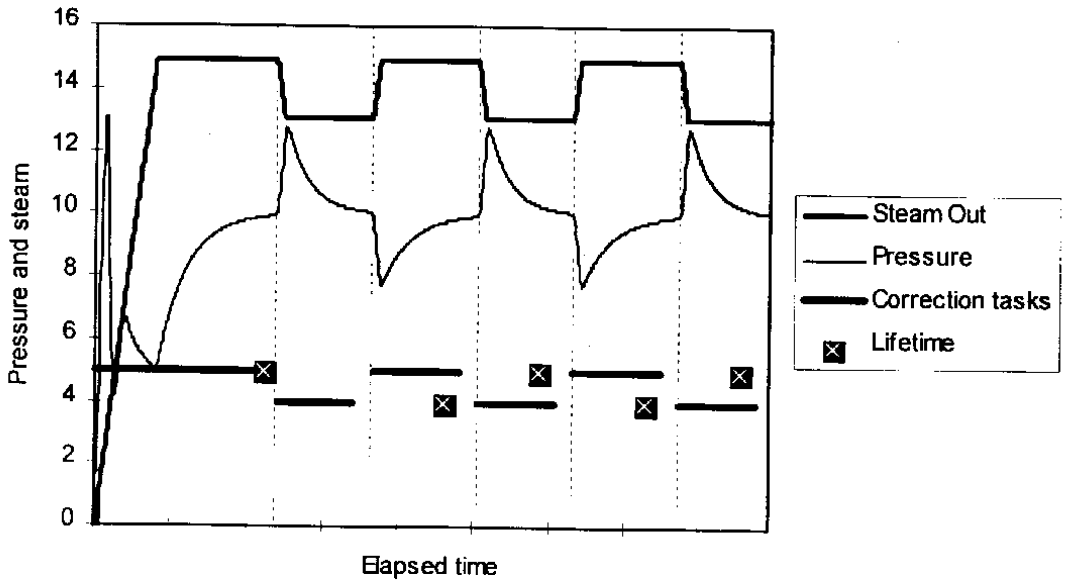

Fig. 16. Boiler control during output fluctuations.

kg/cm^2, V=150 liters, and steam output ranged between 15 and 13 units. Only the Corrective task is shown. A second independent boiler application was placed on the same host so that the host was busy controlling two boilers (Fig. 17). The second boiler maintained P = 9 kg/cm^2 and generated between 11 and 9 units of steam. In order to schedule the second boiler application, the Corrective task timing parameters had to be changed (Td=200) so that both Corrective tasks could execute comfortably during the

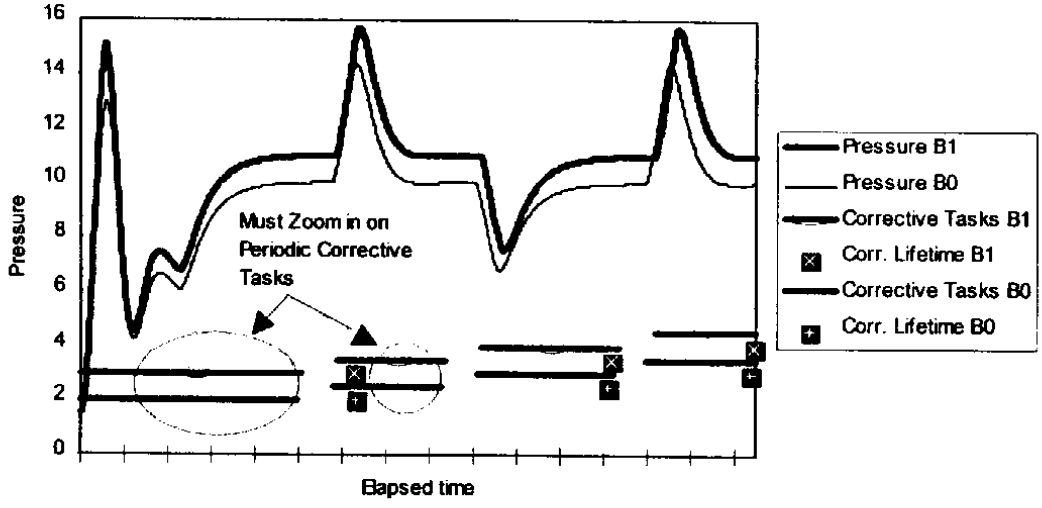

Fig. 17. Two boiler control process.

same 200ms time interval. In order to get a better understanding of the control process, Figure 18 shows execution traces at 30ms granularity. It shows Controller tasks for both boiler 0 (B0) and boiler 1 (B1) along with each boiler's Monitor task.

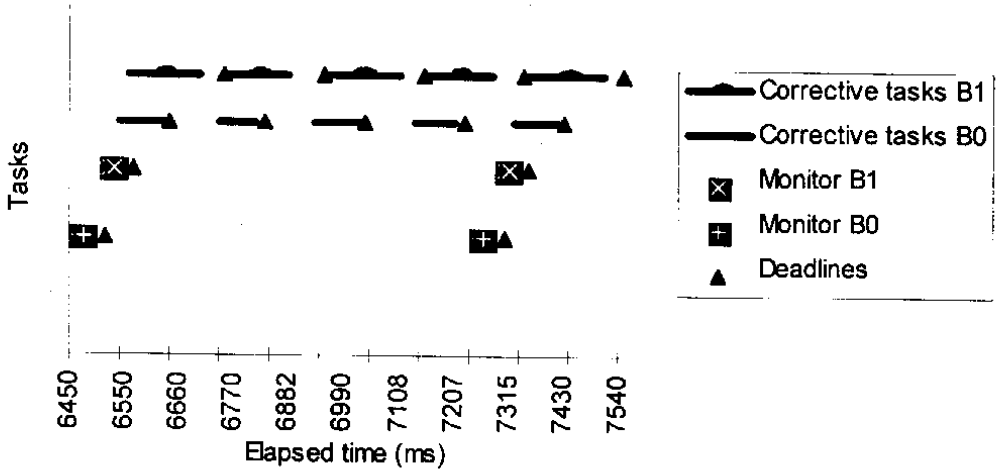

Fig. 18. Corrective task executions.

An unexpected implementation phenomenon arose during one of the tests (Fig.19). Monitors (critical tasks) for both boilers executed during a Corrective task's period. SafeCPU thus froze the Corrective task while the critical task executed. But

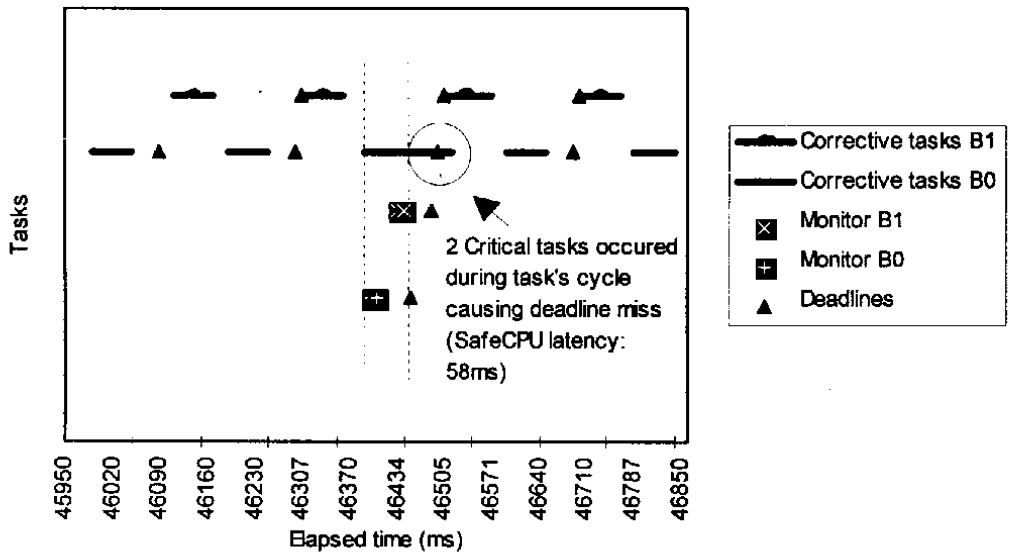

Fig. 19. Critical tasks preempt.

because SafeCPU's period was 58ms, it did not wakeup in time to thaw the frozen Corrective task. Thus it missed its deadline. To solve this problem, task period termination was changed to thaw any frozen tasks, if the terminating task was the only running task. Figure 20 shows the result where the latency was reduced and the Corrective task was able to meet its deadline.

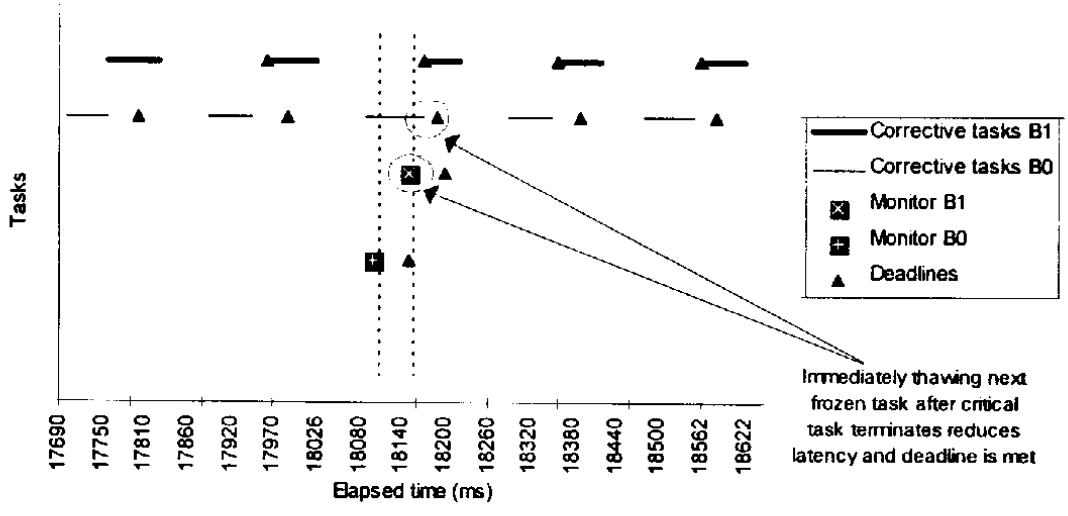

Fig. 20. Thaw after task exits.

Without SafeCPU, task deadlines were vulnerable because task concurrency was unrestrained. Figure 21 shows that under normal conditions, Corrective tasks for both boilers slowed each other down so that a deadline was missed. In the worst case of transient overload, the Corrective task for B1 was changed to execute continuously for its entire period (Fig. 22) and obviously, most deadlines were compromised.

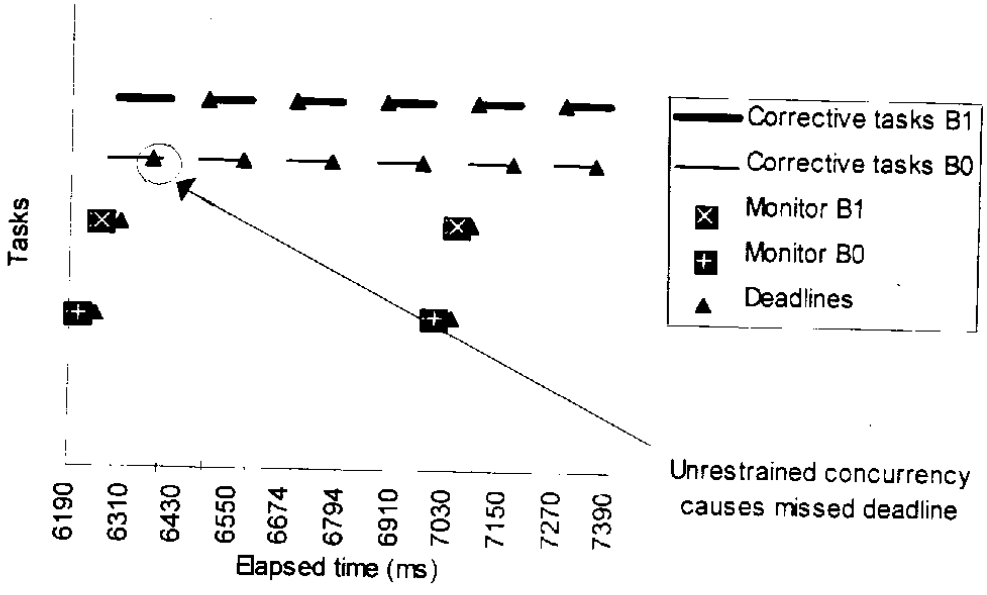

Fig. 21. No SafeCPU - normal operation.

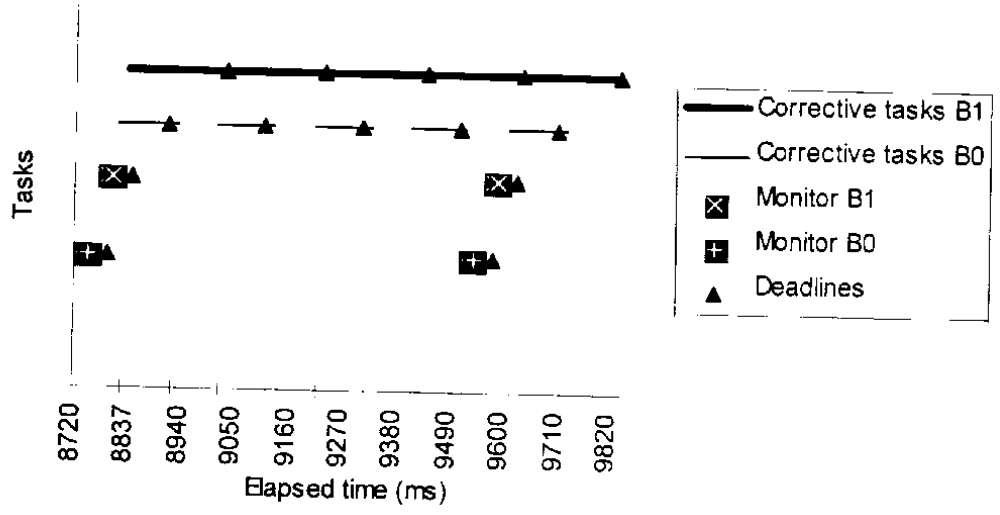

Fig. 22. No SafeCPU - overload condition.

Once SafeCPU was introduced, deadlines were enforced (Fig. 23) favoring temperate tasks over gluttonous tasks and in no case was a critical task deadline compromised. It should be noted in Fig. 23 that sometimes SafeCPU awoke and favored Corrective B1 task for a short time (Corrective B0 was frozen). This is because B1 had a smaller laxity than B0 even though B1 was gluttonous. Nevertheless,B1 was soon terminated at the end of its period which thawed B0. When SafeCPU awoke again, it favored B0 so that its deadline was met.

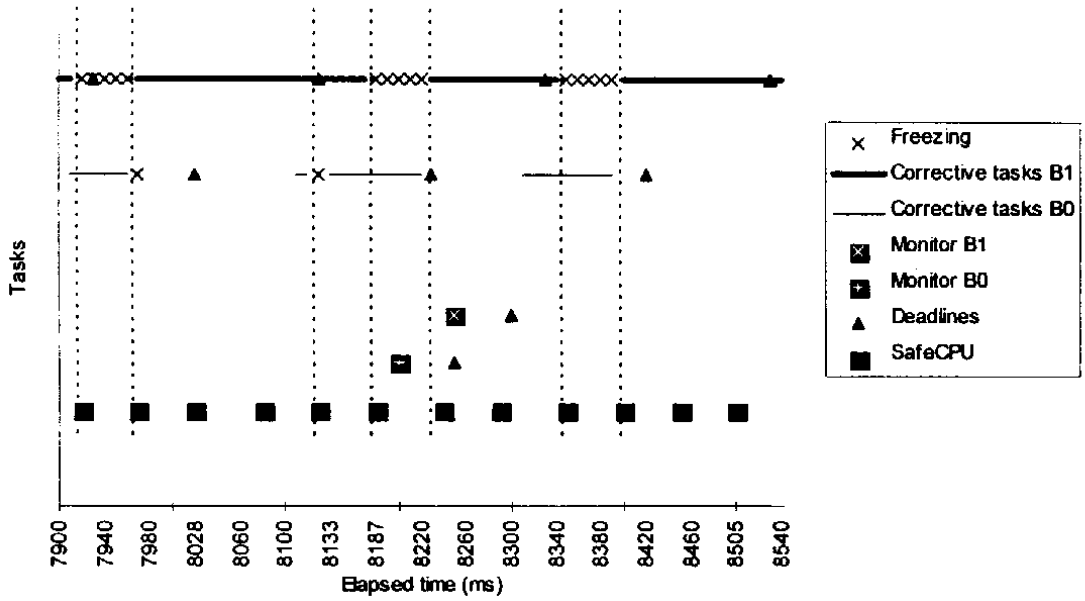

Fig. 23. SafeCPU - deadlines guaranteed.

3.5.4 Cost of SafeCPU

SafeCPU is computationally efficient except for merging Q (R) into Q (F). The complexity of an ordered queue merge is O (n^2) because the entire queue might be searched in order to insert the new element. An implementation of SafeCPU has been incorporated into the RTDOS kernel and the cost of SafeCPU was measured. A task population from 10 to 80 tasks was created and executed on a T425 transputer running at 20 MHz. All tasks were created and started immediately after task creation. All tasks had the same clout (essential) and all tasks were specified with large deadlines so that all tasks were reevaluated and approved during each SafeCPU period. The experiment intended to measure the worst-case cost of one execution of the SafeCPU algorithm, not

including costs associated with starting and stopping each SafeCPU period and costs of user tasks. Table 1 tabularizes the measured results on the T425. The trial test case was faster than $O(n^2)$ and indeed approaches $(n/10)^{1.5}+1$.

Table 1. Actual SafeCPU cost.

number of tasks	actual cost (ms)	$(n/10)^2+1$ approximation	$(n/10)^{1.5}+1$ approximation
10	2	2	2
20	4	5	4
30	7	10	7
40	9	17	9
50	12	26	13
60	15	37	16
70	19	50	20
80	23	65	24

3.5.5 SafeCPU's period

SafeCPU's period is a crucial factor in the responsiveness of the system. If the period is too large, *and if processor utilization is high* (meaning laxity is tight) then potentially only 1 task is running every period. This latency could be unacceptable for certain applications. Nevertheless, if SafeCPU has a moderate period and it is monitoring a reasonably-sized task population, then it incurs acceptable overheads. For example (on the current platform), if the processor's task population is limited to 30 tasks (which is quite large for a transputer), then SafeCPU consumes at the very worst 7ms each period. This allows a SafeCPU period of 50ms to consume 14% overhead in the absolute worst case.

3.5.6 Critical factors for SafeCPU

In summary, SafeCPU requires several factors to be taken into consideration. First, the algorithm assumes that the host processor is not overburdened, i.e. all task deadlines have been guaranteed online or offline by the Local Scheduler. Second, it assumes that critical tasks will never exceed their specified compute time. Third, SafeCPU's period must be balanced with the task set population so that overhead is kept low. The above costs are worst-case. In the experiments of section 3.5.2 using a period of 62.5ms and a task population of 19 tasks, SafeCPU consumed an average of about 1 ms of CPU time each period because many of the tasks were blocked on I/O and thus were not merged into Q (F). Last, any tasks executing after their deadlines (including overrun tasks) receive top priority and are thus terminated.

4. Conclusion

The main thrust of RTDOS is predictable services for dynamic hard real-time distributed systems. The proposed scheduler is unique among current trends in that it distributes the scheduling burden between 3 components:

- a pre-run-time static scheduler for known tasks (periodic tasks),
- a 5-level dynamic scheduler model for the remaining types of known and unknown periodic, aperiodic and background tasks. The dynamic schedulability analysis and topological reconfiguration tools can adapt the network topology to meet unforeseen deadline, resource, and precedence constraints. Also,
- a deadline priority and guarantee scheme that when coupled with the dynamic scheduler and the deadline monitor (SafeCPU) can truly provide a dynamic system adapting to unforeseen configurations.

Pending research is identifying and quantifying the predictability of the proposed scheduling model. Local and global dynamic schedulers are currently under development and measurement.

References

[1] Xu, J. and Parnas, D.L. "On Satisfying Timing Constraints in Hard-real-time Systems." *IEEE Transactions on Software Engineering*, 19, No.1 (Jan. 1993), 70-84.

[2] Stankovic, J.A."Real-Time Computing Systems: The Next Generation." In: *Tutorial Hard Real-Time Systems.*, Stankovic, J.; Ramamritham, K. (eds). IEEE Computer Society Press, (1988), 14-37.

[3] Tayli, M.; Bor, M.; Benmaiza, M., and Eskicioglu, M.R., "RT-DOS A Real-Time Distributed Operating System for Transputers." *Proceedings of the 13th Occam User Group Technical Conference*, Real-Time Systems with Transputers, York, UK, H.Zedan, (ed), IOS Press, (Sept 1990), 1-11.

[4] Benmaiza, M. and Tayli, M. "Circuit-switched IPC for Predictable Message Passing in a Multiloop Transputer Network." *Proceedings of the World Transputer Congress*, Transputer Applications and Systems '93, Aachen, Germany: R. Grebe et al. (eds.), IOS Press, (Sept. 20-22, 1993),890-898.

[5] Xu, J. "Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations." *IEEE Transactions on Software Engineering* 19, No. 2 (Feb 1993), 139-154.

[6] Stankovic, J.A. and Ramamritham, K. "The Design of the Spring Kernel." Tutorial Hard Real-Time Systems. *IEEE Computer Society Press*, Stankovic, J.; Ramamritham, K. (eds.), (1988), 371-382.

[7] Zhao, W.; Ramamritham, K., and Stankovic, J. "Preemptive Scheduling under Time and Resource Constraints."*IEEE Transactions on Computers*, C-36, No.8 (Aug 1987), 949-960.

[8] Ramamritham, K.; Stankovic, J.A., and Zhao, W. "Distributed Scheduling of Tasks with Deadlines and Resource Requirements." *IEEE Transactions on Computers*, 38, No. 8 (Aug. 1989), 1110-23.

[9] Stankovic, J.A. "Decentralized Decision-making for Task Reallocation in a Hard Real-time System." *IEEE Transactions on Computers*, 38, No. 3 (March 1989), 341-55.

[10] Stankovic, J. A."The Spring Architecture." *Proceedings. EUROMICRO, Workshop on Real Time*, Horsholm, Denmark, IEEE Computer Science Press, (June, 6-8, 1990), 104-113.

[11] SGS-Thomson Microelectronics. *The Transputer Databook*, 3rd ed., INMOS, Ltd., #72 TRN 203 02, order code DBTRANST/3, 1992.

[12] Hoare, C.A.R. "Communicating Sequential Processes." *Communications of the ACM*, 21, No.8 (Aug 1978), 666-677.

[13] Tayli, M. and Benmaiza, M. "An Efficient Circuit-switching Mechanism for Inter Process Communication in a Transputer Network." *Proceedings of the IEEE Workshop on Future Trends in Distributed Computing Systems*, Lisbon (Sep,22-24, 1993), 215-220.

[14] Xu, J. and Parnas, D.L. "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations." *IEEE Transactions on Software Engineering*, 16, No.3 (March 1990), 360-369.

[15] Dertouzos, M.L. and Mok, A.K."Multiprocessor on-line Scheduling of Hard-real-time Tasks" *IEEE Transactions on Software Engineering*, 15, No.12 (Dec. 1989), 1497-1506.

[16] Burns, A. "Scheduling Hard Real-time Systems: A Review." *Software Engineering Journal*, 6, No.3 (May 1991), 116-128.

[17] Lehoczky, J.; Sha, L., and Ding, Y."The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior." *Proceedings: Real Time Systems Symposium, IEEE Cat.* No.89CH2803-5 (Dec. 1989), 166-71.

[18] Stankovic, J. and Ramamritham, K."The Spring Kernel: A New Paradigm for Real-Time Operating Systems." *ACM Operating Systems Review*, 23, No.3 (July 1989), 54-71.

[19] Auslander, D.M. and Sagues, P. *Microprocessors for Measurement and Control.* Berkeley, CA: Osborne/McGraw-Hill (1981).

الجدولة الديناميكية في نظام موزع حقيقي للزمن RTDOS

برادلي سويم، مراد تايلي و محمد علي بن معيزه

كلية علوم الحاسب والمعلومات - جامعة الملك سعود

ملخص البحث . من المتوقع أن تتعامل النظم الموزعة في الزمن الحقيقي مع بيئات ديناميكية يصعب التوقع فيها لعدد الوظائف والأعمال التي ينبغي معالجتها، خاصة أن هذا العدد يتغير مع الزمن وحدوده غير معروفة. في مثل هذه الحالات الشاذة، ليست الجدولة وحدها هي التي تصبح ديناميكية مطلقة، بل ينبغي أيضًا أن يتكيف الهيكل العام للنظام والمعمارية مع التشكيلات المحتملة. وتعالج هذه الورقة مشكلة الجدولة الديناميكية في نظام موزع حقيقي للزمن RTDOS . ويُعد هذا النظام فريدًا في تعامله مع هذه المشكلة الصعبة، ذلك أنه يقدم بتنزيل تشكيل شبكي مرن للأجهزة على شبكة من الوظائف الحاسوبية. يحاول النظام RTDOS اكتشاف خصائص المجدول في ضوء التكيف المتوقع حتى يخفف العبء على التطبيقات فيما يتعلق بالمتطلبات قبل التنفيذ. وتوضح الورقة المعمارية العامة للمجدول مع نظام ديناميكي لضمان الموعد النهائي وبعض النتائج التجريبية.