RESEARCH FEATURE

Deterministic Preemptive Scheduling of Real-Time Tasks

Algorithms for the preemptive scheduling of deterministic, real-time tasks can have applications in providing quality-of-service guarantees to packet flows in multichannel optical networks.

Laura E. Jackson
George N. Rouskas
North Carolina State University

Scheduling is the science of allocating limited resources to competing tasks over time. A feasible schedule satisfies the constraints that accompany any particular collection of tasks and resources. Often, however, finding a single feasible schedule is not enough. In some scheduling problems, the goal is to find the optimal schedule from among all feasible schedules, according to a desired optimality principle.

In a multiprocessor system, the limited resources consist of one or more processors, which can be either identical or distinct with respect to function and speed. The systems we focus on process real-time tasks, characterized by their computation time, ready time, and deadline. Task parameters are deterministic: A task first becomes available for processing at its ready time, and it must receive an amount of processing equal to its computation time within the next deadline units of time. Task preemptions are allowed, meaning that the system can interrupt the processing of one task to process another task.

Scheduling problems are characterized by the types of processors and tasks in the system and by the scheduling constraints imposed on them. For a given task set, a number of processors greater than or equal to one, and a set of constraints, scheduling addresses two problems:

- The decision problem: Does a feasible schedule exist?
- The search problem: What scheduling algorithms produce a feasible schedule?

Although researchers typically have studied periodic real-time task scheduling in the context of multiprocessor systems, similar problems arise in the context of ATM networks and the Internet, where the objective is to provide quality-of-service guarantees to packet flows. In a network environment, packet scheduling takes place within an ATM switch or Internet router, and the outgoing link bandwidth is the limited resource of interest.

PACKET-SCHEDULING ALGORITHMS

Many algorithms developed for multiprocessor systems in the 1970s were applied successfully in commercial products in the late 1990s to solve packet-scheduling problems. For example, the infinite time slicing (ITS)[1] technique, also known as generalized processor scheduling (GPS), led to the development of a theory for providing delay bounds and bandwidth guarantees in a network of routers and switches.[2]

Weighted fair queuing (WFQ) is a practical scheduling algorithm that emulates GPS and shares its QoS properties. A network in which each node implements a version of the earliest-deadline-first[3] algorithm can achieve similar bounds and guarantees.[4] Developers based the weighted round-robin (WRR) algorithm on a concept similar to minimal time slicing (MTS). These algorithms are all widely implemented in the Internet today.

So far, packet-scheduling applications use only uniprocessor algorithms because an outgoing link's bandwidth represents a single resource, much like a uniprocessor's capacity. The emergence of wavelength division multiplexing (WDM), on the other

 0018-9162/02/$17.00 © 2002 IEEE

Explanation of Terms

The following terms are used in working with real-time scheduling problems:

- *Static versus dynamic problems.* The parameters describing a task set, such as arrival times, are known in advance in static problems but may be initially unknown in dynamic problems.
- *Slotted versus continuous time.* Slotted time divides time into equal-length slots and requires that a processor begin new work only at slot boundaries. In continuous time, tasks can begin processing or be preempted at any time.
- *Single versus multiple processors.* A system can be a uniprocessor or a multiprocessor.
- *Identical versus distinct processors.* Identical processors can execute all tasks, and each processor executes a given task at the same speed. Distinct processors can vary in processing speed or in functionality in that a given processor may only be capable of processing a subset of tasks.
- *Partitioning versus global schemes.* In a multiprocessor system, a partitioning scheme assigns each task to a processor before execution begins. The processor to which the task initially was assigned must then execute all instances of a periodic task. In a global scheme, any processor is eligible to process a task.
- *Fixed-priority versus dynamic-priority algorithms.* A fixed-priority algorithm assigns priorities to tasks only once, during a preprocessing phase. A dynamic-priority algorithm can reevaluate task priorities over the course of the schedule.
- *Real-time tasks.* These tasks are characterized by ready time R_i, computation time C_i, and deadline D_i, $0 < C_i \leq D_i$. A task requires processing units C_i to begin on or after time R_i. The task must be completed by $R_i + D_i$.
- *Periodic real-time tasks.* These tasks are characterized by R_i, D_i, C_i, and period T_i, $0 < C_i \leq D_i \leq T_i$. The first instance of task i occurs on the interval $[R_i, R_i + D_i)$; the $(k_i + 1)$-th instance of task i occurs on $[R_i + k_iT_i, R_i + k_iT_i + D_i)$. Each task instance requires C_i processing units.
- *Hard real-time versus soft real-time scheduling.* Although soft real-time scheduling can tolerate the lateness of tasks, hard real-time scheduling requires that every deadline be respected.
- *Feasible schedule.* A schedule is feasible if it meets each task's deadline without violating any scheduling-problem constraints.
- *Optimality criteria.* An optimality criterion assesses the relative merits of competing feasible schedules. Examples of such criteria include minimizing schedule length, minimizing maximum lateness, or minimizing the number of late tasks.
- *Real-time versus periodic real-time task systems.* In a real-time task system, (R_i, D_i, C_i) specifies each task. Such a system can include one or more periodic tasks. In this case, each instance of a periodic task is a distinct task. A periodic real-time task system characterizes every task as (R_i, D_i, C_i, T_i).

hand, has led to the deployment of multichannel optical networks that divide a link's bandwidth into several independent channels, each operating on a different wavelength. The channels are identical if they operate at the same data rate; otherwise they are distinct. Therefore, each link in a WDM network can be modeled as a multiprocessor system.

Projects such as the differentiated services (Diffserv) initiative of the Internet Engineering Task Force (http://www.ietf.org/html.charters/diffserv-charter.html) are increasingly emphasizing the importance of guaranteeing QoS for real-time traffic. Combining the deployment of WDM with the need for QoS will spawn considerable interest in developing and implementing appropriate packet-scheduling algorithms. Already, government-sponsored research projects are moving in this direction, including MIT's Onramp project[5] and the Helios project (http://helios.anr.mcnc.org/.

Unlike previous work surveying real-time scheduling results,[6] our survey focuses on the relative performance of preemptive scheduling algorithms that are applicable to computer network traffic. Preemption is appropriate in this setting because the dataflow is already broken into small, atomic chunks—data packets or cells.

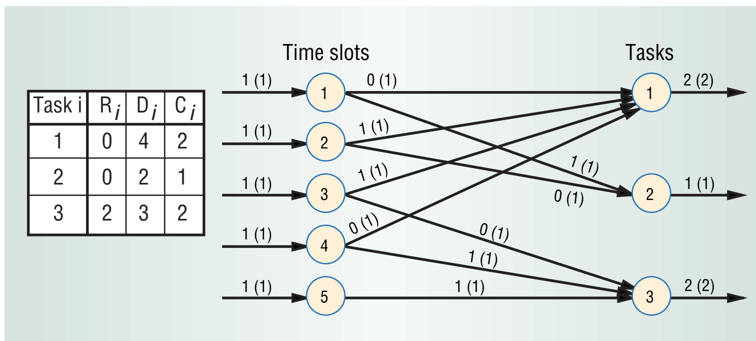
SCHEDULING PROBLEMS

The "Explanation of Terms" sidebar lists and defines the terms we use in working with scheduling problems.

In its general form, we define a real-time task system S with n processors and m tasks, $m > n \geq 1$, as a 4-tuple (R, C, D, T) as follows:

- $R = \{R_1, \ldots, R_m\}$ is the set of ready times for the m tasks.
- $C = [C_{ij}]$ is the $m \times n$ matrix of computation times, where C_{ij} is the computation time of task i on processor j. If all processors are identical, then $C = \{C_1, \ldots, C_m\}$, where C_i is the computation time of task i on any processor.
- $D = \{D_1, \ldots, D_m\}$ is the set of deadlines for the m tasks.
- $T = \{T_1, \ldots, T_m\}$ is the set of periods for the m tasks, with $0 < C_i < D_i \leq T_i$.

The set T is relevant only for *periodic* real-time



Task i	R_i	D_i	C_i
1	0	4	2
2	0	2	1
3	2	3	2

Figure 1. Network representation of a sample task set. A feasible flow assignment—and the arc capacity, in parentheses—appears on each arc.

tasks. The *density*—or *load factor*—of a periodic task is $\rho_i = C_i/T_i$, $i = 1, \ldots, m$, and lies strictly between 0 and 1. The density ρ_i is the fraction of time that a processor must dedicate to the task over the long run to successfully meet its deadlines. The density of a set of m tasks is $\rho = \Sigma_{i=1}^{m} \rho_i$.

Scheduling constraints classification

Three constraints apply to all the deterministic preemptive-scheduling problems we consider. The *task constraint* states that, at any point, no task can be executed at more than one processor. The *processor constraint* requires that, at any point, no processor can work on more than one task. Last, the *zero-time context switch constraint* ensures that a processor loses no time switching from one task to another. Other constraints that uniquely distinguish a particular scheduling problem include the following:

- task sets can involve periodic or nonperiodic real-time tasks ;
- time can be continuous or slotted;
- task parameters can be static or dynamic; and
- the number of processors determines whether the system is a uniprocessor or a multiprocessor.

Stating the applicable constraints for each variation of a scheduling problem provides a quick reference point to facilitate comparisons between problems.

Packet scheduling in WDM networks

Sending packet traffic over WDM networks requires delay and bandwidth QoS guarantees. For real-time tasks such as transmitting voice or video traffic, a deadline D_i typically expresses a delay guarantee in which a packet i of length C_i arriving at time R_i must complete service—be transmitted out of the router or switch—by time $R_i + D_i$. These services can be periodic or nonperiodic. The transmission time for periodic requests such as a typical uncompressed voice application is 20 ms and 1/30 second for high-quality video. In general, the deadline D_i at each router or switch in the packet's path is $D_i < T_i$. Nonperiodic requests are delay-sensitive messages containing network control and monitor information that must be received promptly to be useful. Examples include alarm messages, such as alerts of node or link faults, and network and traffic management messages, such as congestion notifications.

A bandwidth guarantee for a packet flow is the amount of service C_i that the packets must receive within each interval of length T_i. These requests are periodic because the flow must receive C_i units of service every T_i units of time. For example, an uncompressed voice application requires 64 Kbps of bandwidth or it becomes unusable. Usually, the C_i service units can take place anywhere in the time interval of length T_i, thus $D_i = T_i$.

NONPERIODIC REAL-TIME TASKS IN SLOTTED TIME

Figure 1 shows the network formulation from Paul Bratley and colleagues for scheduling a sample nonperiodic real-time task set.[7] The leftmost set of nodes in the network contains the possible time slots for scheduling tasks, where time slot 1 is the interval [0,1), time slot 2 is [1,2), and so on. The rightmost set contains one node for each task in the task set.

Arcs entering each time slot node have unit capacity because the single processor can execute at most one unit of work per time slot. The arcs joining time slots to tasks also have unit capacity because the processor will execute no more than one unit of work from a task in a time slot. An arc joins each task to each time slot in which the processor can execute it. For example, task 3 arrives at time $R_i = 2$ and must finish by time $R_i + D_i = 5$; therefore the processor can execute task 3 in slots 3, 4, or 5. Finally, each task node i has one exiting arc with capacity equal to C_i.

A task set is feasible if a feasible flow assignment fills each arc leaving a task node to capacity. The arc labels indicate one feasible flow assignment for the problem in Figure 1. While Bratley and colleagues offered no conditions to ensure the feasibility of a task set, their variation of the Ford-Fulkerson primal-dual algorithm for the transportation problem maximizes the flow through the network. Successful termination of the algorithm results in a feasible schedule, while unsuccessful termination indicates that the problem is infeasible.

In the multiprocessor version of this problem, increasing the capacity of the arcs entering the time slot nodes from one to n extends the network representation to a system of n identical processors. If there is a feasible schedule, the same primal-dual algorithm will produce a feasible multiprocessor schedule.

PERIODIC REAL-TIME TASKS WITH DEADLINE EQUAL TO PERIOD

In a periodic task scheduling problem in which the deadline equals the period for each task,[3] another task instance becomes available for processing as soon as the deadline from a task instance expires.

In this case, $M_i(t)$ is the amount of processing that the (k_i+1)-th instance of task i has received by time t, where t falls in the interval $[R_i + k_iD_i, R_i + (k_i + 1)D_i)$. A task is *active* at time t if it has requested processing but has not yet received an amount of processing equal to its computation time, C_i. That is, task i is active at time t if $M_i(t) < C_i$, where t falls in the interval $[R_i + k_iD_i, R_i + (k_i + 1)D_i)$. An algorithm is *optimal* if it always finds a feasible schedule when one exists.

Uniprocessor algorithms for continuous and slotted time

Four optimal algorithms apply to the single-processor version of this problem. The earliest deadline first and slack time algorithms achieve scheduling through dynamic priority assignment, while the infinite time slicing and minimal time slicing algorithms use processor sharing.

Earliest deadline first. Developed in 1967,[8] the earliest deadline first (EDF) algorithm—also referred to as the relative urgency algorithm or the deadline driven rule—dictates that at any point, the system must assign highest priority to the active task with the most imminent deadline. The processor executes tasks according to their priority. The assignment is *dynamic* because a task's priority changes as time passes.

The processor only reassigns priorities when a new task instance becomes active. If the highest priority task completes processing before another task instance becomes active, the processor doesn't need to reassign priorities; it simply turns to the task with the next highest priority. Full processor utilization is possible for an arbitrarily large task set.[3] EDF yields a feasible schedule for a set of m tasks whenever

$$\rho = \sum_{i=1}^{m} \rho_i \leq 1 \quad (1)$$

A task set cannot request a quantity of work that exceeds the processor's available capacity. Thus, Equation 1 is a necessary and sufficient condition for the feasibility of a task set, and the EDF algorithm is optimal for building a feasible schedule.

Slack time. The slack time algorithm—also referred to as the least laxity algorithm[6]—is optimal in a continuous-time environment.[9] ST dynamically assigns priorities to active tasks in order of nondecreasing *slack time*—the difference between the task's *relative deadline* and its *remaining computation time*. For the $(k_i + 1)$-th instance of task i at time t, where t lies in the interval $[R_i + k_iD_i, R_i + (k_i + 1)D_i)$, the relative deadline $D_i(t)$ and remaining computation time $C_i(t)$ are

$$D_i(t) = R_i + (k_i + 1)D_i - t$$
$$C_i(t) = C_i - M_i(t)$$

Thus, for task i at time t, the slack time is $D_i(t) - C_i(t)$.

The infinite time slicing algorithm provides insight into the relationship between processor load and individual task densities.

ST measures a task's relative urgency. Suppose that a particular task becomes active while the processor is occupied with other work of higher priority, preventing the task from receiving processing. The task's initial slack time is equal to $D_i - C_i$ but, as time passes, this ignored task's slack time steadily decreases until it reaches 0.

At this point, the processor must begin executing the task—there is just enough time remaining before the deadline to process the task to completion. If the processor does not begin executing the task at this critical moment, the slack time becomes negative, and the task is sure to miss its deadline.

Infinite time slicing. The infinite time slicing algorithm[1] is fundamentally different from the EDF and ST dynamic priority assignment algorithms.[3, 9, 10] ITS provides insight into the relationship between processor load and the individual densities of tasks in the task set. This algorithm divides time into small intervals of length ΔL, and the processor executes each task on each interval in any order for a period $\rho_i\Delta L$. By allowing ΔL to approach zero, ITS achieves infinitely small time slices.

Applying the ITS policy to one processor operating at a speed S is equivalent to a system of m distinct processors, one for each task in the task set, in which processor i operates at a speed ρ_iS and processes task i exclusively. On each interval ΔL, task i receives an amount of processing equal to $\rho_i\Delta L$, and on each period of length T_i, task i receives $(T_i/\Delta L)\rho_i\Delta L = \rho_iT_i = C_i$ units of processing. Since each task receives exactly its computation time on each period, ITS is an optimal scheduling policy.

Minimal time slicing. Since an exact ITS policy requires that ΔL approach 0, processors cannot implement ITS. However, a processor capable of rapid context switching can achieve a good approximation of ITS by allowing ΔL to be small relative to the task periods T_i. Thus, the minimal time slicing algorithm offers a practical alternative to ITS.

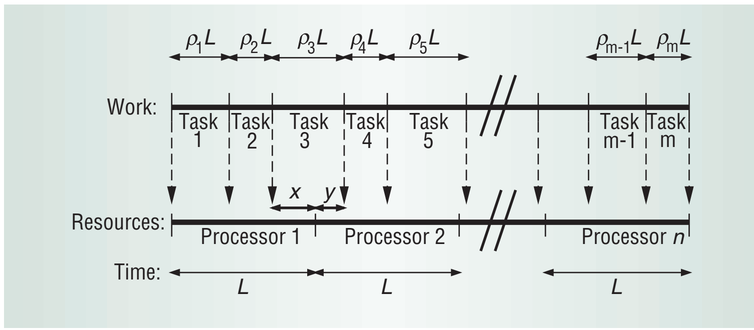


Figure 2. Assigning tasks to processors during a scheduling interval of length L. Task 3 overlays the boundary between Processors 1 and 2 and will thus receive processing time on both processors.

In an MTS implementation, a scheduling event is any event affecting the processor load, such as a task arrival or deadline, and a scheduling interval is the time between two successive scheduling events. Instead of the infinitely short intervals ITS uses, MTS considers successive scheduling intervals.

On each scheduling interval, MTS allocates to each task an amount of processing time commensurate with its density. Thus, for a scheduling interval of length L, each task is processed in an arbitrary order for a length of time $\rho_i L$, which ensures that each task meets its deadline. Consequently, MTS is an optimal scheduling policy.

Periodic and nonperiodic real-time task mix. This set of real-time tasks can contain either periodic or nonperiodic tasks. Both EDF and ST are optimal for scheduling static real-time tasks in continuous time, producing a feasible schedule if one exists. Even better, these two algorithms can handle dynamic task parameters, meaning the scheduler has no knowledge of upcoming tasks until the instant they arrive at the system. Thus, a system using EDF or ST will fare well even when tasks are unexpectedly added to or removed from the task set.

Multiprocessor algorithms

The system containing multiple identical processors has a single necessary and sufficient condition for the feasibility of a task set, for both slotted and continuous time.

Continuous time. In a system with n identical processors, the condition in Equation 1 becomes

$$\rho = \sum_{i=1}^{m} \rho_i \leq n \quad (2)$$

which is both necessary and sufficient for feasibility.[11] In Equation 2, a task set requires at most ρt units of processing time on an interval of length t, and n processors can complete at most nt units of work. To establish sufficiency, it must always be possible to build a feasible schedule whenever Equation 2 is satisfied. An algorithm must only build a schedule from time 0 to D, where D is the least common multiple of all task periods. At D, a new period begins for each task, just as at time 0. Therefore, if there is a feasible schedule for task system S on the interval $[0, D)$, this schedule can repeat at time lD, $l = 1, 2, \ldots$, to effectively form a feasible schedule for S for all times.

Building a simple assignment scheme that achieves feasible scheduling requires making two assumptions. First, we will consider only task sets for which $\rho = n$. If the density ρ of the task system S is strictly less than n, we can add one or more dummy tasks until the total density equals n. Scheduling a dummy task effectively schedules idle time on that processor. Second, we assume that the start time of each task is $t = 0$. At $t = 0$, every task begins requesting its full computation time, and the amount of available processor time is just enough to meet all requests.

To build a feasible schedule, we focus on scheduling intervals, like MTS. On any scheduling interval of length L, each task i will receive an amount of processing proportional to its density, or $\rho_i L$. Because the processors are identical, we can assign tasks to processors arbitrarily, as long as we do not violate the task constraint: At any point, no task can be executed at more than one processor.

We begin by arbitrarily numbering the processors 1 to n and the tasks 1 to m. As Figure 2 shows, we can view the processor resources available on the scheduling interval as a continuous ribbon of length nL, divided into n sections each of length L. Likewise, we can view the work to be completed on the scheduling interval as a ribbon of length nL, divided into m sections, in which section i corresponds to task i and has length $\rho_i L$. We can overlay the work ribbon with the resource ribbon to effectively assign tasks to processors over the scheduling interval without violating the task constraint.

If one task happens to overlay the boundary between two processors, as Task 3 does in Figure 2, it will receive processing time on both processors. Task 3 will receive y units of processing time on Processor 2 at the beginning of the scheduling interval, then x units on Processor 1 at the end of the scheduling interval. No task will be assigned to different processors at the same time because $\rho_i < 1$ for each i. Since each task receives exactly $\rho_i L$ processing time on each scheduling interval of length L, all deadlines will be met.

Slotted time. In the slotted time environment, we first consider the decision problem of whether a feasible schedule exists for a given task set and n processors. In 1996, Sanjoy Baruah and colleagues[12] proved that the condition is necessary and sufficient for the existence of a feasible schedule. They proposed an additional scheduling constraint that is stricter than the requirement that every deadline be respected. A schedule that satisfies this new requirement also respects every deadline.

Task i	R_i	D_i	C_i	ρ_i
1	0	4	3	.75
2	0	4	2	.5
3	0	4	3	.75

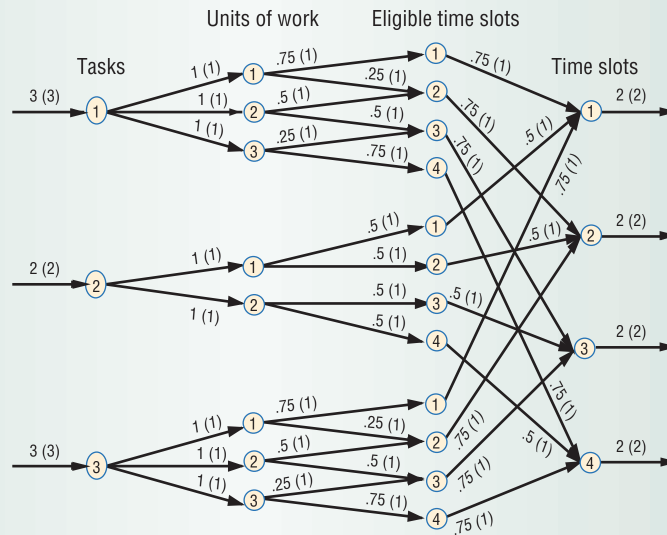


Figure 3. Network representation of a P-fair schedule. A fractional feasible flow assignment—and the arc capacity, in parentheses—appears on each arc.

This *proportionate fairness* or *P-fairness* requirement attempts to allocate slots of processor time to each task proportional to its density. In particular, at each time t, the amount of processing that task i has received must lie between $\lfloor \rho_i t \rfloor$ and $\lceil \rho_i t \rceil$. Baruah and colleagues proved that the condition in Equation 2 is both necessary and sufficient for the existence of a P-fair schedule. This P-fairness requirement applies only to problem instances in which $\rho = n$. If a task set's density falls short of n, the algorithm can add one or more dummy tasks.

To prove the sufficiency of Equation 2 for the existence of a P-fair schedule, Baruah and colleagues first transform the scheduling problem of m periodic tasks on n processors into a directed graph G with integer-valued arc capacities; Figure 3 shows this network representation for a sample problem of three tasks and two processors. They next demonstrate how to map a particular integer-valued flow in G to a P-fair schedule for the scheduling problem, and they prove that such a flow always exists in G. The proof relies on Ford and Fulkerson's Integrality theorem, which states that since each edge in G has an integer-valued capacity, if there is a maximum fractional flow, there is also a maximum integer-valued flow.

An online polynomial-time algorithm generates a P-fair schedule, which chooses the n tasks for processing at each slot with a running time that is linear in the size of the input in bits. Baruah later gave an algorithm that improves the running time at each slot to $O(min\{\, n \log m, m\})$.

PERIODIC REAL-TIME TASKS WITH DEADLINE LESS THAN PERIOD

When a task's deadline equals its period, a new task instance always becomes available for processing the moment the previous instance's deadline expires. Allowing the deadline to be less than the period creates a stretch of dead time—after the deadline passes but before the period ends—during which no instance of the task is available for processing.

Uniprocessor algorithms for continuous time

Researchers have found that both EDF and ST are optimal for this scheduling problem. However, optimality implies that these algorithms will find a feasible schedule whenever one exists.

Joseph Leung and M.L. Merrill[13] proved that the problem of determining whether a feasible schedule exists for a given task set is NP-hard, and they presented an exponential-time algorithm to solve the problem. They also showed that a sufficient but not necessary condition for the feasibility of a task set is

$$\sum_{i=1}^{m} \frac{C_i}{D_i} \le 1$$

while a necessary but not sufficient condition is

$$\sum_{i=1}^{m} \frac{C_i}{T_i} \le 1$$

Later, Leung[10] posed a different decision problem: Can a fixed-priority scheduling algorithm such as ST or EDF build a feasible schedule for a given task set and $n = 1$ processor? He showed this problem to be co-NP-hard and presented an exponential-time algorithm to solve it.

Multiprocessor algorithms for continuous time

Another decision problem concerns whether a feasible schedule exists for a multiprocessor system given a task set and $n > 1$ processors. Eugene Lawler and Charles Martel[14] introduced an exponential-time algorithm to solve this problem. Leung found that ST is more effective than EDF for each fixed $n > 1$: Every task set schedulable by EDF can also be

Partitioning schemes assign tasks to a processor exactly once, and every instance of a given task must be executed on its assigned processor.

scheduled under ST, but one or more task sets are schedulable under ST that EDF fails to schedule.[11]

Finally, Leung considered the multiple processor decision problem: this is co-NP-hard, and the exponential-time algorithm that solves the single-processor equivalent also solves this problem.

PARTITIONING SCHEMES

In global scheduling schemes, any processor can execute successive task instances. In contrast, partitioning schemes assign tasks to a processor exactly once, and every instance of a given task must be executed on its assigned processor. The task constraint is never violated because only one processor can execute a given task. Compared to a global scheme, a partitioning scheme requires less runtime overhead because it assigns tasks to processors only once, before scheduling begins.

A partitioning scheme consists of two phases. During the *task assignment phase*, a bin-packing heuristic treats the processors as bins to fill with as many tasks as possible; a *schedulability condition* tests whether a processor can accept more tasks. During the *scheduling phase*, each processor uses a uniprocessor scheduling algorithm to schedule its assigned tasks. In contrast to a global approach in which the number of processors is fixed, a partitioning scheme usually fixes the uniprocessor scheduling algorithm and then seeks a task assignment algorithm that achieves a feasible schedule using as few processors as possible.

For a given uniprocessor scheduling algorithm, an optimal task assignment algorithm produces a feasible schedule with the smallest number of processors. When either a fixed- or dynamic-priority scheduling algorithm is used, the problem of finding an optimal assignment of tasks to processors is NP-hard.[15] Therefore, most studies concentrate on partitioning schemes that strike a balance between computational complexity and performance. Employing an extra processor or two beyond the optimal number can be a small price to pay when it significantly reduces the complexity.

Almut Burchard and colleagues[16] examined partitioning schemes that use the rate-monotonic algorithm during the scheduling phase. RM, a fixed-priority algorithm, gives higher priority to tasks with shorter periods. A fixed-priority algorithm has lower computational overhead than a dynamic-priority algorithm, and RM in particular has been proven optimal among all fixed-priority algorithms.[3]

Burchard and colleagues considered the problem of scheduling periodic real-time tasks on multiple processors in continuous time for a task set in which each task's deadline equals its period. They studied two variants of this problem, the first requiring that all task parameters be known at the start, and the second allowing tasks to enter and exit the system dynamically. Their major contribution is the introduction of tight schedulability conditions that yield a greater maximum achievable load per processor than previous partitioning schemes. Under the further assumption that task densities are small relative to processor capacity, these partitioning schemes can achieve nearly full processor utilization.

Sylvain Lauzac and colleagues[17] also employed RM during the scheduling phase, but they achieved higher processor utilization than the Burchard scheme through innovations to the partitioning phase. First, they transformed the task set to obtain a tighter schedulability condition. Second, using the first-fit bin-packing algorithm, they assigned compatible tasks—tasks that have the same or nearly the same period—to a single processor, resulting in fuller utilization of that processor. The Lauzac scheme performs best under heavy loads that require approximately 15 to 50 processors, better enabling the first-fit algorithm to assign compatible tasks to the same processor.

Sergio Saez and colleagues[18] relaxed the requirement for an equal deadline and period, allowing a task's deadline to be less than its period. They presented two partitioning schemes, one using RM and the other using EDF, that both sacrifice computational efficiency to obtain close-to-optimal processor utilization. Introducing a two-step schedulability test to determine whether a processor can accept another task increases the complexity in the partitioning phase. This additional requirement transforms the bin-packing problem into the more difficult restricted bin-packing problem. Given this increased complexity, Saez and colleagues advised that their approach is better suited for multiprocessor systems with fewer than 16 processors.

The scheduling of real-time tasks in multiprocessor systems bears similarities to packet scheduling in multichannel optical networks. While many of the algorithms described in this survey can be applied directly in a network context, existing multiprocessor scheduling algorithms do not address certain aspects of packet scheduling.

Examples include the need to provide quantitative delay guarantees, the fair allocation of excess capacity (bandwidth) to packet flows, and the performance of algorithms when a packet must traverse a sequence of identical schedulers in a network path. We are currently studying some of the problems in this exciting research area, which is likely to become more essential in the future.

Acknowledgment

This work was supported by a GAANN fellowship and the NSF under grant NCR-9701113.

References

1. O. Serlin, "Scheduling of Time Critical Processes," *Proc. AFIPS Spring Joint Computer Conf.*, AFIPS Press, Montvale, N.J., 1972, pp. 925-932.
2. A.K. Parekh and R.G. Gallagher, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Multiple Node Case," *IEEE/ACM Trans. Networking*, vol. 2, no. 2, 1994, pp. 137-150.
3. C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM*, Jan. 1973, pp. 46-61.
4. L. Georgiadis et al., "Efficient Network QoS Provisioning based on Per Node Traffic Shaping," *IEEE/ACM Trans. Networking*, vol. 4, no. 4, 1996, pp. 482-501.
5. A.C. Kam, K-Y. Siu, and R.A. Barry, "Toward Best Effort Services over WDM Networks with Fair Access and Minimum Bandwidth Guarantee," *IEEE JSAC*, Sept. 1998, pp. 1024-1039.
6. J.A. Stankovic et al., "Implications of Classical Scheduling Results for Real-Time Systems," *Computer*, June 1995, pp. 16-25.
7. P. Bratley, M. Florian, and P. Robillard, "Scheduling with Earliest Start and Due Date Constraints," *Naval Research Logistics Quarterly*, Dec. 1971, pp. 511-519.
8. M.S. Fineberg and O. Serlin, "Multiprogramming for Hybrid Computation,"*Proc. AFIPS Fall Joint Computer Conf.*, Thompson, Washington, D.C., 1967.
9. M.L. Dertouzos and AK-L. Mok, "Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks," *IEEE Trans. Software Eng.,* Dec. 1989, pp. 1497-1506.
10. J.Y-T. Leung, "A New Algorithm for Scheduling Periodic, Real-Time Tasks," *Algorithmica*, vol. 4, 1989, pp. 209-219.
11. E.G. Coffman Jr., "Introduction to Deterministic Scheduling Theory," *Computer and Job-Shop Scheduling Theory*, E.G. Coffman Jr., ed., John Wiley & Sons, New York, 1976, pp. 1-50.
12. S.K. Baruah et al., "Proportionate Progress: A Notion of Fairness in Resource Allocation," *Algorithmica*, vol. 15, no. 6, 1996, pp. 600-625.
13. J.Y-T. Leung and M.L. Merrill, "A Note on Preemptive Scheduling of Periodic, Real-Time Tasks," *Information Processing Letters,* Nov. 1980, pp. 115-118.
14. E.L. Lawler and C.U. Martel, "Scheduling Periodically Occurring Tasks on Multiple Processors," *Information Processing Letters*, vol. 12, no. 1, 1981, pp. 9-12.
15. J.Y-T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks," *Performance Evaluation*, vol. 2, 1982, pp. 237-250.
16. A. Burchard et al., "New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems," *IEEE Trans. Computers*, Dec. 1995, pp. 1429-1442.
17. S. Lauzac, R. Melhem, and D. Mosse, "An Efficient Rms Admission Control and Its Application to Multiprocessor Scheduling," *Proc. 12th Int'l Parallel Processing Symp.* (IPPS98), IEEE CS Press, Los Alamitos, Calif., 1998, pp. 511-518.
18. S. Saez, J. Vila, and A. Crespo, "Using Exact Feasibility Tests for Allocating Real-Time Tasks in Multiprocessor Systems," *Proc. 10th Euromicro Workshop Real-Time Systems* (Euro-RTS98), IEEE CS Press, Los Alamitos, Calif., 1998, pp. 53-60.

***Laura E. Jackson** is a doctoral candidate in the Department of Computer Science at North Carolina State University and a network research engineer in the Advanced Network Research group at MCNC in Research Triangle Park, N.C. Her research interests include real-time packet scheduling and quality of service in all-optical networks. Jackson received a master's degree in operations research from the College of William and Mary. She is a student member of the IEEE. Contact her at lojack@ mcnc.org.*

***George N. Rouskas** is an associate professor in the Department of Computer Science at North Carolina State University. His research interests include network architectures and protocols, optical networks, multicast communications, and performance evaluation. Rouskas received a PhD in computer science from the College of Computing at Georgia Institute of Technology. He is a member of the IEEE, the ACM, and the Technical Chamber of Greece. Contact him at rouskas@eos.ncsu.edu.*