Scheduling Real-Time Tasks in Distributed Systems: A Survey

Alex Gantman Pei-Ning Guo James Lewis Fakhruddin Rashid

University of California, San Diego
Department of Computer Science

Abstract

We evaluate the options available to the designers of schedulers or real-time tasks in distributed systems. We also present a select subset of scheduling algorithms ranging from the classic Rate Monotonic Scheduling algorithm to the recent Real Time Self Adjusting Dynamic scheduling algorithm. Each algorithm is evaluated on the design choices made and their applicability to the problem at hand.

1. Introduction

Scheduling of real-time tasks is very different from general scheduling. Ordinary scheduling algorithms attempt to ensure fairness among tasks, minimum progress for any individual task, and prevention of starvation and deadlock. To the scheduler of real-time tasks, these goals are often superficial. The primary concern in scheduling real-time tasks is deadline compliance. Another difference between scheduling ordinary and real-time tasks is the dependence of the latter on the real-time clock. Because of this dependence, Dijkstra's assertion that preemption is transparent to the task [Dij68] does not hold for real-time systems, making preemptive schedulers of real-time tasks much more complex than their "ordinary" counterparts.

In this paper we survey several algorithms developed over the last twenty-five years that are designed to schedule real-time tasks in distributed systems. The algorithms are presented in evolutionary manner – starting from the early theoretical works and introducing new optimizations and heuristics as they were developed over time.

Before examining the actual algorithms, it is helpful to establish the exact meanings of the terms *real-time task* and *distributed system*. We provide a basic definition of what a real-time task is and identify the different dimensions along which this definition may vary. We also briefly describe the different types of distributed systems and how the system architecture affects the scheduler.

In order to be consistent in our analysis of the algorithms we identify the major choices available to the designers of schedulers for distributed real-time systems. We then evaluate the choices made by each algorithm and their applicability to the problem being addressed.

The rest of the paper is structured as follows. Section 2 describes in detail the possible definitions of terms *real-time task* and *distributed system*. Section 3 discusses the various options in available to the designers of scheduling algorithms for distributed real-time systems. Section 4 provides a survey of some of the key algorithms for scheduling distributed real-time systems and analyzes them in terms of the design choices introduced in earlier sections. Section 5 discusses general deficiencies of existing algorithms and suggests future research directions. Section 6 concludes.

2. Problem Formulation

Certain definitions must be established before we can begin analyzing the actual algorithms. In this section we attempt to provide a minimal vocabulary necessary for the discussion of distributed real-time systems.

2.1 Task Characteristics

Perhaps the most important term to define in our context is the *real-time task*. The exact meaning varies greatly between the different domains of Computer Science. Terms such as *application*, *task, sub task*, *task force*, and *agent* are used to denote the same object in some instances, and yet, have totally different meanings in others. In order to be consistent in our analysis we have chosen the following basic definition of a real-time task:

Definition 1: An instance[1] of a task is the basic object of scheduling. Its fundamental properties are arrival time and approximate execution time. In addition, a real-time task specifies a deadline by which it must complete execution.

The above definition can be further enhanced/specialized with the following qualifiers.

2.1.1 Arrival Patterns

Based on their arrival patterns over time, tasks can be divided into two categories: *periodic* and *aperiodic*. A task is said to have arrived when it is available for execution, regardless of processor availability and inter-task dependencies. When all the dependencies of a task are satisfied and a processor is available, that task is released to execute.

Aperiodic tasks have arbitrary arrival patterns. Although some characteristics of an aperiodic task may be known before its arrival (e.g. approximate execution time and relative deadline), none of the scheduling algorithms that we know of can utilize such partial information without knowing the arrival time of an instance of a task. Naturally, aperiodic tasks cannot be scheduled statically (see Section 3.1) with any degree of success.

Periodic tasks, on the other hand, have very regular arrival patterns. The difference in time between the arrival of two consecutive instances of a periodic task is always fixed and is referred to as the period of that task. Although the inter-arrival times (period) of instances of a periodic task are fixed, the inter-release times may not be.

2.1.2 Inter-Task Dependencies

Depending on the system's definition of the task, there may be dependencies among tasks. A task j is defined to be dependent on a set of tasks S(j) if j cannot start executing until all tasks in S(j) have completed their execution. Clearly, scheduling independent tasks is much simpler since there is one less constraint to satisfy. Schedulers that have to deal with task dependencies usually construct a directed acyclic[2] graph (DAG) to represent them. A task represented by node j can be scheduled to start execution only after all tasks represented by parent nodes (also referred to as predecessors) of j have completed their execution. The nodes and edges of a task dependency DAG may have weights associated with them, representing computation and communication costs, respectively. A set of interdependent tasks (represented by a possibly disconnected DAG) is often referred to as an *application* or a *task force*. Because most such applications are subdivided into tasks at compile time, interdependent tasks are usually the subject of static schedulers (see Section 3.1).

2.1.3 Deadline Laxity

All real-time tasks define a deadline by which they have to complete executing. However, the problem of finding an optimal schedule for real-time tasks has been shown to be NP-complete [CHE96]. Not all systems can afford to solve such a complex problem. Instead, many systems implement a heuristic scheduler that, rather than guarantying full deadline compliance, either attempts to minimize the deadline-miss rate or introduces some amount of laxity for each deadline (or both). Laxity is defined as the amount of time by which a task can miss its deadline and still avoid severe consequences.

Real-time systems are often labeled as either *hard* or *soft*, depending on the laxities of tasks and severity of consequences of missed deadlines. Hard real-time systems have little laxity and generally provide full deadline compliance. Soft real-time systems are more flexible. They have greater laxity and can tolerate certain amounts of deadline misses.

2.2 System Architecture

The term *distributed system* means different things to different people. It may refer to a set of networked workstations or to a multi-processor parallel machine. These two types of systems vary greatly in the types of jobs they tend to be running. Networked workstations tend to have dynamic hardware configuration and run random, unpredictable workloads. On the other hand, parallel machines have fixed processing nodes and tend to run tasks that belong to an application and thus, have more predictable behavior.

[1] A task may have multiple instances, just like a function may have multiple invocations.
[2] Since cyclical dependencies cannot be satisfied, they are not addressed here.

The hardware architecture of the distributed system plays an important role in the design of the scheduling algorithm as well. In a homogeneous system no information needs to be stored in association with each processor (aside from the list of tasks already scheduled on it) and only the deadline, approximate execution time, and dependencies on other tasks are associated with each task. On the other hand, in heterogeneous systems, tasks might specify resource requirements that can be satisfied by only a few processors. Thus, in a heterogeneous system, each processor must provide a listing of resources available and each task must specify its complete resource requirements. Also, since processors and communication links might vary in speed, predicting execution time for tasks is much more difficult.

The topology of the distributed system is also important to consider. Scheduling algorithms can often be optimized for regular and well-understood structures such as trees, meshes, hypercubes, etc.

3. Design Options

Having established the basic definitions required for the discussion of distributed real-time systems, we can start examining the various choices available to the designers of schedulers for such systems. We have identified three key choices: should the scheduling be performed statically or dynamically, should the scheduler take into account task priorities[3] and allow preemption, and, in case of dynamic scheduling, should the scheduler be assignment-oriented or sequence-oriented.

3.1 Static vs. Dynamic

Schedulers of real-time applications can be categorized as either static or dynamic depending on the time at which the information necessary for the timely completion of an application is available. If real-time applications can be partitioned into tasks with well-known execution times, scheduling can be done statically at compile time. Static scheduling takes advantage of tasks that have a well-defined structure to optimize deadline compliance. When apriori knowledge of task completion constraints is unavailable to the system, dynamic scheduling can be used. A dynamic scheduler uses the current allocation of system resources to determine the feasibility of processing real-time applications.

Static scheduling algorithms can be classified into three main categories: priority based, clustering based, and duplication based [Dar96]. In priority based schemes, the order of task execution is based on priority assignments. Clustering based algorithms attempt to group tasks with data dependencies on the same processor. This serves to minimize the cost of inter-process communication, but does not scale easily to available system resources. To take advantage of the number of available processors, tasks can be duplicated in order to alleviate dependencies. Consideration of possible side effects outside the scope of parent-child interactions should be taken into account when duplicating tasks. The scalability of duplication based schemes is beneficial for constructing efficient task schedules. It is important to note that static schedulers tend to have more time and space at their disposal and therefore, can afford more expensive techniques producing better schedules.

In dynamic scheduling schemes tasks are assigned to processors at run-time. Information about the current and future availability of system resources can be taken into account when making scheduling decisions. A dynamic scheduler can query this information in order to determine if new tasks can be scheduled, current tasks need to be dropped due to deadline constraints, or task priorities need to be re-evaluated. Although the scheduler can make many optimizations, limits should be placed on the complexity of the algorithm. Efficiency is the main concern for dynamic schedulers because the computational costs of scheduling should not conflict with the processing of real-time tasks. Scheduling tasks dynamically has the additional benefit of being able to adjust to changes in the processing environment, thus providing greater system availability.

3.2 Priority and Preemption

Implicitly, all real-time systems are priority based. Real-time tasks are always prioritized based on their deadlines. Additionally, priorities can be based on such factors as processing time, frequency of execution, or the severity of missed deadlines. Priorities may be fixed or dynamic. Fixed priorities are assigned statically and do not change over time. Dynamic priorities, on the other hand, may be changed at run time.

A useful mechanism in supporting priority policies is preemption. Being able to preempt a low priority task in order to schedule a higher priority one allows the scheduler more flexibility in constructing a schedule. While preemptive schedulers might be able to produce better schedules, they are also more complex than non-

[3] In cases were task priorities depend on more than just deadlines.

preemptive schedulers and require more resources. It is also important to remeber that poorly designed preemption strategies can easily lead to starvation of low priority tasks.

3.3 Assignment-Oriented vs. Sequence-Oriented

The problem of scheduling tasks on multiple processors can be viewed as a search for a feasible schedule. Feasibility of a schedule is usually defined as meeting a set of constraints (e.g. deadline compliance, task dependencies). The search-space can be represented as a tree where each node is a task-processor assignment. Any path from the root to a leaf of the tree is a schedule, though not necessarily a feasible one. A path between any other pair of nodes is referred to as a partial schedule. Most scheduling algorithms traverse such a tree in depth-first order, truncating the tree at nodes that are not feasible task-processor assignments. When the traversal is complete, each remaining leaf (truncated nodes are not leaves) represents a feasible schedule. In cases where multiple feasible schedules exist the goal is to find the optimal one where optimality is usually achieved by either maximizing or minimizing some metric function (e.g. average load, maximum load, last completion time).

This search through the task-processor space can often be categorized as either assignment-oriented or sequence-oriented, depending on the semantics of the edges in the search tree. If, for each level of the search tree, an algorithm chooses a task and attempts to assign it to some processor, then it is assignment-based (Figure 1). On the other hand, if an algorithm picks a processor first and then tries to find a task for it, it is sequence-based (Figure 2). Both types of algorithms traverse the tree depth-first, looking for a feasible schedule, both run in exponential time and space, and both produce identical results if allowed to traverse the entire tree in one iteration.

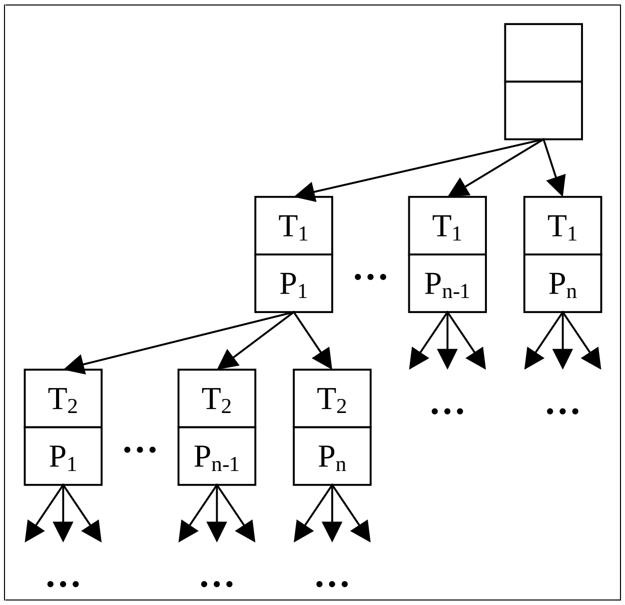

Figure 1: Assignment-oriented search tree.

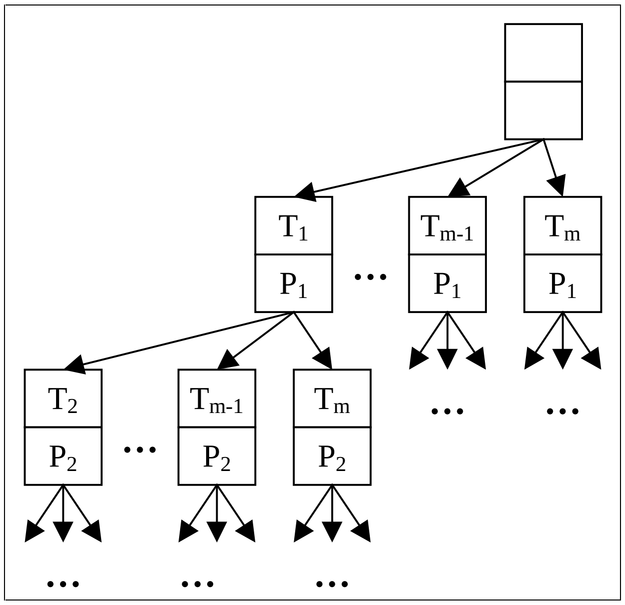

Figure 2: Sequence-oriented search tree.

If the scheduling algorithm is static and has no strict time constraints, it can traverse the entire search tree in one attempt and find all feasible schedules (if any exist). However, if the algorithm is dynamic, i.e. it accepts a stream of new (unpredictable) tasks, then the search must be incremental. Such an algorithm would produce partial schedules at specified intervals. In between intervals, new jobs would be added to the scheduler and jobs with expired deadlines would be removed. Note that for uniprocessor systems, the two approaches remain virtually identical. Although the search trees would have different topologies – a linked list in the assignment-oriented approach and a bush of height 1 in the sequence-oriented case – the task-processor pairs would get evaluated in the same order.

It has been shown that the sequence-oriented approach to scheduling real-time tasks in distributed systems is not very scalable. Because during each iteration it attempts to choose a task to run on the next processor in the queue, a sequence-oriented scheduler is inherently focusing on load balancing rather than deadline compliance. Assignment-oriented schedulers, on the other hand, always seek to schedule the most urgent task.

4. Algorithms

Now that we have introduced the dimensions along which the problems vary and the design choices available to designers of schedulers, it is time to evaluate some of the algorithms developed over the last quarter of the century. Instead of simply going over the most recent developments in the field, we attempt to illustrate the evolution of scheduling algorithms. Each algorithm is described in terms of the problem it attempts to solve, the design choices made by the authors, and how the former influenced the latter.

4.1 Rate Monotonic Scheduling (RMS)

In 1973 Liu and Layland published the paper [Liu73] which provides the theoretical foundation to all modern scheduling algorithms for real-time systems. In their analysis, the authors limit themselves to independent, periodic, preemptable, hard real-time tasks executing on a single processor. The processor is always executing the task with the highest priority. Although Liu and Layland only address scheduling of real-time tasks on a single processor, their results turned out to be applicable to distributed systems as well.

The first scheduling model introduced by Liu and Layland is the RMS. It is a static, fixed-priority scheduler in which the priority of a task is given by a monotonic function of its rate[4] (hence, rate monotonic) – the higher the rate, the higher the priority. Once the priorities are assigned, it is up to the processor to execute the available task with the highest priority. The authors show that if a given set of hard real-time tasks can be scheduled by any algorithm, then it can be scheduled by the RMS.

The RMS algorithm provides a good mathematical basis for determining the upper bound for processor utilization. Unfortunately, according to the authors, that upper bound quickly drops to ln(2) (approximately 70%) as the number of tasks increases. Such poor utilization is not acceptable in most systems. Liu and Layland suggest a better scheduler that uses dynamic priority assignments. This new, deadline-driven scheduling algorithm assigns priorities to tasks according to their deadlines. A task will be assigned the highest priority if its deadline is the nearest (most urgent). Priority assignments are adjusted every time a new task is ready to be scheduled. The deadline-driven algorithm provides full processor utilization. The authors also ascertain that if any algorithm can schedule a set of tasks, then so can the deadline-driven algorithm.

The deadline-driven algorithm still had a major drawback – it was not easily implementable on real systems available at the time[5]. Liu and Layland addressed this issue by developing a mixed algorithm that used RMS for most tasks and deadline-driven scheduling for the rest. Although the mixed algorithm cannot always achieve full processor utilization, in general it provides most of the benefits of the deadline-driven approach while remaining easily implementable on existing systems.

Although the mixed algorithm was developed in response to implementation difficulties, all algorithms presented in this subsection are very analytical (i.e. impractical). For example, the scheduling overhead is never taken into account. Nonetheless, the theoretical analysis presented in [Liu73] provides the mathematical basis for most current scheduling analysis.

4.2 Release Guard Protocol

Synchronization protocols are used to govern the release of tasks so that the precedence constraints among them are satisfied. Essentially, a synchronization protocol is a domain-specific term for a scheduler. In this subsection we discuss three synchronization protocols, namely the Direct Synchronization protocol, Phase Modification protocol and the Release Guard protocol [Sun96], examining strengths, shortcomings, and applicability of each.

These protocols are designed to schedule independent applications consisting of a succession of preemptable, periodic tasks with fixed priorities on a heterogeneous system. The tasks t(a,1),t(a,2),…,t(a,n) of an application a, have dependencies such that t(a,j+1) depends only on t(a,j), for $1 \leq j < n$. Note that in this case, the task dependency DAG is actually a list (each node has at most one parent and at most one child). All tasks in an application have the same period (it is therefore acceptable to refer to it as the application period). Furthermore, each task declares the processor on which its instances must execute. Priorities are assigned to tasks independent of the scheduler. Each processor always runs the released task with the highest priority. Finally, each application has a phase that is defined as the arrival time of the first instance of the first task for that application.

[4] The rate of a task is the reciprocal of its period.
[5] Specifically, no hardware support for dynamic priority assignments.

For the purpose of illustrating the protocols we will use the following example (borrowed from [Sun96]). The system consists of two processors and three applications. Table 1 lists the attributes of each task. Note that the period of each task is chosen to co-incide with the relative deadline.

Application	Task	Computational Cost (in time units)	Period	Relative Deadline	Phase	Priority	Processor
1	t(1,1)	2	4	4	0	1	P1
2	t(2,1)	2	6	6	0	2	P1
2	t(2,2)	2	6	6	0	2	P2
3	t(3,1)	3	6	6	4	1	P2

Table 1: System configuration.

The Direct Synchronization (DS) protocol operates on synchronization signals sent between tasks. A task signals its immediate successor when it has completed executing. Upon arrival of this signal, an instance of its successor is released immediately. If the target processor has a lower priority task running, it is preempted. Figure 3 below illustrates a simple schedule using the DS protocol. Note how the immediate release of the second instance of t(2,2) upon completion of the second instance of t(2,1) preempts task t(3,1). In this case, it causes t(3,1) to miss its deadline.

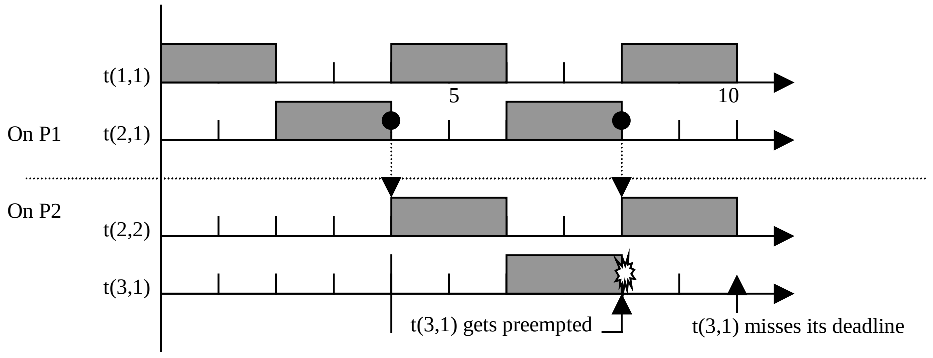

Figure 3 The Direct Synchronization Protocol

The DS protocol has low complexity, minimum overhead, and yields short average end-to-end response (EER)[6] times. It is a reasonable choice for applications with soft timing constraints, short task chains, or with tasks having low processor utilization. However, for applications that have high processor utilization, long task chains, and hard timing constraints, the DS protocol produces large, or even unbounded, worst-case EER times.

Unlike the DS protocol, the Phase Modification (PM) protocol insists that instances of all tasks be released periodically according to the periods of their parent applications. To ensure that the precedence constraints among tasks are satisfied, each task is given its own phase. The phase of the first task, f(a,1), is the same as that of the application. For a later task, we adjust its phase to be the sum of f(a,1) and the bounds on the response times of its predecessors. Since the bound on the response time of a task is the worst case EER time for that task, the estimated worst-case EER time of an application is therefore the sum of the bounds on the response times of all its tasks.

Figure 4 below illustrates the schedule produced by the PM protocol. The bound on the response time of t(2,1) is 4 time units, and therefore the phase of t(2,2) is 4. In this case, the first instance of t(3,1) meets its deadline because the second instance of t(2,2) is not released until time 10 and hence does not preempt the first instance of t(3,1).

[6] The end-to-end response time is defined in [Sun96] as the time between the release of the first task of an application and the completion of the last task.

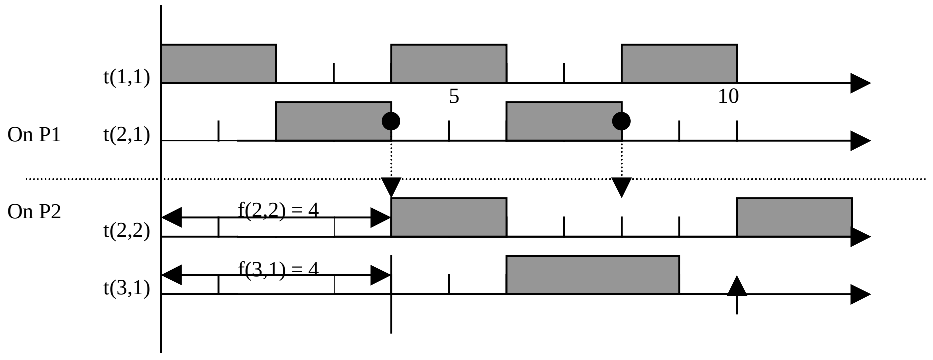

Figure 4 The Phase Modification Protocol

The PM protocol needs strict clock synchronization to guarantee that precedence constraints among tasks are satisfied. Additionally, the inter-release times of tasks are occasionally greater than the period; this too may violate the precedence constraint. To overcome these shortcomings, a slight modification to this protocol is proposed.

The Modified Phase Modification (MPM) protocol addresses these shortcomings at a slightly higher run-time expense. By controlling when the synchronization signal leaves the scheduler, the MPM controls the release of the next instance of a task. This is done is by introducing a delay in sending the synchronization signal between the time the predecessor completes execution and the time the scheduler releases the next instance. Note that when clocks are synchronized, when tasks do not overrun, and when the release of the first tasks is strictly periodic, the PM protocol and the MPM protocol produce identical schedules. However, the MPM achieves this without requiring clock synchronization. This is important since strict clock synchronization is difficult to achieve in practice.

The Release Guard (RG) protocol furnishes a method with which we can obtain an upper bound on the response times of tasks. It combines the strength of the DS and PM protocols while avoiding their shortcomings.

The idea behind the RG protocol is to control the releases of a task such that the inter-release time of any two consecutive instances is no shorter than the period. A release guard, g(a,j), is a timer associated with a task j for application *a* that enforces the earliest allowed release time for t(a,j+1). In an instance of the immediate predecessor of t(a,j) completes after g(a,j) an instance of t(a,j) can be released immediately. Otherwise the instance will not be released until time g(a,j).

The release guard timer, g(a,j), for each task t(a,j) with a period p(a,j) is updated as follows:

```
g(a,j) = 0;  /* Init to 0 so that t(a,j+1) can be released immediately */
        if (idlePoint[7])
                g(a,j) = current_time;
        else
                g(a,j) = current_time + p(a,j);
```

Figure 5 below illustrates the same schedule at Figure 1 and 2 using the RG protocol used. The schedule is similar to the schedule produced by the DS protocol up until when the second instance of t(2,1) completes at time 8. We expect the second instance of t(2,2) to be released at time 10. This was set at time 4 when the first instance of t(2,2) is released. However, time 9 is an idle point on processor P2, and g(2,2) gets updated to the current time 9. Consequently, the second instance of t(2,2) is released at time 9. This early release serves to reduce the average EER times of tasks.

[7] An idle point is a time instant by which all tasks that are released before the instant have completed. Intuitively, it is a time instant when the processor is idle.

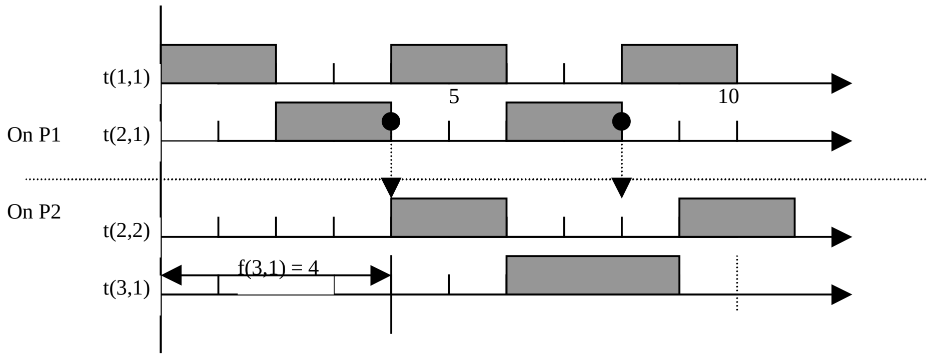

Figure 5 The Release Guard Protocol

The MPM and RG protocols, which have the same bounds on EER times of tasks, are better suited for applications that have high processor utilization, a long task chain in each application, and hard timing constraints. Simulation surveys on estimated worst case and average end to end response times of tasks using the both the MPM and RG protocols show that they outperform the DS protocol [Sun96]. The RG protocol is superior to the MPM protocols because it yields reasonably short average EER times of tasks. However, RG assumes that task execution have small jitters. It also assumes a zero cost of IPC for synchronizing tasks on different processors. Thus, the RG protocol's superiority to MPM protocol is confined to applications that have small output jitters otherwise the MPM protocol is more favorable.

4.3 Search and Duplication Based Scheduling (SDBS)

Real-time applications can be more efficiently scheduled on systems with extra or idle processors through task duplication. The Search and Duplication Based Scheduling algorithm (SDBS) [Dar94] is a static scheduling algorithm designed to minimize the completion times of real-time applications by taking advantage of under-utilized processors. This algorithm produces efficient schedules for applications that are easily partitioned into tasks (Figure 6). Tasks are assumed to be non-preemptive processes running on a homogeneous, connected, and unbounded set of processors.

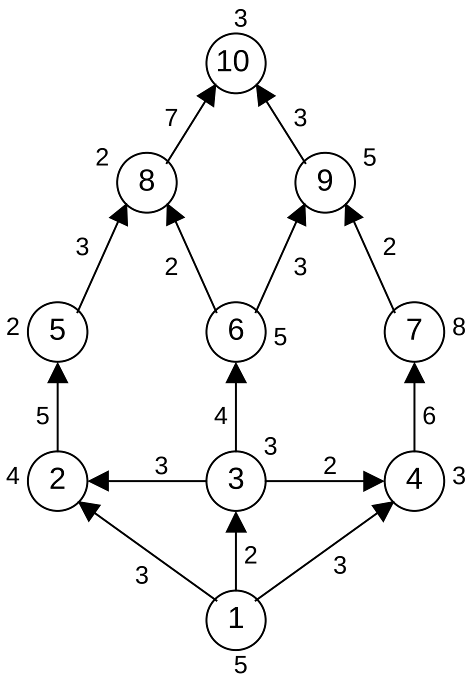

Figure 6: An application partitioned into 10 tasks. The weights of the nodes and the edges represent computation and communication costs, respectively. Node 1 is the entry node, and node 10 is the exit node.

The main focus of the algorithm is to identify critical tasks within the application. A critical task is a node in the task dependency DAG that has more than one parent node. Critical tasks are identified so that the completion times of its predecessors can be evaluated. The predecessor with the latest completion time should run on the same processor as the critical task in order to alleviate inter-process communication costs.

The SDBS algorithm performs three steps to determine an optimal schedule for the tasks of an application. Step one is to compute the earliest start time (EST) and earliest completion time (ECT) for each node in the task dependency DAG. The earliest start time of a node is calculated as follows:

Let PRED(j) be the set of predecessors to node j and $c_{i,j}$ be the communication cost between nodes i and j. Let k be the "bottleneck" node for i, such that

$$\max\{ect(k) + c_{k,i} \mid k \in PRED(i)\}.$$

Then,

$$est(i) = \max\{ect(j) + c_{j,i} \mid j \in PRED(i), j \neq k, ect(k)\}$$

The earliest completion time is simply the sum of EST and the computational cost of the task. The EST of an entry node is always 0. Table 2 contains the EST and ECT for all nodes of the application in Figure 6.

Node	EST	ECT
1	0	5
2	8	12
3	5	8
4	8	11
5	12	14
6	8	13
7	11	19
8	15	17
9	19	24
10	24	27

Table 2: Earliest start and completion times.

Step two of SDBS rewrites the task dependency DAG as an inverted tree (single child, multiple parents) by duplicating any node that has more than one child. The DAG is traversed in reverse breadth first order, starting at the exit node. Any node encountered more than once is duplicated in the new inverted tree. Figure 7 shows the result of step 2 applied to the application in Figure 6.

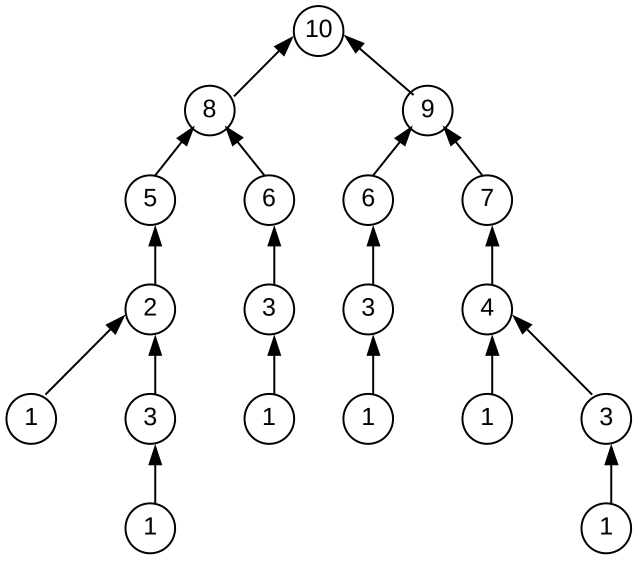

Figure 7: Task dependency inverse tree produced after step 2.

Finally, step three assigns each node in the inverted tree to a processor. A node is always assigned to run on the same processor as the predecessor for which the sum of ECT and communication cost to this node is the greatest. An optimal schedule will be generated when all nodes have been assigned to processors (Figure 8).

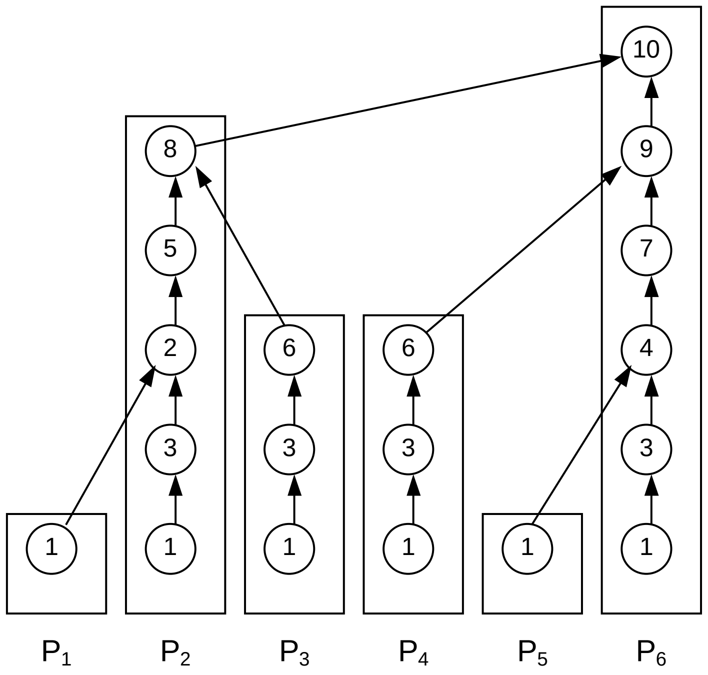

Figure 8: Processor assignments.

Due to the homogeneous nature of the system, task execution times do not vary between processors. Therefore, duplication of tasks on separate processors only serves to minimize communication costs.

One of the shortcomings of the SDBS algorithm is that it makes a rather unrealistic assumption that there are enough available processors to handle any amount of task duplication. Improvements have been made to the SDBS algorithm to address this issue. Darbha and Agrawal, the authors of SDBS, came up with the modification which would make the algorithm more practical [Dar96]. This new algorithm makes the same assumptions about the characteristics of the tasks and the system, with the single exception – the number of available processors is finite. Scheduling is still done statically, so the number of available processors must be known at compile-time.

This new algorithm takes four steps to determine an optimal schedule. Step one computes the earliest start times and earliest completion times as in SDBS. Step two computes the *latest allowable* start end completion times for each task. Step three traverses the graph in a reverse depth fist search, dividing the tasks into clusters. Each cluster represents a path from the first unassigned task to the entry node (Figure 9). The tasks which are clustered together will execute on the same processor. Instead of duplicating all the predecessors to critical tasks (SDBS), this algorithm only makes note of task duplications that would benefit the overall execution time of the application. Once clustering is complete, a schedule (usually sub-optimal) has been determined. Step four uses the information of possible improvements in completion time due to task duplication to scale to the number of processors.

Both scheduling algorithms presented in this section emphasize the strict constraints necessary for producing an optimal schedule with reasonable computation costs. The system must be homogeneous so that the processing times for the same task does not vary between processors. Communication costs between any two processors for the same size message must also be fixed. Although the second algorithm scales to the number of processors, it is not done dynamically. Modifications to the system after compilation of the application are not taken into account.

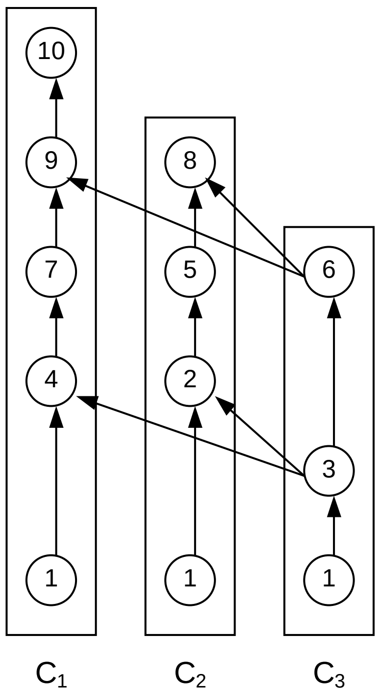

Figure 9: Clustered task duplication

4.4 Real-Time Self-Adjusting Dynamic Scheduling (RT-SADS)

Atif and Hamidzadeh propose a new Real-Time Self-Adjusting Dynamic Scheduling (RT-SADS) algorithm in [Ati98]. RT-SADS possesses a unique ability to tune itself according to the current system state (hence, self-adjusting).

The RT-SADS algorithm is designed for scheduling aperiodic, non-preemptable, independent, soft real-time tasks with deadlines on a set of identical processors with distributed memory architecture. The primary goal of the designers of this scheduling algorithm is scalability of deadline compliance with respect to increases in the number of processors and task arrival rate.

RT-SADS uses a dedicated processor to perform incremental searches through an assignment-oriented task-processor space. Each scheduling stage j receives as input a set of tasks Batch(j) and produces as output Batch(j+1) and a feasible (although possibly only partial) schedule Sj. Batch(j+1) is formed from Batch(j) by removing all scheduled tasks and tasks with missed deadlines and adding tasks which arrived during stage j.

RT-SADS self-adjusts the scheduling stage duration depending on processor load, task arrival rate and slack (maximum time a task can be delayed without missing its deadline). Longer scheduling stages allow the algorithm to traverse a greater part of the search tree and make a more informed choice. The duration of a stage j cannot be greater than the minimum of all slack times for any task in Batch(j). However, there is no benefit in completing stage j before at least one of the processors becomes available. Therefore, the duration of a scheduling stage is chosen such that it runs until at least one processor becomes available and then keeps running as long as all tasks in the current batch have some slack time.

At the end of any scheduling stage there usually more than one feasible partial schedule available. RT-SADS allows the use of a cost (utility) function for determining which of the available schedules to choose.

Atif and Hamidzadeh describe a set of experiments evaluating RT-SADS and comparing it to a leading sequence-oriented algorithm named Distributed Continuous On-Line Scheduling (D-COLS). D-COLS was developed by the authors of RT-SADS just two years before. Both algorithms are similar in all ways but one – their solution-space representation. The results of the experiments demonstrated that the performance (measured in deadline compliance) of the sequence-oriented algorithm increased only moderately as the number of processors in the system was increased from 2 to 8 and then decreased with each additional processor. The assignment-oriented approach, however, yielded a steady (linear, with a sharp slope) increase of performance as the number of processors grew.

The fact that tasks are assumed to be non-preemptable, independent, and without priorities greatly simplifies the design of the scheduler. The absence of preemption means that once a task is added to Sj, it is never again considered by the scheduler. Since the tasks are independent the scheduler does not need to worry about satisfying precedence constraints.

Although the authors of RT-SADS boast impressive results, there are still possible improvements which should be investigated. It would be interesting to explore how RT-SADS can be made distributed. Because the evaluation of any search sub-tree is independent of its siblings, multiple processors can perform the scheduling stage simultaneously (Figure 10). One processor should still be in charge of assigning sub-trees and in choosing the best partial schedule from among the results returned by each scheduling processor. Such an algorithm would, in fact, be neither purely assignment-oriented nor sequence-oriented, but a mix of both. Each scheduling node would still be performing an assignment-oriented search. However, as a whole, the search would resemble the sequence-oriented approach. It would be interesting to observe the tradeoff between allocating more processors to scheduling versus execution of real-time tasks.

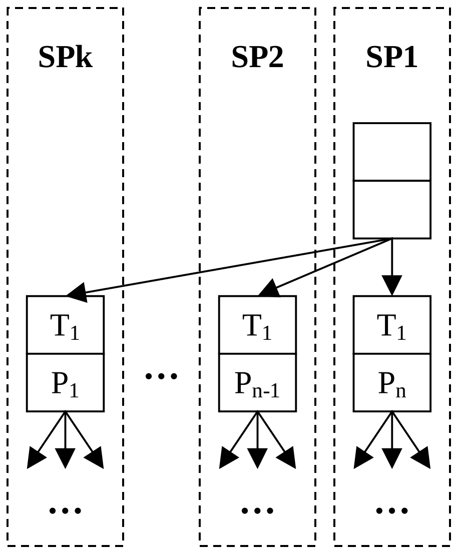

Figure 10: Distributed RT-SADS. SP = Scheduling Processor.

Another area of possible further investigation is "stage granularity." In the current implementation of RT-SADS, at the end of each scheduling stage the entire partial schedule created during that stage is committed. One possible alteration is, in cases of low task arrival rates and/or high slack times, committing only a part of the partial schedule and reconsidering the rest in light of the updated system state.

5. Future Work

In our discussion of the scheduling algorithms we have tried to point out their shortcomings and suggest directions for future research. In addition to those comments, we would like to point out several observations which are not necessarily applicable to any single algorithm but rather to the entire field of scheduling real-time tasks in distributed systems. First, very little work has be aimed at exploring scheduling of real-time tasks in heterogeneous systems. Heterogeneous systems are a lot more difficult to analyze and control from a centralized location than homogeneous systems. This leads into our second observation. There seems to be virtually no work at all on *distributed* scheduling of real-time tasks. All of the scheduling algorithms for real-time tasks that we have encountered were centralized. It would be interesting to investigate the possibilities of scheduling real-time tasks in a distributed fashion.

6. Conclusion

With this paper we hoped to introduce the reader to the problem of scheduling real-time tasks in distributed systems. We presented the different interpretations of the problem and the various options available to the solution designers. Our analysis of some of the existing scheduling algorithms tried to focus on the affect of the specific problem on the choices made in the solution. We hope that what we presented provides the reader with a broad understanding of the problem and a range available solutions. This paper was also aimed at providing the reader with a solid foundation for further research on the subject. Finally, we suggested possible future research directions.

References

[Ati98] Y. Atif and B. Hamidzadeh, "A Scalable Scheduling Algorithm for Real-Time Distributed Systems," Proceedings of the 18th International Conference on Distributed Computing Systems, May 26-29 1998, pp. 352-359.

[Dar94] S. Darbha and D. P. Agrawal, "SDBS: A Task Duplication Based Optimal Scheduling Algorithm," Proceedings of the Scalable High Performance Computing Conference, May 23-25 1994, pp. 756-763.

[Dar96] S. Darbha and D. P. Agrawal, "Scalable Scheduling Algorithm for Distributed Memory Machines," Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing, October 23-26 1996, pp. 84-91.

[Khe97] A. Khemka and R. K. Shyamasundar, "An Optimal Multiprocessor Real-Time Scheduling Algorithm," Journal of Parallel and Distributed Computing, vol. 43, 1997, pp. 37-45.

[Lin96] K. Lin and C. Peng, "Scheduling Algorithms for Real-Time Agent Systems," Proceedings of the IEEE International Workshop on Research Issues in Data Engineering (RIDE), February 26-27 1996, pp. 32-41.

[Liu73] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," Journal of the ACM, vol. 20, no. 1, January 1973, pp. 46-61.

[Man98] G. Manimaran and C. Siva Ram Murthy, "An Efficient Dynamic Scheduling Algorithm For Multiprocessor Real-Time Systems," IEE Transactions on Parallel and Distributed Systems, vol. 9, no. 3, March 1998, pp. 312-319.

[Mun70] R. R. Muntz and E. G. Coffman, Jr., "Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems," Journal of the ACM, vol. 17, no. 2, April 1970, pp.324-338.

[Ram90] K. Ramamrithan, J. A. Stankovic, and P. Shiah, "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems," IEEE Transactions on Parallel and Distributed Systems, vol. 1, no. 2, April 1990, pp. 184-194.

[Sun96] J. Sun and J. Liu, "Synchronization Protocols in Distributed Real-Time Systems," Proceedings of the 16th International Conference on Distributed Computing Systems, May 27-30 1996, pp. 38-45.

[Wan98] H. Wang and D. Guozhong, "Multi-Node Scheduling for Distributed Real-Time Systems," Proceedings of the IEEE International Conference on Intelligent Processing Systems, ICIPS, Part 2 (of 2), October 28-31 1997, pp. 1356-1360.

[Wu97] M. Wu, "On Runtime Parallel Scheduling for Processor Load Balancing," IEEE Transactions on Parallel and Distributed Systems, vol. 8, no. 2, February 1997, pp. 173-186.