A New Strategy for Improving the Effectiveness of Resource Reclaiming Algorithms in Multiprocessor Real-Time Systems

Indranil Gupta G. Manimaran C. Siva Ram Murthy*
Department of Computer Science and Engineering
Indian Institute of Technology
Madras 600 036, INDIA
Email: murthy@iitm.ernet.in

Abstract

The scheduling of tasks in multiprocessor real-time systems has attracted many researchers in the recent past. Tasks in these systems have deadlines to be met, and most of the real-time scheduling algorithms use worst case computation times to schedule these tasks. Many resources will be left unused if the tasks are dispatched purely based on the schedule produced by these scheduling algorithms, since most of the tasks will take lesser time to execute than their respective worst case computation times. Resource reclaiming refers to the problem of reclaiming the resources left unused by a real-time task when it takes lesser time to execute than its worst case computation time. Several resource reclaiming algorithms such as Basic, Early Start and RV algorithms have been proposed in the recent past. But very little attention is paid in these to the strategy by which the scheduler can better utilize the benefits of reclaimed resources. In this paper, we propose an estimation strategy which can be used along with a particular class of resource reclaiming algorithms (like Early Start and RV algorithms) by which the scheduler can estimate the minimum time by which any scheduled but unexecuted task will start or finish early, based solely on the start and finish times of tasks that have started or finished execution. We then propose an approach by which dynamic scheduling strategies, which append or reschedule new tasks into the schedules, can use this estimation strategy to achieve better schedulability. Extensive simulation studies are carried out to investigate the effectiveness of this estimation strategy versus its cost.

1 Introduction

Their capability for high performance and reliability is seeing multiprocessors emerging as a powerful computing tool for safety-critical real-time applications such as nuclear plant control and avionic control [1]. The problem of multiprocessor scheduling, which is determining when and where a given task executes, has attracted considerable research in the past [1-11]. Two classes of scheduling algorithms have emerged - static and dynamic. In static algorithms, the assignment of tasks to processors and the time at which the tasks start execution are determined *a priori* [7, 8]. Static algorithms are often used to schedule periodic tasks with hard deadlines which are known a priori. The advantage is that if a solution is found, one can be sure that all deadlines will be guaranteed.

*Author for Correspondence.

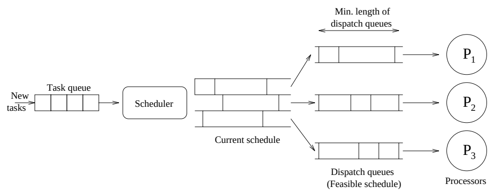

Figure 1: System Model

However, this approach is not applicable to aperiodic tasks whose arrival times and deadlines are not known a priori. Scheduling such tasks in a multiprocessor real-time system requires dynamic scheduling algorithms. In dynamic scheduling, when new tasks arrive, the scheduler dynamically determines the feasibility of scheduling these new tasks without jeopardizing the guarantees that have been provided for the previously scheduled tasks [9-11].

In general, for predictable execution which is essential in a real-time system, schedulability analysis must be done before tasks' start execution. For schedulability analysis, tasks' worst case computation times must be taken into account. A feasible schedule is generated if the timing, precedence, and resource constraints of all the tasks can be satisfied, i.e., if the schedulability analysis is successful. Tasks are dispatched according to this feasible schedule.

Dynamic scheduling algorithms can be either distributed or centralized. In a distributed dynamic scheduling scheme, tasks arrive independently at each processor. The local scheduler at the processor determines whether or not it can satisfy the constraints of the incoming task. If so, the task is accepted, otherwise the local scheduler tries to find another processor to accept the task. In a centralized scheme, all the tasks arrive at a central processor called the *scheduler*, from where they are distributed to other processors in the system for execution. In this paper, we will assume a centralized scheduling scheme. The communication between the scheduler and the processors is through *dispatch queues* (DQs). Each processor has its own dispatch queue. This organization, shown in Fig 1, ensures that the processors always find some tasks in the dispatch queues when they finish the execution of their current tasks. The scheduler runs in parallel with the processors, scheduling the newly arriving tasks, and periodically updating the dispatch queues. The scheduler has to ensure that the dispatch queues are always filled to their minimum capacity (if there are tasks left with it) for this parallel operation. This minimum capacity depends on the worst case time required by the scheduler to reschedule its tasks upon the arrival of a new task [12, 13]. The schedule constructed by the scheduler is assumed to be stored in a set of schedule queues (or SQs, one queue per processor), presumably in the scheduler's memory itself. The dispatch queues are updated by the scheduler from these schedule queues just before the invocation of the scheduling algorithm or when they become empty so that the processors can execute tasks in parallel with the scheduler's running.

The worst case computation times, deadlines, and the constraints of the tasks are taken into account by the scheduler to arrive at a *feasible schedule* [14]. However, at run time, the actual time taken by a task to execute may be smaller than its worst case computation time because of data-dependent loops and conditional statements. Moreover, task deletion could take place when extra tasks are initially scheduled to account for fault tolerance. When no faults occur, there is no necessity for these temporally redundant tasks to be executed and hence they can be deleted. Hence, executing the tasks strictly based on the starting times specified in the feasible schedule leads to a lot of resources remaining unused. *Resource reclaiming* is required to utilize the resources[1] left unused by a task when it executes less than its worst case computation time, or when a task is deleted from the current schedule.

1.1 Motivation and Objectives

Any resource reclaiming algorithm should possess the properties of correctness, bounded complexity, and effectiveness [13]. The *correctness* condition ensures that there are no *run-time anomalies*. Run-time anomalies are said to occur when a task that is scheduled to meet its deadline in the feasible schedule misses the deadline during actual execution. The *bounded complexity* requirement states that the cost of resource reclaiming should be independent of the number of tasks in the schedule, so that it can be incorporated into the worst case computation times of the tasks. *Effectiveness* of a resource reclaiming algorithm aims at improving the *guarantee ratio*, which is defined as the ratio of the number of tasks guaranteed to the number of tasks arrived. The larger the amount of resource reclaimed by the reclaiming algorithm, the better will be the guarantee ratio (or schedulability of the system).

The earliest work [13] considered resource reclaiming in multiprocessor real-time systems with independent tasks having resource constraints. Two algorithms, *Basic reclaiming* and *Early Start* were proposed. [12] presents the RV algorithm which extends the task model presented in [13] to include precedence constraints among tasks. The resource reclaiming algorithm is run at the end of a task's execution. All these algorithms satisfy the bounded complexity requirement. It has also been shown that the RV algorithm reclaims more resources than the Early Start and Basic reclaiming algorithms, in that order.

To realize the full use of reclaiming in increasing the schedulability in a dynamic system, the processors should report the reclaimed resource time to the scheduler. The scheduler should be able to utilize this time efficiently in achieving a higher schedulability. One approach is to use the *holes* (idle processor time intervals in the schedule) created due to reclaiming by immediately scheduling new tasks into them. However, this involves considerable processor-scheduler communication overhead and it may not always be possible to find tasks to fill these holes. Another approach reported in [13] makes sure that the resource time reclaimed simultaneously on all the processors is used by the scheduler. However, such an occurrence is rare. Also, we can intuitively say that this approach is a rather pessimistic way of estimation. The paper on the RV algorithm [12] makes no mention about how processors running the RV algorithm can report reclaimed time to the scheduler. Thus the gain from the reclaiming algorithms is mostly realized only when the last scheduled task (of a task set) on a processor actually finishes execution, and the time reclaimed so far is reported to the scheduler.

In this paper, we present a simple approach to *estimate* the minimum time by which unexecuted

[1]resource refers to processor and other system resources.

events i.e., task start and finish, in a schedule will occur earlier than their scheduled times based on the time reclaimed on different processors in the executed schedule until that time. This strategy will be seen to be superior to that in [13]. This strategy can be used with either the Early Start or RV algorithms. Through this strategy, the scheduler will be able to predict the early finish of tasks much earlier even before these tasks have been started. The second objective of this paper is to develop an algorithmic approach by which a class of dynamic scheduling algorithms namely, those which append their schedules to already guaranteed schedules, like the Spring scheduling algorithm, can utilize this estimation to achieve better schedulability. Of course, the prime considerations to be kept in mind while developing any such strategy would be to not violate any of the four properties of resource reclaiming algorithms and to see whether the cost of the new addition is worth the increase in schedulability.

The rest of the paper is organized as follows. In the next section, we describe the background: first the task model we are using, then (informally) the three existing resource reclaiming algorithms. In section 3, we present our estimation technique and in section 4 the proposed scheduling strategy. The simulation results and their analysis appear in section 5. Finally, we summarize our work in section 6.

2 Background

In this section, we first describe the task model we will be using. Then we present an informal description of the three existing resource reclaiming algorithms.

2.1 Task Model

In the rest of the paper, we shall be using the following task model.

1. Tasks are aperiodic, i.e., task arrival times are not known *a priori*. Every task T_i has the attributes: *arrival time (a_i)*, *worst case computation time (c_i)*, and *deadline (d_i)*. The ready time of a task is equal to its arrival time.

2. The actual computation time of a task is denoted as $\bar{c}_i$. This is the actual time taken by the task to execute at run-time. The difference ($c_i - \bar{c}_i$) is due to the execution of data dependent loops, conditional statements in the task code and architectural features of the system.

3. Tasks may also have resource constraints. The foremost example of a resource that a task might require is a processor. A task might need some other resources such as data structures, variables, and communication buffers for its execution. Every task can have two types of accesses to a resource: (a) exclusive access, in which case, no other task can use the resource with it or (b) shared access, in which case, it can share the resource with another task (The other task also should be willing to share the resource). We say that a resource conflict exists between two tasks T_i and T_j if one of these tasks cannot share the resources it requires, with the other. This resource conflict between T_i and T_j is denoted as $T_i \odot T_j$. If the system has r resources, the resources required by a given task is an r-tuple $((R_1, m_1) \ldots (R_r, m_r))$, where each m_i $(1 \leq i \leq r)$ is one of *EXCLUSIVE*, *SHARED*, and *NOT-REQUIRED*.

4. Tasks may have precedence constraints. If there is a precedence relation from task T_j to task T_i, then T_j has to finish its execution before the beginning of T_i. We denote this precedence relation from T_j to T_i as $T_j \prec T_i$.

5. Tasks are non-preemptable.

6. The multiprocessor system has $(m+1)$ processors of which one processor performs the scheduling and the remaining processors execute the tasks. Each processor runs the resource reclaiming algorithm at the completion of a task on it. The processors communicate through shared memory.

2.2 Existing Work on Resource Reclaiming

Here we give a brief description of the three algorithms namely Basic, Early Start and the RV algorithm which are the best known algorithms for the resource reclaiming problem.

2.2.1 Basic Reclaiming Algorithm

The idea behind the Basic reclaiming algorithm is to see at the end of each task's execution whether *all* the processors are idle. If so, the entire rest of the schedule can be moved forward by $min_{(i=1 \ldots m)}\{$ *Scheduled start time of next task on processor* p_i $\} - [CurrentTime]$ and started early by this value. This approach has a complexity of $O(m)$ per task finish.

2.2.2 Early Start Algorithm

In this algorithm, whenever a task finishes execution, it checks the dispatch queues of all idle processors. If the next task T_{pi} scheduled on a currently idle processor p_i is such that *all* tasks scheduled to finish before T_{pi} have already finished execution by now ($t = CurrentTime$), T_{pi} is started immediately. The motivation here is to avoid *passing*. *Passing* is said to occur in a system when a task T_i starts execution before another task T_j that is scheduled to finish execution before T_i was originally scheduled to start. It is clear that no passing would mean that no resource/precedence constraints among the tasks are violated. In other words, the Early Start algorithm avoids passing and ensures that no two non-overlapping tasks in the pre-run schedule overlap in the post-run schedule [13]. For example, Fig 2a shows a pre-run schedule for a set of tasks and Fig 2b the post-run schedule for the same with tasks executing for a time less than the worst case scheduled time. Fig 2c shows the post-run schedule when the Early Start algorithm is used on the pre-run schedule of Fig 2a. The Early Start approach has a complexity of $O(m^2)$ per task finish. Although the Basic approach has a complexity of $O(m)$ only, Early Start reclaims more resources than the Basic approach [13].

2.2.3 Restriction Vector Algorithm

Here, the scheduler builds a restriction vector (RV) for each task T_j that it schedules. The RV is an m-component vector (m being the number of processors), where each entry $RV_j(p_i)$ is the *last* task scheduled prior to T_j on processor p_i which has a resource conflict or precedence relation with T_j. When a task finishes execution, the processor runs the RV algorithm. It checks the first task in the DQs (dispatch queues) of all idle processors and starts that task immediately if all the tasks in its RV have finished execution. The RV algorithm thus is a more refined version of the Early

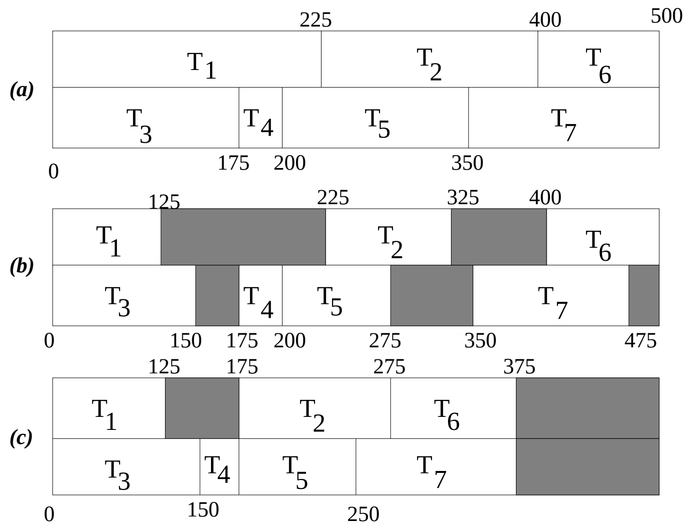

Figure 2: (a) Pre-run schedule (b) Post-run schedule without resource reclaiming and (c) Post-run schedule with Early Start algorithm, for a task set

Start Algorithm as it allows a task T_j to *pass* a task T_k as long as T_k has no resource/precedence conflicts with T_j. [12] presents a comparison by simulation of the three algorithms for various system parameters. Note that the Early Start Algorithm is a special case of the RV algorithm when the scheduler uses a look-back window of size one while constructing the RVs. The RV algorithm has a complexity of $O(m^2)$ per task finish, which is the same as that of early Start. However, RV is a more general case of the Early Start approach and therefore reclaims more resources than the Early Start approach [12].

3 Proposed Estimation in Resource Reclaiming

In this section, we first present an abstracted view of the Early Start and RV algorithms by setting forth two properties which are common to the way both algorithms work. Following this, we present our estimation strategy which will use these two properties.

3.1 Properties of the Early Start and RV Algorithms

Property RR1: We assume that the resource reclaiming algorithm runs as follows. (a) For every scheduled task T_i, there exists an array $PT_i[1 \ldots m]$ where $PT_i[j]$ stands for the last task scheduled on processor p_j that is scheduled to finish execution before T_i can start execution and also has a resource/precedence conflict with T_i. The aim of the PT arrays is to capture the resource and precedence constraints among tasks as they are scheduled. For example, for task T_2 in figure 2(a), if T_4 has a resource or precedence conflict with it, then $PT_4[2] = T_4$, otherwise $PT_4[2] = T_3$. (b) When a task finishes execution on a processor, the processor checks every idle processor p_j and decides whether the next task T_i scheduled on p_j can be executed by checking the execution status of tasks in $PT_i[1 \ldots m]$. If all tasks in $PT_i[1 \ldots m]$ have finished execution (other than this task), T_i is started immediately.

Note that this implies that the only way a task can start execution is when some other task finishes execution and runs the resource reclaiming algorithm. The Early Start and RV algorithms satisfy the property $RR1$, but the Basic Algorithm does not. In the Early Start algorithm, $PT_i[j]$ is the immediately preceding non-overlapping task on p_i that is scheduled to finish execution before T_i's start time. In the RV algorithm, $PT_i[1 \ldots m]$ is nothing but the RV of T_i. These PT arrays are constructed in such a way that a resource reclaiming algorithm running according to Property RR1 will not lead to any resource/precedence conflicts and will thus avoid run-time anomalies [12, 13]. In both Early Start[2] and RV algorithms, the PT arrays are constructed by the scheduler. Note that following property RR1 ensures avoidance of runtime anomalies and thus resource/precedence conflicts.

Property RR2: The finishing of a task T_i, and the starting of the tasks chosen by the resource reclaiming algorithm run at the end of T_i, occur simultaneously. That is, the time taken by the resource reclaiming algorithm run by the processor at every task T_i's end is taken to be a part of T_i's computation time itself.

3.2 The Estimation Strategy

Definition 1: A *schedule* is a set of tasks scheduled in one invocation of the scheduler and put into the schedule queues at once. For example, Fig 2a is one schedule.

Consider a pre-run schedule, as in Fig 2a, created by the scheduler. This is executed by the processors with some resource reclaiming algorithm to give a post-run schedule (Fig 2c is the post-run schedule with Early Start algorithm). The aim of the estimation strategy in a dynamic scheduling environment is to inform or report to the scheduler, at any time, the minimum amount by which the rest of the scheduled but unexecuted tasks in a schedule will start/finish early [13]. This is achieved by maintaining a variable *Reclaim_del* which denotes the above said minimum (predicted) time. The scheduler takes this variable into account while scheduling newly arriving task sets. Thus the scheduler is more likely to accept tasks with tighter deadlines as the scheduler assumes that already scheduled tasks are guaranteed to finish at least *Reclaim_del* earlier than they are scheduled to. The problem here is the strategy involved in updating *Reclaim_del* so that its definition holds true, i.e., the rest of the tasks in the schedule do finish early by at least *Reclaim_del*.

One way to do this, as described in [13], is for the processors to set *Reclaim_del*, when all the processors become idle, to the minimum amount by which the next scheduled tasks will start early. Let us call this the *simultaneous approach.* When implemented with the Basic Reclaiming algorithm, this strategy accurately predicts the amount, at any time, by which the rest of the scheduled but unexecuted tasks will start/finish early. This is because the Basic algorithm moves the whole schedule forward by this minimum amount when all processors are idle. However, when this strategy is used with Early Start, *Reclaim_del* does not accumulate a large value as quickly as the Basic algorithm does in a short time. But in the long run it does reach a larger value than the Basic algorithm as Early Start is a better reclaiming algorithm [13]. This is illustrated in Table 1 which shows the values of *Reclaim_del* when the pre-run schedule of Fig 2a is run according to the Basic and Early Start algorithms (second and third rows of table respectively). Note that the Early Start algorithm ultimately accumulates a greater value of *Reclaim_del* than Basic algorithm but accumulates it more towards the end. On the other hand, the Basic algorithm may finally

[2]Early Start algorithm can also be implemented without using PT arrays.

Time	0	125	150	175	250	275	300	375
Simultaneous time reclaimed on all processors-Basic algorithm as in [13]	0	0	25	25	25	25	50	50
Simultaneous time reclaimed on all processors-Early Start algorithm as in [13]	0	0	25	25	25	25	25	125
$M(t) = min_{i=1}^{m}\{$[Scheduled start/finish time of last seen task on p_i at time t - its actual start/finish time]$\}$	0	0	25	25	50	125	125	125

Table 1: The values of *Reclaim_del* at each task completion using Basic algorithm and Early Start with and without estimation for the example in Fig 2.

accumulate a lower *Reclaim_del* value but it accumulates it throughout the schedule's execution. In our estimation strategy, which follows, we propose a way of maintaining the value of *Reclaim_del* so that the prediction is much faster than when the simultaneous approach of [13] is used.

Consider a variable $M(t) = min_{i=1}^{m}\{$[Scheduled start/finish time of last seen task on processor p_i at time t - its actual start/finish time]$\}$. This variable is maintained as follows. An array $M\_arr[1..m]$ is used (in global memory). When a task T_j starts or finishes on a processor p_i at time t the processor does two things. First it updates $M\_arr[i]$ (by p_i) to the value of [Scheduled start/finish time of T_j - actual start/finish time of T_j]. p_i then finds the minimum of $\{M\_arr[i] \mid 1 \leq i \leq m\}$ and sets $M(t)$ to this value.

Let us examine the values of $M(t)$ at different times during the execution of the pre-run schedule in Fig 2a using Early Start algorithm. We observe that $M(t)$ (fourth row of Table 1) increases faster than the *Reclaim_del* of the simultaneous approach (third row of Table 1) although both finally reach the same value.

Suppose we could prove formally that after time t, the rest of the scheduled but unexecuted tasks in the schedule will finish early by at least $M(t)$ (we will do this in Theorem 1). Then, using $M(t)$ as *Reclaim_del* would be a better prediction strategy than the simultaneous approach as $M(t)$ would increase much faster. Thus, in a way, we will be incorporating the quickness of predictability that the simultaneous approach shows when used with the Basic algorithm, into the Early Start algorithm. Note that $M(t)$ is equal to *Reclaim_del* at all times when the Basic algorithm is used i.e., this strategy boils down to the simultaneous approach when it is used with the Basic algorithm. In the rest of the paper, we will be dealing with the application of our estimation strategy to the Early Start and RV algorithms only, using properties RR1 and RR2. Now we formally prove a lemma needed for Theorem 1.

Definition 2: st_i and ft_i denote the start and finish times of the task T_i in the feasible schedule S, whereas, st'_i and ft'_i denote the actual start and finish times of the task T_i when it executes, as depicted in the post-run schedule S'. $proc(T_i)$ denotes the processor on which task T_i has been scheduled.

Lemma 1.1: The actual start time of any task T_j, $st'_j = max\{ft'_l \mid T_l \in PT_j[1 \ldots m]\}$.

Proof: By the definition of PT and property *RR1*, it is clear that $st'_j \geq max\{ft'_l \mid T_l \in PT_j[1 \ldots m]\}$. Let T_k be the last task to finish execution among all the tasks in $PT_j[1 \ldots m]$. The resource reclaiming algorithm run at the end of T_k's execution will find $proc(T_j)$ to be idle. This is because

$PT_j[proc(T_j)]$, which is actually the task scheduled on $proc(T_j)$ just before T_j, has finished execution. Also, all other tasks in $PT_j[1 \dots m]$ would have finished execution by ft'_k. Hence by properties *RR1* and *RR2*, T_k will start T_j immediately. Thus $st'_j = ft'_k = max\{ft'_l \mid T_l \in PT_j[1 \dots m]\}$. △

Definition 3: Let S be the schedule of a set of tasks. The set of m processors is said to have *totally started (executing)* the schedule S if $\forall i, 1 \le i \le m$, p_i has started (perhaps finished) the execution of at least one task belonging to S. When at least one of the processors p_i finishes executing all the tasks of S scheduled on p_i, S is said to have *ended.* When a schedule S has been started by the system and has not yet ended, we say that it is being executed by the system.

Theorem 1: Consider a set of tasks T scheduled onto m processors as S in such a way that one or more task is scheduled on each processor, and a resource reclaiming algorithm which guarantees correctness that is, no run-time anomalies, and satisfies assumptions *RR1* and *RR2*. Let $M_S(t) = min_{i=1}^{m}\{$[Scheduled start/finish time of last seen task on p_i at time t - its actual start/finish time]$\}$ (note that $M_S(t) = 0$ until schedule S starts). If the schedule has *totally started* (definition 2), the following holds:
(a) $M_S(t)$ is monotonically non-decreasing with t, and
(b) at any time t after the start of schedule S, all unexecuted events of S will occur at least $M_S(t)$ earlier than scheduled as in S.
Proof : Note that $M_S(t)$ can be updated only at the beginning or end of a task's execution. As we are considering a system with resource reclaiming obeying *RR1* and *RR2*, the only event which changes the value of $M_S(t)$ during the execution of schedule S is the finishing of a task T_i, its running the resource reclaiming algorithm leading to the (early) start of tasks on some of the idle processors. These events occur simultaneously by property *RR2*.

As $M_S(t) = 0$ until S starts, for $t =$ minimum start time of a task of S, (b) is true.

Let the value of $M_S(t)$ at some time t during the execution of schedule S be M_1. This means that for processors which are not idle, the tasks currently in execution on them have started at least M_1 earlier than their scheduled start time. These tasks will thus finish at least M_1 earlier than their scheduled finish time because $ft_j - ft'_j \ge st_j - st'_j$. Thus, from time t, all events on currently executing tasks will occur at least M_1 early.

No task will start execution until one of the currently executing tasks finishes (consequence of Property $RR1$). Consider the first task T_k, among the tasks being executed at time t , that might finish its execution at a later time. As $ft_k - ft'_k \ge st_k - st'_k$, the update of $M_S(t)$ at the end of T_k can only change its value to some $M_2 \ge M_1$. Also, T_k runs the resource reclaiming algorithm and chooses a set of tasks from idle processors at that time, for starting immediately. For any such task T_i chosen to start by T_k, $st'_i = ft'_k$ and $st_i \ge ft_k \Rightarrow st_i - st'_i \ge M_2$. Thus any such task T_i updates $M_S(t)$ to a value $M_3 \ge M_2$.

Thus brings us again to a state in the system where all currently executing tasks have started at least M_1 early (actually M_3 early, but anyway $M_3 \ge M_1$) and will thus have all events in them occurring at least M_1 early. As all tasks in S are anyway going to be chosen to start sometime in the future, we can use the same above argument recursively (or inductively) on all un-started tasks at time t to say that all events in the system occurring at a time after t will occur at least as early as $M_S(t) = M_1$. Thus (b) is proved.

Also, as the only event that affects $M_S(t)$ can only increase it to a higher value (as we have shown above: $M_3 \geq M_2 \geq M_1$), $M_S(t)$ is monotonically non-decreasing. Thus (a) is proved. $\square$

4 Proposed Scheduling Strategy

Now we seek to answer the question: how a dynamic scheduling algorithm can utilize the reclaimed time($M_S(t)$). To see this, we have to examine how dynamic scheduling algorithms work. There are three broad classes of dynamic scheduling algorithms depending on the manner in which they schedule newly arriving tasks into the system while not jeopardizing the guarantees of already scheduled tasks :

1. Algorithms which attempt to fill in holes in the schedule with newly arrived tasks. These holes might be created due to resource reclaiming.

2. Algorithms which do not fill in holes but try to schedule the new tasks so that they can be appended to the end of current schedule. The algorithm might

   (a) *Append:* Construct a separate schedule for the newly arrived tasks and then append the schedule to the schedule queues.

   (b) *Reschedule:* Remove some of the tasks from the schedule end, which have not yet started execution, reschedule them with the new tasks and then append the schedule to the schedule queues as in Spring scheduling algorithm [13, 14]

We shall first look at how algorithms of type 2 can be made to utilize $M_S(t)$ to achieve better schedulability.

Broadly, algorithms of type 2 work as follows. When a new set of tasks are to be scheduled, the scheduler might decide to reschedule some of the already scheduled tasks . It removes some of the last scheduled-to-finish tasks from the schedule queues (SQs) - by this, we mean that the scheduler comes from the end of the schedule in the SQs, removing tasks T_i (which have not yet started execution) for rescheduling according to decreasing st_i, until some criteria are satisfied (such as, maximum number of tasks that can be rescheduled, etc.). The idea behind this is to schedule the newly arrived tasks earlier if they have tight laxities (and hence deadlines). Also, the scheduler, being in a real-time system, will have to decide the latest (worst case) time at which it can append the new schedule to the SQs. At that time (or whenever the scheduler arrives at final schedule with possible rejection of tasks) the scheduler appends the new schedule to SQs. If it has failed to schedule any of the new tasks, it leaves the old schedule intact in the SQs. To guarantee that these tasks will still meet their deadline in the event of a such a failure, the obvious condition that the scheduler will have to satisfy is:

Assumption Sch1: The scheduled start time of any of the tasks the scheduler chooses to reschedule should be later than the worst case time for the scheduler to append the new schedule to the SQs.

Now, with a schedule S in the SQs, how does the scheduler construct a schedule S' for a new set of arriving tasks ? Once some tasks have been removed from the schedule S in the SQs for rescheduling, the scheduler calculates $ProcAvTime_{S'}[1 \ldots m]$, where $ProcAvTime_{S'}[i]$ stands for the scheduled finish time of the last task on processor p_i not chosen for rescheduling (the subscript

S' stands for the fact that this information is going to be used in scheduling S'). Also the scheduler needs to see the part of the schedule on all processors for time $t \geq min(ProcAvTime_{S'}[1 \ldots m])$. We shall call this the *end-jack* of the schedule in the SQs. The scheduler can now use these two parameters to schedule the tasks using an appropriate scheduling heuristic. Some examples of heuristics are the least-laxity first heuristic and earliest-deadline first heuristic [14].

Definition 4: For a set of tasks T, $SchGetTime(S)$ and $SchPutTime(S)$ refer to the start and end time of the scheduling algorithm which constructs the schedule S for T. The schedule S is thus appended into the SQs at time $t = SchPutTime(S)$.

Assume that the set of m processors is executing a schedule S. Now, when a new set of tasks (includes newly arrived tasks and those that have been chosen for rescheduling) are considered for scheduling, the scheduler starts an invocation to construct a new schedule S'. The scheduler can utilize Theorem 1 by assuming that any unexecuted event (here the only events we are interested in are task start and end times) in a schedule S supposed to occur at time t' will occur at least $M_S(SchGetTime(S'))$ earlier. Thus, for example, it can safely assume that the end-jack of the schedule S will move forward by at least $M_S(SchGetTime(S'))$. Thus it will be able to schedule the new set of tasks that much early and the chances of scheduling the new tasks are more because of this extra reclaimed time. The PT arrays for tasks in S' are also constructed by considering this offsetting.

Fig 3 shows an example of this. The first schedule S (tasks 1 to 6) is put into the SQs at time $t = 0$ (fig 3a). At $t = 11$, $M_S(t = 11) = 2$ and at this time, a new set of tasks (a-f) arrives (fig 3b). The scheduler predicts that the last tasks in S , 3 and 6, will both finish early by at least 2 time units. It thus considers the end-jack to be displaced by $M_S(t = 11) = 2$ time units (right bottom corner of fig b) and uses this in scheduling the new set of tasks. It is able to utilize the value of $M_S(t = 11) = 2$ to schedule the new tasks earlier. If for example, the deadline of task a had been 40, then not using the value of $M_S(t = 11)$ would have led to its rejection by the scheduler whereas using it has resulted in its successful scheduling. The final schedule is added to the SQs at $t = 19$ (fig 3c).

This means that some of the tasks in S' may be *scheduled to start earlier* than the *scheduled finish times* of tasks in S(pre-run schedule) with which they have resource/precedence constraints. There is thus an *overlapping* of the two schedules. The overlapping of schedules S and S' should not lead to run-time anomalies when S' starts execution. In the following theorem, we prove the important result that if the PT relations are followed by the resource reclaiming algorithm (i.e., Property $RR1$), no run-time anomalies occur during the run-time transition from schedule S to S'. Further, we prove that the estimation strategy can be extended to tasks of S' also, i.e, tasks of schedules other than the currently being executed schedule can also be guaranteed to start early by a value determined from $M_S(t)$ and some other parameters.

Assumption Sch2: For any schedule S', at time $t = SchPutTime(S')$, the scheduler executes the resource reclaiming algorithm.

The purpose behind this is that any task T_j in S' for which all tasks in $PT_j[1 \ldots m]$ are in S and have finished execution by $SchPutTime(S')$, can be started immediately at $t = SchPutTime(S')$. This will add only an $O(m)$ term to the scheduler's execution time.

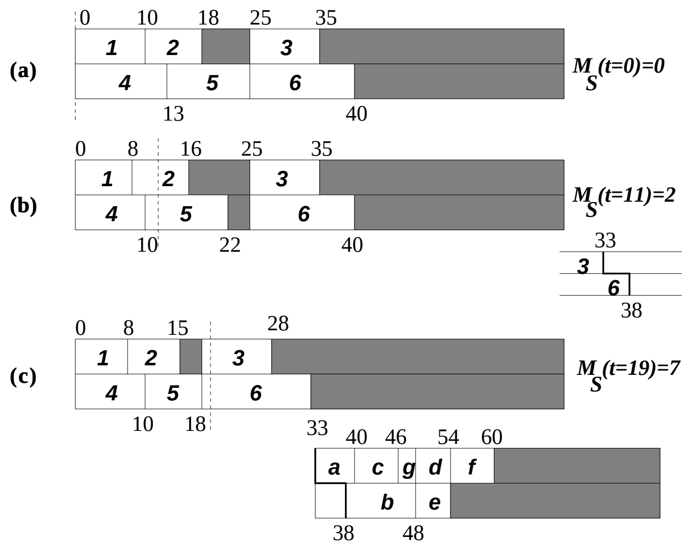

Figure 3: Example

Theorem 2: Consider a schedule S_i currently under execution by the system. We denote $M_{S_i}(t)$ as $M_i(t)$, $SchGetTime(S_{i+1})$ as $SchGetTime(i+1)$ and $ProcAvTime_{S_{i+1}}[j]$ as $ProcAvTime_{i+1}[j]$. At $t = SchGetTime(i+1)$, let M_{old} be the value of $M_i(t)$. The scheduler uses this value of M_{old} in scheduling S_{i+1} and in constructing the PT arrays for tasks in S_{i+1}. If we denote the start and end times of a schedule S as $ST(S)$ and $ET(S)$, respectively and $M_S(ET(S)) = M'$, ($\geq M_{old}$ by Theorem 1), then all events in S_{i+1} will start early by at least
(a) $M' - M_{old}$ if $SchPutTime(i+1) \leq max(ProcAvTime_{i+1}[1 \ldots m]) - (M' - M_{old})$
(b) $max(ProcAvTime_{i+1}[1 \ldots m]) - SchPutTime(i+1)$ if
$SchPutTime(i+1) > min(ProcAvTime_{i+1}[1 \ldots m]) - (M' - M_{old})$

Further, all precedence/resource constraints among the tasks are obeyed and no run-time anomalies occur.

Proof: We first make the following observations.

1. No tasks of S_{i+1} can start execution on any processor until schedule S_i ends - this means that just before the first task of S_{i+1} starts execution, we can state that all tasks of S_i will finish at least M' earlier than their scheduled finish times (by Theorem 1).

2. For any task T_i in S_{i+1}, $st'_i = max\{SchPutTime(i+1), max\{ft'_j \mid T_j \in PT_j[1 \ldots m]\}\}$.

3. There is at least one task $T_j \in S_{i+1}$ such that $PT_j[1 \ldots m] \cap S_{i+1} = \phi$; this is the task with the least scheduled start time in S_{i+1}.

4. The resource reclaiming algorithm run by the scheduler at $t = SchPutTime(i+1)$ does not cause any PT relations to be violated.

(a) The first set of tasks chosen (by the resource reclaiming algorithm run at the end of a task of S_i) to start from S_{i+1} will be tasks T_j of the type $\{T_j \mid PT_j[1 \ldots m] \cup S_{i+1} = \phi\}$. As the tasks in $PT_j[1 \ldots m]$ will finish at least M' earlier than their scheduled finish time (Theorem 1) and M_{old} of this M' has been taken into account in the scheduling of T_j, from (1) and (2) the first set of tasks chosen to start from S_{i+1} will start at least $M' - M_{old}$ earlier than their scheduled start time. This

argument can be easily extended to all tasks T_k in S_{i+1} as all tasks in $PT_k[1\dots m]$ will finish at least $M' - M_{old}$ earlier than their scheduled finish times. Hence (a) is proved.

(b) $SchPutTime(i+1) > min(ProcAvTime_{i+1}[1\dots m]) - (M' - M_{old})$ implies that some task(s) of S_{i+1} may start before $SchPutTime(i+1)$ (consider a task with start time scheduled as $min(ProcAvTime_{i+1})$), which is not possible. The difference from (a) is that the schedule S_{i+1} is put into the SQs later than the predicted start time of some of the tasks in it.

Notice first that $SchPutTime(i+1) > min(ProcAvTime_{i+1}[1\dots m]) - (M' - M_{old})$ means that S_i has ended before $t = SchPutTime(i+1)$. For all $T_j \in S_{i+1}, st_j \geq min(ProcAvTime_{i+1}[1\dots m]$. If there is any task $T_j \in S_{i+1}$ such that $PT_j[1\dots m] \cup S_{i+1} = \phi$ and all tasks in $PT_j[1\dots m]$ have finished execution by $t = SchPutTime(i+1)$, the resource reclaiming algorithm will start such a task at $t = SchPutTime(i+1)$. As $st_j \geq min(ProcAvTime_{i+1}[1\dots m])$, T_j will start at least $min(ProcAvTime_{i+1}[1\dots m]) - SchPutTime(i+1)$ earlier than st_j(i.e., $st_j - st'_j \leq min(ProcAvTime_{i+1}[1\dots m]) - SchPutTime(i+1)$.

Extending this argument to all tasks in S_{i+1} (as all tasks in S_i will finish at least M' earlier than their scheduled start time and $M' - M_{old} < min(ProcAvTime_{i+1}[1\dots m]) - SchPutTime(i+1)$), we can say that all events in S_{i+1} will occur at least $min(ProcAvTime_{i+1}[1\dots m]) - SchPutTime(i+1)$ earlier than their scheduled times.

If the resource reclaiming algorithm run by the scheduler at $t = SchPutTime(i+1)$ does not start any task of S_{i+1}, then applying the same argument as (a), we can say that all tasks in S_{i+1} will start at least $M' - M_{old}$ earlier than their scheduled start times. As $M' - M_{old} > min(ProcAvTime_{i+1}[1\dots m]) - SchPutTime(i+1)$, all events in S_{i+1} will occur at least $min(ProcAvTime_{i+1}[1\dots m]) - SchPutTime(i+1)$ earlier than their scheduled times.

Note that as we have maintained the correctness of the PT relations constructed by the scheduler, all tasks execute correctly i.e., no precedence/resource constraints are violated. □

We thus see that the estimation can be extended to two schedules in the SQs without jeopardizing the PT relations. Consider again the example of Fig 3. At any time $t \leq SchGetTime(S') = 11$, from Theorem 1, we can say that all unexecuted events in the SQs will occur at least $M_S(t)$ earlier than their scheduled times. At any time $t > 11$ we can say from Theorem 1 that all unexecuted events of S are guaranteed to occur at least $M_S(t)$ early, and from Theorem 2 that all unexecuted events of S' are guaranteed to occur early by at least

$$min(M_S(ET(S)) - M_S(SchGetTime(S')), max(ProcAvTime_S[1..m] - SchPutTime(S'))$$

Thus, a dynamic scheduling algorithm using the append/reschedule approach can utilize the 'minimum' heuristic (Theorem 1) to schedule a set of tasks earlier.

However, in the general case, we may have the SQs of the system containing a series of task schedules $S_p..S_M$ (S_p being the one currently under execution). The strategy presented so far has to be extended to take care of complications such as a number of schedules being totally removed for rescheduling along with a newly arriving set if tasks and predicting the early finish of tasks in all schedules $S_p..S_M$. The approach used is similar to the one suggested in [13]. We present a simplified version of the strategy here. A variable $M(t)$ is maintained, denoting the minimum early start/finish time of unexecuted tasks of the *last* schedule added to the SQs (here S_M). At the *SchPutTime* of the next new task set (S_{M+1}), $M(t)$ has to be updated using an expression similar to equation (1). This is because a part of $M(t)$ has already been taken into accounting the scheduling of S_{M+1}. For example, in Fig 3, the values of $M(t)$ at different times are as follows:$M(t=0)=0; M(t=11)=2; M(t=19)=7-2=5$. Note that at $t=19$, 5 reflects the minimum predicted execution of events of S'. Further, an array $\mathit{OldReclaim\_dels}[MAX]$ is maintained where $\mathit{OldReclaim\_dels}[j]$ is set to the value of $M(\mathit{SchGetTime}(j))$. Then it can be proved that at any time t, we can predict that all unexecuted events in schedule S_j ($p \leq j \leq M$) will occur at least $\sum_{k=j+1}^{M} \mathit{OldReclaim\_dels}[k] + M(t)$ earlier than they are scheduled to. Thus the scheduler can predict the early execution of events in *any* of the schedules $S_p..S_M$.

How can an algorithm that uses resource reclaiming by filling holes in the schedule (i.e., algorithms of type 1) utilize this strategy ? Such algorithms can benefit from Theorem 1 by being able to schedule the new tasks into holes which need not necessarily be *visible* in the schedule. It could schedule new tasks to start at a later time in the schedule by using the estimation. For example, in Fig 2c, if a task arrived T_8 at $t=250$, the scheduler could schedule it on p_2 at $t=250$ or $t=425$ (when T_7 would have finished, according to the estimation), or on p_1 at $t=325$ (when T_2 would have finished or $t=450$ (when T_6 would have finished) or later, depending on the task characteristics and in such a way as to not violate the guarantees of already scheduled tasks.

4.1 Added Complexity of Estimation

At the end of a task T_j's execution on processor p_i, it updates $M\_arr[i]$ (this takes $O(1)$), then sets $M(t)$ to the minimum among $M\_arr[1] \ldots M\_arr[m]$ (this takes $O(m)$ time). Thus the added complexity of the Estimation approach is $O(m)$ per task finish. As Early Start and RV both run in time $O(m^2)$ per task finish, adding Estimation to them does not increase their complexity. Maintaining $\mathit{OldRelcaimdels}[k]$ takes time $O(1)$ per scheduler invocation - this does not add any extra complexity as long as at least one task is scheduled (or rescheduled) per scheduler invocation. The Estimation approach thus does not increase the complexity of the resource reclaiming algorithms.

5 Simulation Studies

To evaluate the effectiveness of the proposed reclaiming estimation on the schedulability of a system already using resource reclaiming, we conducted extensive simulation studies. In these simulation studies, we study the effect of the estimation, when used with a resource reclaiming algorithm (either Early Start or RV), on the schedulability of a dynamic multiprocessor system. In other words, we compared the schedulability of the system when it used just the resource reclaiming algorithm with its schedulability when it used the estimation strategy in addition to the resource reclaiming algorithm. We decided to use the RV algorithm as the Early Start algorithm is a special case of it (as explained in section 2.2.3). As such, the relative trends seen in our simulation results between the RV algorithm with estimation and without estimation, can be expected to be similar to the trends between Early Start algorithm with and without estimation. The dynamic scheduling algorithm was the *Spring scheduling algorithm* [13, 14], which is a rescheduling algorithm, with an integrated heuristic that takes deadline and resource and precedence constraints of a task into account. The simulation parameters used in these studies are shown in Table 2. The values indicated for the parameters in the table are used in all the following graphs unless otherwise stated.

In order to ensure a wide applicability of the experimental results, in our simulation studies, we varied the task and system parameters to a wide range values instead of choosing them based on particular applications and/or architecture(s). Our experiments might have been more meaningful if application-dependent parameters (e.g., task deadline) were not linked to system-dependent parameters (e.g., task computation time). However, such a study would also have the drawback of having limited scope i.e., applicability only to a specific class of applications implemented on a given system architecture.

The tasks were generated using the above parameters as follows.

- The worst case computation time (c_i) of a task (T_i) is chosen uniformly between *MinCompTime* and *MaxCompTime*. The cost due to reclaiming = (*RecCost* $\times$ *NumProcs*) is added to c_i. The cost of estimation, when used, is added as (*RDelCost* $\times$ *NumProcs*) to this.
- The actual execution time ($\bar{c}_i$) of a task at run-time is determined using a multiplicative factor *aw-ratio* (actual to worst case computation ratio) on c_i. *aw-ratio* is chosen uniformly between *min-aw-ratio* and *max-aw-ratio*.
- The deadlines of the tasks are chosen using a *laxity* chosen uniformly between *min-laxity* and *max-laxity*. The deadline of T_i is chosen as $(laxity \times Maxtask \times \frac{MaxCompTime+MinCompTime}{2} + a_i)$
- The resource requirements of T_i are determined by *UseP* and *ShareP*.
- The precedence constraints among the tasks arriving at the scheduler at a time are chosen based on a parameter P which is $= \frac{\text{number of precedence links in the task graph}}{\text{number of tasks in the precedence graph}}$.
- An average of 10 tasks arrive at the scheduler. The average inter-arrival time of these task sets is *TaskFreq* with Poisson distribution with $\lambda = 1/TaskFreq$.
- *NumProcs* is the number of processors. *K*, the window lookahead in the Spring scheduling algorithm is chosen to be 4.

The performance metric used is the *guarantee ratio* which defined as the ratio of number of tasks found schedulable by an algorithm to the number of tasks considered for scheduling. The simulation considers dynamic task arrival only. Pre-run tasks are not considered. As a result, the guarantee ratio refers to the acceptance ratio for dynamically arriving tasks. Each point in the performance curves (fig 4) is the average of several simulation runs each with 1000 tasks with a 95% confidence level. In Figs 4, 'RV-EST' and 'RV-NO-EST' stand for RV resource reclaiming with estimation and without estimation, respectively.

In all the curves, in general, RV-EST gives a higher schedulability than RV-NO-EST. This is because the former is able to predict or estimate the reclaimed time much earlier and is thus able to aid the scheduler in accepting more tasks.

Fig 4a shows the effect of varying *NumProcs* in the system from 2 to 10. As the number of processors rises, both approaches show an increasing guarantee ratio, which saturates and then begins to decline slowly. The rise is a result of more resources (processors) being available for scheduling. The saturation is the point where the laxity restricts the schedulability of the tasks and no increase in the number of processors has any effect on the schedulability. The slow decline is due to the reclaiming cost increasing with the number of processors and an added estimation cost in the case of the estimation algorithm. For the system parameters chosen, as *NumProcs* rises, the gap

Parameter	Explanation	Values used
wcc_max	Task's maximum worst case computation time	50
wcc_min	Task's minimum worst case computation time	30
task graph density (P)	(number of links in task graph)/(number of tasks) a value of 0.0 indicates independent tasks.	0.85
UseP	Probability of a task using a given resource	0.5
ShareP	Probability of a task using a given resource in SHARED mode	0.5
RecCost	Cost of RV algorithm per processor	1.0
RDelCost	Cost of estimation per processor	1.0
max_laxity	Maximum laxity	1.5
min_laxity	Minimum laxity	1.3
max_aw_ratio	Maximum actual to worst case computation ratio	0.65
min_aw_ratio	Minimum actual to worst case computation ratio	0.60
NumProcs	Number of processors	6
TaskFreq	Average period of a task arrival at scheduler	225

Table 2: Simulation Parameters

between RV-NO-EST and RV-EST narrows and above some value of *NumProcs*, RV-EST performs worse than RV-NO-EST as the cost of estimation becomes high. However, note that this occurs in a region where the performance of the reclaiming algorithm (RV-NO-EST) itself is declining. In an actual real-time system running in such a region, the option of reclaiming itself is not likely to be used because of the high cost.

Fig 4b shows the effect of varying the average *TaskFreq* from 75 to 475. At high task arrival rates ($TaskFreq = 0$ to 100), RV-EST performs as bad as RV-NO-EST as the tasks arrive too quickly for the reclaimed time to be useful. As *TaskFreq* rises, RV-EST starts performing better than RV-NO-EST. But at low arrival rates ($TaskFreq \geq 350$), the low load allows the scheduler to schedule almost all the arriving tasks and the reclaimed time does not add anything to the schedulability, thus resulting in RV-NO-EST coming up to perform as well as RV-EST.

Fig 4c shows the effect of varying the average *wcc-min* in the system from 5 to 30. *wcc-max* is always chosen to be 50. Both algorithms show a declining guarantee ratio as the *wcc-min* goes up with RV-NO-EST performing progressively worse than RV-EST. This is because the laxity of tasks becomes tighter as *wcc_min* increases. Fig 4d shows the effect of varying the average *min_aw_ratio* in the system from 0.3 to 0.9. *max-aw-ratio* is always chosen to be 0.05 more than *min_aw_ratio*. Both algorithms show a declining guarantee ratio as the *aw_ratio* goes up. This is obviously because the reclaimable time decreases as *aw_ratio* increases increasing run-time load.

Fig 4e shows the effect of varying the average laxity from 0.5 to 1.9. The *min_laxity* and *max_laxity* values varied keeping $max\_laxity = min\_laxity + 0.2$. The laxity values shown on the graph is the average of *min_laxity* and *max_laxity* at that point. At low values of laxity, RV-NO-EST performs as well as RV-EST because the low deadline (tight laxity) of tasks causes the scheduler to reject most of the tasks. There is no effect of the reclaimed time on the schedulability. However, as laxity rises, RV-EST starts performing better than RV-NO-EST. Both algorithms show an increasing guarantee ratio as laxity rises, which then saturates. The rise is a result of increasing task deadlines. The saturation is the point where the resource constraints restrict the schedulability of the tasks.

Fig 4f shows the effect of varying *UseP* from 0.0 to 1.0. The value of *ShareP* is fixed. Both RV-EST and RV-NO-EST show a declining guarantee ratio with *UseP*'s rise because of the increasing resource constraints. Fig 4g shows the effect of increasing the precedence constraints on the tasks. A value of 0 stands for tasks without precedence constraints. A higher value of *Graph density* indicates more precedence relations among the tasks. For this experiment, *TaskFreq* is set to 125. Throughout this experiment, the same task load is maintained on the system. Only the precedence constraints among arriving tasks are increased. As the graph density increases, the performance of both RV-EST and RV-NO-EST fall but RV-EST stays above RV-NO-EST. This shows that all the above results (graphs a-f) will hold at whatever the extent of precedence constraints among the tasks.

6 Conclusions

Many real-time multiprocessor systems use resource reclaiming algorithms to utilize resources left unused by a task when it finishes early or when it is deleted from a fault-tolerant schedule due to fault-free operation. The Basic, Early Start, and RV algorithms are the best known algorithms to the resource reclaiming problem. However, there is a lack of efficient approaches to report this reclaimed time to the scheduler well in advance so that the scheduler can use this (reclaimed) time for efficient scheduling of newly arriving tasks. In this paper, we have proposed an estimation strategy which can be used along with a particular class of resource reclaiming algorithms (like Early Start and RV algorithms) by which the scheduler can estimate the minimum time by which any scheduled but unexecuted task will start or finish early, based solely on the start and finish times of tasks that have started or finished execution. We then proposed an approach by which dynamic scheduling strategies, which append or reschedule new tasks into the schedules, can use this estimation strategy to achieve better schedulability. We have also shown that the proposed estimation and scheduling lead to schedule (and hence task) executions that are correct i.e., never result in run-time anomalies. Our simulation studies reveal that the proposed strategies significantly improve the schedulability of the system for a wide range of task and system parameters.

References

[1] Kang G. Shin and P.Ramanathan, "Real-time computing: A new discipline of computer science and engineering," *Proc. IEEE,* vol.82, no.1, pp.6-24, Jan 1994.

[2] W.Zhao, K.Ramamritham, and J.A.Stankovic, "Scheduling tasks with resource requirements in hard real-time systems," *IEEE Trans. Software Eng.,* vol.13, no.5, pp.564-577, May 1987.

[3] Kwang S. Hong and J.Y-T. Leung, "On-line scheduling of real-time tasks," *IEEE Trans. Comput.,* vol.41, no.10, pp.1326-1331, Oct 1992.

[4] M.L.Dertouzos and A.K.Mok, "Multiprocessor on-line scheduling of hard real-time tasks," *IEEE Trans. Software Eng.,* vol.15, no.12, pp.1497-1506, Dec 1989.

[5] B.Sprunt, L.Sha, and J.P.Lehoczky, "Aperiodic task scheduling for hard real-time systems," *J. Real-Time Systems,* vol.1, pp.27-60, 1989.

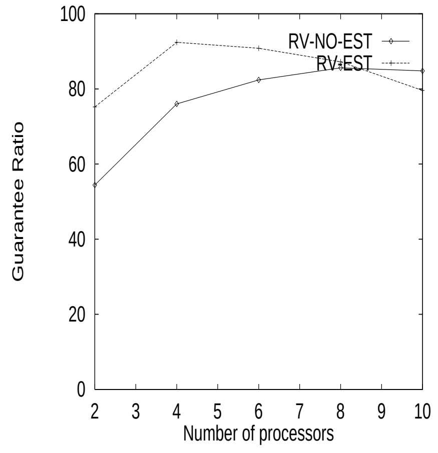

Fig 4a Effect of NumProcs

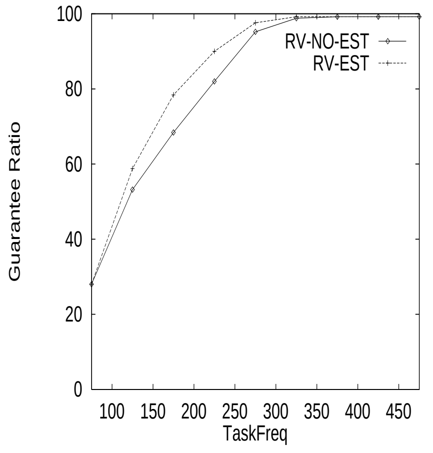

Fig 4b Effect of task arrival rate

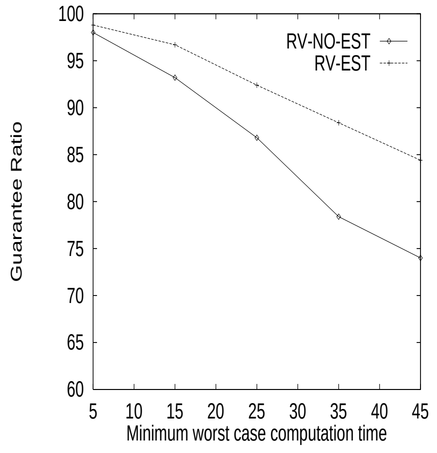

Fig 4c Effect of MinCompTime

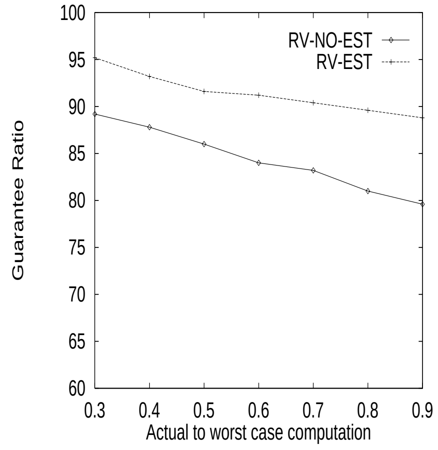

Fig 4d Effect of aw-ratio

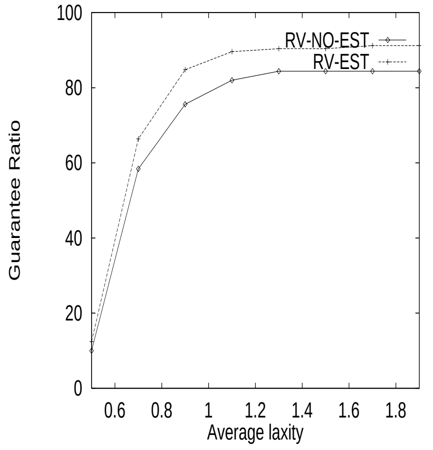

Fig 4e Effect of laxity parameter

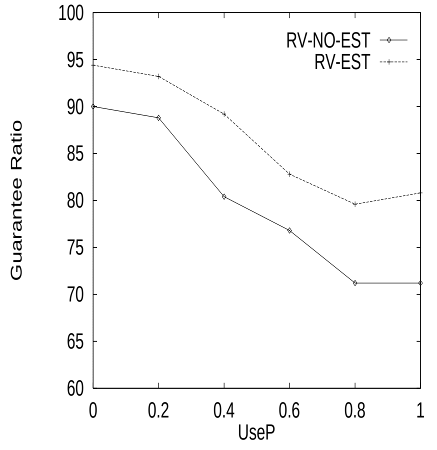

Fig 4f Effect of resource usage

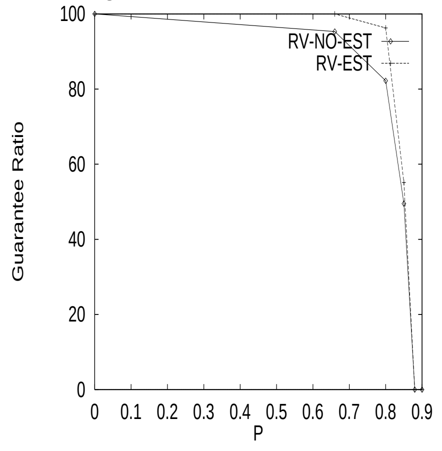

Fig 4g Effect of graph density

Figure 4: Simulation Results

[6] M. Spuri and J.A. Stankovic, "How to integrate precedence constraints and shared resources in real-time scheduling," *IEEE Trans. Computers,* vol.43, no.12, pp.1407-1412, Dec. 1994.

[7] C.-J. Hou and K.G. Shin, "Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems," *IEEE Trans. Computers,* vol.46, no.12, pp.1338-1356, Dec. 1997.

[8] K. Ramamritham, "Allocation and scheduling of precedence-related periodic tasks," *IEEE Trans. Parallel and Distributed Systems,* vol.6, no.4, pp.412-420, Apr. 1995.

[9] K. Ramamritham, "Dynamic priority scheduling," in *Real-time Systems - Specification, Verification, and Analysis* (Mathai Joseph, ed.), pp.66-96, Prentice Hall, 1996.

[10] K. Schwan and H. Zhou, "Dynamic scheduling of hard real-time tasks and real-time threads," *IEEE Trans. Software Engg.,* vol.18, no.8, pp.736-748, Aug. 1992.

[11] J.A. Stankovic and K. Ramamritham, "The Spring Kernel: A new paradigm for real-time operating systems," *ACM SIGOPS, Operating Systems Review,* vol.23, no.3, pp.54-71, July 1989.

[12] G. Manimaran, C. Siva Ram Murthy, M. Vijay, K. Ramamritham, "New algorithms for resource reclaiming from precedence constrained tasks in multiprocessor real-time systems", *Journal of Parallel and Distributed Computing*, vol. 44, no. 2, pp. 123-132, Aug. 1997.

[13] C. Shen, K. Ramamritham, J.A. Stankovic, "Resource reclaiming in multiprocessor real-time Systems", *IEEE Trans. Parallel and Distributed Systems*, vol.4, no.4, pp.382-397, Apr 1993.

[14] K. Ramamritham, J.A. Stankovic, P. F. Shiah, "Efficient scheduling algorithms for real-time multiprocessor systems", *IEEE Trans. Parallel and Distributed Systems*, vol.1, no.2, pp.184-194, Apr 1990.