New Algorithms for Resource Reclaiming from Precedence Constrained Tasks in Multiprocessor Real-Time Systems*

G. Manimaran C. Siva Ram Murthy
Dept. of Computer Science & Engineering
Indian Institute of Technology
Madras 600 036, INDIA
gmani@bronto.iitm.ernet.in murthy@iitm.ernet.in

Machiraju Vijay† Krithi Ramamritham†
Dept. of Computer Scinece Dept. of Computer Science
University of Utah University of Massachusetts
Salt Lake City, UT 84112, USA Amherst, MA 01003, USA
machiraj@facility.cs.utah.edu krithi@nirvan.cs.umass.edu

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. In this paper, we propose two algorithms to reclaim these resources from real-time tasks that are constrained by precedence relations and resource requirements, in shared memory multiprocessor systems. We introduce a notion called restriction vector for each task which captures its resource and precedence constraints with other tasks. This will help not only in the efficient implementation of the algorithms, but also in obtaining an improvement in performance over the reclaiming algorithms proposed in earlier work [2]. We compare our resource reclaiming algorithms with the earlier algorithms and, by experimental studies, show that they reclaim more

*This work was supported by the Department of Science and Technology, New Delhi.

†This work was done when the authors were at the Indian Institute of Technology, Madras.

resources, thereby increasing the guarantee ratio (the ratio of the number of tasks guaranteed to meet their deadlines to the number of tasks that have arrived), which is the basic requirement of any resource reclaiming algorithm. From our simulation studies, we demonstrate that complex reclaiming algorithms with high reclaiming overheads do not lead to an improvement in the guarantee ratio.

1 Introduction

Multiprocessors have emerged as a powerful computing means for real-time applications such as avionic control and nuclear plant control, because of their capability for high performance and reliabilty [1]. The problem of multiprocessor scheduling is to determine when and where a given task executes [1]. This can be done either statically or dynamically. In static algorithms [3, 4], the assignment of tasks to processors and the time at which the tasks start execution are determined *a priori*. Static algorithms are often used to schedule periodic tasks with hard deadlines. The main advantage is that, if a solution is found, then one can be sure that all deadlines will be guaranteed. However, this approach is not applicable to aperiodic tasks whose arrival times and deadlines are not known apriori. Scheduling such tasks in a multiprocessor real-time system requires dynamic scheduling algorithms [5, 6]. In dynamic scheduling, when new tasks arrive, the scheduler dynamically determines the feasibilty of scheduling these new tasks without jeopardizing the guarantees that have been provided for the previously scheduled tasks. Thus for predictable executions, schedulability analysis must be done before a task's execution is begun. 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. When a task arrives at a processor, the local scheduler at the processor determines whether or not it can satisfy the constraints of the incoming task. The task is accepted if they can be satisfied, otherwise the local scheduler tries to find another processor which can 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*. Each processor has its own dispatch queue. This organization, shown in Fig.1, ensures that the processors will always find some tasks in the dispatch queues when they finish the execution of their current tasks. The scheduler will be running 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 average time required by the scheduler to reschedule its tasks upon the arrival of a new task [2].

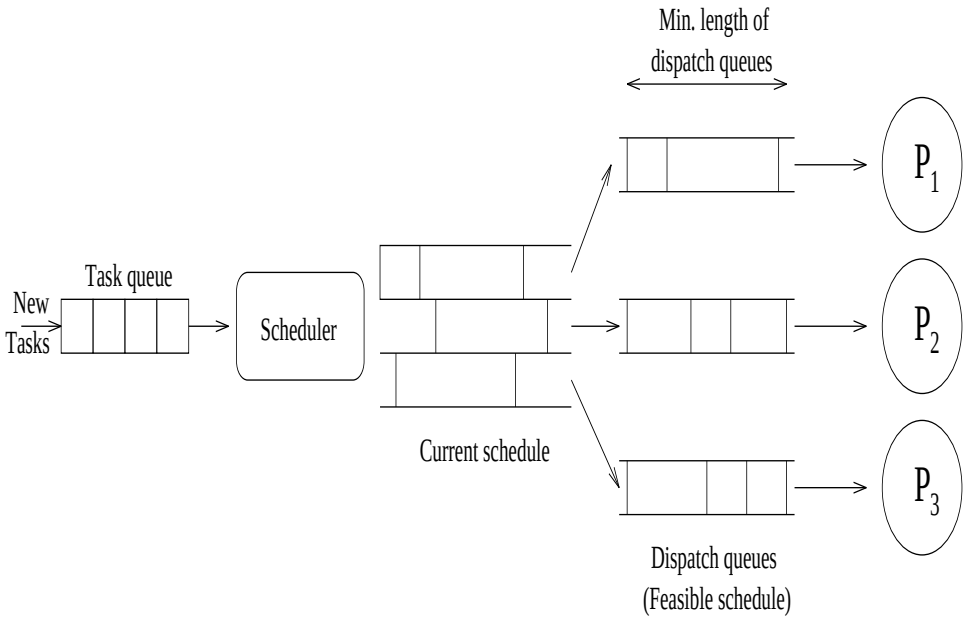


Fig.1 Parallel execution of scheduler and processors

The scheduler arrives at a *feasible schedule* taking the worst case computation times, deadlines, and the constraints of the tasks into account. The actual time taken by a task during execution can be smaller than its worst case computation time. Hence, a lot of resources remain unused if we dispatch the tasks strictly based on their starting times of the feasible schedule. *Resource reclaiming* is required to utilize the resources 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, and is invoked by each processor on completion of its currently executing task. Task deletion takes 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. Resource reclaiming on multiprocessor systems with independent tasks is straightforward. The resource reclaiming in such systems is *work-conserving* which means that the reclaiming never leaves a processor idle if there is a dispatchable task. But, resource reclaiming on multiprocessor systems with resource and precedence constrained tasks is more complicated. This is due to the potential parallelism provided by a multiprocessor, and potential resource and precedence constraints among tasks.

When the actual computation time of a task differs from its worst case computation time in a nonpreemptive multiprocessor schedule with resource constraints, run-time anomalies [7] may occur. These anomalies

may cause some of the already guaranteed tasks to miss their deadlines. In particular, one cannot simply use a work-conserving scheme without verifying that the task deadlines will not be missed. Any resource reclaiming algorithm should possess four properties namely, correctness, inexpensive, bounded complexity, and effeciveness [2]. 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 meet their deadlines 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 performance of the system). The bounded complexity requirements necessiate the need for running the resource reclaiming algorithm by the processors rather than by the scheduler.

Earlier work [2] considered resource reclaiming in multiprocessor real-time systems with resource constrained tasks. Two algorithms, *Basic reclaiming* and *Early start* were proposed. Both these algorithms satisfy the bounded complexity requirement. In this paper, we extend the task model presented in [2] to include precedence constraints among tasks. We propose two new algorithms, called *RV-algorithms*, for this enhanced task model, which reclaim more resources than the earlier algorithms, basic reclaiming and early start. The first RV-algorithm reclaims resources under the constraint that once a task is scheduled on a processor, it must execute on the same processor. However, in certain circumstances, all that the resource reclaiming algorithm must ensure is that the tasks complete execution before their deadlines. The processor on which a task executes can be different from the one on which it is originally scheduled. Hence, in this paper, we study the effect of migrating tasks from one processor to another in an attempt to increase the guarantee ratio. Towards the end, we propose a second algorithm that includes *task-migration*, which will be called *RV-algorithm with task-migration*, and show that it will reclaim more resources than the original RV-algorithm, under certain circumstances. Throughout the paper, the term task-migration refers to swapping of dispatch queues.

In Section 2 of this paper, we will describe the task model and some basic definitions. We will also introduce a notation called *restriction vectors*, which will be used in the *RV-algorithms*. Section 3 discusses our (original) *RV-algorithm* and in Section 4, we present the *RV-algorithm with task-migration*. In Section 5, we support our claims by simulation studies and in Section 6, we summarize our work with some concluding remarks.

2 Basics

2.1 Task model

We make the following assumptions about the task model. Each task T_i has the following attributes:

1. worst case computation time c_i, which is the upper bound on the computation time of T_i, when all the overheads of scheduling and resource reclaiming are included.

2. Deadline d_i.

3. precedence constraints with other tasks: 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 $T_j \prec T_i$.

4. resource constraints: A task might need some resources such as data structures, variables, and communication buffers for its execution. Every task can have two types of access 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 $T_i \odot T_j$.

In addition, we assume as follows:

1. At any instant, atmost one task can be executed on a given processor. Tasks are not preemptable.

2. Tasks cannot be migrated from one processor to another, i.e., if the scheduler assigns a processor P_i to a task T_i, then T_i has to be executed on P_i. (We later relax this assumption.)

3. All the processors are identical and do not fail. Processor faults can be dealt with, for example, by scheduling a task on multiple processors (spatial redundancy) or multiple times on the same processor (temporal redundancy). But in this paper, we do not consider processor faults.

4. There is a memory shared by all the processors, in which the dispatch queues are placed (shared memory multiprocessor model).

Fig.2a shows a task graph with 13 tasks. A directed arc between two tasks in this figure indicates the precedence relation between them. The resource requirements of a resource r are also shown in this figure. Tasks T_7 and T_9 need exclusive accesses to the resource r, whereas tasks T_4 and T_{12} can share the resource r. Fig.2b gives the worst case computation times and deadlines of these tasks with ready times equal to 0.

We will use a convenient notation, called *Restriction Vectors (RVs)*, for the description of these resource and precedence constraints. As we will see in the later sections of this paper, the same RVs can be used to reclaim the resources efficiently in our *RV-algorithms*. In what follows, we will assume that m is the number of processors and n is the number of tasks currently available with the scheduler. Before introducing RVs, we will look at some basic definitions.

2.2 Terminology

Definition 1: The scheduler fixes a *feasible schedule* S taking into account the precedence and resource constraints of all the tasks. The feasible schedule uses the worst case computation time of a task for scheduling it and ensures that the deadlines of all the tasks in S are met.

Definition 2: Starting from a feasible schedule, a *post-run schedule* S' is the layout of the tasks in the same order as they are executed at run time with respect to their actual computation times. The *actual computation time* c'_i of a task T_i is the actual time taken by the task during execution. By definition, the actual computation time of any task is always less than or equal to its worst case computation time, i.e., $c'_i \leq c_i$.

Definition 3: 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'.

Definition 4: Given a post-run schedule S', a task T_i starts *on time* if $st'_i \leq st_i$. A post-run schedule S' is *correct* if $\forall i\ 1 \leq i \leq n$, $ft'_i \leq d_i$. As proved in [2], if all the tasks start on time, then the post-run schedule will be correct.

Definition 5: $T_{<i} = \{T_j : ft_j < st_i\}$; $T_{>i} = \{T_j : st_j > ft_i\}$; $T_{\simeq i} = \{T_j : T_j \notin T_{<i}, T_j \notin T_{>i}\}$. $T_{<i}$ denotes the set of tasks that are scheduled in S to finish before T_i starts. $T_{>i}$ denotes the set of tasks that are scheduled after T_i finishes and $T_{\simeq i}$ denotes the set of tasks that overlap with T_i in S.

Definition 6: $T_{<i}(j) = \{T_k : T_k \in T_{<i}$ and T_k is assigned to processor $P_j\}$. From this it follows that, for any task T_i, $T_{<i} = \bigcup_{j=1}^{m} T_{<i}(j)$. In the feasible schedule, the tasks of $T_{<i}(j)$ are ordered according to their start times in the feasible schedule.

For the example shown in Fig.2, a feasible schedule is given in Fig.3a. We notice that all the precedence, resource constraints, and deadlines are met in this feasible schedule. The start and finish times of all the tasks are clearly shown in this figure. Figs.3b-3e show the post-run schedules, when various algorithms are used for resource reclaiming. We will take a detailed look at these post-run schedules in the next section. We notice that all these post-run schedules are correct since every task in these schedules finishes before its deadline.

Definition 7: A task T_i *passes* another task T_j if $st'_i < st'_j$, but $ft_j < st_i$. Thus *passing* occurs 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.

2.3 Restriction Vectors (RVs)

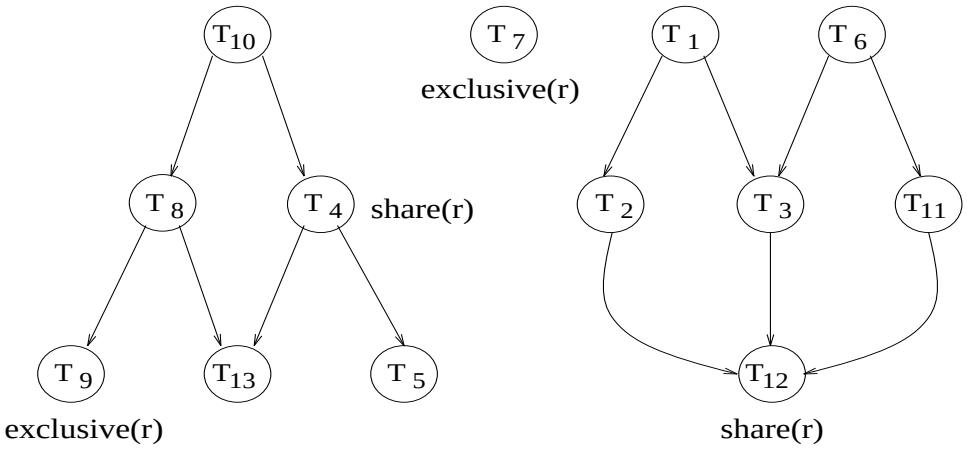
Definition 8: Each task T_i has an associated m-component vector, $RV_i[1..m]$, called Restriction Vector. $RV_i[j]$ for a task T_i contains the *last task* in $T_{<i}(j)$ which must be completed before the execution of T_i begins.

$$RV_i[j] = \begin{cases} T_k & \text{if } Proc(i) = j, \text{ where } T_k \in T_{<i}(j) \text{ and } \nexists T_l \text{ such that } st_l > st_k, \\ & \text{where } T_l \in T_{<i}(j), \text{ and } Proc(i) \text{ denotes the processor on which task } T_i \text{ is scheduled} \\ T_m & \text{if } Proc(i) \neq j, \text{ where } T_m \in T_{<i}(j) \text{ and } (T_m \prec T_i \text{ or } T_m \odot T_i), \text{ and} \\ & \nexists T_l \text{ such that } st_l > st_m, \text{ where } T_l \in T_{<i}(j) \text{ and } (T_l \prec T_i \text{ or } T_l \odot T_i) \\ \text{“}-\text{”} & \text{if no such task exists.} \end{cases}$$

The RVs for the tasks in the task graph of Fig.2 are given in Fig.4. For example, $RV_3 = [T_2, T_6, -]$ indicating that T_2, T_6 have to finish execution on processors P_1 and P_2 respectively before T_3 starts execution. $RV_3[1] = T_2$ indicates that T_2 is the immediate predecessor of task T_3 on processor P_1 and $RV_3[2] = T_6$ indicates that T_6 is the *last task* in $T_{<3}(2)$ which has precedence constraint with T_3. $RV_3[3] = \text{”}-\text{”}$ indicates that there is no direct restriction for T_3 from any task on processor P_3. Similarly, $RV_9 = [T_4, T_8, T_{12}]$, because T_9 has resource conflicts with T_4 and T_{12}, and a precedence constraint from T_8. Thus, RVs take both the precedence constraints and resource conflicts into account. In the feasible schedule shown in Fig.3a, notice that $T_7 \in T_{<3}(2)$, but $RV_3[2] \neq T_7$ because T_7 does not have either resource conflicts or precedence constraints with T_3. Therefore, the execution of T_3 can overlap with T_7, and this is exploited by the *RV-algorithm* unlike the early start algorithm which considers only the tasks in $T_{\simeq 3}$ as possible tasks with which T_3 can overlap in execution.

3 RV-algorithm for resource reclaiming

Fig.3b is the post-run schedule for the feasible schedule of Fig.2 without any resource reclaiming. In this post-run schedule, notice that all the processors are idle between time 125 and 150. This fact is used by the basic reclaiming algorithm [2] to reclaim the unused resources. The post-run schedule using the basic reclaiming algorithm is shown in Fig.3c. Refering to this post-run schedule (Fig.3c), notice that, T_9 could have started 50 units of time earlier, i.e. at time 350, because $T_9 \in T_{\simeq 5}$ and $T_9 \in T_{\simeq 13}$. Similarly T_{13} can also be started 50 units of time earlier. This is what the early start algorithm [2] does. The post-run schedule using the early start algorithm is given in Fig.3d. These algorithms are discussed in detail in [2].



(a) Precedence constraints and resource requirements of real-time tasks

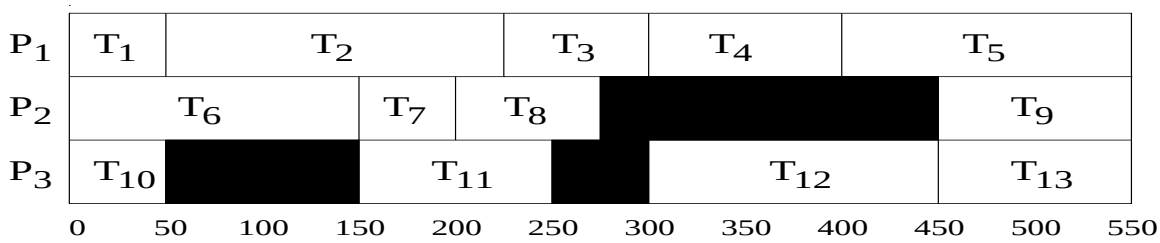
task	deadline	worst case comp. time	resource requirement (r)
T_1	100	50	
T_2	250	175	
T_3	350	75	
T_4	500	100	share
T_5	600	150	
T_6	150	150	
T_7	200	50	exclusive
T_8	300	75	
T_9	600	100	exclusive
T_{10}	50	50	
T_{11}	300	100	
T_{12}	450	150	share
T_{13}	575	100	

(b) parameters for the tasks

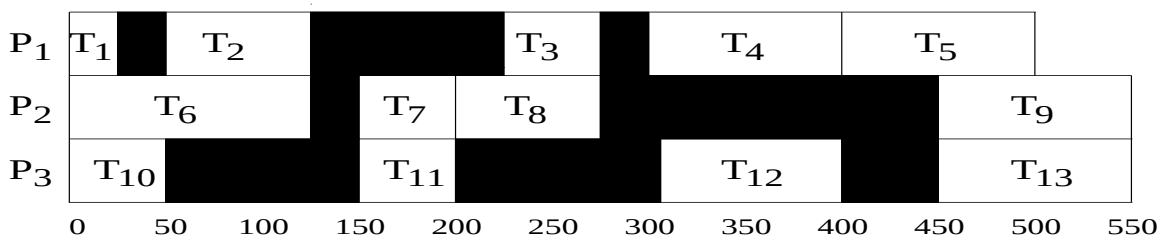
Fig.2 An example of real-time tasks

task	RV	task	RV
T_1	$[-,-,-]$	T_8	$[-,T_7,T_{10}]$
T_2	$[T_1,-,-]$	T_9	$[T_4,T_8,T_{12}]$
T_3	$[T_2,T_6,-]$	T_{10}	$[-,-,-]$
T_4	$[T_3,T_7,T_{10}]$	T_{11}	$[-,T_6,T_{10}]$
T_5	$[T_4,-,T_{10}]$	T_{12}	$[T_3,T_7,T_{11}]$
T_6	$[-,-,-]$	T_{13}	$[T_4,T_8,T_{12}]$
T_7	$[-,T_6,-]$		

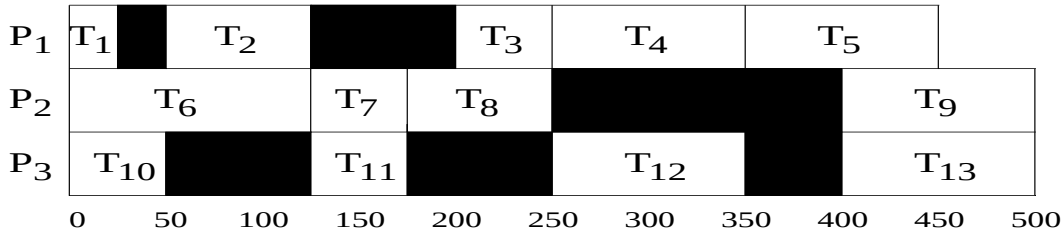
Fig.4 RVs for the real-time tasks example



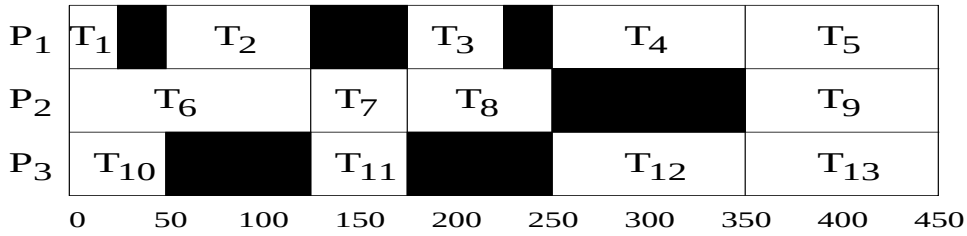
(a) Feasible schedule in dispatch queues



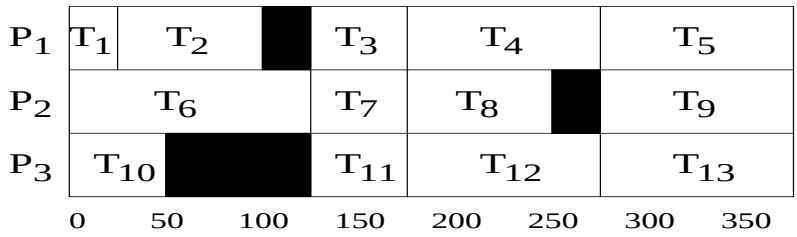
(b) Post-run schedule without resource reclaiming



(c) Post-run schedule with Basic reclaiming



(d) Post-run schedule with Early start



(e) Post-run schedule with RV algorithm

Fig.3 Feasible and post-run schedules for the real-time tasks

We demonstrate that, for the same example, more resources can be reclaimed. In Fig.3d, T_3 could have started at time 125 because neither T_7 nor T_{11} have any resource or precedence constraints with T_3. Early start algorithm does not reclaim this resource since $T_3 \notin T_{\simeq 7}$. The RVs defined in the previous section contain a better picture of the resource and precedence constraints from the reclaiming point of view. From the restriction vector of T_3 (RV_3), it is clear that there are no constraints between T_3 and T_7. $RV_3[2] = 6$ which indicates that T_2 is only restricted by T_6 and not by any task after T_6 on P_2. Our *RV-algorithms* exploit this fact to reclaim more resources.

Before describing the algorithms we take a look at the other data structures used in them.

(a) Completion Bit Matrix (CBM) : an $n \times m$ boolean matrix indicating whether a task has completed execution.

$$CBM[i,j] = \begin{cases} 0 & \text{iff } T_i \text{ scheduled to execute on } P_j \text{ has not yet completed its execution,} \\ 1 & \text{otherwise.} \end{cases}$$

(b) Dispatch queues ($DQ[1..m]$) : Each processor P_i has its own dispatch queue $DQ[i]$. These dispatch queues are the same as those described in Section 1. Fig.5 gives the *RV-algorithm* for reclaiming resources from precedence and resource constrained tasks.

/* Whenever a task T_i finishes execution on processor P_j,*/

1. Set $CBM[i,j]$ to 1.
2. For all idle processors P_k do,
   - 2.1 Let T_f be the first task in $DQ[k]$.
   - 2.2 For all the components of the RV_f, check CBM to see if the tasks in RV_f have finished execution.
   - 2.3 If all of them have finished execution, start the execution of T_f on P_k and remove T_f from DQ_k

Fig.5 *RV-algorithm*

1. For all processors P_i, $1 \leq i \leq m$,
   - 1.1 Take the first task T_f from $DQ[i]$.
   - 1.2 If T_f can be executed, start execution.
   - 1.3 While $DQ[i]$ is not empty do
     - 1.3.1 If there is a task under execution on P_i,
       - 1.3.1.1 Wait for its completion.
       - 1.3.1.2 Invoke *RV-algorithm*.

(a) Execution of the tasks on processors

1. while (true)

    1.1 repeat

        1.1.1 Whenever a new task arrives,

        1.1.2 Perform the schedulability check.

        1.1.3 Form a feasible schedule taking into account

            the precedence and resource constraints.

    1.2 until cut-off-time.

    1.3 Compute the restriction vectors for the tasks in the feasible schedule.

    1.4 Update the dispatch queues.

    1.5 Compute new cut-off-time.

(b) Scheduler

Fig.6 Parallel operation of the processors and the scheduler

All the elements of CBM are initialized to 0. The DQs are constantly updated by the scheduler as observed in Section 1. Each DQ can be maintained as a linked list with the first element pointing to the first task ready for execution in it. The entire parallel operation of the scheduler and the processors is demonstrated in Fig.6. The cut-off-time in the above algorithm is to ensure this parallel operation and is directly related to the minimum length of the dispatch queues mentioned in Section 1. Further, the scheduler should be aware of the amount of time reclaimed by the reclaiming algorithm for it to be able to schedule the new tasks correctly and effectively. A protocol for achieving this is suggested in [2].

3.1 Complexity of the RV-algorithm

The basic reclaiming and early start algorithms have a resource reclaiming complexity of $O(m)$ and $O(m^2)$, respectively. Step 2.2 in the *RV-algorithm* of Fig.5 takes $O(m)$ time since all the m components of the restriction vector have to be checked to find out whether the corresponding tasks have finished execution. There can be $O(m)$ idle processors and hence the time complexity of the algorithm is $O(m) * O(m)$, which is $O(m^2)$. Hence, the *RV-algorithm* satisfies the bounded complexity requirement mentioned in Section 1, and has the same run-time complexity as the early start algorithm.

3.2 Proof of correctness

Lemma: Given a feasible real-time multiprocessor schedule S, if $\exists T_i$ such that T_i does not start *on time* in a post-run schedule then *passing* should have occured.

Proof: Since T_i does not start on time, $st'_i > st_i$. Assume the contradiction, i.e., assume no passing occurred. Then the tasks in $T_{<i}$ must have been dispatched before T_i started and the tasks in $T_{>i}$ must have been dispatched after T_i finished execution. By definition of a feasible schedule, the tasks in $T_{\simeq i}$ overlap with task T_i in S which means that they do not have resource or precedence constraints with T_i, therefore, no matter in what order these tasks were dispatched with respect to the dispatching time of T_i, they would not have delayed the dispatching time of T_i. This contradicts the premise that T_i did not start on time. □

Theorem: Given a feasible multiprocessor schedule with resource and precedence constraints, the post-run schedule produced by the RV-algorithm is correct.

Proof: To prove this we need to show that RV-algorithm does not lead to any run-time anomalies.

By lemma 1, if there is a task T_i on processor P_k which does not start on time, then passing must have occured [2]. Consider a schedule in which T_i and T_j are scheduled on P_k and P_j, respectively, with $st_j \geq ft_i$. Assume that T_i has missed its deadline in the post-run schedule because T_j has passed T_i. We consider two cases.

a) Case 1: T_i and T_j have resource or precedence constraints.

If there was a precedence or resource constraint, then $RV_j[k] = T_i$ and hence the RV-algorithm does not permit this passing.

b) Case 2: T_i and T_j have no constraints between them.

In this case, T_i need not wait for the completion of T_j. They can overlap in execution and hence T_i can still start on time.

Since all the tasks start on time, there will be no run-time anomalies. □

It may be noted that the RV-algorithm allows passing in a restricted way. A task T_i is allowed to pass all those tasks in $T_{<i}$ which do not have any precedence constraints or resource conflicts with it. The above theorem proves that this passing does not lead to run-time anomalies.

4 RV-algorithm with task-migration

In this section we will study the potential performance improvement in the guarantee ratio by migrating the tasks from one processor to another processor. By task-migration here, we mean swapping of dispatch queues. For this, we assume a shared-memory model, in which the dispatch queues are global. A processor has access to the tasks in the dispatch queues of all the processors.

In the RV-algorithm, DQ_i is always assigned to processor P_i. In this section, DQ_i represents a pointer to the dispatch queue that is *currently* assigned to processor P_i. The assignment of dispatch queues to

processors varies dynamically during the execution of the algorithm. At any point of execution, only one DQ will be assigned to any processor, and no DQ will be assigned to more than one processor. We consider the following mechanism for migrating tasks from one processor to another. When a task T_i completes execution on processor P_i, P_i looks at the first task in DQ_i to see if it can be executed immediately. This is done by examining the RV of that task. If it cannot be executed, P_i examines the dispatch queues of k other processors to see if the first task in any of them can be executed immediately. If P_i finds that such a dispatch queue exists, say DQ_j $(i \neq j)$, then P_i *swaps* pointers DQ_i and DQ_j, so that the dispatch queues now associated with P_i and P_j will be different. Then P_i starts the execution of the first task in its new dispatch queue. The value of k, which is the number of dispatch queues that have to be checked to find a new task for execution, determines both the efficiency of the algorithm and its complexity.

5 Experimental studies

To evaluate the performance of the proposed resource reclaiming *RV-algorithms*, we conducted extensive simulation studies and compared its performance with that of the algorithms given in [2]. The parameters used in these simulation studies are shown in Fig.7.

The worst case computation time (wcc) of a task was generated by adding a value between wcc-min and wcc-max to the algorithm-costs (AC). This value was generated by a uniform distribution between wcc-min and wcc-max. AC includes the scheduler cost (SC) and the reclaiming cost (RC). The SC for the RV-algorithms includes the overhead costs incurred in the computation of RVs. In the case of RV-algorithm with task migration, RC is a function of number of migration attempts.

Guarantee ratio is defined as the ratio of the number of tasks guaranteed to meet their deadlines to the number of tasks arrived. In order to study the pure effects of reclaiming algorithms, and to compare their effectiveness in reclaiming resources, we have not inlcuded the scheduling costs in our simulation studies. Hence AC was taken to be equal to the value of RC. However, the extra costs incurred in the computation of the RVs was included in the algorithm costs.

5.1 Effect of precedence and resource constraints

Fig.8 shows the effect of varying the precedence constraints among the tasks keeping the num-procs, wcc-min and wcc-max fixed (at 4, 25, and 50, respectively). The parameter P_p, which is the *precedence constraint probability* is varied from 0.3 to 0.9. A higher value of P_p indicates that the newly arrived tasks will have more precedence constraints from the already arrived tasks. Further, a high value of P_p will also indicate

that any given task will have larger number of precedence constraints from the immediately preceding tasks (on other processors) in the feasible schedule. A smaller value of P_p indicates that new tasks will be less constrained by precedence relations (more independent). No reclaiming, basic reclaiming, and early start gave horizontal lines for this variation because, there is no effect of precedence relations on these algorithms. In the RV-algorithm, as the tasks are more constrained by precedence relations, they will be more restricted from starting earlier, and hence the guarantee ratio decreases. It is also clear from the graph that, when P_p approaches its maximum value of 1.0, the RV-algorithm tends to behave like the early start algorithm.

parameter	explanation
wcc-min	task's minimum worst case computation time
wcc-max	task's maximum worst case computation time
task graph density (P_p)	probability that the new task will have precedence constraints with the already existing tasks, ranges from 0.0 to 1.0, a value of 0.0 indicates independent tasks.
aw-comp-ratio	ratio between actual computation time and worst case computation time, ranges from 60% to 90%.
num-procs	number of processors used during simulation
RV-comp-cost	cost incurred in the computation of RVs, taken as 1 for 4, 5, and 6 processors, and 2 for 12, 14, and 16 processors
mig-attempts	number of dispatch queues checked by the RV-algorithm with task migration (defined as k in Section 4), taken as 1
task arrival rate	arrival rate of tasks, ranges from 0.4 to 0.9
$RC_{no-reclaiming}$	0
RC_{Basic}	1
$RC_{Early start}$	num-procs * RC_{Basic}
$RC_{RV-algorithm}$	$RC_{Early start}$ + RV-comp-cost
$RC_{RV-migration}$	$RC_{RV-algorithm} + f$(mig-attempts,$RC_{Early start}$), taken as 6, 7, and 8 for 4, 5, and 6 processors, respectively, and 15, 17, and 19 for 12, 14, and 16 processors, respectively

Fig.7 Simulation parameters

5.2 Effect of worst case computation time

As the worst case computation time of a task decreases, the ratio of resource reclaiming cost to worst case computation time increases. Since RC is a part of worst case computation time, it will not be feasible to perform reclaiming if the worst case computation time decreases beyond a certain limit. This experiment was performed to get an estimate of these limits for the various resource reclaiming algorithms. The graphs

are plotted in Fig.9. wcc-min was varied from 20 to 34 units of time, and the worst case computation time of a task was uniformly distributed between wcc-min and wcc-max before adding the RC. When wcc-min = 20, the ratio of RC to worst case computation time will range between 10% and 20% (using $RC_{Early\ start}$). Hence, the reclaiming overhead will be 10% to 20%. When wcc-min = 34, this overhead will only be 6% to 12%. From the graphs, it is clear that RV-algorithm performs no better than no reclaiming and early start when wcc-min = 20, since at this point the reclaiming costs overtake the improvement in performance obtained by reclaiming. It only pays off to use RV-algorithm when wcc-min $\geq$ 22 units.

5.3 Effect of actual to worst case computation ratio

Fig.10 shows the effect of varying actual to worst-case computation time ratio keeping num-procs, wcc-min, and P_p fixed at 4, 25, and 0.7, respectively. For smaller values of aw-comp-ratio ($\leq$ 0.65), the guarantee ratio is 100% both in early start and RV-algorithms. As the aw-comp-ratio increases, the guarantee ratio decreases. RV-algorithm (without task migration) gives better results than the early start algorithm, even if the amount of its reclaimable computation time is lesser than that of the latter, because it is more effective in fully utilizing the reclaimable computation time, whereas early start does only a partial utilization as discussed in Section 3.

5.4 Effect of number of processors

Fig.11 shows the effect of varying the number of processors on the performance of various algorithms. The effect of varying the task load on the guarantee ratio, was studied for 12, 14, and 16 processors. All the graphs exhibit the same behaviour in the sense that the guarantee ratio decreases as the task load increases. For the same task load, as the number of processors increases, the guarantee ratio increases, as more number of processors are available for dispatching the new tasks. The graphs clearly indicate the superiority of RV-algorithms.

5.5 Effect of task-migration

The RC in this case depends on the number of migration attempts (mig-attempts). Graphs were plotted to study the influence of these additional costs over the guarantee ratio offered by this algorithm. For smaller values of worst case computation times (20), the ratio of RC to wcc-min is as high as 35%. Hence, the guarantee ratios offered by the RV-algorithms increase with increase in the average worst case computation time of a task (Fig.9). From the graph in Fig.10, the RV-algorithm with task migration offers poorer guarantee ratio than the other two. This is because of the fact that the overhead costs in the case of RV-algorithm

with task migration are greater than those in the other two. The amount of reclaimable computation time is less in the case of RV-algorithm with task migration and hence the guarantee ratio offered by this algorithm is less. From the graph shown in Fig.8, RV-algorithm with task migration is effective when P_p is greater than 0.7. When the tasks are more constrained by precedence relations, the RV-algorithm gives a poorer performance compared to RV-algorithm with task migration since the former checks only its own dispatch queue and it is more likely that the first task in its dispatch queue cannot be immediately executed because of the higher value of P_p.

6 Conclusions

In this paper, we have studied the problem of resource reclaiming for tasks with precedence constraints and resource requirements in real-time multiprocessor systems. We have proposed new algorithms called RV-algorithms for resource reclaiming for this task model and showed that they are correct (causing no run-time anomalies) and satisfy the bounded complexity requirement. We studied their effectiveness through simulation and found that the RV-algorithm (without task migration) gives better guarantee ratio than the early start algorithm under all circumstances. We have observed the trade-offs between the algorithm costs and the improvements in performance of the various reclaiming algorithms. No-reclaiming, basic reclaiming, early start, RV-algorithm, and RV-algorithm with task migration are the resource reclaiming algorithms in the increasing order of computational overheads. As we proceed to use the latter algorithms in an attempt to reclaim more resources, at one stage, the overheads incurred will be so high that they nullify the advantages obtained by reclaiming more and more resources. From our experimental studies, we conclude that RV-algorithm with task migration is one such algorithm, beyond which, it will not pay off to use more sophisticated algorithms to reclaim more resources.

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

[3] J. Xu and L. Parnas, "Scheduling processes with release times, deadlines, precedence, and exclusion relations," *IEEE Trans. on Software Engg.,* pp.360-369, Mar. 1992.

[4] K. Ramamritham, "Allocation and scheduling of precedence-related periodic tasks," *IEEE Trans. on Computers,* vo.6, no.4, pp.412-420, Apr. 1995.

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

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

[7] R.L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM J. Appl. Math.,* vol.17, no.2, Mar. 1969.

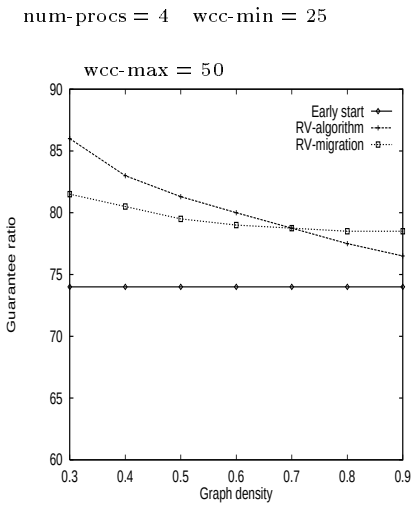


Fig.8 Effect of precedence and resource constraints

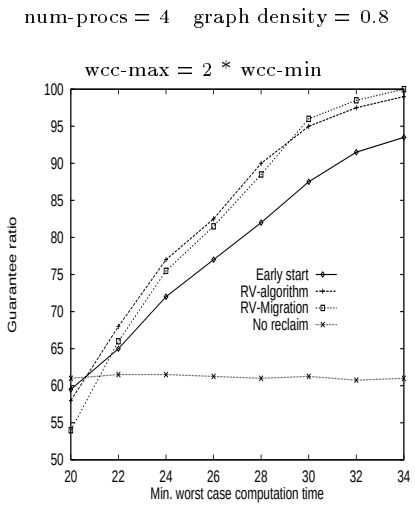


Fig.9 Effect of worst case computation time

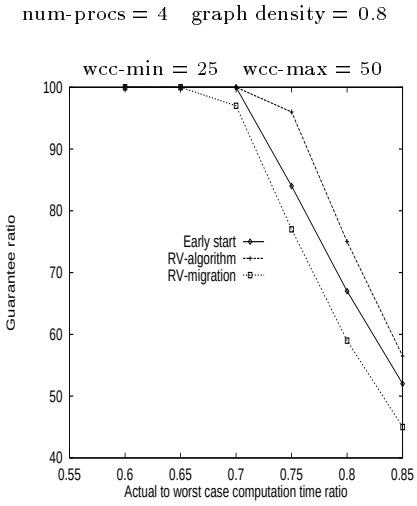


Fig.10 Effect of actual to worst case computation ratio

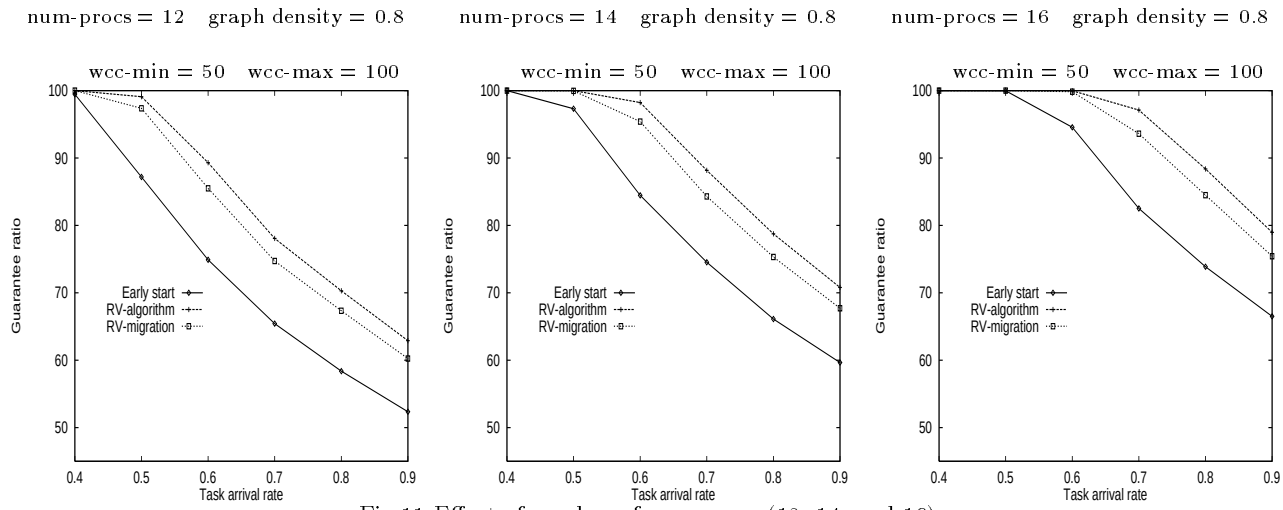


Fig.11 Effect of number of processors (12, 14, and 16)