RESOURCE RECLAIMING IN REAL-TIME

C. Shen, K. Ramamritham, J. A. Stankovic
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 90-89

Resource Reclaiming in Real-Time *

Chia Shen
Krithi Ramamritham
John A. Stankovic
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

February 11, 1991

Abstract

Most real-time scheduling algorithms schedule tasks with respect to their worst case computation times. *Resource reclaiming* refers to the problem of utilizing 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. Resource reclaiming is a very important issue in dynamic real-time multiprocessor environments. Resource reclaiming algorithms must be designed with three issues in mind. Firstly, a resource reclaiming algorithm should be effective in reclaiming resources; that is, it should improve the performance of the system. Secondly, since most of the tasks execute less than their worst case computation times, resource reclaiming will occur frequently. To be effective, resource reclaiming overheads should be small. More precisely, we seek an algorithm with bounded overhead, i.e., with time complexity that is *independent* of the number of tasks in a schedule, and include this overhead cost in the worst case computation time of a task. Thirdly, when the actual computation time of a task differs from its worst case computation time in a multiprocessor schedule with resource constraints, run time anomalies may occur if greedy schemes are used for utilizing idle resources. These anomalies may cause some of the already feasibly scheduled tasks to miss their deadlines. Thus, *dynamic resource reclaiming* algorithms must be effective in reclaiming unused time, have low, bounded overheads, and also avoid any run time anomalies. In this paper, we present resource reclaiming algorithms with these properties. The effectiveness of the algorithms is demonstrated through simulation studies. The algorithms have also been implemented in the Spring Kernel [17].

index terms — deadlines, dynamic real-time systems, multiprocessor scheduling, resource constraints, worst case computation times.

*This work is part of the Spring Project at the University of Massachusetts and is funded in part by the Office of Naval Research under contract N00014-85-K-0398 and by the National Science Foundation under equipment grants DCR-8500332 and CCR-8716858.

1 Introduction

In real-time applications such as space stations, avionics, and command and control systems, many tasks have execution deadlines. Among these real-time tasks, some are *safety-critical*, i.e., their deadlines must be met under all circumstances, otherwise the result could be catastrophic; while others are not safety-critical, i.e., missing their deadlines will seriously degrade the performance of a system but will not cause catastrophe. In such real-time applications, the resources required by the safety-critical tasks should be preallocated and a schedule should be statically produced with respect to the worst case timing and resource requirements of these tasks so that their deadlines will be met. On the other hand, due to the dynamic and nondeterministic nature of these applications, other real-time tasks have to be scheduled *on-line* as they arrive since it is impossible to statically reserve enough resources for all contingencies with respect to the worst case requirements of these tasks. Thus, one of the performance metrics for these real-time systems should be the *guarantee ratio* defined as $\frac{\text{the number of tasks guaranteed}}{\text{the number of tasks arrived}}$.

When real-time tasks arrive in a dynamic real-time environment, the scheduler dynamically determines the feasibility of scheduling the new task and the previously scheduled tasks, including safety-critical tasks, given their worst case requirements and current system state. A feasible schedule is generated if all the timing and resource requirements of tasks can be satisfied. Tasks are dispatched according to this feasible schedule. In order to guarantee that real-time tasks will meet their deadlines once they are scheduled, most real-time scheduling algorithms schedule tasks with respect to their worst case computation times [6, 9, 13, 18]. Since this worst case computation time is an upper bound, the actual execution time may vary between some minimum value and this upper bound, depending on various factors, such as the system state, the amount and value of input data, the amount of resource contention, and the types of tasks. *Resource reclaiming* refers to the problem of utilizing 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. Task deletion occurs either during an operation mode change [14], or when one of the copies of a task completes successfully in a fault-tolerant system [2] and the fault semantics permits deletion of the other copies from the schedule. Resource reclaiming is a very important issue in dynamic real-time systems, and it has not been addressed in practice.

The design of dynamic resource reclaiming algorithms in real-time systems has four requirements:

(1) *correctness*: A resource reclaiming algorithm must maintain the feasibility of guaranteed tasks, i.e., any possible run time anomalies must be avoided.

(2) *inexpensive*: The overhead cost of a resource reclaiming algorithm should be very low compared to tasks' computation times since a resource reclaiming algorithm may be invoked very frequently.

(3) *bounded complexity*: The complexity of a resource reclaiming algorithm should be *independent* of the number of tasks in the schedule, so that its cost can be incorporated into the worst case computation time of a task.

(4) *effective*: A resource reclaiming algorithm should improve the performance of the system, i.e., increase the guarantee ratio.

The *correctness* requirement addresses the issue of avoiding run time anomalies in a multiprocessor system. Resource reclaiming is straightforward given a uniprocessor schedule because there is only one task executing at any moment on the processor. Resource reclaiming on multiprocessor systems for tasks with resource constraints is much more complicated. This is due to the potential parallelism provided by a multiprocessor system and the potential resource conflicts 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 [8] may occur. These anomalies may cause some of the already guaranteed tasks to miss their deadlines. In particular, one cannot simply use any work-conserving scheme, one that will never leave a processor idle if there is a dispatchable task, without verifying that task deadlines will not be missed [1]. For tasks with precedence constraints, Manacher [10] proposed an algorithm to avoid these anomalies by imposing additional precedence constraints on tasks to preserve the order of tasks which can run in parallel. However, the complexity of the algorithm is not independent of the number of tasks in the schedule and the algorithm does not deal with resource constraints among tasks. Moreover, the primary purpose of the algorithm is to ensure the feasibility of the original schedule in the event of tasks executing less than their worst case computation times in a static system, rather than to dynamically reclaim unused resource.

Predictability is one of the most important issues in a real-time operating system. The system overhead incurred in scheduling, dispatching, and resource reclaiming should not introduce uncertainty into the system. In particular they should not cause already guaranteed tasks to miss their deadlines. Since every task might complete early (i.e., execute less than its worst case computation time), every task might incur resource reclaiming overhead. Hence, the resource reclaiming cost must be *low* (i.e., inexpensive) so that it is insignificant compared to the computation time of a task. Moreover, the entire dispatching cost, which includes the resource reclaiming cost, should be included in the worst case computation time of a task. Consequently, the overheads of a resource reclaiming algorithm must be *bounded* so that its maximum run time cost does not vary. One straightforward approach to resource reclaiming when a task finishes early is to reschedule the entire set of tasks that is remained in the feasible schedule. In practice, this will not be beneficial if the rescheduling cost exceeds the time

[1]See Appendix for a complete description and analysis of the anomalies for the multiprocessor model dealt with in this paper.

reclaimed. Further, most scheduling algorithms have time complexities that depend on the number of tasks to be scheduled, i.e., use of these algorithms for resource reclaiming would result in unbounded overhead costs. Thus a resource reclaiming algorithm which employs rescheduling does not meet the requirements of predictability. One of the challenging issues in designing resource reclaiming algorithms is to reclaim resources with a *bounded* complexity and *low* overhead, in particular, a complexity that is not a function of the number of tasks in the schedule.

In this paper, we study the resource reclaiming problem for multiprocessor systems in which each processor has local memory for task code and private resources. Tasks might also require other non-local resources, such as shared data structures, and communication ports. We present two resource reclaiming algorithms, Basic Reclaiming and Reclaiming with Early Start. These two algorithms employ strategies that are a form of on-line *local optimization* on a feasible multiprocessor schedule. Both of these algorithms have *bounded* time complexity although Reclaiming with Early Start is more expensive to run than Basic Reclaiming. We prove the correctness of these algorithms. To understand the performance impact of these algorithms, we have done extensive simulation studies of the resource reclaiming algorithms for a five processor multiprocessor system. We tested a wide range of task parameters, including different worst case computation times and actual computation times of tasks, task laxities, and task resource usage probabilities. Through simulation results, we demonstrate that

- Low complexity run time local optimization can be very effective in improving the system performance in a dynamic real-time system.
- Using complete rescheduling as a resource reclaiming scheme is not a practical choice.
- It only pays to do resource reclaiming if one can ensure that the overhead cost of the resource reclaiming algorithm is below 10% of tasks' worst case computation times.
- Resource reclaiming can compensate for the performance loss due to the inaccuracy of the estimation of the worst case computation times of real-time tasks.

Further, to demonstrate the applicability of the algorithms, we have implemented the resource reclaiming algorithms in the Spring Kernel [17] — a real-time kernel on a *NUMA* multiprocessor (Non-Uniform Memory Access multiprocessor) system with shared resources. In this paper the important issues in implementing the resource reclaiming algorithms as part of this multiprocessor kernel and the interplay between the scheduler and the resource reclaiming algorithms are also presented.

The remainder of the paper is organized as follows. Section 2 defines our task model, and introduces the terminology used throughout the paper. In Section 3 we study the resource reclaiming problem and present our resource reclaiming algorithms. In Section 4, we apply the resource reclaiming algorithms to dynamic real-time systems with independent tasks, describe the implementation issues on a multiprocessor, and present experimental results. The applicability of the resource reclaiming algorithms to tasks and systems with other characteristics is discussed in Section 5. In Section 6 we summarize the paper.

2 Definitions and Assumptions

In this section we first define the types of real-time tasks and resources considered in this paper. Then we define some of the terminology used. n is the number of tasks $\{T_1, T_2, \ldots\ T_n\}$, m the number of processors $\{P_1, P_2, \ldots\ P_m\}$, and s the number of resources $\{r_1, r_2, \ldots\ r_s\}$.

2.1 Task Model

Tasks are well-defined schedulable entities. A task is not preemptable and, once a task starts execution, it will releases its resources after it completes. Resources that can be required by a task include variables, data structures, memory segments, and communication buffers. Resources can either be used in exclusive mode or shared mode [18]. Two tasks conflict on a resource if both of them need the same resource in exclusive mode, or one of them needs a resource in exclusive mode while the other needs the same resource in shared mode. Two tasks with resource conflict(s) cannot be scheduled in parallel. Each task T_i has the following attributes:

c_i : the worst case computation time of T_i. At scheduling time, this value is known to the scheduling algorithm. But at execution time, a task may have an actual computation time $c_i' \leq c_i$.

d_i : the deadline of T_i;

$\{R_i^j\}$: a resource requirement vector for $1 \leq j \leq s$, denoting the set of resource requirements of T_i; each element of the vector indicates exclusive_use, shared_use, or no_use;

p_q : a processor id for $1 \leq q \leq m$; this is the *processor constraint* attribute of a task. Processor constraints arise because:

(1) in a heterogeneous multiprocessor system, task T_i requires some particular processor p_q, or

(2) in a NUMA (Non-Uniform Memory Access) multiprocessor system, task T_i has been allocated to some processor p_q in order to maximize the potential parallelism and minimize remote memory access.

The processor constraint implies that a task can only be executed on that processor. However, tasks on different processors may share resources; thus all the tasks and their resource needs must be considered together at scheduling time. We assume that every task has a processor associated with it.

2.2 Terminology

The following definitions will be used in the remainder of the paper.

Definition 1: A *feasible* schedule S is a task schedule in which tasks' worst case computation time, resource constraints and processor constraints are all guaranteed to be met. In this paper, we consider nonpreemptive feasible schedules in which a *scheduled start time* (st_i) and *scheduled finish time* (ft_i) are assigned to each task T_i in the schedule such that $\forall i, ft_i \leq d_i$.

Definition 2: Given a feasible schedule S, a *post-run* schedule S' is a layout of the tasks in the same order as they are executed at run time with respect to their actual computation times c'_i, where $\forall i$, $c'_i \leq c_i$. Associated with each task T_i in a post-run schedule S' is a *start time* st'_i and a *finish time* ft'_i. st'_i and ft'_i are the actual times at which T_i starts and completes execution, respectively, and they may be different from st_i and ft_i.

Definition 3: Given a post-run schedule S', a task T_i starts *on-time* if $st'_i \leq st_i$, that is, if the task T_i starts execution by or before its scheduled start time.

Definition 4: A post-run schedule S' is *correct* if $\forall i\ 1 \leq i \leq n$, $ft'_i \leq d_i$.

Lemma 1: If $\forall i\ 1 \leq i \leq n$, T_i starts *on-time* in a post-run schedule S', then S' is *correct.*

Proof: Given nonpreemptive task executions, by definition 3, if T_i starts on time, i.e., $st'_i \leq st_i$, then $ft'_i \leq ft_i \leq d_i$. So the resulting post-run schedule S' is correct. □

This lemma forms the basis for the correctness of our reclaiming algorithms. Note that the lemma gives us a sufficient condition for task starting times. Our reclaiming algorithms will be designed to start tasks *on-time*. As we shall see, this strategy results in reclaiming algorithms that have bounded reclaiming overhead.

We illustrate the terminology introduced above through the following example.

Example:

Table 1 provides the attributes of a set of tasks. Each task requires a processor and some need an additional resource r_1. Figure 1 shows a two processor *feasible* schedule S for this set of tasks. The scheduled start times st_i and scheduled finish times ft_i are given in Table 1. Table 2 and Figure 2 show one of the possible *post-run* schedules S' and the corresponding start times st'_i and finish times ft'_i of the tasks. All the tasks are *on-time* in S'. Hence S' is correct. On the other hand, Table 3 and

Tasks	pid	c_i	c'_i	d_i	r_1	st_i	ft_i
T_1	2	225	125	225	-	0	225
T_2	2	175	100	400	*shared*	225	400
T_3	1	175	150	175	-	0	175
T_4	1	25	25	200	*exclusive*	175	200
T_5	1	150	75	350	-	200	350
T_6	2	100	100	500	-	400	500
T_7	1	150	125	500	*shared*	350	500

Table 1: Task Parameters for **Example 1**.

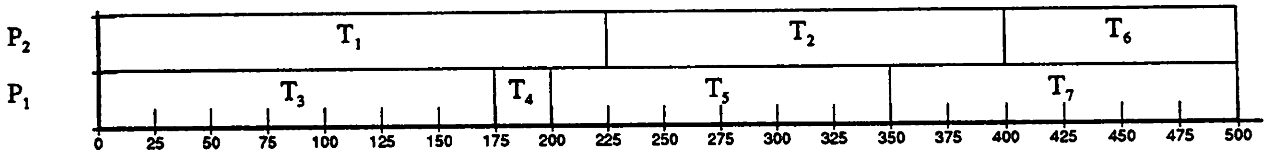

Figure 1: A *feasible* schedule S according to tasks' worst case computation times.

Tasks	T_1	T_2	T_3	T_4	T_5	T_6	T_7
st'_i	0	225	0	175	200	400	350
ft'_i	125	325	150	200	275	500	475

Table 2: Start Times and Finish Times produced by no resource reclaiming.

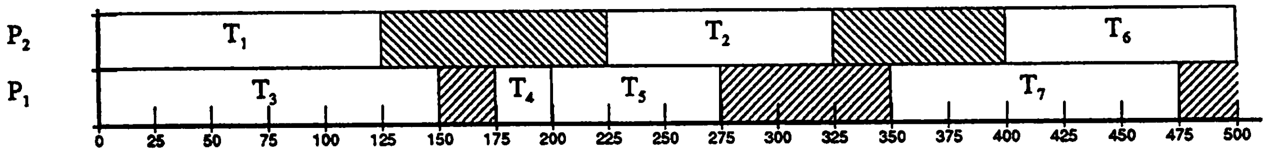

Figure 2: A *post-run* schedule S' when tasks execute only up to their actual computation times and no resource reclaiming is done.

Tasks	T_1	T_2	T_3	T_4	T_5	T_6	T_7
st'_i	0	125	0	225	250	225	325
ft'_i	125	225	150	250	325	325	450

Table 3: Start Times and Finish Times produced by the work-conserving algorithm.

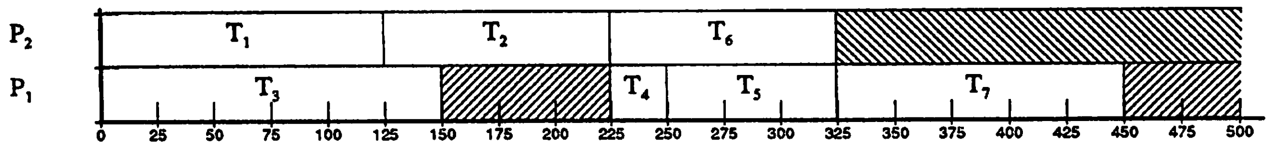

Figure 3: A *post-run* schedule S' produced by a work-conserving algorithm.

Figure 3 demonstrate one of the possible *incorrect post-run* schedules caused by using a work-conserving algorithm. In this *post-run* schedule, T_2 starts execution at time 125 because, as soon as T_1 completes execution, both the resource and the processor that T_2 requires are available. This *work-conserving* action causes task T_4 to eventually miss its deadline. Thus a correct resource reclaiming algorithm must be able to guarantee that this kind of run time anomaly does not occur in a post-run schedule.

3 Resource Reclaiming Algorithms

We first discuss the resource reclaiming problem with respect to its time complexity. Then we present our two resource reclaiming algorithms, (1) Basic Reclaiming and (2) Reclaiming with Early Start.

3.1 Multiprocessor Resource Reclaiming

Since we are working in a dynamic real-time environment, efficiency and predictability are of major concern for the on-line resource reclaiming algorithms. There are two extreme cases that provide the lower and upper bounds on the cost in terms of time.

EXTREME CASE 1. Dispatching tasks strictly according to their scheduled start times (st). This implies no resource reclaiming and, obviously, the cost of resource reclaiming is zero.

EXTREME CASE 2. Total rescheduling of the rest of the tasks in the schedule whenever a task executes less than its worst case computation time. Suppose the cost of a particular scheduling algorithm is $f(n)$ for scheduling n tasks. Then, the cost of total rescheduling would be $O(f(n))$, assuming no new task arrivals. Note that *total rescheduling* can be used only if the cost of this rescheduling is less than the time left unused by a task.

Note that because the resource constrained multiprocessor scheduling problem is NP-complete in the nonpreemptive case [5] and only has high degree polynomial linear programming solutions in the preemptive case [1], any practical scheduling algorithm used in dynamic real-time systems must be approximate or heuristic. This implies that it is not always the case that the same scheduling algorithm will definitely find a feasible schedule when a task is removed from the original set of tasks when the task finishes execution. Thus, even though extreme case 2 provides us with an upper bound on the time complexity of the resource reclaiming problem, it does not represent the *optimal* solution in terms of being able to find feasible schedules whenever they exist. It does provide an indication of the best a system can do in reordering tasks according to available resources.

Clearly, a useful resource reclaiming algorithm should have a complexity less than the total rescheduling extreme, while being just as effective. We distinguish between two classes of resource reclaiming algorithms. One is resource reclaiming with *passing*, and the other is resource reclaiming without *passing*.

Definition 5: A task T_i *passes* task T_j if $st'_i < st'_j$, but $ft_j < st_i$. Thus *passing* occurs when a task T_i starts execution before other task(s) that are scheduled to *finish* execution before T_i was originally scheduled to start.

If T_i is a task in a feasible schedule, then we can divide the rest of the tasks in the schedule into three disjoint subsets with respect to T_i defined as follows:

Definition 6:

$$T_{<i} = \{T_j : ft_j < st_i\}$$

$$T_{>i} = \{T_j : st_j > ft_i\}$$

$$T_{\simeq i} = \{T_j : T_j \notin T_{<i} \ \mathit{and} \ T_j \notin T_{>i}\}$$

Thus, $T_{<i}$ is the set of tasks that are scheduled to finish *before* T_i starts. $T_{>i}$ is the set of tasks that are scheduled *after* T_i finishes. $T_{\simeq i}$ is the set of tasks whose scheduled execution times overlap with the execution time of T_i. For example, in Figure 1, $T_{<5} = \{T_3, T_4\}$, $T_{>5} = \{T_6, T_7\}$, and $T_{\simeq 5} = \{T_1, T_2\}$.

Given a feasible schedule S, if we assume tasks never execute longer than their worst case computation times, and there are no interruptions or arbitrary idle times inserted during the execution of the tasks in S, then we have the following Lemma. This Lemma in essence tells us when run-time anomalies can occur, and will be used in proving the correctness of our resource reclaiming algorithms in the next section.

Lemma 2: Given a feasible real-time multiprocessor schedule S, if $\exists T_i$, such that task T_i does not start *on time* in a post-run schedule, then *passing* must have occurred.

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 $T_{\simeq i}$ have no resource conflicts with T_i, therefore, no matter what order these task were dispatched with respect to the dispatching time of T_i, they would not have delayed the dispatching of T_i. This contradicts the premise that T_i did not start on time. □

A resource reclaiming algorithm that allows *passing* will inevitably incur higher complexity in terms of time than another that does not allow *passing.* This is because *passing* implies altering the ordering of tasks imposed by the feasible schedule, thus is similar to rescheduling. To determine which task in the remaining schedule can utilize an idle period involves searching (since the scheduling problem is in fact a search problem [18]). Any searching will have a complexity of at least $O(\log n)$. Since we are interested in designing resource reclaiming algorithms with *bounded* cost that can be used for dynamic real-time systems, we will concentrate on resource reclaiming algorithms without *passing.*

3.2 Algorithms for Multiprocessor Resource Reclaiming

In this section, we present our two multiprocessor resource reclaiming algorithms, the Basic Reclaiming algorithm and the Reclaiming with Early Start algorithm. Before the details of the algorithms are presented, we would like to motivate the ideas behind the algorithms. Let us reexamine the correct post-run schedule portrayed in Figure 2. Actually, this post-run schedule is a result of no run time resource reclaiming. Notice that between time 150 to 175 all the processors are idle. Clearly, every task in the remaining feasible schedule, i.e., tasks T_2, T_4, T_5, T_6, and T_7, could have been started at least 25 time units earlier than their scheduled start times without in any way jeopardizing the meeting of their deadlines. This is in essence what our Basic Reclaiming algorithm does as illustrated in Figure 9. However, with a more careful inspection of Figures 1 and 9, one can see that we can do even better in utilizing the idle time left in the post-run schedule of Figure 9. For example, T_2 could have started even earlier than in this post-run schedule. In particular, it can be started at time 175 because $T_2 \in T_{\simeq 5}$ (see Definition 6). This can be accomplished if we can in some way represent and utilize the information given in Definition 6. Our second resource reclaiming algorithm, Reclaiming with Early Start, does this and produces the post-run schedule shown in Figure 11.

Now we are in the position to present our resource reclaiming algorithms. The following definitions are needed to describe our resource reclaiming algorithms.

Definition 7: Given a feasible schedule S, a *projection list PL* is an ordered list of the tasks in the feasible schedule, arranged in nondecreasing order of st_i. A projection list is in one-to-one correspondence with some feasible schedule. If $st_i = st_j$ for some tasks T_i and T_j, we place the task with the smaller processor id in the PL first. Thus PL imposes a *total ordering* on the guaranteed tasks.

Definition 8: Given a projection list PL, a *processor projection list PPL_q* is an ordered list of all the tasks processor id P_q in the PL, also arranged in nondecreasing order of st_i, for $1 \leq i \leq n$ and $1 \leq q \leq m$.

Therefore, for the feasible schedule given in Figure 1, the *projection list* of S is $PL = \{T_3, T_1, T_4, T_5, T_2, T_7, T_6\}$. The *processor projection lists* are: $PPL_1 = \{T_3, T_4, T_5, T_7\}$, and $PPL_2 = \{T_1, T_2, T_6\}$.

In the following, we assume the existence of (1) a feasible schedule for n tasks $\{T_1, T_2, ..., T_n\}$, which have been guaranteed with respect to their timing, resource and processor constraints (e.g., using the algorithm presented in [13]), (2) the corresponding *projection list PL* and *m processor projection lists* PPL_1 ... PPL_m, and (3) a scheduled start time st_i, and scheduled finish time ft_i for each task entry T_i in the feasible schedule. We also assume that we can associate a *constant* cost to access the first task in the PL and the first task in each of PPL_q (These assumptions are very practical and easily achievable.).

The resource reclaiming algorithms are presented in pseudo code in Figure 4, 5, 6, and 7. Figure 4 gives the outline of the resource reclaiming algorithms. Recall that resource reclaiming occurs when a task completes, say on processor P_q. There are two steps involved in resource reclaiming. In the first step, the length of the idle time resulting from the early completion of tasks is determined. Details of this step are the same for both the Basic Reclaiming Algorithm and the Reclaiming with Early Start Algorithm. In the second step, the next task in PPL_q is examined to decide whether it can be immediately dispatched. Figure 6 and 7 present **Step2** for each of these algorithms, respectively. In the following, we describe the resource reclaiming algorithms in detail.

- **Step1**: (see Figure 5) Resource reclaiming occurs when a task completes execution and another task is to be dispatched. A task scheduled on processor q is not removed from the PL and PPL_q until it finishes execution. This restriction is important to ensure a consistent view of the amount of time reclaimable. Upon completion of a task, **Step1** tries to identify idle periods on all processors and resources by computing a function $reclaim\_\delta = st_f - current\_time$ (lines 6 to 8 in **Step1**); where st_f is the scheduled start time of the current first task in the PL. The computation complexity of this function is $O(1)$. Since the PL imposes a total ordering on the guaranteed tasks, st_f must be the minimum scheduled start time among all tasks in the schedule, including the one(s) still in execution. Any positive value of $reclaim\_\delta$ indicates the length of the idle period resulting from tasks finishing early. Since a task is removed from the schedule only upon its completion (line 1 in Figure 5), $temp\_reclaim\_\delta$ could have a negative value (if the first task in the PL is still in execution) and, in this case, $reclaim\_\delta$ retains its original value. For example, let us examine Figure 9. At time 125 when task T_1 completes execution, the current first task in the PL is T_3 which is still in execution, and so $temp\_reclaim\_\delta = 0 - 125 = -125$ since $st_3 = 0$ (refer to Table 1 for scheduled start times and scheduled finish times). On the other hand, at time 150 when T_3 finishes execution, T_4 becomes the first task in the PL, and so $temp\_reclaim\_\delta = 175 - 150 = 25$.

- **Step2.BASIC**: (see Figure 6) The idea behind the Basic Reclaiming algorithm is very simple. When a processor completes a task, it checks to see if *all* the processors are idle. If so, the entire schedule can be shifted forward. Now let us be more precise and discuss the pseudo code for the algorithm. We immediately start the execution of the first task T_{r_f} on processor P_r only if the task is the current first task in the PL (i.e., it is the next task in the total order of tasks) or if it has the same st (scheduled start time) as the current first task (line 3 to 4 in Figure 6). Otherwise we compute a function ast_{r_f} for T_{r_f} to decide the *actual start time* (vs. the scheduled

```
/* $m$ — the number of processors */
/* $reclaim\_\delta$ — the amount of time that has been reclaimed. */
/* $reclaim\_\delta$ is set to zero initially. */
/* $T_{q_i}$ — the newly completed task in $PPL_q$ for some processor $q$. */
Algorithm Resource Reclaiming

    Whenever a task $T_{q_i}$ completes execution on a processor $q$, do
    {
        Step1($T_{q_i}$, $reclaim\_\delta$, $PPL_q$);
        if $reclaim\_\delta > original\_reclaim\_\delta$
          then
          {
                for all $r$ such that processor $r$ is idle do
                    Step2.{BASIC|EARLYSTART}($r$, $reclaim\_\delta$, $PL$, $PPL_1, \dots, PPL_m$);
          }
    }

end Algorithm Resource Reclaiming
```

Figure 4:

```
Step1 ($T_{q_i}$, $reclaim\_\delta$, $PL$, $PPL_q$);

/* Task $T_{q_i}$ just completed execution on processor $q$.*/
1.  REMOVE($T_{q_i}$, $PL$, $PPL_q$);
2.  $T_f \leftarrow$ the first task in the current $PL$;
3.  if ($current\_time < (ft_{q_i} - reclaim\_\delta)$)
4.     then
5.     {
6.          $temp\_reclaim\_\delta = st_f - (current\_time)$;
7.          if $temp\_reclaim\_\delta > 0$
8.             then $reclaim\_\delta \leftarrow temp\_reclaim\_\delta$;
9.          end if
10.    }
11. end if

end Step1
```

Figure 5:

```
Step2.BASIC ($r$, $reclaim\_\delta$, $PL$, $PPL_1$, ... , $PPL_m$);

1.  $T_f \leftarrow$ the first task in the current $PL$;
2   $T_{r_f} \leftarrow$ the first task in the current $PLL_r$;
3.  if ($T_{r_f} == T_f$)
      or ($st_{r_f} == st_f$)
4.    then startexecution($T_{r_f}$);
5.    else
6.    {
7.          $ast_{r_f} = st_{r_f} - reclaim\_\delta$;
8.          pend($T_{r_f}$,$ast_{r_f}$);
9.    }
10. end if

end Step2.BASIC
```

Figure 6:

```
Step2.EARLYSTART (r, $reclaim\_\delta$, $PL$, $PPL_1$, ... , $PPL_m$);

1.  $T_f \leftarrow$ the first task in the current $PL$;
2.  $T_{r_f} \leftarrow$ the first task in the current $PLL_r$;
3.  $can\_start\_early \leftarrow$ true;
4.  if $st_{r_f} \neq st_f$
5.    then
6.    {
7.          $q \leftarrow 0$;
8.          while ($can\_start\_early$ and $q < m$) do
9.          {
10.                 $q \leftarrow q + 1$;
11.                 if ($q \neq r$) and ($st_{r_f} > ft_{q_f}$)
12.                   then $can\_start\_early \leftarrow$ false;
13.                 end if
14.         }
15.   }
16. endif
17. if $can\_start\_early$
18.   then startexecution($T_{r_f}$);
19.   else
20.   {
21.         $ast_{r_f} = st_{r_f}$ - $reclaim\_\delta$;
22.         pend($T_{r_f}$,$ast_{r_f}$);
23    }
24. end if

end Step2.EARLYSTART
```

Figure 7:

start time given in the schedule) for it, taking into consideration the idle periods that have been accumulated up to now. This function is $ast_{r_f} = st_{r_f} - reclaim\_\delta$, where st_{r_f} is the original scheduled start time of task T_{r_f}. This function is also $O(1)$. Once this function is computed, processor r will pend until (1) either the calculated ast_{r_f} has arrived, or (2) some other task finishes early and $reclaim\_\delta$ is incremented. In the latter case, **Step2.BASIC** will be invoked again (see Figure 4).

- **Step2.EARLYSTART**: (see Figure 7) Notice that the Basic Reclaiming algorithm will start a task early by an amount of time equal to $reclaim\_\delta$ which is the length of time that all the processors can reclaim. The Reclaiming with Early Start algorithm dispenses with this requirement. It allows a task T_{r_f}, the first task in PPL_r, to start as long as the first task T_{q_f} in each of the other PPL_q does not conflict over any resources with T_{r_f} and no *passing* will occur. More precisely, T_{r_f} can start if for $1 \leq q \leq m$ and $q \neq r$, T_{q_f} is either in $T_{\simeq r_f}$ or in $T_{>r_f}$. Now let us define that a task T_{r_f} is being *early started* if $st'_{r_f} < st_{r_f} - reclaim\_\delta$. In Reclaiming with Early Start, we first compute (line 8 to 14) a Boolean function $can\_start\_early = st_{r_f} < ft_{q_f}$, $\forall q$ such that $q \neq r$ and $1 \leq q \leq m$, where st_{r_f} is the scheduled start time of the first task on processor r and ft_{q_f} is the scheduled finish time of the first task on processor q. This function identifies parallelism between the first task on processor r and the first tasks on all other processors by checking to see whether the first tasks on all other processors are in $T_{\simeq r_f}$ (see Definition 6). That is, for any two tasks T_{r_f} and T_{q_f}, if $st_{r_f} < ft_{q_f}$, then $T_{t_f} \in T_{\simeq q_f}$. The complexity of this function is $O(m)$. The task will be dispatched if the value of the Boolean function is true. Only when the value of the function can_start_early is false, we will compute the ast_{r_f} for task T_{r_f} as in **Step2.BASIC**.

For both algorithms, whenever a positive value of $reclaim\_\delta$ is obtained in **Step1**, **Step2** must be executed for all currently idle processors. Thus the complexity of the basic version is: $O(1) + m * O(1)$ $= O(m)$, while Reclaiming with Early Start has a complexity of $O(1) + m * O(m) = O(m^2)$.

These two resource reclaiming algorithms are based on the idea that a feasible multiprocessor schedule provides task ordering information that is *sufficient* to guarantee the timing and resource requirements of tasks in the schedule. If two tasks T_i and T_j are such that $T_j \in T_{\simeq i}$ (see Definition 6) in a schedule, then we can conclude that no matter which one of them will be dispatched first at run time, they will never jeopardize each other's deadlines. On the other hand, if $T_j \in T_{<i}$ or $T_j \in T_{>i}$, we cannot make the same conclusion without re-examining timing and resource constraints or without total re-scheduling. Assume each task T_i is assigned a scheduled start time st_i and a scheduled finish time ft_i in the given feasible schedule, our resource reclaiming algorithms utilize these two task attributes to infer the information in Definition 6 at run time, i.e., to identify tasks in $T_{\simeq f}$ where T_f is the first task in PL, and to reclaim unused resources using these tasks. Thus our resource reclaiming algorithms perform *local* optimization, optimization that is *local* to the T_f portion of a feasible schedule S. By doing so, we do not have to explicitly examine the availability of each of the resources needed by a task in order to dispatch a task when reclaiming occurs. This keeps the complexity of the algorithms independent of the number of tasks in the schedule and the number of resources in the system — a desirable property of any algorithm that has to be used in dynamic real-time systems at run time.

3.3 Properties of the Resource Reclaiming Algorithms

The two resource reclaiming algorithms just presented guarantee that run time anomalies as shown at the end of Section 2 will not occur. In the section we shall illustrate the two resource reclaiming algorithms through an example and prove the correctness of the algorithms in this section. We also discuss some interesting aspects of the algorithms.

3.3.1 Discussion Through An Example

Assume we have the same feasible schedule in Figure 1 for the set of tasks defined in Table 1. The post-run schedule produced by the Basic Reclaiming Algorithm is shown in Figure 9 and the post-run schedule produced by the Reclaiming with Early Start Algorithm is shown in Figure 11. We show the values of *reclaim_δ* at the time of each task completion in Figures 8 and 10 for the two algorithms respectively[2]. Figure 2 is the post-run schedule when no resource reclaiming is done. Thus from Figures 2, 9, and 11, one can see the effects of resource reclaiming.

Note that once the new value of *reclaim_δ* is determined in **Step1**, *every* task T_i in the rest of the schedule can in fact be started *reclaim_δ* time units earlier than its st_i, e.g., at time 150 when T_3 completes execution, T_4 can start execution (see Figures 9 and 11). This is equivalent to a time translation of *reclaim_δ* units of time on the remaining feasible schedule, i.e., the st_i and ft_i of every task T_i in the remaining feasible schedule can be translated to st_i − *reclaim_δ* and ft_i − *reclaim_δ*. However, we do not explicitly carry out this time translation in the remaining feasible schedule because we will incur a time complexity of $O(n)$ to modify the st_i and ft_i of each task, thus violating our *boundedness* premise.

From the description of the algorithms, it seems obvious that Reclaiming with Early Start should be more effective than Basic Reclaiming. However, there are two interesting aspects of the Reclaiming with Early Start Algorithm that are not easily seen.

[2]Note that although there is no task completion at time 300 in Figure 11, we include the value of *reclaim_δ* in Table 10 for comparison purposes.

time	0	125	150	175	250	300	425	450
reclaim_δ	0	0	25	25	25	50	50	50

Figure 8: The values of *reclaim_δ* at each task completion when the Basic Reclaiming Algorithm is used.

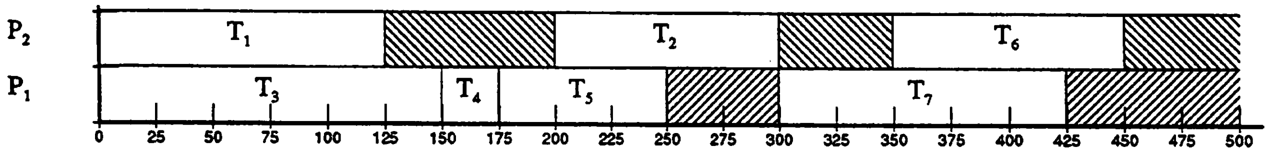

Figure 9: The *post-run* schedule S' produced by the Basic Reclaiming Algorithm.

time	0	125	150	175	250	275	300	375
reclaim_δ	0	0	25	25	25	25	25	125

Figure 10: The values of *reclaim_δ* at each task completion when *early start* is allowed.

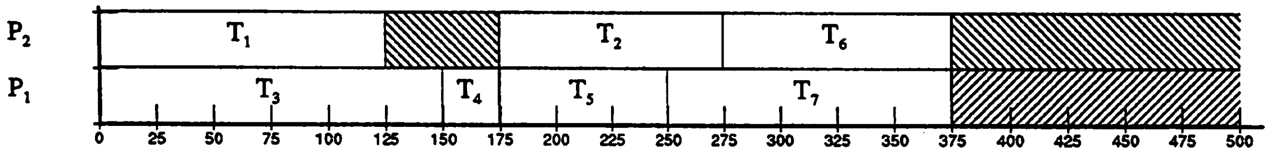

Figure 11: The *post-run* schedule S' produced by the Reclaiming with Early Start Algorithm.

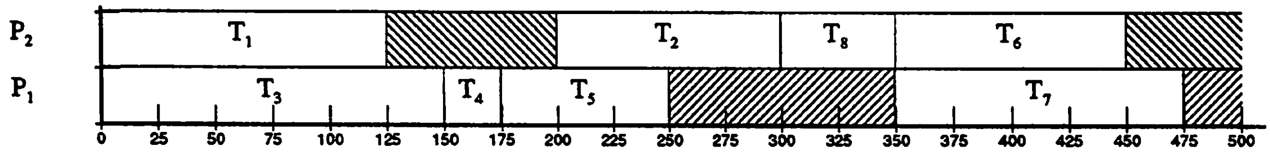

Figure 12: The *post-run* schedule S' produced by the Basic Reclaiming with the addition of T_8.

- First, Reclaiming with Early Start does not necessarily accumulate a larger value of *reclaim_δ* in the *short term.* For example, compare the values of *reclaim_δ* at time 300 in Figures 8 and 10. The value of *reclaim_δ* from using Basic Reclaiming is larger than from using Reclaiming with Early Start at time 300, even though at time 375, the converse is true. This is because *reclaim_δ* reflects the time reclaimed on *all* processors and resources. In general Reclaiming with Early Start keeps the processors and resources busier than Basic Reclaiming does. So when using Reclaiming with Early Start, *temp_reclaim_δ* might be found to be positive *less* frequently in **Step1**. But in the long run, such as by time 375, Reclaiming with Early Start can have a large value of *reclaim_δ*.

- Second, since we are dealing with dynamic real-time systems, tasks can arrive stochastically. Whether a task can be feasibly scheduled depends very much on the particular time the task arrives at the system, i.e., the current system state including the number of tasks and their worst case requirements, and which tasks are already in execution. Therefore, even though Reclaiming with Early Start can *eventually* have a larger value of *reclaim_δ*, it does not outperform the Basic Reclaiming algorithm with respect to guaranteeing dynamic task arrivals at *every* task arrival instance. This is because starting the execution of a task as early as possible is not necessarily always the best choice in a system with nonpreemptive scheduling and dynamic arrivals. For example, assume we have the same feasible schedule as in Figure 1 and, for the ease of explanation, let us assume scheduling occurs instantaneously. If a task T_8 arrives at time 300 with $c_8 = c'_8 = 50$, $d_8 = 375$, $p_8 = 2$ and $R^1_8 = \textit{exclusive}$ (i.e., having a resource conflict with T_7), a system using the Basic Reclaiming algorithm will be able to feasibly schedule T_8 as shown in Figure 12, while a system using the Reclaiming with Early Start will not be able to schedule T_8 (since T_6 and T_7 are already in execution). Thus, through experimental studies, we need to examine the effectiveness of Reclaiming with Early Start with respect to Basic Reclaiming.

3.3.2 Correctness

In the following, we shall prove that the two resource reclaiming algorithms presented in this section are *correct*, that is, they will not cause the type of run time anomalies discussed in Section 2.

Theorem 1: Given a feasible multiprocessor schedule S with resource and processor constraints, the Basice Reclaiming Algorithm will produce a correct post-run schedule.

PROOF. By Lemma 1, we only have to prove that all tasks start *on-time* in the post-run schedule produced by the Basic Reclaiming Algorithm.

By Definition 3, if tasks are dispatched according to their *st* in the feasible schedule, they all start *on-time*. We only have to observe that the value of *reclaim_δ* in **Step1** reflects the idle time units on all resources and processors. Therefore, for $reclaim\_\delta > 0$, we can have a time translation of $reclaim\_\delta$ units of time (i.e., time moved forward) on the portion of the feasible schedule remaining to be dispatched. Since the feasible schedule remains feasible under time translation, and since **Step2.BASIC** dispatches every task at $st'_i = st_i - reclaim\_\delta$, it follows that the tasks in the post-run schedule produced by the Basic Reclaiming Algorithm must have been started *on-time*. □

Theorem 2: Given a feasible multiprocessor schedule S with resource and processor constraints, the post-run scheduled produced by the Reclaiming with Early Start Algorithm is correct.

PROOF. We shall prove that *passing* does not occur when Reclaiming with Early Start is used. Then by Lemma 2, we know that all tasks start *on time*.

We prove this by contradiction. Consider a task T_j to be dispatched in **Step2.EARLYSTART**. Suppose $\exists$ T_i such that T_j were dispatched at some time $st'_j < st_i$ while $st_j > ft_i$. This implies that T_j *passed* T_i. But this is impossible; because if $st_j > ft_i$, *can_start_early* would have become false in line 12 of **Step2.EARLYSTART**, and hence T_j would not have been dispatched. □

4 An Application of the Resource Reclaiming Algorithms

In many real-time applications, the system is required to execute tasks in response to external events and signals. To improve the guarantee ratio (the number of tasks guaranteed / the number of tasks arrived) of tasks, the resource reclaiming algorithms presented in the last section can be used. In this section, we shall be concerned with the application of the resource reclaiming algorithms to such a real-time operating system kernel [17] and demonstrate the effectiveness of the algorithms through simulation results in the next section.

4.1 Concurrent Implementation of Resource Reclaiming Algorithms in a Multiprocessor System

Both resource reclaiming algorithms have been implemented in the Spring Kernel [17]. In this section, we discuss the important issues in implementing the resource reclaiming algorithms in a *NUMA* multiprocessor (Non-Uniform Memory Access multiprocessor) system with shared resources, and the interplay between the scheduler and the resource reclaiming algorithms. Recall that we are dealing with real-time tasks with resource constraints, thus there exists an integrated schedule for all the processors on a multiprocessor. There are two different ways to implement a resource reclaiming algorithm on a multiprocessor system — *centralized* and *concurrent*. In a centralized scheme, the algorithm can be implemented by a single reclaiming daemon process. In a concurrent scheme, each processor will do its own reclaiming and all the processors in the multiprocessor system can be concurrently reclaiming unused time as tasks complete execution. We have taken the concurrent approach in the Spring Kernel. This choice has two major merits. First, the parallelism provided by a multiprocessor can be more effectively explored with a concurrent implementation. Second, solutions developed for the concurrent approach can be implemented on either NUMA or symmetric shared-memory multiprocessors efficiently, but this is not true for solutions developed for the centralized approach which in general assumes a symmetric shared-memory multiprocessor architecture.

Since predictability and consistency are two important issues and are difficult to maintain in a dynamic system, in the following discussion, we concentrate on how to achieve boundedness in terms of overhead cost, and how to achieve data consistency in this concurrent implementation.

Parallel Execution of the Scheduler and Guaranteed Tasks

In order to maximize the potential parallelism provided by multiprocessor systems, the Spring Kernel supports the concurrent execution of application tasks and the scheduling algorithm. This is accomplished by using one processor on a multiprocessor node as the system processor to offload task scheduling and other operating system overhead, while using the remaining processors to execute guaranteed application tasks. The scheduler on the system processor is responsible for dynamically producing a feasible schedule for the multiprocessor as tasks arrive. There is a *dispatcher process* on each application processor. Effectively, whenever a task completes, this dispatcher process executes steps 1 and 2 of the reclaiming algorithm. Thus, reclaiming occurs concurrently on the application processors.

Figure 13 illustrates the scheme we use to schedule dynamic task arrivals with resource reclaiming. GUARANTEE uses the heuristic scheduling algorithm proposed in [13]. To achieve concurrent execution of application tasks and the scheduler while maintaining the predictable of the feasible schedule, at each task arrival, a time line called the *cut_off_line* is calculated in the existing feasible schedule based on the time cost of the scheduling algorithm in use. In order to bound the cost of running the scheduler, we set a value N as the maximum number of tasks that the scheduler will schedule at a time. So the maximum value of the *cut_off_line* is capped by a value $current\_time + max_{SC}$. Any task T_i with $st_i - reclaim\_\delta < cut\_off\_line$ in the schedule will not be considered in the re-scheduling process. This ensures that the scheduling algorithm can execute in parallel with application tasks. The details of this concurrent implementation can be found in [11].

Parallel Execution of the Dispatchers

To ensure consistency of *reclaim_δ*, **Step 1** of the algorithms must be within a critical section . To bound the cost of this mutual exclusion among m processors, we used the predictable multiprocessor synchronization mechanisms developed in [12]. Therefore, the time complexity of *Step 1* for each task completion now becomes $O(mC)$, where C is the critical section size. Step 2 of the algorithm does

Scheduler

```
Whenever a task $T_i$ arrives, do
{
    $reclaim\_\delta \longleftarrow$ the amount of time that has been *reclaimed*;
    • Calculate the run time cost $SC$ of the scheduling
    algorithm based on the number of tasks in
    the current $PL$ plus the new task arrival;
    • $cut\_off\_line = current\_time + SC$;
    • $\mathcal{T}_{nr} \longleftarrow \{T_j | st_j - reclaim\_\delta < cut\_off\_line\}$;
    • Calculate the earliest available time
    of each resource and processor,
    based on the resource and processor requirements,
    and $ft_j$ of the tasks $T_j$ in $\mathcal{T}_{nr}$, and the value of $reclaim\_\delta$;
    • $\mathcal{T}_r \longleftarrow \{T_j | st_j - reclaim\_\delta > cut\_off\_line\}$;
    • GUARANTEE($\mathcal{T}_r$, $T_i$);
}
```

Figure 13: Scheduling Dynamic Real-Time Tasks with Resource Reclaiming

not involve any locking. As the overheads of reclaiming are *bounded*, predictability of the system is maintained even with reclaiming.

Multiple Invocations of the Scheduler

When a new task arrives, its worst case computation time, deadline, and resource and processor requirements are assumed to be known. The system will try to guarantee the new task arrival together with all the tasks T_i, in the original feasible schedule, for which $st_i - reclaim\_\delta > cut\_off\_line$. With the knowledge of the value of $reclaim\_\delta$, i.e., the amount of time that has been reclaimed on all resources and processors, those tasks T_i with $st_i - reclaim\_\delta < cut\_off\_line$ will finish at least $reclaim\_\delta$ time units earlier than their scheduled finish time ft_i. Thus, in calculating the earliest available time of resources and processors in trying to schedule the new task arrival in Figure 13, the scheduler takes the current value of $reclaim\_\delta$ into consideration.

If the new task arrival is guaranteed, the newly generated feasible schedule S_{new} must be *appended* to the original feasible schedule at the *cut_off_line*. Since the scheduler's cost SC is the scheduler's worst case computation time, it is very likely that there are still tasks in the original feasible schedule before the *cut_off_line* at the time when the scheduler finishes scheduling. Meanwhile, $reclaim\_\delta$ will be continuously updated by the resource reclaiming algorithm. Let us call the value of $reclaim\_\delta$ that has been updated since the new scheduling instance occurred $reclaim\_delta'$. Thus for the tasks that are in the section of the feasible schedule before the *cut_off_line*, the value of $reclaim\_\delta'$ is valid. However, for the tasks that are in the section of the feasible schedule produced after the *cut_off_line*, the *reclaim_δ* portion of the value of *reclaim_δ'* has already been taken into consideration in calculating the tasks' scheduled start times. Moreover, there can be more than one *cut_off_line* in a feasible schedule since more than one task can arrive, causing the scheduler to be invoked multiple times during the execution of a feasible schedule. We must develop a protocol to maintain the correct view of the value of *reclaim_δ* between the tasks that are before and that are after each of the *cut_off_lines*, i.e., between any two portions of the current feasible schedule that have been constructed at two different scheduling instances. Otherwise, inconsistent usage of the value of *reclaim_δ* may result in incorrect post-run schedules.

To handle this problem, we have designed a protocol. Due to space limitations, we present only a simplified version of this protocol in the following. See [15] for a complete description and correctness analysis of this protocol.

- Each task T_i in the feasible schedule has a *reset_δ* field.

- The value of this field is zero for all tasks except for the task T_{f_k} which is the first task in the total ordering PL for S_{new_k}, where S_{new_k} is the section of the feasible schedule produced by the kth invocation of the scheduler. *reset_δ*(T_{f_k}) is set to be equal to the value of *reclaim_δ* that has been assimilated by the kth invocation of the scheduler.

- As soon as T_{f_k} is dispatched, *reclaim_δ* = *reclaim_δ* − *reset_δ*(T_{f_k}).

This protocol ensures the correct view of the value of *reclaim_δ* throughout a feasible schedule at any time. One may be tempted to adopt a conceptually simpler protocol, one that explicitly modifies the st_i and ft_i of all the tasks after the *cut_off_line* by the amount of *reclaim_δ* − *reset_δ*(T_{f_k}) at the end of each scheduler's invocation. The *drawback* to this protocol is that its run time cost is $O(n)$ and *reclaim_δ* must be locked while this protocol is in progress to avoid race conditions between the scheduler and the dispatchers. This means that the dispatchers may have to wait for an amount of time that is $O(n)$, i.e. not *bounded.* So this is not acceptable.

4.2 Experimental Results

To evaluate the performance of the resource reclaiming algorithms and to study the tradeoff between system overhead costs and runtime savings due to resource reclamation, we present experimental results in this section. Since it is difficult to collect elaborate performance statistics without affecting the true performance of the actual Spring Kernel, we have implemented our resource reclaiming algorithms not only on the Spring Kernel, but also on a software simulator which simulates the multiprocessor Spring Kernel. In our simulations, the system overhead costs are the worst case costs measured on the Spring Kernel. The scheduler's cost SC is calculated before each invocation of the scheduler as follows: SC

= *overhead_cost* + n * *per_task_cost*, where *overhead_cost* is the portion of the scheduler's cost that is constant for each invocation and *per_task_cost* is the portion of the cost dependent on the number of tasks in the schedule. As mentioned in the previous section, in order to bound the cost of running the scheduler, we set a value N as the maximum number of tasks that the scheduler will schedule at a time, i.e., $n \leq N$ in calculating SC. The worst case values of *overhead_cost* and *per_task_cost* are 4 milliseconds and 5 milliseconds, respectively, unless specified otherwise. The run time cost of Basic Reclaiming and Reclaiming with Early Start are 1 and 2 milliseconds, respectively. In all the experiments, whenever the resource reclaiming algorithms are used, the cost of the algorithms are added onto a task's worst case computation time before the task is scheduled.

We present simulation results for a five processor system. There are five additional resources in all the simulations. Tasks' worst case computation times are uniformly distributed between *wcc_min* and *wcc_max*. We have tested two cases for *wcc_min* and *wcc_max*. One is *wcc_min* = 50 and *wcc_max* = 150. The other is *wcc_min* = 50 and *wcc_max* = 1000. These two cases represent the two kinds of task systems in which the worst case computation times of tasks have small/large variance. We have found that in most cases, the performance of the resource reclaiming algorithms is almost the same for both cases of tasks' worst case computation times. We also present results for which we linearly increase the value of *wcc_min*, thus causing the ratio of the cost of resource reclaiming to the average worst case computation time among tasks to decrease. The actual computation time of a task is uniformly distributed between 50% and 90% of its worst case computation time in all cases except where specified differently. The laxity of a task is calculated based on the worst case computation time of the task, and it is uniformly distributed between 9 to 10 times the worst case computation time in all cases except specified otherwise. A task requires any resource with probability P_{use}. If a task requires a particular resource, it uses the resource in shared/exclusive mode with probability 0.5.

The combination of the mean interarrival time $\frac{1}{\lambda}$ of tasks, the value of P_{use}, the number of resources S, and *wcc_min* and *wcc_max* determines the average load of the system. In our simulation, tasks arrive as a Poisson process. Every processor has the same $\frac{1}{\lambda_i}$, for $1 \leq i \leq m$. We use the following three formulas to measure the average processor load L_{pi}, the average resource load L_{ri}, and the resource conflict probability P_c for two tasks.

$$\begin{aligned} L_{pi} &= \lambda_i * E[wcc] \\ L_{ri} &= P_{use} * \lambda_i * E[wcc] * m \\ P_c &= 1 - (2(1 - P_{use}) * P_{use} + (1 - P_{use})^2 + (0.5 * P_{use})^2)^S \end{aligned}$$

$E[wcc]$ is the expected value of the worst case computation time of a task; thus it is either 100 or 525 for the two kinds of worst case computation times in our simulations. m is the number of processors and T is the number of tasks in a schedule. The first two formulas are straightforward. Note that the average resource load L_{ri} goes up as P_{use} increases even if the expected worst case computation time $E[wcc]$ and the mean arrival rate λ_i stays the same. In the third formula, P_c is the probability that two tasks will conflict on *any* of the given S resources (vs. P_{use} which is the probability that a task will require a resource). Thus P_c is a measure of the resource conflicts in a task load. In order to simulate task arrivals that have sufficient parallelism to be run on a multiprocessor system, we must keep the value of P_c fairly low. A high value of P_c would indicate the inherent resource conflicts among many tasks. P_c is calculated as 1 minus the probability that the two tasks will not conflict on any of the S resources. With respect to any one of the S resources, the first term in the summation is the probability that only one task will require the resource, the second term is the probability that none of the two tasks will require the resource, and the third term is the probability that both tasks will require it in shared mode (i.e., no resource conflict). P_c increases when the value of P_{use} or the value of S increases. So if we keep P_{use} the same for all the tasks, the more resources there are in a system, the more resource conflicts tasks will have.

The heuristic scheduling algorithm proposed in [13] is used in the scheduler in all of our simulations presented in this section. The performance metric we use is the *guarantee ratio* of an algorithm with respect to dynamic task arrivals. The *guarantee ratio* is defined as $\frac{\text{the number of tasks guaranteed}}{\text{the number of tasks arrived}}$. In all the simulation experiments, each data point consists of ten runs. Our requirement on the statistical data is to generate 95% confidence intervals for the guarantee ratio whose width is less than 5% of the point estimate.

To evaluate the effectiveness of the proposed resource reclaiming algorithms, we have also implemented the following three schemes for comparison purposes:

- **guarantee with actual computation time**: This is an ideal scheduling scenario. In this scheme, when a task arrives, the scheduler omnisciently knows the *actual*, rather than the *worst case*, computation time of the task. Therefore, resource reclaiming is not necessary.

- **rescheduling**: In the rescheduling scheme, whenever a task executes less than its worst case computation time, the scheduler is invoked to reschedule the tasks in the existing schedule in the same manner as when a new task arrives. The scheduler is invoked to do resource reclaiming only if the difference between the worst case computation time and the actual computation time of the completed task is greater than the scheduler's cost.

- **no resource reclaiming**: Here no resource reclaiming is done. Tasks are dispatched according to their scheduled start times. The case of no resource reclaiming provides a lower bound on performance.

A. *Performance comparison of the two resource reclaiming algorithms*

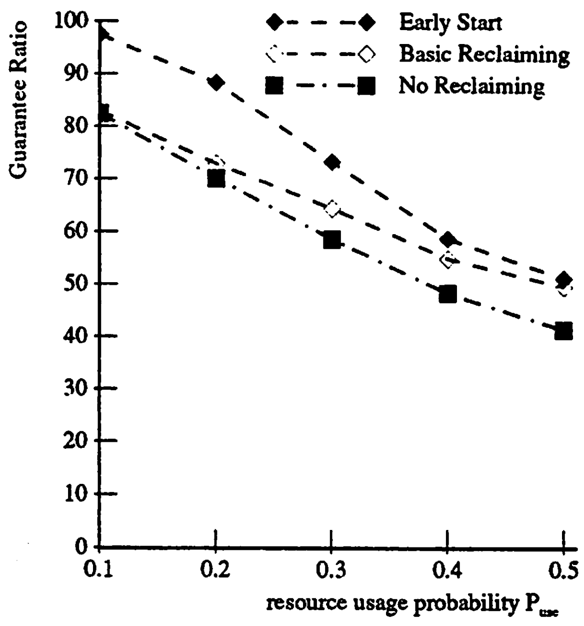

Figure 14: Performance of Basic Reclaiming and Reclaiming with Early Start

In this section, we compare the performance of the two resource reclaiming algorithms with no resource reclaiming. In Figure 14, L_{pi} = 0.75 and P_{use} varies from 0.1 to 0.5. This represents a heavy to overloaded system. For example, when P_{use} is 0.3, L_{ri} is 1.13, and when P_{use} is 0.5, L_{ri} is 1.9. Reclaiming with Early Start is very effective for all the resource usage probabilities. Its guarantee ratio is 18.4% higher than that of no resource reclaiming when P_{use} = 0.2. When the resource conflict is small (i.e., when $P_{use} \leq 0.3$ and thus $P_c \leq 0.3$), Reclaiming with Early Start performs much better than Basic Reclaiming since it can exploit more parallelism. When the value of P_{use} is 0.5, the performance of the Basic Reclaiming algorithm approaches that of Reclaiming with Early Start. When the value of P_{use} is too high, P_c is even larger, indicating high resource conflicts among tasks, thus little parallelism among tasks. For example, for P_{use} = 0.5, P_c = 0.65. In this case there is a very high probability that any two arriving tasks will have resource conflicts. This will result in schedules in which very few tasks can be run in parallel. Since in using a multiprocessor system, one would expect certain levels of parallelism to exist among the tasks, it is more appropriate to keep the value of $P_{use} \leq 0.3$ (thus, $P_c \leq 0.3$) in the rest of our experiments. From the above results, we see that Reclaiming with Early Start does outperform Basic Reclaiming in most of the cases. Thus in the following experiments, we concentrate on evaluating the performance of Reclaiming with Early Start.

B. *Performance Comparison with Rescheduling*

The scheduler has a more *global* view of the tasks in the schedule than the resource reclaiming algorithm does, but it also has a higher run time cost. The purpose of this study is to answer the

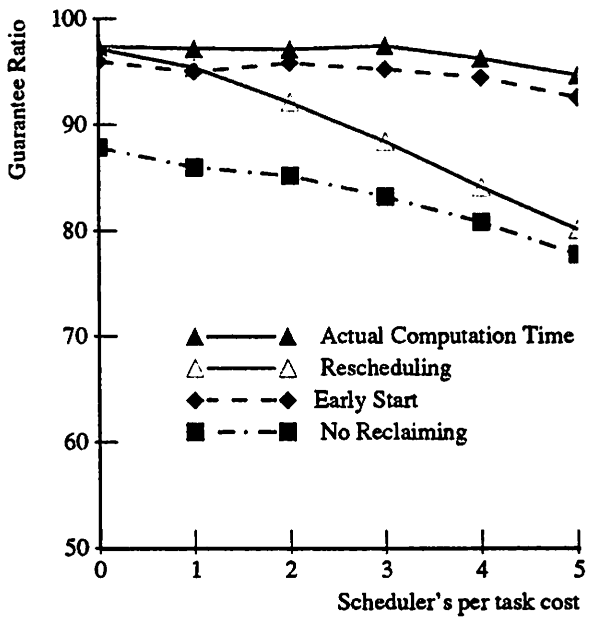

Figure 15: Effects of Scheduler's Runtime Cost

following question: 'Suppose we can reduce the cost of the scheduler, will the rescheduling scheme be a better choice?' We compare the performance of the rescheduling scheme with that of (1) the guarantee with actual computation time, (2) Reclaiming with Early Start, and (3)the no reclaiming schemes. Here we artificially vary the scheduler's *per_task_cost* from 0 to 5, where 5 is the actual worst case cost we have measure on the Spring Kernel. The task loads simulated have $L_{pi} = 1.0$ and $P_{use} = 0.2$.

The simulation results in Figure 15 indicate that the performance of rescheduling degrades 17.1% when the cost of the scheduling algorithm increases from 0 to 5. Only when the scheduler's *per_task_cost* is zero, does rescheduling perform better than Reclaiming with Early Start. In real systems, the cost of the scheduler will be nonzero. So the rescheduling scheme is not a practical choice. The performance of Reclaiming with Early Start is very close to the performance of the guarantee with actual computation time scheme no matter how the cost of the scheduler changes. This demonstrates that low complexity run time local optimization, such as the one used in Reclaiming with Early Start, can be very effective in a dynamic real-time system.

C. *Effects of Task Laxity*

We now examine the performance of the various schemes with respect to different task laxities. Figure 16 shows the results of the experiments in which $P_{use} = 0.2$ and $L_{pi} = 1.0$. Here tasks' laxities are plotted along the X-axis. At each x point, a task's laxity is drawn from a uniform distribution between $x\% * wcc$ and $x + 100\% * wcc$, where wcc is the average worst case computation time of tasks. With tight task laxities, e.g., $x \leq 200$, resource reclaiming is not very effective, since, in this case,

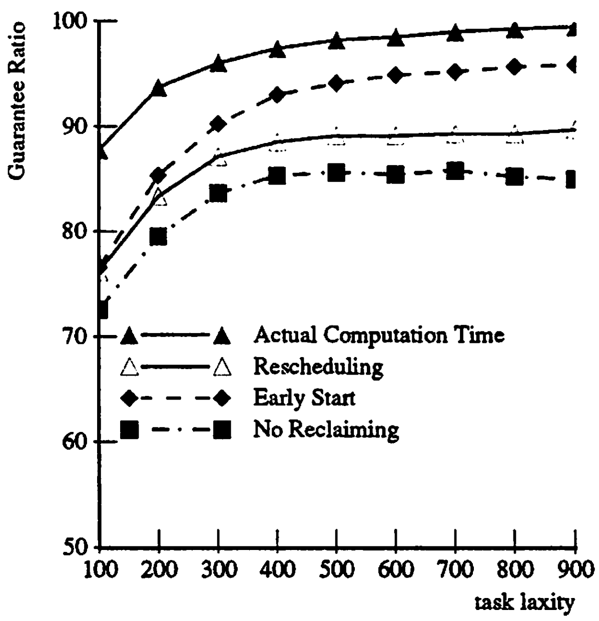

Figure 16: Effects of Task Laxity

tasks arrive at the system with very small laxities, thus many of them cannot even be guaranteed. As the laxities of the tasks are relaxed, the performance of Reclaiming with Early Start approaches the performance of the guarantee with actual computation time scheme, and is much better than that of rescheduling and no resource reclaiming. At $x = 900$, the difference between the guarantee ratios of using Reclaiming with Early Start and of using no resource reclaiming is 11%. On the other hand, rescheduling performs as well as Reclaiming with Early Start only when the laxity is very tight, i.e., when $x = 100$. It performs poorly as the laxity increases. The more tasks there are in the feasible schedule, the more rescheduling will cost. With larger task laxities, more tasks can be guaranteed, thus the feasible schedule contains more tasks. We have found that resource reclaiming is most effective when there are tasks to be dispatched continuously from the schedule.

D. *Effects of Worst Case Computation Time*

In Figure 17, we compare the performance of Reclaiming with Early Start with no reclaiming with respect to different worst case computation times. As the worst case computation times of tasks increase, the ratio $\frac{\text{resource reclaiming cost}}{\text{worst case computation time}}$ decreases. Recall that the run time cost of Reclaiming with Early Start is 2 (milliseconds). So for the two kinds of worst case computation times we have tested so far, i.e., uniformly distributed between (50, 150) and between (50, 1000), the resource reclaiming overhead cost is at most 0.4% of a task's worst case computation time (since the minimum worst case computation time $wcc\_min = 50$ in both cases and $2/50 = 0.4$). What happens to the

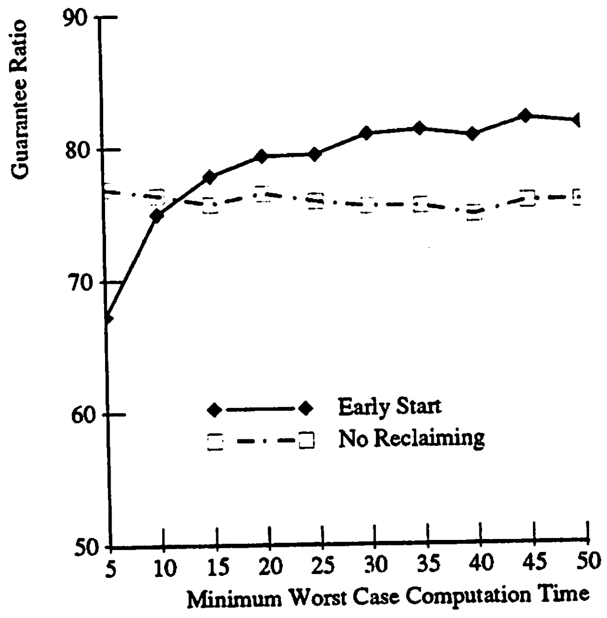

Figure 17: Effects of WCC to Resource Reclaiming Cost Ratio

performance of resource reclaiming if *wcc_min* is smaller so that the ratio of the resource reclaiming overhead to the minimum worst case computation time becomes larger? In this experiment, we varied *wcc_min* from 5 to 50, and the worst case computation time of a task is uniformly distributed between *wcc_min* and 2 * *wcc_min*. The average processor load L_{pi} is 1.0 and P_{use} is set to 0.3. We did not include any scheduling overhead in this experiment for the purpose of examining the *pure* effects of the resource reclaiming overhead costs. In Figure 17, we plot the values of *wcc_min* on the X-axis. When *wcc_min* = 5, the resource reclaiming overhead ranges from 20% to 40% of tasks' worst case computation times. When *wcc_min* = 50, the resource reclaiming overhead is only 0.2% to 0.4% of tasks' worst case computation times. As one can see, if the resource reclaiming overhead can be more than 10% of tasks' worst case computation time, i.e., when *wcc_min* < 20 on the X-axis, the guarantee ratio using Reclaiming with Early Start can be even worse than without any resource reclaiming. So it only pays to do resource reclaiming if one can ensure that the overhead cost of the resource reclaiming algorithm is below a reasonable percentage of tasks' worst case computation times, such as below 10%.

E. *Effects of Average Processor Load*

In all the above experiments, we have simulated heavy load situations. In Figure 18, we examine the performance of Reclaiming with Early Start with respect to different average processor loads L_{pi}. We vary the value L_{pi} from heavily loaded (1.0) to lightly loaded (0.3). In this experiment, P_{use} is 0.2. A task's laxity is uniformly distributed between 1 to 10 times its worst case computation time, so that no matter what the average processor load is, tasks arrive with a large variance of laxities. We

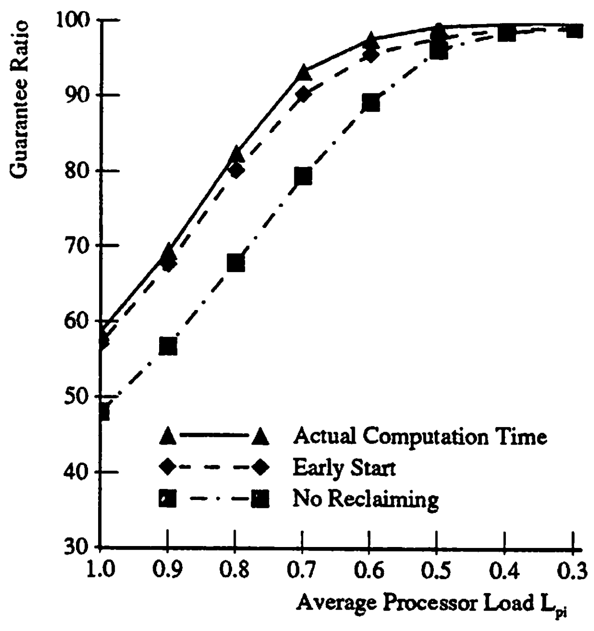

Figure 18: Effects of Average Processor Load

compare the performance of Reclaiming with Early Start with the performance of guarantee with actual computation time and no resource reclaiming. As the performance graphs indicate, the guarantee ratio of Reclaiming with Early Start follows closely to that of guarantee with actual computation time for all the different loads. Except when the system is very lightly loaded, i.e., when $L_{pi} < 0.4$, Reclaiming with Early Start has much higher guarantee ratio than no resource reclaiming. At $L_{pi} = 0.8$, the difference between the guarantee ratios of Reclaiming with Early Start and no resource reclaiming is 14.3. When the load of the system is extremely low, e.g., at $L_{pi} = 0.3$, resource reclaiming is not necessary.

F.*Effects of Actual Computation Time to Worst Case Computation Time Ratio*

In all the simulations presented above, the actual computation time of a task is between 50% to 90% of its worst case computation time, drawn from a uniform distribution. Figure 19 shows the results for the case in which all the tasks in a task load for each simulation point have the same *ratio* of actual computation time to worst case computation time. We plot the percentage of the unused computation time on the x axis. This test studies the effect of the accuracy of worst case execution times upon performance. This ratio is varied from 100% to 10%. Note that for each test, even if all the tasks have the same actual computation time to worst case computation time ratio, their actual computation times are still very different due to the uniform distribution of their worst case computation times. P_{use} is set 0.2. The average processor load has been calculated according to tasks' actual computation times rather than their worst case computation times, i.e., $L_{pi} = \lambda_i * E[\text{actual computation time}]$. At each simulation point, we generated the same average processor load $L_{pi} = 0.6$ with respect to the expected

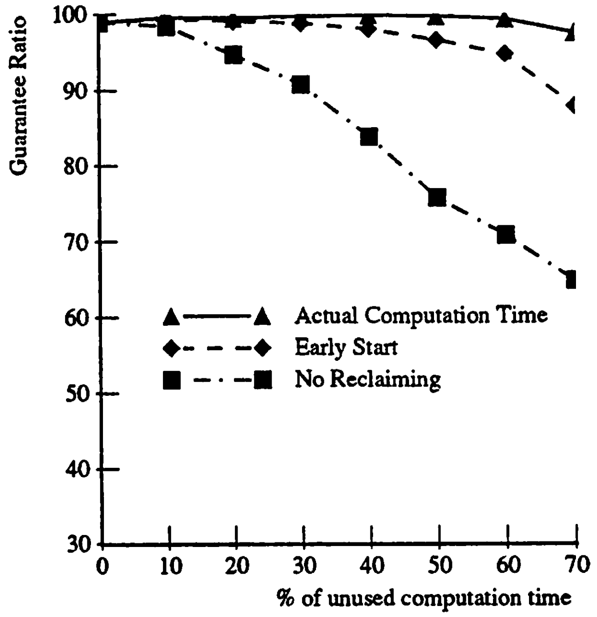

Figure 19: Effects of different actual to worst case computation time ratios

actual computation time, so that if we had known the actual computation times of tasks, the task load was mostly feasible as demonstrated by the performance of guarantee with actual computation time. However, since in using Reclaiming with Early Start and no resource reclaiming we do not know the actual computation times at schedule time, the smaller the ratio of the actual computation time to the worst case computation time (as the tasks leave more unused computation time), the larger the worst case load the system has to handle.

The simulation results indicate that

- For a large range of the accuracy of worst case computation time estimation (from 30% to 100%), Reclaiming with Early Start performs very close to that of the guarantee with actual computation time scheme. This is because:
  - Reclaiming with Early Start is very effective in reclaiming the unused time dynamically, reflecting the actual computation times of tasks in a timely fashion.
- The improvement on the guarantee ratio of Reclaiming with Early Start over no resource reclaiming is substantial. The guarantee ratio is improved by 23.9% when tasks' actual computation time is 40% of their worst case computation times.

5 Additional Remarks

In the previous section, we demonstrated the applicability of the resource reclaiming algorithms in dynamic real-time systems. Here we discuss their applicability and necessary extensions for task systems with other characteristics.

Precedence Contraints among Tasks:

In this paper, we have assumed that tasks are independent. There are many applications in which tasks are related by *precedence constraints.* Precedence constraints specify the partial ordering among tasks such that a task can start execution only when all of its predecessors have completed execution. Since neither of the resource reclaiming algorithms proposed in this paper allows *passing* (as defined in Section 3), they are both applicable for task systems with precedence constraints. If tasks have precedence constraints in a feasible schedule, the resource reclaiming algorithms will never violate these precedence constraints.

Tasks with Explicit Ready Times:

Some systems may have tasks that cannot be started until after some specific time, called a *ready time.* For example, periodic tasks cannot be started until the beginning of their periods. In such systems, a task with a ready time may have been placed in the feasible schedule, but it cannot be moved forward to pass its ready time in the schedule. In this case, our resource reclaiming algorithms can be modified to take into consideration a task's ready time. In Step 2 of either of the algorithm, we need to consider the ready time of a task when we try to start a task. Specifically, first at line 4 in Figure 6 and line 18 in Figure 7, the following condition should be added:

- if $current\_time \geq ready\_time(T_{r_f})$.

Second, at line 7 in Figure 6 and line 21 in Figure 7 we need to modify the calculation of the actual start time of a task to the following:

- $ast_{r_f} = max(sst_{r_f} - reclaim\_\delta, ready\_time(T_{r_f}))$.

Other Types of Tasks:

In addition to dynamic hard real-time tasks, a system may have (1) monotone tasks [16], (2) dual-copy fault-tolerant tasks [2], and (3) non-real-time tasks. Real-time systems with these types of tasks can all benefit from resource reclaiming. Instead of using the reclaimed time *reclaim_δ* for the tasks that have already been *guaranteed* in the feasible schedule, a system can use it to (1) execute the optional part of a monotone task, (2) increase the time assigned to the primary copy of a dual-copy fault-tolerant task in a feasible schedule, or (3) preemptively execute non-real-time background tasks.

6 Conclusion

In this paper, we have investigated the problem of resource reclaiming in real-time multiprocessor systems. A correctness criterion was defined for designing correct resource reclaiming algorithms. We presented two simple resource reclaiming algorithms, Basic Reclaiming and Reclaiming with Early Start. The complexity of the algorithms is *bounded* by the number of processors in a multiprocessor node. Practical issues for supporting predictability in multiprocessor real-time systems were considered and the algorithms were shown to be implementable. In fact, both resource reclaiming algorithms have been implemented in the Spring Kernel. The resource reclaiming algorithms have also been studied under dynamic real-time task arrivals and experimental results are presented.

From the simulation studies, the following can be observed:

- Good local optimization can be very effective in a dynamic real-time system.
- In a real-time system, it is important to employ run time algorithms with *bounded* time complexity. The complexity of the algorithm should be *independent* of the number of tasks.
- Beside having *bounded* time complexity, it is essential for a resource reclaiming algorithm to be *inexpensive* in terms of overhead cost. Our simulation results indicated that it only pays to do resource reclaiming if one can ensure that the overhead cost of the resource reclaiming algorithm is below 10% of tasks' worst case computation times.
- Resource reclaiming can compensate for the performance loss due to the inaccuracy of the estimation of the worst case computation times of real-time tasks.
- Resource reclaiming is very useful for real-time systems that have to guarantee tasks with respect to their worst case computation times. For a large range of accuracy of the worst case computation time estimation (from 30% to 100%) that we have experimented with, Reclaiming with Early Start performs very close to that of an ideal scheduling scenario – *guarantee with actual computation time.*
- Even though Reclaiming with Early Start has a higher run time cost than that of Basic Reclaiming, it performs much better than Basic Reclaiming in most of the situations except (1) when the system is lightly loaded with $L_{pi} < 0.5$ and/or (2) when the resource usage probability of tasks is high with $P_{use} \geq 0.5$.
- Simple resource reclaiming algorithms are needed most when the system is heavily loaded and the invocation of the scheduling algorithm is expensive compared with the resource reclaiming algorithms.

- When the load of the system is extremely low, e.g., $L_{pi} \leq 0.3$, resource reclaiming is not necessary.
- Dynamic resource reclaiming is applicable to a wide range of task resource usage probabilities, task laxities, and system loads.

In summary, the results show that, although the resource reclaiming algorithms proposed are very simple, they are very effective with respect to a wide range of system and task parameters. We believe that resource reclaiming substantially improves average system performance.

Acknowledgment

The authors wish to thank Lory D. Molesky for implementing the resource reclaiming algorithms on the Spring Kernel, and Fuxing Wang for providing the Load Generator to generate the task loads for the evaluation of the algorithms.

A Appendix — Multiprocessor Resource Reclaiming Anomalies

Anomalies can arise at run time in a dynamic real-time multiprocessor schedule with resource and processor constraints when resource reclaiming is not done correctly. These anomalies may jeopardize the deadlines of the real-time tasks that have already been guaranteed. In this appendix, we examine two very simple *work-conserving* resource reclaiming algorithms and the possible anomalies they can cause at execution time to a resource constrained multiprocessor task schedule. The worst case bounds on these anomalous cases are presented.

A.1 Greedy Resource Reclaiming Rule

The first is the *Greedy Resource Reclaiming Rule.*

Definition 9: Given a *Projection List* PL (See Definition 7) of a feasible schedule S, the *Greedy Resource Reclaiming Rule* will scan the PL from left to right, and dispatch a task T_i if the resources and the processor T_i needs are all available.

The *Greedy Resource Reclaiming Rule* reclaims resources by not *intentionally* leaving any processor or resource idle at run time. Figure 3, which is in fact the post-run schedule produced by the *Greedy Resource Reclaiming Rule*, demonstrates the run-time anomaly that can occur when the *Greedy Resource Reclaiming Rule* is used. Although all the tasks' timing and resource constraints are satisfied in S in Figure 1, T_4 misses its deadline in S' in Figure 3. Then the questions to ask are (1) why does this anomaly occur, and (2) how much performance degradation can this anomaly bring to the system? The anomaly occurs because tasks are not dispatched in the same order as in the given feasible schedule, i.e., *passing* has occurred, and this run-time *reordering* is done without verification of the resource conflicts and timing constraints among tasks. For example in Figure 3, since the *Greedy Resource Reclaiming Rule* always keeps a processor busy (i.e. work-conserving) whenever possible, T_2 was dispatched *earlier* than its scheduled start time. When T_1 executes less than its worst case computation time, T_2 is dispatched immediately since both the processor and resource it needs are available. Because of the resource conflicts between T_2 and T_4, i.e. due to their resource conflict over r_1, T_4 cannot be dispatched on time. The following theorem provides the answer to the second question.

Theorem 3: Let m be the number of processors and n the number of tasks. Given a feasible schedule S of length L, when the computation times of some task(s) decreases, in the worst case, the length L' of the resulting post-run schedule S' produced by the *Greedy Resource Reclaiming Rule* is $\frac{m+1}{2} * L$, i.e.

$$\frac{L'}{L} \leq \frac{m+1}{2}$$

Proof Recall that c_i is the worst case computation time of task T_i in the feasible schedule, and c'_i is the actual computation time of T_i in the corresponding post-run schedule.

Suppose e(t) = the set of tasks being executed at time t. If $S = \{t||e(t)| = 1\}$, i.e. S is the set of time intervals t in which only one task is being executed in L', then let $\tau(S)$ denote the sum of all the time intervals in S, and let $\tau(\bar{S})$ denote the sum of the rest of the time intervals in L'. Then we have

$$L' = \tau(S) + \tau(\bar{S})$$

Define

$$T = \bigcup_{T_i \in S} e(t)$$

i.e. T is the set of tasks executed in $\tau(S)$.

Since no two tasks T_i, T_j in T can execute in parallel due to their resource constraints, (otherwise they would have been dispatched in parallel by the *Greedy Resource Reclaiming Rule*),

$$L \geq \tau(S)$$

This is because the scheduling algorithm that produced the feasible schedule with length L could not have scheduled any of the tasks in T in parallel either due to their resource constraints. For example, $T = \{A_3, B_1\}$ for the post-run schedule in Figure 21.

Since

$$mL \geq \sum_{i=1}^{n} c_i \quad and \quad c_i \geq c'_i,$$

$$\sum_{i=1}^{n} c_i \quad \geq \quad \tau(S) + 2\tau(\bar{S})$$

(since there are at least 2 tasks executing in $\tau(\bar{S})$), we have

$$mL \geq \tau(S) + 2\tau(\bar{S})$$

And since

$$\begin{aligned} L' &= \tau(S) + \tau(\bar{S}) \\ 2L' &= \tau(S) + \tau(S) + 2\tau(\bar{S}) \\ \tau(S) &+ 2\tau(\bar{S}) = 2L' - \tau(S) \end{aligned}$$

Then

$$\begin{aligned}
mL &\geq \tau(S) + 2\tau(\bar{S}) \geq 2L' - \tau(S) \\
mL &\geq 2L' - L \\
(m+1)L &\geq 2L' \\
\frac{L'}{L} &\leq \frac{m+1}{2}
\end{aligned}$$

□

Even though our task model assumes processor constraints, we found that there is a direct mapping of the *Greedy Resource Reclaiming Rule* in our model to the *list-scheduling* problem [3, 4, 7] which assumes identical processors with global shared memory. In [4], a similar bound and proof were given for the list-scheduling problem; however their bound was not derived in the context of task computation time decreasing at run time. The following example demonstrates that the worst case ratio of Theorem 3 is tight for our multiprocessor scheduling model.

Example *Greedy*:

Let m = the number of processors, n = the number of tasks = $4m - 1$, and r = the number of resources, $r \geq m$. Figures 20 and 21 illustrate this worst case example with $m = 3$, $B = 10$, $\epsilon = 2$, and $\delta = 1$. The tasks and their parameters are specified in Table 4, and Table 5 contains tasks' resource requirements. Here L' is 43 and L is 26, i.e. $L' < \frac{m+1}{2} * L$. We now show that if $\epsilon \longrightarrow 0$ and $\delta \longrightarrow 0$, $L' \longrightarrow \frac{m+1}{2} * L$.

- Calculation of the Asymptotic Worst Case Ratio of L'/L:

  Let L be the schedule length of the feasible schedule S, and L' be the schedule length of the post-run schedule L'. We have

$$\begin{aligned}
L &= m\epsilon + 2B \\
L' &= mB + B + m(\epsilon - \delta) \\
&= (m+1)B + m(\epsilon - \delta)
\end{aligned}$$

  Then as $\epsilon \rightarrow 0$ and $\delta \rightarrow 0$,

$$\begin{aligned}
L &= m\epsilon + 2B \longrightarrow 2B \\
L' &= (m+1)B + m(\epsilon - \delta) \longrightarrow (m+1)B
\end{aligned}$$

  Thus,

$$\frac{L'}{L} = \frac{(m+1)B}{2B} = \frac{m+1}{2}$$

Task Type	# of Tasks	pid	c_i	c'_i	d_i
A	m	i	B	B	$m\epsilon + B$
B	m	i	B	B	$m\epsilon + 2B$
E	m	i	ϵ	$\epsilon - \delta$	$m\epsilon$
F	m−1	i	ϵ	ϵ	$m\epsilon$

Table 4: Task Parameters: pid = processor id, c_i = worst case computation time, c'_i = actual computation time, d_i = deadline, for $1 \leq i \leq m$.

resources	F_1	F_3	E_1	E_2	E_3	A_1	A_2	A_3	B_1	B_2	B_3
R_1			e	e	e	e					s
R_2	s			e	e		e		s		
R_3	s	e						e	s	s	

Table 5: Task resource requirements.

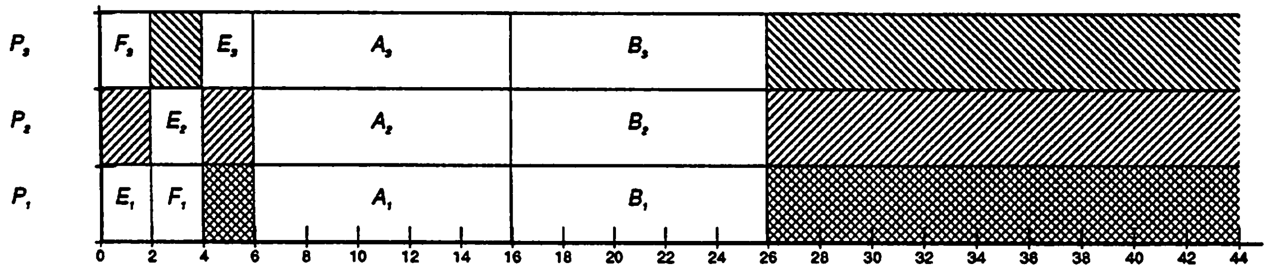

Figure 20: A feasible schedule S.

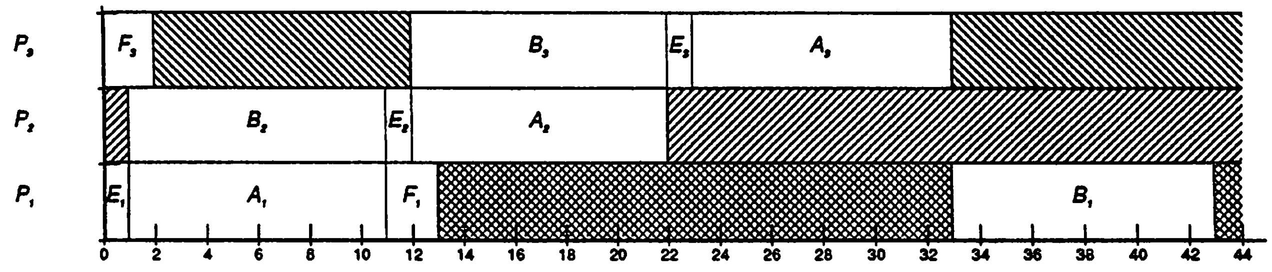

Figure 21: The worst-case post-run schedule produced by the *Greedy* algorithm.

From the above construction of the worst case bounds for the *Greedy Resource Reclaiming Rule*, the following corollary can be derived.

Corollary 1: Given a feasible schedule S of length L, in the case of decreasing computation times of some task(s), a task T_i can miss its deadline by as much as $L' - L$ in the resulting post-run schedule S' produced by the *Greedy Resource Reclaiming Rule*, where L' is the length of the post-run schedule.

Proof It follows directly from Example *Greedy* that, as $\epsilon \to 0$ and $\delta \to 0$, $ft_{B_1} = d_{B_1} + (L' - d_{B_1})$ and $d_{B_1} = L$, where ft_{B_1} is the finish time in S' and d_{B_1} is the deadline of task B_1. □

Observation 1: The worst case of L'/L, i.e. when L'/L is maximum, occurs when S is produced by an *optimal* scheduling algorithm such that L is smallest. Here by *optimal* we mean an algorithm that can always find a feasible schedule if one exists, and if more than one feasible schedule exists, it will always find the shortest one. Since any practical dynamic scheduling algorithm is likely to be heuristic, the resulting L will be larger than the optimal schedule length. Thus in general the worst case ratio of L'/L will be smaller than $\frac{m+1}{2}$.

A.2 Bounded Greedy Resource Reclaiming Rule

Since scanning the PL is part of the *Greedy Resource Reclaiming Rule*, it has a run time complexity linear to the number of tasks in the PL. The second simple resource reclaiming algorithm, the *Bounded Greedy Resource Reclaiming Rule* reduces this complexity by being less *greedy*.

Definition 10: Given a *projection list* PL and its corresponding *processor projection lists* PPL_j of a feasible schedule S, the *Bounded Greedy Resource Reclaiming Rule* will dispatch a task T_i if (1) T_i is the very first task in PPL_j, and (2) the resources, including the processor, that T_i needs are all available. If more than one task satisfies (1) and (2), dispatch the task T_i such that st_i is minimum.

The *Bounded Greedy Resource Reclaiming Rule* limits the time complexity by not scanning the rest of the PL, but by only examining the tasks that are in front of the individual PPL_j. This in effect reduces the time complexity to be *bounded.* However, even though the *greediness* is bounded, run time anomalies can still occur when the *Bounded Greedy Resource Reclaiming Rule* is used. The worst case ratio bound of L'/L resulting from using the *Bounded Greedy Resource Reclaiming Rule* is even worse than from using the *Greedy Resource Reclaiming Rule*, as shown in the following theorem.

Theorem 4: Given m processors, if the *Bounded Greedy Resource Reclaiming Rule* is used, the length L of the feasible schedule S and the length L' of the post-run schedule S' can have a worst case ratio of $L'/L = m$ when task computation time decreases at execution time.

Proof Any feasible schedule S must have a schedule length $L \geq \frac{\sum_{i=1}^{n} c_i}{m}$, so we have $mL \geq \sum_{i=1}^{n} c_i$. Since at no time all processors and resources are idle in S' and $c_i' \leq c_i$, we must have

$$L' \leq \sum_{i=1}^{n} c_i \leq m * L$$

where m is the number of processors and n is the number of tasks. □

The following example demonstrates that the worst case ratio in Theorem 4 is tight for our multiprocessor scheduling model.

Example *Bounded Greedy*:

Let m = the number of processors, n = the number of tasks = $3m - 1$, and r = the number of resources, $r \geq 1$. The tasks and their parameters are specified in Table 6, and Table 7 contains tasks' resource requirements. Figures 22 and 23 illustrate this worst case example with $m = 3$, $B = 20$, $\epsilon = 2$ and $\delta = 1$.

- Calculation of the Asymptotic Worst Case Ratio of L'/L:

  Let L be the schedule length of the feasible schedule S, and L' be the schedule length of the post-run schedule S'. We have

$$\begin{aligned} L &= m\epsilon + B \\ L' &= mB + m(\epsilon - \delta) \end{aligned}$$

  Then as $\epsilon \rightarrow 0$ and $\delta \rightarrow 0$,

$$\begin{aligned} L &= m\epsilon + B \longrightarrow B \\ L' &= mB + m(\epsilon - \delta) \longrightarrow mB \end{aligned}$$

  Thus,

$$\frac{L'}{L} = \frac{mB}{B} = m$$

The above construction of the worst case bounds for the *Bounded Greedy Resource Reclaiming Rule* leads us to the following corollary.

Corollary 2: Given a feasible schedule S of length L, in the case of decreasing computation times of some task(s), a task T_i can miss its deadline by as much as $L' - L$ in the resulting post-run schedule S' produced by the *Bounded Greedy Resource Reclaiming Rule*, where L' is the length of the post-run schedule.

Proof It follows directly from Example *Bounded Greedy* that $ft_{A_1} = d_{A_1} + (L' - d_{A_1})$ and $d_{A_1} = L$, where ft_{A_1} and d_{A_1} are the finish time and deadline of task A_1. □

Task Type	# of Tasks	pid	c_i	c'_i	d_i
A	m	i	B	B	$m\epsilon + B$
E	m	i	ϵ	$\epsilon - \delta$	$i\epsilon$
F	m−1	j	ϵ	ϵ	$i\epsilon$

Table 6: Task Parameters: pid = processor id, c_i = worst case computation time, c'_i = actual computation time, d_i = deadline, for $1 \le i \le m$ and $2 \le j \le m$.

resources	E_1	E_2	E_3	F_2	F_3	A_1	A_2	A_3
R_1	e	e	e			s	s	
R_2	s			s	e	s		s

Table 7: Task resource requirements for the worst case construct of the *Bounded Greedy* algorithm.

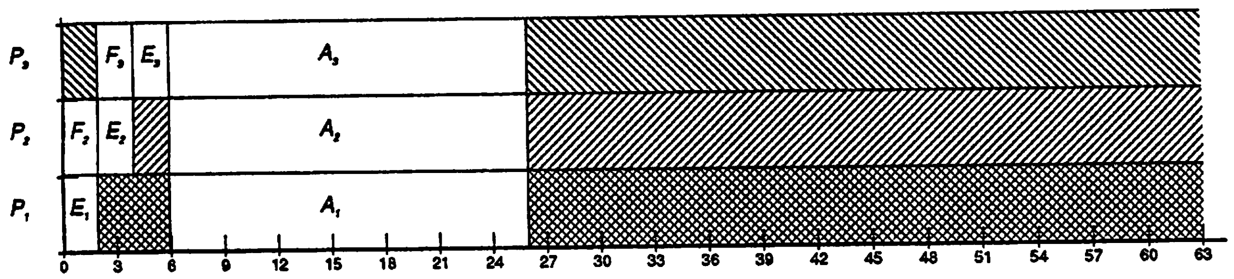

Figure 22: A feasible schedule S.

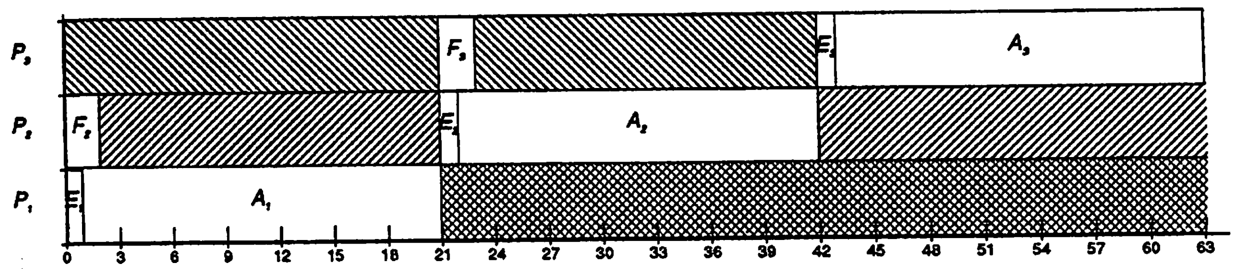

Figure 23: The worst-case post-run schedule produced by the *Bounded Greedy* algorithm when tasks execute only up to their actual computation times.

References

[1] J. Blazewicz, W. Cellary, R. Slowinski, and J. Weglarz. *Scheduling under Resource Constraints – Deterministic Models* . Annals of Operations Research. J.C. Baltzer AG Scientific Publishing Company, 1986.

[2] H. Chetto and M. Chetto. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Transaction on Software Engineering*, 15(10), Oct. 1989.

[3] Jr. E. G. Coffman. *Computer and Job-Shop Scheduling Theory.* John Wiley & Sons, 1976.

[4] M. R. Garey and R. L. Graham. Bounds for Multiprocessor Scheduling with Resource Constraints . *SIAM Journal on Computing*, 4(2):200–187, June 1975.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[6] M. R. Garey and D. S. Johnson. Scheduling Tasks with Nonuniform Deadlines on Two Processors. *Journal of the Association for Computing Machinary*, 23(3), July 1976.

[7] D. W. Gillies and J. W. S. Liu. Greed in Resource Scheduling . In *the 10th IEEE Real-Time System Symposium*, 1989.

[8] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.*, 17(2), March 1969.

[9] K. S. Hong and J. Y-T. Leung. On-Line Scheduling of Real-Time Tasks. In *the Proceedings of Real-Time Systems Symposium*, Dec 1988.

[10] G. K. Manacher. Production and stabilization of real-time task schedules. *Journal of the ACM*, 14(3), July 1967.

[11] L. D. Molesky, K. Ramamritham, C. Shen, J. A. Stankovic, and G. Zlokapa. Implementing a Predictable Real-Time Multiprocessor Kernel – The Spring Kernel . *Seventh IEEE Workshop on Real-Time Operating Systems and Software*, May 1990.

[12] L. D. Molesky, C. Shen, and G. Zlokapa. Predictable Synchronization Mechanisms for Multiprocessor Real-Time Systems . *Real-Time Systems Journal*, 2(3), Sept. 1990.

[13] K. Ramamritham, J. A. Stankovic, and P-F. Shiah. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2), April 1990.

[14] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode Change Protocols for Priority-Drive Preemptive Scheduling . *Real-Time Systems, The International Journal of Time-Critical Computing Systems*, 1(3), Dec. 1989.

[15] C. Shen. *An Integrated Approach to Real-Time Task and Resource Management in Multiprocessor Systems.* PhD thesis, University of Massachusetts, 1991.

[16] W-K. Shih, J. W. S. Liu, and J-Y. Chung. Fast Algorithms for Scheduling Imprecise Computation. In *the Proceedings of the IEEE Real-Time Systems Symposium*, 1989.

[17] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Operating Systems. *ACM Operating Systems Review*, 23(3), July 1989.

[18] W. Zhao and K.g Ramamritham. Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints. *Journal of Systems and Software*, 1987.