Resource Reclaiming in Real-Time *

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

Abstract

Most real-time scheduling algorithms schedule tasks with respect to their worst case computation times. Resources 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. In dynamic real-time multiprocessor environments, resource reclaiming can be used to improve the average performance. In a multiprocessor schedule with resource constraints, when the actual computation time of a task differs from its worst case computation time, run time anomalies may occur if another task is allowed to begin execution immediately. These anomalies may cause some of the already guaranteed tasks to miss their deadlines. Thus, dynamic resource reclaiming algorithms must be effective in reclaiming unused time and also avoid any anomalies. In this paper, we present resource reclaiming algorithms with these properties. These algorithms are designed to have time complexity independent of the number of tasks in a schedule. The effectiveness of the algorithms is demonstrated through simulation studies. The algorithms have also been implemented in the Spring Kernel [14].

1 Introduction

In order to guarantee that hard real-time tasks will meet their deadlines in the worst case, most real-time scheduling algorithms schedule tasks with respect to their worst case computation times [4; 6; 10; 11; 16]. 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 correctly and effectively utilizing resources left unused by a task when a task 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 [12], or when one of the copies of a task completes successfully in a fault-tolerant system [2].

One straightforward approach to resource reclaiming is to reschedule the remaining tasks upon each task completion. In practice, this may not pay if rescheduling cost exceeds the time reclaimed. This is corroborated by our simulation results. Moreover, since every task might complete early, then every task might initiate resource reclaiming. To maintain predictability, we can incorporate the resource reclaiming cost into the worst case computation time of a task. Consequently, the cost must be low and bounded. Most of the scheduling algorithms have time complexities that depend on the number of tasks to be scheduled. Thus they are not suitable for rescheduling tasks to reclaim unused time. Therefore, one of the challenging issues in designing resource reclaiming algorithms is to reclaim resources with bounded overhead, in particular, overhead that is not a function of the number of tasks in the schedule.

When tasks arrive in a dynamic real-time environment, the operating system guides the tasks through two identifiable phases:

(1) scheduling phase (done upon task arrival) — determines the feasibility of a set of tasks given their worst case constraints and the current system state, and generates a feasible task schedule;

(2) execution phase (done at task run time) — given a feasible schedule, correctly reclaims resources and dispatches tasks,

These two phases interact closely in that the results produced by one are used by the other. The resource and processor time reclaimed in the execution phase can be used in the scheduling phase to produce a new schedule when new tasks arrive. The task and system characteristics impose certain restrictions on the way resource reclaiming is done. For example, 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 [5] 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. For tasks with precedence constraints, Manacher [7] 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 it does not deal with resource constraints among tasks.

In this paper, we study resource reclaiming algorithms for multiprocessor systems in which each processor has local memory for task code and private resources. Tasks also might require other non-local resources, such as shared data structures, files, and communication ports. We present two resource reclaiming algorithms that (1) avoid any run time anomalies that may occur in a feasible multiprocessor schedule, (2) have time complexities independent of the number of tasks in the given schedule, and (3) effectively reclaim the unused portions of resource time and processor time. Due to space limitation, we do not present the analysis of the run time anomalies for our multiprocessor model. See [13] for a complete description and analysis of the anomalies.

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 discuss the resource reclaiming problem and present our resource reclaiming algorithms. In Section 4, we present experimental results for the resource reclaiming algorithms applied to dynamic hard real-time multiprocessor systems. In Section 5 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 needs all its resources throughout its execution and it is not preemptable. Resources can either be used in exclusive mode or shared mode [15]. 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 $T_i$;
- $\{R_j^i\}$: 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. A task $T_i$ has processor constraints if:
  (1) in a heterogeneous multiprocessor system, task $T_i$ requires some particular processor $p_q$;
  (2) in a NUMA 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.

2.2 Terminology

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

Definition 1: A multiprocessor 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 for which it is always possible to assign a scheduled start time (sstart) and scheduled finish time (sfinish) to each task $T_i$ in the schedule such that $\forall_i, sfinish_i \leq d_i$.

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 $start_i$, and a finish time $finish_i$, $start_i$ and $finish_i$ are the actual times at which $T_i$ starts and completes execution, respectively, and they may be different from $sstart_i$ and $sfinish_i$.

Definition 3: Given a post-run schedule $S'$, a task $T_i$ starts on-time if $start_i \leq sstart_i$, 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 $\forall_i 1 \leq i \leq n, finish_i \leq d_i$.

Theorem 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., \( s_{t_i} \leq s_{f_i} \), then \( f_{t_i} \leq s_{f_i} \leq d_i \). So the resulting post-run schedule \( S^r \) is correct. □

This theorem forms the basis for our reclaiming algorithms. Note that the theorem gives us a (stronger than necessary) 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.

3 Resource Reclaiming Algorithms

We first discuss the resource reclaiming problem with respect to its time complexity. Then we present our resource reclaiming algorithms, Basic Reclaiming and its extension, 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 of resource reclaiming 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 (sst). 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 \( \Omega(f(n)) \), assuming no new task arrivals.

Note that because the resource constrained multiprocessor scheduling problem is NP-complete in the nonpreemptive case [3] 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 taken out of 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. But it does provide an indication of the best a system can do in reordering tasks according to the available time.

Clearly, a useful resource reclaiming algorithm should have a complexity less than the total rescheduling extreme, while being as effective as it with low overhead. 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 \( s_{t_i} < s_{t_j} \), but \( s_{f_i} \leq s_{f_j} \). Thus passing occurs when a task \( T_i \) passes some task(s) that are scheduled to finish execution before \( T_i \) was originally scheduled to start.

Given a feasible schedule \( S \), if we assume tasks never execute longer than their worst case computation times, and there are no interruptions during the execution of all the tasks in \( S \) other than task completion and task dispatching, then we have the following Lemma which we shall state without proof.

Lemma 1: Given a feasible real-time multiprocessor schedule \( S \), if some task does not start on time in a post-run schedule, then passing must have occurred.

A resource reclaiming algorithm that allows passing will inevitably incur higher complexity in terms of time than another that does not allow passing. 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 scheduling problem is in fact a search problem [15]). Any searching will have a complexity of at least \( \tilde{O}(\log n) \). 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 resource reclaiming algorithms, the Basic Reclaiming algorithm and the Reclaiming with Early Start algorithm. The following definitions are needed to describe our resource reclaiming algorithms.

Definition 6: 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 \( s_{st} \). A projection list is in one-to-one correspondence with some feasible schedule. If \( s_{st} = s_{st'} \) for some tasks \( T_i \) and \( T_j \), we put the task with the smaller processor id first in the \( PL \). Thus \( PL \) imposes a total ordering on the guaranteed tasks.

Definition 7: Given a projection list \( PL \), a processor projection list \( PPL \) is an ordered list of all the tasks belonging to processor \( p_t \) in the \( PL \), also arranged in nondecreasing order of \( s_{st} \), for \( 1 \leq i \leq n \) and \( 1 \leq t \leq m \).

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. via the algorithm presented in
[10], (2) the corresponding projection list $PL$ and $m$ processor projection lists $PPL_1$, ..., $PPL_m$, and (3) a scheduled start time $sst_i$ and scheduled finish time $sft_i$ 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 $PPL_q$.

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 overlap in time on different processors 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 two tasks do not overlap in time, we cannot make the same conclusion without re-examining resource constraints or without total rescheduling. Assume each task is assigned a scheduled start time $sst$ and a scheduled finish time $sft$ in the given feasible schedule, our resource reclaiming algorithms utilize this information to do local optimization at run time, while preserving the correct relative ordering among the scheduled tasks, thus ensuring the original guarantee. This local optimization is accomplished by considering the characteristics of the first task on each $PPL_q$ for $1 \leq q \leq m$. 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 runtime.

A task scheduled on processor $q$ is not removed from the $PL$ and $PPL_q$ until it finishes execution. Resource reclaiming occurs when a task completes execution and another task is to be dispatched. Figure 1 gives the outline of the resource reclaiming algorithms. There are two steps involved in each of our algorithms. Suppose a task completion occurs on processor $q$. First, the length of the idle time resulting from the early completion of tasks is determined. In the second step, the first task in $PPL_q$ is examined to decide whether it can be immediately dispatched. Figure 2 presents the pseudo code for Step1. Details of this step are the same for both the Basic Reclaiming Algorithm and the Reclaiming with Early Start Algorithm. Figure 3 and 4 present Step2 for each of these algorithms, respectively.

Upon completion of a task, Step1 tries to identify idle periods on all processors and resources by computing a function $\text{reclaim.b} = \text{sst}_t - \text{current.time}$ (line 6 to line 8 in Step1); where $\text{sst}_t$ 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, $\text{sst}_t$ must be the minimum scheduled start time among all tasks in the schedule, including the one(s) still in execution. Any positive value of $\text{reclaim.b}$ 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 2), $\text{temp.reclaim.b}$ could have a negative value (if the first task in the $PL$ is still in execution) and, in this case, $\text{reclaim.b}$ remains its original value.

In Step2 for Basic Reclaiming (Figure 3), we only start the execution of the first task $T_r$, on processor $r$ immediately 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 $sst$ (scheduled start time) as the current first task (line 3 to 4 in Figure 3). Otherwise we compute a function $\text{ast}_r$, for $T_r$, to decide the actual start time (vs. the scheduled start time given in the schedule) for it, taking into consideration the idle periods that have been accumulated up to now. This function is $\text{ast}_r = \text{sst}_r - \text{reclaim.b}$, where $\text{sst}_r$ is the original scheduled start time of task $T_r$. This function is also $O(1)$. Once this function is computed, processor $r$ will pend until (1) either the calculated $\text{ast}_r$ has arrived, (2) or some other task finishes early and $\text{reclaim.b}$ is incremented. In the second case, Step2.BASIC will be invoked again (see Figure 1).

Notice that the Basic Reclaiming algorithm will start a task early by only $\text{reclaim.b}$ 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 $T_r$, in front of $PPL_r$ to start early as long as the first tasks in each of the other $PPL_q$, for $1 \leq q \leq m$ and $q \neq r$, do not conflict for resources with $T_r$. Now let us define that a task $T_r$ is being early started if $\text{sst}_r < \text{sst}_r - \text{reclaim.b}$. In the extended algorithm Reclaiming with Early Start, we first compute a Boolean function $\text{can.start.early} = \text{sst}_r < \text{sft}_r \forall q$ such that $q \neq r$ and $1 \leq q \leq m$, where $\text{sst}_r$ is the scheduled start time of the first task on processor $r$ and $\text{sft}_r$ 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 task on all other processors and has a complexity $O(m)$. The task will be dispatched if the value of the Boolean function is true. Only when the value of the function $\text{can.start.early}$ is false, we will compute the $\text{ast}_r$ for task $T_r$ as in Step2.BASIC.

For both algorithms, whenever a positive value of $\text{reclaim.b}$ is obtained in Step1, Step2 must be executed for all currently idle processors. Thus the complexity of the basic version is: $O(1) + m \cdot O(1) = O(m)$, while the extended version has a complexity of $O(1) + m \cdot O(m) = O(m^2)$. 

44
Algorithm Resource Reclaiming

```
/* m - the number of processors */
/* reclaim_d - the amount of time that has been reclaimed */
/* reclaim_d is set to zero initially */
/* Tn - the newly completed task in PPL */
/* for some processor q */

Whenever a task Tn completes execution on a processor q, do
{original_reclaim_d = reclaim_d;
  Step2.BASICEARLYSTART(q, reclaim_d, PL, PPL1, ..., PPLm);
}
end Algorithm Resource Reclaiming
```

**Figure 1:**

Step1(Tn, reclaim_d, PL, PPLq);

```
/* Task Tn just completed execution on processor q.*/
1. REMOVE(Tn, PL, PPLq);
2. Tj - the first task in the current PL;
3. if (current_time < (start(Tj) - reclaim_d))
4. then
5. { temp_reclaim_d = (start(Tj) - current_time);
6. if temp_reclaim_d > 0
7. then reclaim_d = temp_reclaim_d;
8. end if
9. }
10. end if
11. end Step1
```

**Figure 2:**

Step2.BASIC (r, reclaim_d, PL, PPL1, ..., PPLm);

```
1. Tf - the first task in the current PPL;
2. Tj - the first task in the current PLL;
3. if (Tf == Tj)
4. or (setj == setf)
5. then startexecution(Tf);
6. else
7. astf = setf - reclaim_d;
8. pend(Tf, astf);
9. }
10. end if
end Step2.BASIC
```

**Figure 3:**

### 3.3 Implementing Resource Reclaiming Algorithms in a Local-Memory Multiprocessor System

Both resource reclaiming algorithms have been implemented in the Spring Kernel [14]. In this section, we briefly discuss the important issues in implementing the resource reclaiming algorithms in a local-memory multiprocessor system with shared resources, and the interplay between the scheduler and the resource reclaiming algorithms. In the last section, we presented two resource reclaiming algorithms for real-time multiprocessor systems. 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 process. In a concurrent scheme, each processor will have its own reclaiming process and all the processors in the multiprocessor system can concurrently reclaiming unused time. We have taken the concurrent approach in the Spring Kernel.

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 rest of the processors to execute guaranteed application tasks. The scheduler on the system processor is responsible for dynamically producing a feasible schedule. There is a dispatcher process on each application processor. Effectively, this executes steps 1 and 2 of the reclaiming algorithm. Thus, reclaiming occurs concurrently on the application processors. Since the overheads of reclaiming are bounded, predictability of the system is
maintained even with reclaiming. In order to correctly support the concurrency, Step 1 of the algorithms must be within a critical section to ensure the data consistency of reclaim.δ. 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 not involve any locking.

Figure 5 gives the scheme we use to schedule dynamic task arrivals with resource reclaiming. GUARANTEE uses the heuristic scheduling algorithm proposed in [10]. To achieve concurrent execution of application tasks and the scheduler, 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 $max SC$. Any task $T_i$ with $sst_i < cut-off line$ in the schedule is not going to be considered in the rescheduling process. This ensures that the scheduling algorithm can execute in parallel with application tasks. The detailed implementation of this concurrency can be found in [8; 9].

When a new task arrives, its worst case computation time, deadline, resource and processor requirements are assumed to be known. One can try to guarantee the new task arrival together with the tasks $T_i$ with $sst_i > cut-off line$ in the original feasible schedule, using some given scheduling algorithm. With the knowledge of the value of reclaim.δ, i.e., the amount of time that has been reclaimed on all resources and processors, those tasks $T_i$ with $sst_i < cut-off line$ will finish at least reclaim.δ time units earlier than their scheduled finish time $sft_i$. Thus, the beginning time of the new schedule can be shifted reclaim.δ time units forward in time. If the new task arrival is guaranteed, we reset the value of reclaim.δ to zero as soon as the first task in the new feasible schedule is dispatched. On the other hand, if the new task arrival cannot be guaranteed, we return to the original feasible schedule and reclaim.δ remains the same until the next rescheduling occurs.

4 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 reclaimation, 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 invoca-

Scheduler
Whenever a task $T_i$ arrives, do

1. $reclaim.δ$ = amount of time that has been reclaimed;
2. Calculate the run time cost of the scheduling algorithm based on the number of tasks in the current $P_k$ plus the new task arrival;
3. Calculate the cut-off line:
4. $T_m = \{T_j | sst_j < cut-off line\};$
5. Calculate the earliest available time of each resource and processor, based on the resource and processor requirements, and $sft_j$ of the tasks $T_j$ in $T_m$, and the value of reclaim.δ;
6. $T_i = \{T_j | sst_j > cut-off line\};$
7. GUARANTEE($T_i$, $T_m$);

Figure 5: Scheduling Dynamic Real-Time Tasks with Resource Reclaiming

The combination of the mean interarrival time $\lambda$ of tasks, the value of $P_{recl}$, 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 $\lambda$, 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.

$$L_{ri} = \lambda_i \cdot E[wcc]$$

$$L_{ri} = P_{recl} \cdot \lambda_i \cdot E[wcc] \cdot m$$

$$P_c = 1 - ((1 - P_{recl}) \cdot P_{wcc} + (1 - P_{wcc})^2 + (0.5 - P_{wcc})^2)$$

$E[wcc]$ is the expected value of the worst case computation time of a task; thus it is 100 in our simulation.
\( m \) is the number of processors and \( T \) is the number of tasks in a schedule. The first two formulas are straightforward. In the third formula, \( P_c \) is the probability that two tasks will conflict on any of the given \( S \) resources (vs. \( P_{usc} \), 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, 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 \( P_{usc} \) or the value of \( S \) increases. So if we keep \( P_{usc} \) 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 [10] is used in the scheduler in all of our simulations presented in this section. To evaluate the effectiveness of the proposed resource reclaiming algorithms, whenever appropriate, we compare their performance with the performance of an ideal scheduling scenario and a no resource reclaiming scenario. In the former case, the scheduler guarantees tasks with respect to their actual computation times (Guarantee with AC), and thus resource reclaiming is not necessary. The case of no resource reclaiming provides a lower bound on performance.

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.

A. Comparison with Total Rescheduling

In this section, we compare the performance of (1) the total rescheduling extreme case, (2) the two resource reclaiming algorithms, (3) guarantee with actual computation time, and (4) no resource reclaiming. In total rescheduling, 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. In Figure 6, \( L_{pc} = 0.72 \) and \( P_{usc} \) varies from 0.1 to 0.7.

From the Figure, the following can be observed:

- For small values of \( P_{usc} \), the performance of total rescheduling is even below that of no resource reclaiming. For example, when \( P_{usc} = 0.1 \), the guarantee ratio of total rescheduling is 8.2% below that of no resource reclaiming. This is because it costs more to run the scheduler than what can be reclaimed.

![Figure 6: Comparison with Total Rescheduling](image)

- On the other hand, because the resource reclaiming algorithms have much lower overhead costs, they both perform much better than total rescheduling. For example, at \( P_{usc} = 0.2 \), Reclaiming with Early Start guarantees 18.9% more task arrivals than total rescheduling does.

- When the resource conflict is small (i.e., when \( P_{usc} \leq 0.4 \) and thus \( P_c < 0.5 \)), Reclaiming with Early Start performs much better than Basic Reclaiming since it can exploit more parallelism. When the resource conflict is high, the Basic Reclaiming algorithm is just as effective as Reclaiming with Early Start.

- The performance of guarantee with actual computation time is about the same as that of both of the resource reclaiming algorithms when \( P_{usc} \geq 0.5 \). This is because large values of \( P_{usc} \) translate to large resource conflict probability \( P_c \). If \( P_{usc} \geq 0.5 \), then \( P_c \geq 0.65 \). In this case resource contention becomes the major factor affecting the guarantee ratio.

Suppose we can reduce the cost of the scheduler, will the total rescheduling scheme be a better choice? This question is answered in Figure 7. Here we artificially vary the scheduler's per-task.cost from 0 to 5. The task loads simulated have \( L_{pc} = 0.72 \) and \( P_{usc} = 0.3 \). The simulation results indicate that:

- The performance of total rescheduling degrades the most when the cost of the scheduling algorithm increases.

- Even when the scheduler's per-task.cost is zero,
total rescheduling performs only slightly better than Reclaiming with Early Start. This demonstrates that certain low complexity run time local optimization can be very effective, such as Reclaiming with Early Start.

- Reclaiming with Early Start is the most stable one with respect to the scheduler's cost, i.e., its guarantee ratio drops the least amount as the scheduler's cost increases.
- The performance of guarantee with actual computation time can also be affected by the scheduler's cost because the complexity of the scheduling algorithm depends on the number of tasks in the schedule. Thus the more tasks that are guaranteed in a schedule, the more expensive it is to run the scheduler to guarantee new task arrivals.

B. Effects of the Number of Shared Resources

In this section, we examine the effects of the number of resources that can be required by any tasks on our resource reclaiming algorithms. We increase the number of resources from one to ten while keeping the average processor load \( L_{pt} \) the same at 0.72 and \( P_{rate} \) at 0.3. This simulates a heavily loaded situation for the resources since the average resource load \( L_{rt} \) in this case is 1. The performance is shown in Figure 8. From Figure 8, we can make the following observation:

- Reclaiming with Early Start performs very closely to that of guarantee with actual computation time.
- The performance of Basic Reclaiming is very close to that of no resource reclaiming when there are very few resources. With less than or equal to four resources, the resource conflict probability \( P_e \) is less than 0.25. Thus there is a lot of parallelism among the tasks. It is very unlikely for Basic Reclaiming to find the situation in which all processors and resources are idle.
- As the number of resources increases, this parallelism reduces and the performance of Basic Reclaiming approaches that of Reclaiming with Early Start.

C. Effects of Average Processor Load

In Figure 9, we compare the performance of the two resource reclaiming algorithms with respect to different average processor loads \( L_{pt} \). We vary the value \( L_{pt} \) from lightly loaded (0.4) to heavily loaded (0.9). Then the difference between the guarantee ratio of the two resource reclaiming algorithms is computed. The three curves correspond to simulations with three values of \( P_{rate} \). Based on the simulation results, we make the following observations:

- When the system is very lightly loaded, the difference between the two algorithms is negligible. Their difference peaks when \( L_{pt} \) ranges from 0.7 to 0.8, depending on the value of \( P_{rate} \). This is because when the system is lightly loaded, not too many tasks arrive in parallel, so Basic Reclaiming is just as effective.
- When the system is heavily loaded, many tasks will be scheduled in parallel. In this case the amount of resource reclaimed by the Reclaiming with Early Start algorithm is much larger than by...
the Basic Reclaiming algorithm.

D. 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 10 shows the results for the case in which all the tasks in a load have the same ratio of actual computation time to worst case computation time. This test studies the effect of the accuracy of worst case execution times upon performance. This ratio is varied from 100% to 10%. We plot the percentage of the unused computation time on the x axis. Note that for each test, even if all 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. \( P_{\text{acc}} \) is 0.2. Both the average processor load \( L_{p_i} \) and the average resource load \( L_{r_i} \) are saturated, representing an overloaded situation with respect to the worst case computation times of tasks. The simulation results indicate that

- For a large range of the accuracy of worst case computation time estimation (from 100% to 40%), Reclaiming with Early Start performs very close to that of guarantee with actual computation time. This is because:
  
  1. Reclaiming with Early Start is very effective in reclaiming the unused time dynamically, reflecting the actual computation times

- The improvement on the guarantee ratio of Reclaiming with Early Start over no resource reclaiming is substantial. For example, the guarantee ratio is improved by 15% when tasks' actual computation time is 40% of their worst case computation times.

- The performance of Basic Reclaiming is much lower than that of Reclaiming with Early Start until tasks use only 20% of their worst case computation times.

5 Conclusion

In this paper, we have investigated the problem of resource reclaiming in real-time multiprocessor systems. A correctness criterion is 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. 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:

- In a real-time system, it is important to employ run time algorithms with bounded costs. The cost of the algorithm should be independent of the number of tasks.
- Good local optimization can be very effective.
- Resource reclaiming is very useful for real-time systems that have to guarantee tasks with respect to their worst case computation times.
- 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 most needed when the system is heavily loaded and the invocation of the scheduling algorithm is expensive compared with the resource reclaiming algorithms.

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. Another positive fallout is that with resource reclaiming, the necessity to produce tight worst case computation times is reduced.

Acknowledgment

The authors would like to thank the referees for the helpful comments in the review, especially the one who suggested the comparison between the performance of the resource reclaiming algorithms and the performance of guaranteeing tasks with respect to their actual computation times. The authors also 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.

References


