Bounding Worst-Case Response Time for Tasks under PIP

Harini Ramaprasad
Dept. of Electrical and Computer Engg.
Southern Illinois University Carbondale
Carbondale, IL 62901
harinir@siu.edu

Frank Mueller
Dept. of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
mueller@cs.ncsu.edu

Abstract

Schedulability theory in real-time systems requires prior knowledge of the worst-case execution time (WCET) of every task in the system. One method to determine the WCET is known as static timing analysis. Determination of the priorities among tasks in such a system requires a scheduling policy, which could be either preemptive or non-preemptive. While static timing analysis and data cache analysis are simplified by using a fully non-preemptive scheduling policy, it results in decreased schedulability. In prior work, a methodology was proposed to bound the data-cache related delay for real-time tasks that, beside having a non-preemptive region (critical section), can otherwise be scheduled preemptively.

While the prior approach improves schedulability in comparison to fully non-preemptive methods, it is still conservative in its approach due to its fundamental assumption that a task executing in a critical section may not be preempted by any *other task. In this paper, we propose a methodology that incorporates resource sharing policies such as the Priority Inheritance Protocol (PIP) into the calculation of data-cache related delay. In this approach, access to shared resources, which is the primary reason for critical sections within tasks, is controlled by the resource sharing policy. In addition to maintaining correctness of access, such policies strive to limit resource access conflicts, thereby improving the responsiveness of tasks.*

To the best of our knowledge, this is the first framework that integrates data-cache related delay calculations with resource sharing policies in the context of real-time systems.

1. Introduction

Real-time schedulability requires *a-priori* knowledge of the worst-case execution times (WCET) of all tasks in the system. One method to estimate this WCET is static timing analysis. Static timing analysis is the process of analytically modeling the architectural features of a system and, using these models and the control flow graph of a program, trying to determine the longest path through the program.

While architectural features such as the data cache improve the performance of computer systems significantly, they are inherently unpredictable, thereby complicating static timing analysis. The analysis of data cache behavior for a single task has been the focus of much research. Several analytical techniques have been proposed in the recent past. In prior work [16], we extended the Cache Miss Equations framework by Ghosh *et al.* [8] to produce exact data cache reference patterns. However, this is insufficient since practical systems have multiple tasks executing in a prioritized environment.

In recent work, we proposed a framework that calculated a tight upper bound on the response times of tasks in a fully preemptive scheduling system [17], [18]. The primary assumption in that analysis is that all tasks are completely preemptive. In other words, a task may be interrupted by a task with higher priority *at any time* during its execution. However, this assumption may need to be relaxed for some tasks. A task may have a period in its execution during which it executes in a *critical section*. While a task is in a critical section, no other task may enter a critical section.

In more recent work, we proposed a methodology to analyze tasks with a critical section within their execution [19]. In that work, logical correctness of tasks is maintained by executing all critical sections as *non-preemptive regions (NPRs)*. Using that methodology, schedulability of task sets is improved in comparison to a fully non-preemptive scheduling policy by allowing (legal) preemptions outside the NPR. A fundamental assumption there is that a task executing in a NPR cannot be preempted by *any* higher-priority task for the entire duration of the NPR.

The need for a critical section typically arises due to access of shared resources by multiple tasks. While it is important to prevent two tasks from accessing a shared resource at the same time, it is not necessary to disallow preemptions altogether in such a critical section. In other words, although a shared resource has to be relinquished voluntarily by a task that has acquired it (making the *resource* non-preemptible), the task holding the resource may still be preempted. Several resource sharing policies

This work was supported in part by NSF grants CCR-0310860, CCR-0312695 and CNS-0720496.

have been proposed to control accesses to shared resources in the context of real-time systems [21]. The fundamental aim of all these polices is to maintain correctness of all tasks while maximizing schedulability by reducing the waiting time for tasks that *do not use* a particular resource that has been acquired by some lower-priority task.

In this paper, a framework that incorporates resource sharing policies within the process of estimation of worst-case response times of hard real-time tasks, in the presence of data caches, is presented. The use of resource-sharing policies introduces significant changes to the analysis algorithm compared to that used in our prior work on non-preemptive regions [19]. Furthermore, using resource-sharing policies introduces an additional blocking-related delay component, resulting in further changes to the algorithm and significant additions to the mathematical formulation for our approach.

Throughout this paper, we consider a periodic real-time task model. Every task is assumed to have a deadline less than or equal to its period. The notation used in the remainder of this paper is as follows. A task T_i has characteristics represented by the 5 tuple (Φ_i, P_i, C_i, c_i, D_i). Here, Φ_i is the phase, P_i is the period, C_i and c_i are the worst-case and best-case execution times, respectively and D_i is the relative deadline of the task. In the context of a specific task set, every task has a set of derived characteristics represented by the 3 tuple (B_i, $\Re_i$, Δ_i). Here, B_i is the blocking time and $\Re_i$ is the response time of the task. Δ_i is the data-cache related delay incurred by a task due to interruptions by other tasks. $J_{i,j}$ represents the jth instance (job) of task T_i.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 gives an overview of our prior work on data cache analysis. Section 4 discusses our current framework and Section 5 discusses the calculation of data-cache related delay. Section 7 presents experimental results and the contributions of our work are summarized in Section 8.

2. Related Work

Data cache analysis in the context of real-time systems has been the focus of much research in the recent past. Recently, several methods have been proposed to analytically characterize data cache behavior with respect to a single task [8], [6], [5]. In prior work [16], we extended the Cache Miss Equations framework by Ghosh *et al.* [8] to produce exact data cache reference patterns.

Several techniques have been proposed to analyze multiple tasks executing in a prioritized manner and calculate the cache related preemption delay [11], [12], [22], [23]. These techniques primarily focus on instruction caches. In prior work, [17], [18] we propose a framework to calculate worst-case response times of tasks using a significantly different methodology and in the context of data caches rather than instruction caches.

The question of whether non-preemptive systems are better than preemptive ones or not has been debated for a long time. Several pieces of research provide analysis and tests for non-preemptive systems [7]. However, in all that work, the primary assumption is that every task is completely non-preemptive. Making all tasks fully non-preemptive significantly decreases schedulability. In order to overcome this disadvantage, methods have been proposed to defer preemptions to predetermined points by splitting a job into sub-jobs [3], [4], [13]. Recent work demonstrates some flaws in this method [2], [1],

In prior work, we propose a methodology to analyze tasks that have critical sections during which they are made non-preemptive, but are allowed to be preempted in other areas, thereby improving schedulability over fully non-preemptive systems [19].

In order to address the issue of scheduling tasks with critical sections in a preemptive environment, several resource-sharing policies have been proposed. The aim of these policies is to maintain correctness of accesses to shared resources while maximizing the schedulability of task sets. In this paper, we employ one such policy, namely the Priority Inheritance Protocol [21], to arbitrate accesses to shared resources.

3. Prior Work

In prior work, we proposed a methodology that bounds the data cache related preemption delay and the worst-case response times of hard real-time tasks in a prioritized, fully-preemptive environment [17], [18]. This analysis consists of three fundamental steps: 1) Calculation of an upper bound on the number of preemptions for a task; 2) Identification of the placement of these preemption points within the execution of the preempted task; and 3) Calculation of the data-cache related preemption delay incurred at each identified preemption point.

In more recent work, we extended our methodology to allow a task to have a critical section, termed as a *non-preemptive region* (NPR), within its execution instead of being completely preemptive at all times [19]. In that work, a task is split into three regions, namely $r1$, $r2$ and $r3$. Regions $r1$ and $r3$ are preemptive whereas region $r2$ is a non-preemptive one. The complexity of analysis in such a system arises from the fact that the actual execution time of a task cannot be accurately determined by static timing analysis, forcing analysis techniques to provide upper (worst-case) and lower (best-case) bounds instead. Hence, when a higher priority task (say T_0) is released while a lower-priority task (say T_1) that has a non-preemptive region is executing, there are three possible scenarios. **Case 1**: T_1 has entered its NPR (region $r2$) in both the best and worst cases. In this case, T_0 has to wait for T_1 to finish executing its NPR. **Case 2**: T_1 is still executing region $r1$ in both best and worst cases

or has already entered region $r3$ in both best and worst cases. In this case, T_1 gets preempted and T_0 gets scheduled immediately. **Case 3**: T_1 has entered its NPR in the best case, but is still executing region $r1$ in the worst case. In this situation, our technique considered parallel executions of worst-case scenarios for *individual tasks* to provide safe and tight estimates of the worst-case response times of all tasks. The incorporation of resource-sharing policies such as PIP introduces significant changes to the algorithm and mathematical formulation compared to our prior work.

4. Methodology

In Section 3, we briefly described recent prior work in which we analyzed tasks with critical sections and, in the presence of data caches, calculated safe and tight estimates of the worst-case response times of tasks. That work assumed that, when a task is executing in its critical section, it cannot be preempted by *any* other higher-priority task for the *entire duration* of its critical section, thereby effectively making the region a *non-preemptive region (NPR)*.

In this paper, we remove this restrictive assumption and incorporate resource sharing policies into our analysis to maximize schedulability while maintaining correctness of accesses to shared resources. Hence, although the access to a shared resource is still non-preemptible, the task holding the resource is now preemptible until contention for the shared resource arises. In our current implementation, we use the Priority Inheritance Protocol (PIP) to manage accesses to shared resources [20]. Although our analysis can conceptually support different resource sharing policies with minor extensions to our algorithm, the discussion through the rest of this paper is in the context of the Priority Inheritance Protocol for the sake of simplicity.

In PIP, when a task (say T_0) with a priority higher than the currently executing task (say T_1) is released, T_1 is preempted and T_0 is scheduled immediately. If T_0 later requests access to a shared resource, there are two possibilities. The resource could be available, in which case it is immediately granted to T_0. Alternatively, the resource could have been acquired by T_1 before it was preempted. In such a case, T_1 is now scheduled again and is allowed to execute until it relinquishes the required resource. For this duration of time, T_1 executes at the priority of T_0. In other words, T_1 *inherits* the priority of T_0 until the required resource is relinquished. The reason for this is to ensure that T_1 cannot be preempted by tasks with priority between those of T_0 and T_1, thus preventing a situation termed *priority inversion*.

Every task is split into multiple regions, namely regions that access some shared resource(s) and regions that do not. A total ordering is assumed among all shared resources in the system being analyzed. If a task needs multiple resources simultaneously, it requests them in accordance with the total order and releases them in the reverse order of request to avoid deadlocks in the system.

4.1. Motivating and Illustrative Examples

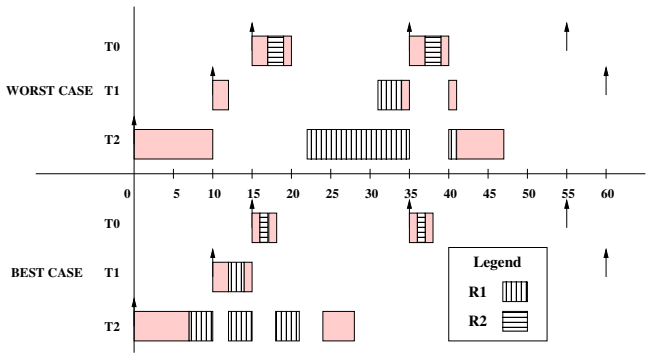
In this section, the methodology is illustrated using examples. In all examples, the deadline of a task is assumed to be equal to its period. Consider the task-set shown in Table 1. The first column indicates task names. The second and third columns show the phases and periods of tasks, respectively. The fourth and fifth columns show the worst-case and best-case execution times (WCET and BCET) of each of the regions of a task. In this example, every task has three regions, the second of which is the one in which resource requests are made. The sixth column indicates the name of the resource being used in the second region of a task. Figure 1(a) shows the results obtained for this task set

Task	**Phase**	**Period**	**WCET** ($r1$/$r2$/$r3$)	**BCET** ($r1$/$r2$/$r3$)	**Res. in** $r2$
T_0	15	20	2/**2**/1	1/**1**/1	R_2
T_1	10	50	2/**3**/2	2/**2**/1	R_1
T_2	0	200	10/**14**/6	7/**9**/4	R_1

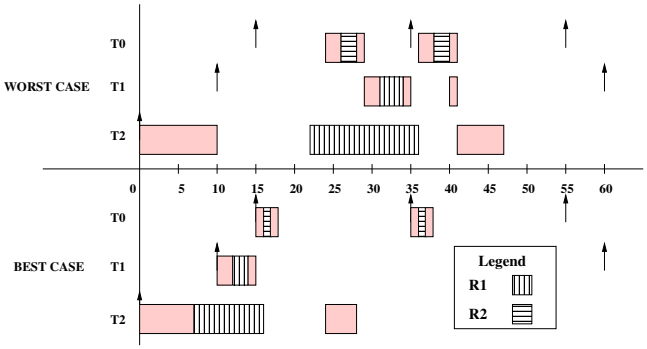
Table 1. Task Set Characteristics - Task Set 1 [RM policy $\rightarrow T_0$ has Highest Priority]

using a resource-sharing protocol, specifically the Priority Inheritence Protocol. This method will henceforth be refered to as ResourceSharingAnalysis. For the sake of comparison, for the same task-set, results obtained using the analysis technique developed in prior work [19], where a critical section is assumed to be a non-preemptive region, are also presented. That method is referred to as NPRAnalysis. It is to be noted that, in NPRAnalysis, a task executing in a critical section cannot be preempted by any other task. These results are shown in Figure 1(b). For the sake of simplicity, data cache related delays are assumed to be zero in these examples. Calculation of data cache related delays will be discussed in Section 5.

In both Figures 1(a) and 1(b), the x-axis represents time. Best-case scenarios and worst-case scenarios for *each individual task* are shown below and above the x-axis, respectively. It is important to note that the timelines do not indicate an actual schedule, but rather best and worst-case possibilities for each task. The arrows represent releases of tasks. Since the deadline of a task is assumed to be equal to the period of the task, a release of a task serves as the deadline for the previous release of the task. The shaded rectangles represent task execution in a preemptive region where no shared resource is accessed and the hatched rectangles represent task execution in a region where it accesses one or more shared resources. Different resources are depicted using different styles of hatching.



(a) Results using ResourceSharingAnalysis



(b) Results using NPRAnalysis

Figure 1. Best and Worst Case Results for Task Set 1

Instead of examining the entire timeline, the portions of the timeline that exhibit differences between NPRAnalysis and ResourceSharingAnalysis are examined. First, consider time 10. At this time, task T_2 is executing and the first instance of task T_1 is released. There is a possibility that T_2 has already finished executing region $r1$ and has entered region $r2$ (best case), but it is not guaranteed to be so (worst case). In NPRAnalysis (shown in Figure 1(b)), the assumption is that a task executing in its NPR is not preemptible by any other task. In this situation, the worst-case scenario for task T_1 is that it has to wait for task T_2 to finish executing its NPR. On the other hand, in ResourceSharingAnalysis (shown in Figure 1(a)), although a shared resource is not preemptible until a task volutarily relinquishes it, the task itself can be preempted until some shared resource being held by it is required by a higher-priority task. Hence, in this situation, task T_1 preempts task T_2 in both the best and the worst cases.

While using ResourceSharingAnalysis (1(a)), at time 12, task T_1 finishes executing region $r1$ in the worst-case and enters region $r2$. On entering region $r2$, T_1 requests access to resource R_1. Since there is a possibility that T_2 has already acquired that resource (best case for T_2), the worst case for T_1 is to allow T_2 to execute and wait until the resource is relinquished. On the other hand, since there is a possibility that the resource R_1 has still not been acquired by T_2, the best case for T_1 is that it acquires R_1 immediately.

At time 15, an instance of task T_0 is released. Once again, in Figure 1(b), since there is a possibility that T_2 has already entered its NPR, the worst case for T_0 is for it to wait for the completion of T_2's NPR. Hence, it is scheduled to start at time 24. On the other hand, in Figure 1(a), since T_2 can be preempted, T_0 gets scheduled immediately in both the best and the worst cases. Furthermore, in ResourceSharingAnalysis, since T_2 never requests resource R_1, it can complete executing all its regions, resulting in a response time equal to its WCET. This example demonstrates the advantages of using a resource sharing policy as opposed to assuming that a task in a NPR is not preemptible at all.

5. Data-Cache Related Delay

In the examples provided in Section 4.1, the data-cache related delays are assumed to be zero. In this section, calculation of data-cache related delays is described. As explained in prior work [18], the data-cache related preemption delay (D-CRPD) of a task is calculated by identifying the range of iteration points at which a task is guaranteed to be within at the time that it is preempted. The delay at each point in this range is calculated using the access chains for the preempted task and the maximum among these is assumed to be the preemption delay at the given preemption point.

When a resource-sharing policy is used to control accesses to shared resources in a system, a situation may arise where a task that requests a resource is denied access to the resource because another lower-priority task has already acquired the same resource at an earlier point in time. In this situation, the task requesting the resource gets *blocked* and the lower-priority task is scheduled and allowed to execute until the required resource is relinquished by it. Since the task requesting the resource has already started its execution, there is a possibility that it loads some of its data into the data cache. When it gets blocked and lower-priority tasks are allowed to execute, some of these data cache lines may potentially be evicted from the data cache by the lower-priority tasks. Consequently, when the blocked task resumes execution at a later point in time, it experiences an additional delay to reload the evicted data cache lines into the data cache, similar to that experienced due to preemption. This delay is termed Data-Cache Related Blocking Delay (D-CRBD).

Calculations of the D-CRBD of a task are performed in a manner similar to that of D-CRPD. However, there are two distinctions. In the case of D-CRPD, the exact point of execution of the preempted task at the time of preemption is unknown. Instead, a range of iteration points where the task

may be is identified. In the case of D-CRBD, since blocking occurs at the time when a resource is requested, the exact iteration point of the requesting task at the time is known.

The second distinction occurs in the identification of data cache lines that may be used by tasks that are responsible for causing the delay. In the case of D-CRPD, all cache lines used by all tasks with priority higher than the preempted task may potentially be candidates for eviction and, hence, need to be considered as such. On the other hand, in the case of D-CRBD, only the data cache lines used in specific resource-usage regions of specific tasks need to be considered.

The algorithm used to calculate the blocking time for a task that is blocked due to request for a particular resource is shown in Figure 2. In addition to the resource usage time remaining for the task that currently holds the requested resource, nested resource usage must be taken into account. For example, assume that a task T_0 requests a resource R_1 and gets blocked on that account by task T_1. The blocking time for task T_0 includes the resource usage time remaining for R_1 by task T_1 and the blocking times that T_1 might in turn incur due to other resources that it requests while holding resource R_1. The union of data cache lines used in the regions thus identified forms the set of data cache lines that may potentially be used while task T_0 is blocked and, hence, may contribute to the D-CRBD experienced by task T_0.

6. Correctness of Analysis

In the context of resource-sharing tasks, response time of a job is the sum of five components, namely the base WCET of the task, the execution time of higher-priority jobs, the D-CRPD incurred due to preemption by higher-priority jobs, the blocking time incurred due to shared resources and the D-CRBD incurred due to blocking by lower-priority jobs. Formulations for each of these components for a job $J_{i,j}$ and proofs of their correctness are presented in this section.

The new symbols introduced in this formulation are explained as follows. nr_i represents the number of regions within a task T_i. $C^r_{i,j}$ represents the WCET of region r of a job $J_{i,j}$. An added superscript of rem represents remaining WCET of job $J_{i,j}$ at a given time and a superscript of $base$ represents its base WCET. $rel_{i,j}$ represents the time of release of job $J_{i,j}$, calculated as $(\phi_{i,j} + (j-1) \cdot P_i)$. I represents an interval between two consecutive releases of higher-priority jobs for $J_{i,j}$ and t^{rem}_I represents the time remaining before the end of an interval I. $\Delta^I_{i,j}$ represents the data-cache related delay experienced by job $J_{i,j}$ due to preemption by the release of a higher-priority job at the end of an interval I. $hp(i,j)$ and $lp(i,j)$ represent the sets of jobs that have a higher and lower priority, respectively than job $J_{i,j}$. $bst^r_{i,j}$ and $wst^r_{i,j}$ represent the earliest and latest possible start time, respectively for region r of job $J_{i,j}$. $\Delta^R_{i,j}$ represents the data-cache related delay experienced by job $J_{i,j}$ due to blocking by a lower-priority job using a resource R that is required by $J_{i,j}$. $\delta^r_{i,j}$ represents the delay incurred by a higher-priority job due to blocking by region r of a lower-priority job $J_{i,j}$. $Res_{i,j}$ represents the set of resources used by job $J_{i,j}$ and $Res^r_{i,j}$ represents the set of resources used by $J_{i,j}$ in a specific region r. $bcreq^{R,r}_{i,j}$ and $wcreq^{R,r}_{i,j}$ represent the best and worst-case request times, respectively for resource R within region r of job $J_{i,j}$. Similarly, $bcrel^{R,r}_{i,j}$ and $wcrel^{R,r}_{i,j}$ represent the best and worst-case release times for R. Resource request and release times are relative to the start of the region in which they are used. Due to the usage of resource-sharing protocols, the priority of a job may be different at different points of time. $cp^t_{i,j}$ represents the current priority of $J_{i,j}$ at time t and $chp(i,j)^t$ represents the set of jobs that have a higher priority than job $J_{i,j}$ at a time t.

Theorem 6.1: The response time of a job $J_{i,j}$, calculated as the sum of the values produced by Equations 1, 6, 7, 10 and 12, is a safe upper bound on the worst-case response time of $J_{i,j}$ in the context of resource-sharing tasks.

The correctness of the theorem is proved using Lemmas 6.2 , 6.3, 6.4 and 6.5.

Lemma 6.2: An upper bound on the execution time of a job $J_{i,j}$, without considering the effects of interference from other jobs, is given by Equation 1.

$$Base\ WCET\ of\ J_{i,j} = \sum_{r=1}^{nr_i} C^{base,r}_{i,j} \tag{1}$$

The calculation of the base WCET of a job is performed using the static timing analyzer. The correctness of the static timing analyzer and, hence, that of Lemma 6.2 is assumed in this paper (see [15], [9], [14], [10] for details).

The execution time of higher-priority jobs within the response time of $J_{i,j}$ is calculated by counting the number of instances of every higher-priority task that may execute within the response time of $J_{i,j}$ and multiplying it by the execution time of the specific job. In the context of resource-sharing tasks, there may be some lower-priority job executing at an inherited priority that is higher than $J_{i,j}$ and, hence, need to be considered as a higher-priority job. Calculation of the set of lower-priority jobs that need to be considered as higher-priority jobs is shown in Equation 2. Calculation of the execution time of higher-priority jobs is shown in Equation 3.

$$\begin{aligned} setlp_{i,j} = \{(m,n)\}\ s.t. & ((m,n) \in (lp(i,j) \cap chp(i,j)^t)) \\ & \wedge (cp^{t-1}_{m,n} \neq cp^t_{m,n}), \\ & \forall t\ s.t.\ (t > rel_{i,j}) \wedge (t < \Re_{i,j}) \end{aligned} \tag{2}$$

```
function: calculateWaitTime(requesting_task, res,
  task_holding_resource, wait_time)
  wait_time ← wait_time +
    resource usage time remaining for task_holding_resource
  for (every resource other_res requested by
    task_holding_resource while holding res) {
    wait_time_res ← 0
    checkIfAvailableAndCalculateWaitTimeIfNot
      (task_holding_resource, other_res, wait_time_res)
    wait_time ← wait_time + wait_time_res
  }
```

```
function: checkIfAvailableAndCalculateWaitTimeIfNot
  (requesting_task res, wait_time)
  acquirers_bc ← possible acquirers of res in best case
  acquirers_wc ← possible acquirers of res in worst case
  if (requesting_task has not already waited for res) {
    if (either acquirers_bc or acquirers_wc contains tasks) {
      for (every task acquirer in acquirers_bc)
        calculateWaitTime(requesting_task, acquirer,
          res, wait_time)
      for (every task acquirer in acquirerswc)
        calculateWaitTime(requesting_task, acquirer,
          res, wait_time)
    }
  }
```

Figure 2. Algorithm to Calculate D-CRBD

$$hpex_{i,j} = \sum_{(k,l)\in hp(i,j)} (\lceil \frac{\Re_{i,j}}{P_k} \rceil \cdot \sum_{r=1}^{nr_k} C_{k,l}^r) + \sum_{(m,n)\in setlp_{i,j}} (\lceil \frac{\Re_{i,j}}{P_m} \rceil \cdot \sum_{r=1}^{nr_m} C_{m,n}^{rem,r}, \forall r \; s.t. \; Res_{m,n}^r \neq \emptyset) \quad (3)$$

Since our methodology calculates response times for every *job* in the task set, the relative phasing between jobs is known. Using this information, the calculation in Equation 3 is tightened. After the release of $J_{i,j}$, the time during which no other higher-priority job is released may be calculated using information about relative phasing as shown in Equation 4. The execution time remaining after the release of $J_{i,j}$ for any higher-priority job released *before* $J_{i,j}$ is calculated as shown in Equation 5.

$$at_{i,j} = \min_{(k,l)\in hp(i,j)} [\max(\lceil \frac{rel_{i,j} - \phi_{k,l}}{P_k} \rceil \cdot P_k, 0) + \phi_{k,l} - rel_{i,j}] \quad (4)$$

$$rem_{reli,j} = \sum_{(k,l)\in hp(i,j), rel_{k,l} < rel_{i,j}} C_{k,l}^{rem} \quad (5)$$

The difference between the times calculated in Equations 4 and 5 gives the time for which $J_{i,j}$ may execute without being preempted. Equation 6 shows the calculation for the new, tighter estimate on the execution time of higher-priority jobs within the response time of $J_{i,j}$, performed in accordance with our methodology.

$$hpex_{i,j} = \sum_{(k,l)\in hp(i,j)} (\lceil \frac{\Re_{i,j} - \max((at_{i,j} - rem_{reli,j}), 0)}{P_k} \rceil \cdot \sum_{r=1}^{nr_k} C_{k,l}^r) + (\sum_{(m,n)\in setlp_{i,j}} (\lceil \frac{\Re_{i,j} - \max((at_{i,j} - rem_{reli,j}), 0)}{P_m} \rceil \cdot \sum_{r=1}^{nr_m} C_{m,n}^{rem,r}, \forall r \; s.t. \; Res_{m,n}^r \neq \emptyset)) \quad (6)$$

Lemma 6.3: An upper bound on the execution time of higher-priority jobs within the response time of a job $J_{i,j}$, in the context of resource-sharing tasks, is given by Equation 6.

Proof: Assume that $\max((at_{i,j} - rem_{reli,j}), 0)$ is not subtracted from the iterative portion of Equation 6. It means that this time can be stretched due to *execution of higher-priority jobs* in between. By definition of $(at_{i,j} - rem_{reli,j})$, *all higher-priority jobs* released before $J_{i,j}$ *have completed execution* and no higher-priority jobs have been released yet after $J_{i,j}$. Contradiction. Hence, $\max((at_{i,j} - rem_{reli,j}), 0)$ can be subtracted from the iterative portion of Equation 6 without jeopardizing safety of the analysis. □

Every release of a higher-priority job is a potential preemption point for $J_{i,j}$. Consider an interval between two such consecutive releases. According to our methodology, the job release at the end of this interval can be a feasible preemption point for $J_{i,j}$ if a) there is a possibility that $J_{i,j}$ is scheduled in the interval and b) there is a possibility that $J_{i,j}$ has not completed execution before the end of the interval. These conditions are mathematically represented

and the preemption delay is given by Equations 7.

$$cond_a = \sum_{(k,l)\in hp(i,j)} c_{k,l}^{rem} < t_I^{rem}$$
$$cond_b = \sum_{(k,l)\in hp(i,j)} C_{k,l}^{rem} + C_{i,j}^{rem} > t_I^{rem}$$
$$PD_{i,j} = \sum \Delta_{i,j}^{I}, \forall I \; s.t. \; (cond_a \wedge cond_b) \quad (7)$$

Lemma 6.4: An upper bound on the data-cache related delay experienced by job $J_{i,j}$ due to preemptions by higher-priority jobs, in the context of fully preemptive tasks, is given by Equation 7.
Lemma 6.4 has been proved in prior work [18].

Calculation of the set of regions within a lower-priority job that could block a higher-priority job $J_{i,j}$ requesting a resource R is shown in Equation 8.

$$setreg_{i,j}^{R} = \{r\} \; s.t. R \in Res_{k,l}^{r} \wedge bst_{k,l}^{r} + bcreq_{k,l}^{R,r} < rel_{i,j} \wedge wst_{k,l}^{r} + wcrel_{k,l}^{R,r} > rel_{i,j} \quad (8)$$

The calculation of the blocking time that $J_{i,j}$ experiences due to denial of resource R is given by Equation 9 and the total blocking time for $J_{i,j}$ is given by Equation 10.

$$B_{i,j}^{R} = \max_{(k,l)\in lp(i,j), R\in Res_{k,l}, r\in setreg_{i,j}^{R}} [C_{k,l}^{rem,r} + \sum_{R'\in Res_{k,l}^{r}, req(R')\geq req(R), rel(R')\leq rel(R)} B_{k,l}^{R'}] \quad (9)$$

$$B_{i,j} = \sum_{R\in Res_{i,j}} B_{i,j}^{R} \quad (10)$$

Due to potential blocking by the regions identified in Equation 8, job $J_{i,j}$ experiences data-cache related delay. The calculation of the data-cache related that $J_{i,j}$ experiences due to denial of resource R is given by Equation 11 and the total data-cache related blocking delay for $J_{i,j}$ is given by Equation 12. Note that the formulae for blocking time and blocking delay are specific to the Priority Inheritance Protocol.

$$\Delta_{i,j}^{R} = \uplus_{(k,l)\in lp(i,j), R\in Res_{k,l}, r\in setreg_{i,j}^{R}} [\delta_{k,l}^{r} + \sum_{R'\in Res_{k,l}^{r}, req(R')\geq req(R), rel(R')\leq rel(R)} \Delta_{k,l}^{R'}] \quad (11)$$

$$\Delta_{i,j} = \sum_{R\in Res_{i,j}} \Delta_{i,j}^{R} \quad (12)$$

Lemma 6.5: a) An upper bound on the blocking time that job $J_{i,j}$ experiences due to lower-priorty jobs holding resources required by $J_{i,j}$ is given by Equation 10.

b) An upper bound on the data-cache related delay that job $J_{i,j}$ experiences due to all possible blocking scenarios identified using Equation 10 is given by Equation 12.

Proof: Priority inheritance is transitive. If a job $J_{i,j}$ is blocked on resource R by a lower-priority job $J_{k,l}$, it is possible that $J_{k,l}$ in turn gets blocked on resource $R^{'}$ by $J_{m,n}$, which has a priority lower than $J_{k,l}$. By definition of the Priority Inheritance Protocol, $J_{m,n}$ transitively inherits the priority of $J_{i,j}$ and finishes using resource $R^{'}$. Then, it resumes its initial priority and $J_{k,l}$ executes at the priority of $J_{i,j}$ until it relinquishes R. This transitive property of PIP proves the recursive part of the calculation shown in Equations 9 and 11.

The correctness of the direct blocking time and blocking delay is now proved. Assume region r of $J_{k,l}$ directly blocks $J_{i,j}$ due to resource R.

a) $R \in Res_{k,l}$ is a necessary condition since the resource has to be used in region r in order to block.

b) Assume $bst_{k,l}^{r} + bcreq_{k,l}^{R,r} >= rel_{i,j}$. This implies that, even in the best case, resource R *has not yet been acquired* by the lower-priority job $J_{k,l}$ before the release of $J_{i,j}$. Hence, region r *of* $J_{k,l}$ *cannot directly block* $J_{i,j}$. Contradiction. Hence, $bst_{k,l}^{r} + bcreq_{k,l}^{R,r} < rel_{i,j}$.

c) Assume $wst_{k,l}^{r} + wcrel_{k,l}^{R,r} <= rel_{i,j}$. It means that, even in the worst case, resource R *has already been relinquished* by the lower-priority job $J_{k,l}$ before the release of $J_{i,j}$. Hence, region r *of* $J_{k,l}$ *cannot directly block* $J_{i,j}$. Contradiction. Hence, $wst_{k,l}^{r} + wcrel_{k,l}^{R,r} > rel_{i,j}$.

a), b) and c) demonstrate that the three conditions are necessary in order to ascertain whether a region can directly block a higher-priority job that requests a particular resource.

Assume region r of $J_{k,l}$ does not directly block $J_{i,j}$ due to resource R. Assume that all three conditions for region r in Equation 9 are satisfied. It means that there is a *possibility that resource* R *has been acquired* by $J_{k,l}$, but *no guarantee that it has been relinquished*, before the release of $J_{i,j}$. Hence, region r *of* $J_{k,l}$ *directly blocks* $J_{i,j}$. Contradiction. This proves that the three conditions specified in Equation 9 are sufficient to determine whether a region directly blocks a higher-priority job. Once the regions that could block job $J_{i,j}$ are identified, the data-cache related blocking delay calculation is a union of delays due to each region. □

Proof: Assume that the sum of the values produced by Equations 1, 6, 7, 10 and 12 is not a safe upper bound on the worst-case response time of a job. This implies that the value produced by at least one of the equations *is an underestimation of the specific component* represented by the equation. Lemmas 6.2, 6.3, 6.4 and 6.5 *demonstrate the correctness of each component* of the response time of a job as a *safe upper bound*. Contradiction. Hence, the sum of the values produced by Equations 1, 6, 7, 10 and 12 *is a safe upper bound* on the worst-case response time of the job. □

7. Experimental Results

The experimental setup used in this paper is similar to that used in prior work [18] and is omitted due to space constraints. In all experiments, task sets that have a base utilization (utilization without considering data cache related delays) of 0.5 and 0.8 are used. Task sets of different sizes (2, 4, 6, 8) are constructed for both these utilizations. For a utilization of 0.8, a task set consisting of 10 tasks is also constructed.

The characteristics of the task sets constructed are shown in Table 2 for a base utilization of 0.8. The table indicates the task IDs, phases (cycles) and periods (cycles) of each task in the various task sets. The task IDs correspond to those assigned to tasks in prior work [18]. Task IDs that only have a single number indicate that the corresponding task does not use any shared resource. In contrast, IDs of tasks that use a shared resource are assigned a suffix of a dash followed by a number. This new ID is used to distinguish between different resource usage characteristics.

Table 2. Task Set Characteristics for Resource Sharing Tasks - U = 0.8

# Tasks	**2**	**4**	**6**	**8**
IDs	27-1, 26-1	28, 13-1, 27-2, 19	21-1, 8-1, 20, 13, 25, 19	8, 26, 20-1, 15-2, 9, 11, 8, 21
Phases	60K, 0	100K, 70K, 0, 0	60K, 0, 0, 0, 0, 0	27K, 27K, 27K, 0, 0, 0, 0, 0
Periods	300K, 500K	500K, 500K, 1000K, 2000K	400K, 500K, 500K, 1000K, 1000K, 2000K	400K, 500K, 800K, 800K, 1000K, 2000K, 2000K, 4000K
# Tasks=10				
IDs	10-1, 8, 15, 9, 5, 11-2, 20-2, 27, 22, 17			
Phases	85.2K, 85.2K, 85.2K, 85.2K, 85.2K, 54K, 0, 0, 0, 0			
Periods	100K, 625K, 625K, 625K, 1000K, 1000K, 1250K, 1250K, 2500K, 5000K			

Table 3 shows the resource usage characteristics for tasks that use some shared resource. The first column indicates a task ID that corresponds to task IDs in Table 2. The second column shows the resource being used and the third and fourth columns indicate the iteration points at which the resource is requested and released, respectively. The format of the iteration point is as follows. Each pair of numbers within parantheses indicates one loop level, starting with the outermost level and proceeding inwards. Within each pair, the first number indicates the number of the loop in the current level (in case of sequential loop nests) and the second number indicates the iteration number within that loop.

Results for the task sets in Table 2, obtained using both the RM and the EDF scheduling policies, are shown in Figure 3, respectively. For each task set, results using two different analysis techniques are presented. The first technique is ResourceSharingAnalysis, which employs a resource-sharing protocol (specifically, the Priority Inheritance Protocol) to control accesses to shared resources as described in this chapter. The second technique is NPRAnalysis (discussed in prior work [19]) and results obtained using this analysis are shown for the sake of comparison. In the case of NPRAnalysis, any region where a shared resource is used is assumed to be a non-preemptive region, *i.e.*, a region during which a task cannot be preempted by *any* other task.

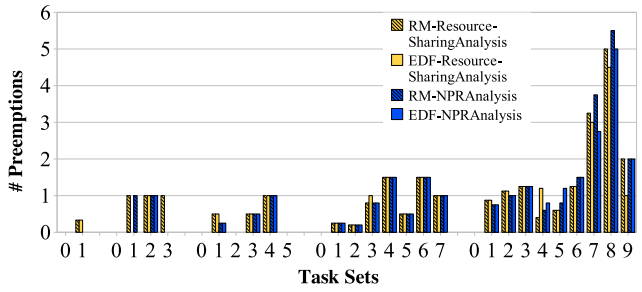
Table 3. Resource Usage Characteristics

Task ID	**Resource**	**Request Iter**	**Release Iter**
8-1	R_1	(1, 0), (2, 400)	(1, 0), (2, 600)
	R_2	(1, 0), (2, 500)	(1, 0), (2, 600)
	R_3	(1, 0), (2, 800)	(1, 0), (2, 850)
10-1	R_1	(1, 0), (2, 30)	(1, 0), (2, 50)
11-1	R_1	(1, 0), (2, 150)	(1, 0), (2, 250)
11-2	R_1	(1, 0), (2, 150)	(1, 0), (2, 200)
	R_2	(1, 0), (2, 170)	(1, 0), (2, 190)
13-1	R_2	(1, 0), (2, 300)	(1, 0), (2, 450)
15-1	R_1	(1, 0), (4, 5), (1, 5)	(1, 0), (4, 6, (1, 2)
	R_2	(1, 0), (4, 8), (1, 2)	(1, 0), (4, 9, (1, 8)
15-2	R_1	(1, 0), (4, 5), (1, 5)	(1, 0), (4, 8, (1, 8)
18-1	R_2	(1, 0), (2, 300)	(1, 0), (2, 400)
19-1	R_1	(1, 0), (2, 350)	(1, 0), (2, 500)
20-1	R_1	(1, 0), (2, 300)	(1, 0), (2, 400)
20-2	R_2	(1, 0), (2, 450)	(1, 0), (2, 500)
21-1	R_1	(1, 0), (2, 300)	(1, 0), (2, 400)
23-1	R_1	(1, 0), (2, 50)	(1, 0), (2, 75)
26-1	R_1	(1, 0), (2, 650)	(1, 0), (2, 750)
27-1	R_2	(1, 0), (2, 450)	(1, 0), (2, 650)
27-2	R_1	(1, 0), (2, 400)	(1, 0), (2, 800)
	R_2	(1, 0), (2, 650)	(1, 0), (2, 750)

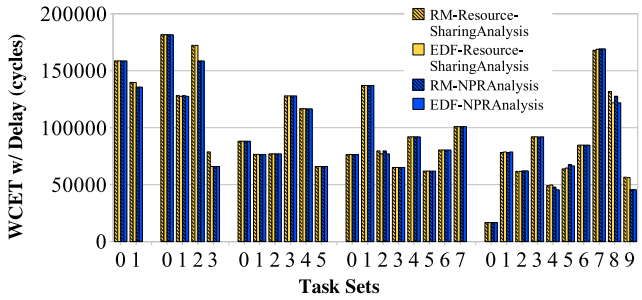
The technique presented in this paper extends from NPRs to resource-sharing protocols without loss of tightness. The method itself, bounding D-CRPD for resource-sharing tasks, is without precedence. Hence, no comparison with prior work can be presented.

From the graphs, several observations may be made. First of all, RM and EDF exhibit little or no differences. In cases where they do exhibit differences (some task sets with utilization = 0.8), the behavior is as expected. The EDF policy sometimes increases the response times of tasks with shorter periods (higher priority according to the RM policy) and sometimes decreases the response times of tasks with longer periods, compared to the RM policy. This is due to the fact that the relative deadlines of jobs alter their priorities.

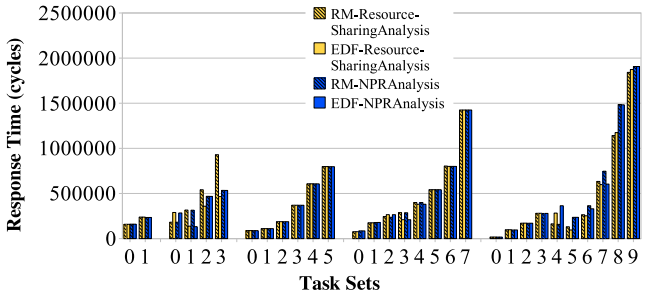
It may be observed from the graphs that tasks with a high priority sometimes have a higher response time in the case of NPRAnalysis compared to ResourceSharingAnalysis. This is expected since NPRAnalysis disallows preemptions altogether when a task is executing in a critical section and could thereby cause a delay in the start of execution of



(a) # Preemptions for U = 0.8



(b) WCET w/ Delay for U = 0.8



(c) Response Time for U = 0.8

Figure 3. Results for U=0.8 using RM and EDF Policies

some higher-priority tasks. On the other hand, in the case of ResourceSharingAnalysis, a higher-priority does not have to wait unless (and until) it requires a resource that has been acquired by some lower-priority task.

For some higher-priority tasks, however, it may be observed that the response time is higher in the case of ResourceSharingAnalysis compared to NPRAnalysis. This is possible due to the fact that, in ResourceSharingAnalysis, a task may get blocked by a lower-priority task when it requests a resource and, consequently, may experience some data-cache related blocking delay. In the case of NPRAnalysis, although the start of execution of a higher-priority task may get *delayed* if a lower-priority task is executing in its critical section, a higher-priority task can never get *blocked* by a lower-priority task once it begins to execute. Hence, in NPRAnalysis, a task does not experience data-cache related blocking delay.

Another observation that may be made from the graphs is that some lower-priority tasks have a higher number of preemptions in the case of ResourceSharingAnalysis compared to NPRAnalysis while others have a lower number of preemptions. Although this may seem contradictory, both these results are valid. Some lower-priority task might be executing in a critical section when a higher-priority task is released. In this situation, NPRAnalysis disallows preemption of the lower-priority task whereas ResourceSharingAnalysis does not. Hence, the number of preemptions could be more in the case of ResourceSharingAnalysis. In some cases, due to relative positioning of jobs and the data-cache related delays experienced by tasks, lower-priority tasks could have a lower number of preemptions in ResourceSharingAnalysis compared to NPRAnalysis.

Based on the above observations, there is no clear answer to the question of whether using resource-sharing protocols is a better option than making critical sections completely non-preemptive when *data-cache related delays* are taken into account. The answer is dependent on the characteristics of the task set at hand. Analysis techniques, such as the ones presented in prior work [19] and in this paper, may be used to statically determine which method is better suited for a given task set.

8. Conclusions and Future Work

In this paper, we have presented a technique to incorporate resource-sharing protocols into the calculation of the data-cache related delays and, hence, the worst-case execution times and the worst-case response times of hard real-time tasks that may contain critical sections within their execution. Through experimental results, we demonstrate that the responsiveness of higher-priority tasks that do not use a resource that has possibly been acquired by some lower-priority task is improved compared to making every critical section non-preemptive. However, we also observe from experimental results that there is no clear overall choice between using resource-sharing protocols and making a critical section non-preemptive when data-cache related delays are considered. The answer is dependent on the characterisitcs of the task set being analyzed. The techniques presented in this paper and in our prior work may be employed to determine the suitability of each choice, in terms of overall schedulability, for a specific task set. To the best of our knowledge, this is the first framework that calculates data-cache related delay in the context of resource-sharing tasks. As part of future work, we wish to extend our framework to incorporate other resource-sharing protocols such as the Priority Ceiling Protocol, the Stack Resource Protocol, etc.

References

[1] R. Bril. Existing worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption refuted. In *Work in Progress (WiP) session of the 18th Euromicro Conference on Real-Time Systems*, July 2006.

[2] R. Bril, J. Lukkien, and W. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. Cs-report 06-34, Technische Universiteit Eindhoven (TU/e), The Netherlands, Dec. 2006.

[3] A. Burns. Pre-emptive priority based scheduling: An appropriate engineering approach. *S. Son, editor, Advances in Real-Time Systems*, pages 225–248, 1994.

[4] A. Burns and A. Wellings. Restricted tasking models. In *8th International Real-Time Ada Workshop*, pages 27–32, 1997.

[5] S. Chatterjee, E. Parker, P. Hanlon, and A. Lebeck. Exact analysis of the cache behavior of nested loops. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–297, June 2001.

[6] B. B. Fraguela, R. Doallo, and E. L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[7] L. George, N. Rivierre, and M. Spuri. Pre-emptive and non-pre-emptive real-time uni-processor scheduling. Technical report, Institut National de Recherche et Informatique et en Automatique (INRIA), France, Sept. 1996.

[8] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[9] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, Jan. 1999.

[10] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, Dec. 1995.

[11] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis or cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[12] C.-G. Lee, K. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering*, 27(9):805–826, Nov. 2001.

[13] S. Lee, C.-G. Lee, M. Lee, S. Min, and C.-S. Kim. Limited preemptible scheduling to embrace cache memory in real-time systems. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES), Lecture Notes in Computer Science (LNCS) 1474*, pages 51–64, 1998.

[14] F. Mueller. *Static Cache Simulation and its Applications*. PhD thesis, Dept. of CS, Florida State University, July 1994.

[15] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.

[16] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 148–157, Mar. 2005.

[17] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *IEEE Real-Time Embedded Technology and Applications Symposium*, Apr. 2006.

[18] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemption points. In *IEEE Real-Time Systems Symposium*, pages 212–222, Dec. 2006.

[19] H. Ramaprasad and F. Mueller. Bounding worst-case response times for tasks with non-preemptive regions. In *IEEE Real-Time Embedded Technology and Applications Symposium*, Apr. 2008.

[20] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols - an approach to real-time synchronization. Technical report, Carnegie Mellon University, Departments of CS, ECE and Statistics, Pittsburgh, Pennsylvania, 1987.

[21] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept. 1990.

[22] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *ACM International Conference on Embedded Software*, 2004.

[23] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Euromicro Conference on Real-Time Systems*, 2005.