Analysis of Global Fixed-Priority for Sporadic DAG Tasks

Son Dinh, Christopher Gill, Kunal Agrawal
Washington University in Saint Louis,
Department of Computer Science and Engineering
sonndinh,cdgill,kunal@wustl.edu

ABSTRACT

We consider global fixed-priority (G-FP) scheduling of parallel tasks, in which each task is represented as a directed acyclic graph (DAG). We highlight issues of the state-of-the-art analyses for G-FP and propose a novel technique for bounding workloads generated by carry-in and carry-out jobs. Our technique works by constructing optimization problems for which the optimal solutions serve as safe upper bounds for carry-in and carry-out workloads. Using the interference bounds, two schedulability tests are derived: a response-time analysis and a slack-based iterative test.

1 INTRODUCTION

With the prevalence of multiprocessor platforms and parallel programming languages and runtime systems such as OpenMP [22], Cilk Plus [12, 16], and Intel's Threading Building Blocks [17], the demand for computer programs to be able to run in parallel in order to exploit the parallelism offered by the hardware is inevitable. In recent years, the real-time community is also working towards catching up to this trend as there are many real-time programs that require parallel execution to satisfy their deadlines, such as real-time hybrid structural simulation [10], and applications in autonomous vehicles [18]. Much effort has been made to develop analysis techniques and schedulability tests for scheduling parallel real-time tasks under scheduling algorithms such as Global Earliest Deadline First (G-EDF), and Global Deadline Monotonic (G-DM).

Schedulability analysis for parallel tasks is inherently more complex than conventional sequential tasks. This is because *intra-task parallelism* is allowed within individual tasks, which enables each individual task to execute simultaneously upon multiple processors. The parallelism of each task can also vary as it is executing, as it depends on the precedence constraints imposed on the task. Consequently, this raises questions of how to account for the inter-task interference caused by other tasks on a task and the intra-task interference caused by the task itself when some but not all threads of the task are interfered. In this paper, we consider task systems that consist of parallel tasks scheduled under Global Fixed-Priority (G-FP), in which each task is represented by a Directed Acyclic Graph (DAG). Our analysis is based on the concepts of critical interference and critical chain introduced in [8, 9, 21]. These concepts allow the analysis to focus on a special chain of sequential segments of each task, hence enable us to reuse techniques developed for sequential tasks [2, 4–6].

Our contributions in this paper are the following:

- We highlight the issues of the state-of-the-art analyses for G-FP, specifically the calculation of interferences of carry-in jobs and carry-out jobs.
- We propose a new technique for computing safe interference bounds by transforming the problem to maximization problems that can be solved by existing optimization solvers.
- We derive a response-time analysis and a slack-based schedulability test using the interference bounds computed by the new bounding technique.

The rest of this paper is organized as follows. In Sections 2 and 3 we discuss related work and present the task model we consider in this paper. Section 4 reviews the concepts of critical interference and critical chain and Section 5 presents a general method for bounding response-times that is used by the previous analyses and our analysis. In Section 5 we also highlight the issues with the state-of-the-art analyses. In Sections 6 and 7 we propose a new technique to bound carry-in and carry-out workloads. A response-time-based schedulability test and a slack-based schedulability test are derived in Sections 8 and 9. We conclude our work in Section 10.

2 SURVEY OF RELATED WORK

For the sequential task model, Bertogna et al. [4] proposed a response-time analysis that works for G-EDF and G-FP. They bound the interference of a task in a problem window by the worst-case workload it can generate in the problem window. The worst-case workload is then bounded by considering a worst-case release pattern of the interfering task. This technique was later extended by others to analyze parallel tasks and is used in this work. Bertogna et al. [6] proposed a sufficient slack-based schedulability test for G-EDF and G-FP in which the slack values for the tasks are used in an iterative algorithm to improve the schedulability gradually. In this paper, we extend that idea for DAG tasks scheduled under G-FP (Section 9). Later Guan et al. [13] proposed a new response-time analysis for both constrained-deadline and arbitrary-deadline tasks.

Initially, simple parallel real-time task models were studied, such as the fork-join task model and the synchronous task model. Lakshmanan et al. [19] presented a transformation algorithm to schedule fork-join tasks where all parallel segments of each task must have the same number of threads, which must be less than the number of processors. They also proved a resource augmentation bound of 3.42 for the algorithm. Saifullah et al. [23] improved on that work by removing the restriction on the number of threads in parallel segments. They proposed a task decomposition algorithm and proved resource augmentation bounds for the algorithm under G-EDF and Partitioned Deadline Monotonic (P-DM) scheduling. Axer et al. [1] presented a response-time analysis for fork-join tasks under Partitioned Fixed-Priority (P-FP) scheduling. Chwa et al. [9] developed an analysis for synchronous parallel tasks scheduled under G-EDF. They introduced the concept of critical interference and presented a sufficient test for G-EDF. Maia et al. [20] reused the concept of

critical interference to introduce a response-time analysis for synchronous tasks scheduled under G-FP.

A general parallel task model was presented by Baruah et al. [3] in which each task is modeled as a Directed Acyclic Graph (DAG) and can have an arbitrary deadline. They presented a polynomial test and a pseudo-polynomial test for a DAG task scheduled with EDF and proved their speedup bounds. However, they only consider a single DAG task. Bonifaci et al. [7] later developed feasibility tests for task systems with multiple DAG tasks, scheduled under G-EDF and G-DM. Melani et al. [21] proposed a response-time analysis for conditional DAG tasks where each DAG can have conditional vertices to model conditional constructs. Their analysis utilizes the concepts of critical interference and critical chain, and works for both G-EDF and G-FP. However, the bounds for carry-in and carry-out workloads are likely to be overestimated since they ignore the internal structures of the tasks. Chwa et al. [8] extended their work in [9] for DAG tasks scheduled under G-EDF. They proposed a sufficient, workload-based schedulability test and improved it by exploiting slack values of the tasks. In this paper, we present a schedulability test for G-FP applying a similar idea but with our own technique to bound interfering workloads (Section 9). Fonseca et al. [11] proposed a response-time analysis for sporadic DAG tasks scheduled under G-FP that improves upon the response-time analysis in [21]. They improve the upper bounds for interference by taking the DAGs of the tasks into consideration. In particular, by explicitly considering the DAGs the workloads generated by the carry-in and carry-out jobs can be reduced compared to the ones in [21], and hence schedulability can be improved. When bounding carry-in workloads they assume that the maximum carry-in workloads are generated when (i) vertices of carry-in jobs execute as soon as they are ready and (ii) all vertices execute for their whole WCETs. However, these assumptions do not always correspond to maximum carry-in workloads, and thus may not yield safe upper bounds. We discuss this observation and differentiate our work in Section 5.

3 SYSTEM MODEL

We consider a set τ of n real-time parallel tasks, $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$, scheduled preemptively by a global fixed-priority scheduling algorithm upon m identical processors. Each task τ_i is a recurrent, sporadic process which may release an infinite sequence of jobs and is modeled by $\tau_i = \{G_i, D_i, T_i\}$, where D_i denotes its relative deadline and T_i denotes the minimum inter-arrival time of two consecutive jobs of τ_i. We assume that all tasks have constrained deadlines, i.e., $D_i \leq T_i, \forall i \in [1, n]$. Each task τ_i is represented as a Directed Acyclic Graph (DAG) $G_i = (V_i, E_i)$, where $V_i = \{v_1, v_2, ..., v_{n_i}\}$ is the set of vertices of the DAG G_i and $E_i \subseteq (V_i \times V_i)$ is the set of directed edges of G_i. In this paper, we also use *subtasks* and *nodes* to refer to the vertices of the tasks. Each subtask $v_{i,a}$ of G_i represents a section of instructions that can only be run sequentially. A subtask $v_{i,a}$ is called a *predecessor* of $v_{i,b}$ if there exists an edge from $v_{i,a}$ to $v_{i,b}$ in G_i, i.e., $(v_{i,a}, v_{i,b}) \in G_i$. Subtask $v_{i,b}$ is then called a *successor* of $v_{i,a}$. Each edge $(v_{i,a}, v_{i,b})$ represents a precedence constraint between the two subtasks, meaning that subtask $v_{i,a}$ must finish before subtask $v_{i,b}$ may start. A subtask is called *ready* if all of its predecessors have finished. Whenever a task releases a job, all of its subtasks are released and have the same deadline as the job's deadline. Let J_i^l denote the l^{th} job of task τ_i and r_i^l and d_i^l denote its release time and absolute deadline, respectively. We abort the superscript l when referring to a general job of a task or when there is no further confusion of which job is being mentioned — that is J_i, r_i, d_i are used instead.

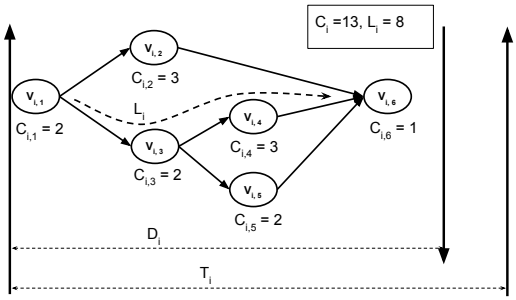


Figure 1: An example of a DAG task.

A subtask $v_{i,a}$ has a worst-case execution time (WCET), denoted by $C_{i,a}$. The sum of WCETs of all subtasks of a task τ_i is the worst-case execution time of the whole task, and is denoted by $C_i = \sum_{v_{i,a} \in V_i} C_{i,a}$. The WCET of a task is also called its *work*. A sequence of subtasks $(v_{i,u_1}, v_{i,u_2}, ..., v_{i,u_t})$ of τ_i, in which $(v_{i,u_j}, v_{i,u_{j+1}}) \forall 1 \leq j \leq t-1$ is an edge of G_i, is called a *chain* of τ_i and is denoted by λ_i. The length of a chain λ_i is the sum of the WCETs of subtasks in λ_i and denoted by $len(\lambda_i)$, i.e., $len(\lambda_i) = \sum_{v_{i,u_j} \in \lambda_i} C_{i,u_j}$. A chain of τ_i which has the longest length is a *critical path* of the task. We note that there can be more than one critical path in a DAG. The length of a critical path of a DAG is called its *critical path length* or *span*, and denoted by L_i. Intuitively, the work C_i is the worst-case execution time of τ_i on a single processor, and the span L_i is the worst-case execution time of τ_i on a hypothetical platform with infinite number of processors. Figure 1 presents an example for the DAG of a task τ_i with 6 subtasks. The work and span of τ_i are $C_i = 13$ and $L_i = 8$, respectively. In this paper, we consider that tasks are scheduled using a preemptive, global fixed-priority algorithm where each task is assigned a fixed task-level priority. Without loss of generality, we assume that tasks have distinct priorities, and task τ_i has higher priority than task τ_k if $i < k$.

4 BACKGROUND

Chwa et al. [8] extended the analysis of G-EDF for synchronous tasks (Chwa et al., 2013 [9]) to general DAG tasks. They introduced the notions of *critical chain* and *critical interference* which are useful to analyze schedulability of parallel DAG tasks. Unlike sequential tasks, analysis of DAG tasks for which internal parallelism is allowed is inherently more complicated due to: (i) some subtasks of a task can be interfered by some other subtasks of the same task (i.e., *intra-task interference*); (ii) subtasks of a task can be interfered by subtasks of higher-priority tasks (i.e., *inter-task interference*); and (iii) the parallelism of a DAG task may vary during execution,

subject to the precedence constraints imposed by its graph. The critical chain and critical interference concepts alleviate the complexity of the analysis by focusing on a special chain of a DAG task which accounts for its scheduling window, i.e., from its release time to its finish time, thus bringing the problem closer to a more familiar analysis for sequential tasks. Although they are originally proposed for analysis of G-EDF [8, 9], these concepts are also useful for analyzing G-FP. We thus use them in our analysis and include their discussion later in this section.

With regard to analysis of G-FP, Melani et al. [21] were among the first to propose a response-time analysis for DAG tasks which contain conditional vertices — which allow modeling conditional constructs, such as `if-then-else` — besides regular vertices. They derived a fixed-point iteration calculation for a response-time bound that works for both G-EDF and G-FP. However, they ignore the internal DAGs of the interfering tasks when computing interference on another task. Their calculated inter-task interference and response-time bounds are thus pessimistic. Later Fonseca et al. [11] improved the response-time bound for G-FP by taking the internal structures of the tasks' DAGs into account. They also considered similar task systems to the ones studied in this paper, i.e., constrained-deadline, sporadic DAG tasks. Our analysis differentiates from theirs in the way we calculate inter-task interference. We present our method to calculate safe bounds for inter-task interference in Sections 5, 6 and 7, and first discuss the concepts of critical chain and critical interference on which our analysis relies.

Consider any job J_k of a task τ_k and its corresponding schedule. A *last-completing subtask* of J_k is a subtask that completes last among all subtasks in the schedule of J_k (there can be more than one last-completing subtask in a schedule for J_k as multiple subtasks of τ_k may finish at the same time instant). A *last-completing predecessor* of a subtask $v_{k,a}$ is a predecessor that completes last among all predecessors of $v_{k,a}$ in the schedule of J_k (similarly, there may be more than one last-completing predecessor for $v_{k,a}$). Note that a subtask can only be ready after that last-completing predecessor finishes, since only then are all the precedence constraints for the subtask satisfied. Starting from a last-completing subtask of J_k we can recursively trace back all last-completing predecessors until we reach a subtask with no predecessors. If during that process, a subtask has more than one last-completing predecessors, we arbitrarily pick one. The chain that is reconstructed by appending those last-completing predecessors and the last-completing node is called a *critical chain* of job J_k. We call the subtasks that belong to a critical chain *critical subtasks*.

Example 4.1. Figure 2 presents an example for a critical chain of a job J_k of task τ_k which has the same DAG as shown in Figure 1. In this figure, boxes with bold, solid borders denote the execution of critical subtasks of J_k; boxes with bold, dashed borders denote the execution of the other subtasks of J_k. The others are for jobs of other tasks. In this example, $v_{k,6}$ is the last-completing subtask. From $v_{k,6}$ we can find a last-completing predecessor, which is subtask $v_{k,5}$. Similarly, a last-completing predecessor of $v_{k,5}$ is $v_{k,3}$, and a last-completing predecessor of $v_{k,3}$ is $v_{k,1}$. Hence a critical chain of J_k is $(v_{k,1}, v_{k,3}, v_{k,5}, v_{k,6})$.

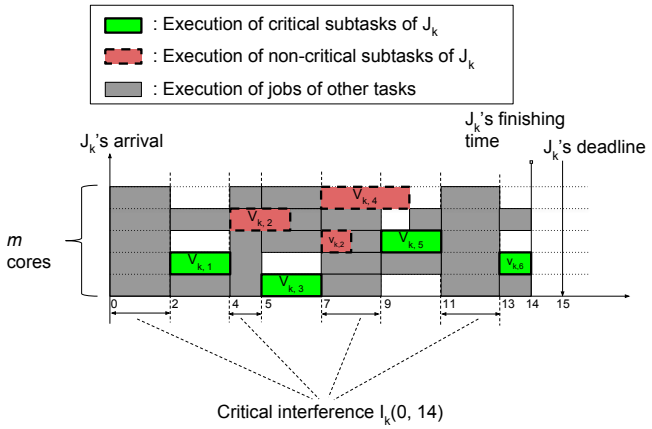


Figure 2: Critical chain and critical interference of J_k.

The critical chain concept has some properties that make it useful for schedulability analysis of parallel DAG tasks. First, the first subtask of any critical chain of a job is ready to execute as soon as the job is released since it does not have any predecessor. Second, when the last subtask of a critical chain completes, the corresponding job finishes. This is from the construction of the critical chain. Thus the scheduling window of a critical chain of J_k — i.e., from the release time of its first subtask to the completion time of its last subtask — is also the scheduling window of job J_k — i.e., from the job's release time to its completion time. Third, consider a critical chain λ_k of J_k: at any time instant during the scheduling window of J_k, either a critical subtask of λ_k is executed *or* a critical subtask of λ_k is ready but not executed because all m processors are busy executing subtasks not belong to λ_k, including non-critical subtasks of job J_k and subtasks from other tasks (see Figure 2). Therefore the "response-time" of a critical chain of J_k is also the response-time of the job. Hence if we can upper-bound the "response-time" of a critical chain for any job J_k of task τ_k, that bound also serves as an upper-bound for the response-time of task τ_k.

The third property of the critical chain suggests that we can partition the scheduling window of a job J_k into two sets of intervals. One includes all intervals during which critical subtasks of J_k are executed and the other includes all intervals during which a critical subtask of J_k is ready but not executed. The total length of the intervals in the second set is called the *critical interference* of J_k. We include the definitions for critical interference and interference caused by individual task on τ_k as follows.

Definition 4.2. Critical interference $I_k(a, b)$ on a job of task τ_k is the aggregated length of all intervals in [a, b) during which a critical subtask of τ_k is ready but not executed.

Definition 4.3. Critical interference $I_{i,k}(a, b)$ on a job of task τ_k due to task τ_i is the aggregated processor time from all intervals in [a, b) during which one or more subtasks of τ_i are executed and a critical subtask of τ_k is ready but not executed.

In Figure 2, the critical interference $I_k(0, 14)$ of J_k is the sum of the lengths of intervals $[0, 2)$, $[4, 5)$, $[7, 9)$, and $[11, 13)$ which is 7. The critical interference $I_{i,k}(0, 14)$ caused by a task τ_i is the total processor time of τ_i in those four intervals. Note that τ_i may execute simultaneously on multiple processors, and we must sum its processor time on all processors. From the definition of critical interference, we have the following equation:

$$I_k(a, b) = \frac{1}{m} \sum_{\tau_i \in \tau} I_{i,k}(a, b). \tag{1}$$

5 A GENERAL APPROACH FOR BOUNDING RESPONSE-TIME

In this section we discuss a general method for response-time analysis of G-FP that was employed by the state-of-the-art analyses [11, 21]. The high-level idea for calculating response-time bound is built on top of the ideas of critical chain and critical interference. This work is also based on that general approach for bounding response-time. However, we present a different method to bound the inter-task interference which can find tight bounds for inter-task interference and works for when subtasks require less than their WCETs. First we look at a common calculation for response-time bound that is shared by [11, 21] and our work.

Based on the definitions of critical chain and critical interference, we can derive the following equation for the response-time R_k of J_k:

$$R_k = len(\lambda_k) + I_k(r_k, r_k + R_k),$$

where λ_k is a critical chain of J_k and $len(\lambda_k)$ is its length (see Figure 2 for an example). Apply Equation 1 we have:

$$R_k = len(\lambda_k) + \frac{1}{m} \sum_{\tau_i \in \tau} I_{i,k}(r_k, r_k + R_k)$$

$$\Rightarrow R_k = \left(len(\lambda_k) + \frac{1}{m} I_{k,k}(r_k, r_k + R_k)\right) + \frac{1}{m} \sum_{\tau_i \in hp(\tau_k)} I_{i,k}(r_k, r_k + R_k), \tag{2}$$

where $hp(\tau_k)$ is the set of tasks with higher priorities than τ_k's. Tasks with lower priorities than τ_k's cannot interfere with τ_k, and thus the terms for them are absent in Equation 2, which holds for any job of task τ_k. Thus if we can bound the right-hand side of Equation 2, we can bound the response-time of τ_k. To do so, we bound the contribution to J_k's response-time caused by subtasks of J_k itself and the contribution of inter-task interference separately.

5.1 Intra-Task Interference

The sum $\left(len(\lambda_k) + \frac{1}{m} I_{k,k}(r_k, r_k + R_k)\right)$, which includes the intra-task interference on the critical chain of J_k caused by non-critical subtasks of J_k, is bounded by Lemma V.3 in [21]. We include the bound below with a small change to fit our notations.

Lemma 5.1. *The following inequality holds for any task τ_k scheduled by any work-conserving algorithm:*

$$len(\lambda_k) + \frac{1}{m} I_{k,k}(r_k, r_k + R_k) \le L_k + \frac{1}{m}(C_k - L_k)$$

5.2 Inter-Task Interference

Now we need to bound the inter-task interference on the right-hand side of Equation 2. Since the interference caused by a task in an interval is at most the workload generated by the task during that interval, we can bound the critical interference $I_{i,k}(a, b) \forall \tau_i \in hp(\tau_k)$, using the bound for the workload generated by τ_i in the interval $[a, b)$. Let $W_i(a, b)$ denote the maximum workload generated by τ_i in an interval $[a, b)$. Let $W_i(L)$ denote the maximum workload generated by τ_i in any interval of length L. The following inequality holds for all τ_i:

$$I_{i,k}(r_k, r_k + R_k) \le W_i(r_k, r_k + R_k) \le W_i(R_k). \tag{3}$$

Let the *problem window* be the interval of interest with length L. The jobs of τ_i that overlap with the problem window are classified into three types, depending on their positions with respect to the problem window: (i) A *carry-in job* is released strictly before the problem window and has a deadline within it, (ii) A *carry-out job* is released within the problem window and has its deadline strictly after it, and (iii) *body jobs* have both release time and deadline within the problem window. Similar to analysis for sequential tasks (Bertogna et al. [4]), the maximum workload generated by τ_i in the problem window can be attained with a release pattern in which (i) jobs of τ_i are released as quickly as possible, meaning that the gap between any two consecutive releases is exactly the period T_i, (ii) subtasks of the carry-in job are executed as late as possible without making the carry-in job to miss its deadline, and (iii) subtasks of all other jobs including the body jobs and the carry-out job are executed as soon as possible. Figure 3 shows an example of such a job-release pattern of an interfering task τ_i with a DAG presented in Figure 1.

For sequential tasks, the workload of τ_i in the problem window is maximized when the jobs of τ_i are released with the above pattern, and the start of the problem window aligns with the start time of the carry-in job, i.e., the time when the carry-in job starts executing [4]. However, for parallel DAG tasks the alignment of the problem window's start time with the start time of the carry-in job does not always produce the maximum workload. This is because the parallelism of a DAG task can vary subject to its graph structure. For instance, in Figure 3 if we shift the problem window to the right 2 time units, the carry-in job's workload loses 2 time units but the carry-out job's workload gains 5 time units. The total workload thus increases 3 time units. Therefore in order to find the maximum workload generated by τ_i we must slide the problem window to search for a position that corresponds to the maximum sum of the carry-in workload and carry-out workload. Regardless of the position of the problem window, the workload contributed by the body jobs is bounded as follows.

Lemma 5.2. *The workload generated by the body jobs of task τ_i in a problem window with length L is upper-bounded by*

$$\left(\left\lfloor \frac{L - L_i + R_i}{T_i} \right\rfloor - 1\right) C_i.$$

Proof. Consider the case where the start of the problem window is aligned with the starting time of the carry-in job, as shown in Figure 3. The number of body jobs is at most $\left(\left\lfloor \frac{L - L_i + R_i}{T_i} \right\rfloor - 1\right)$ since

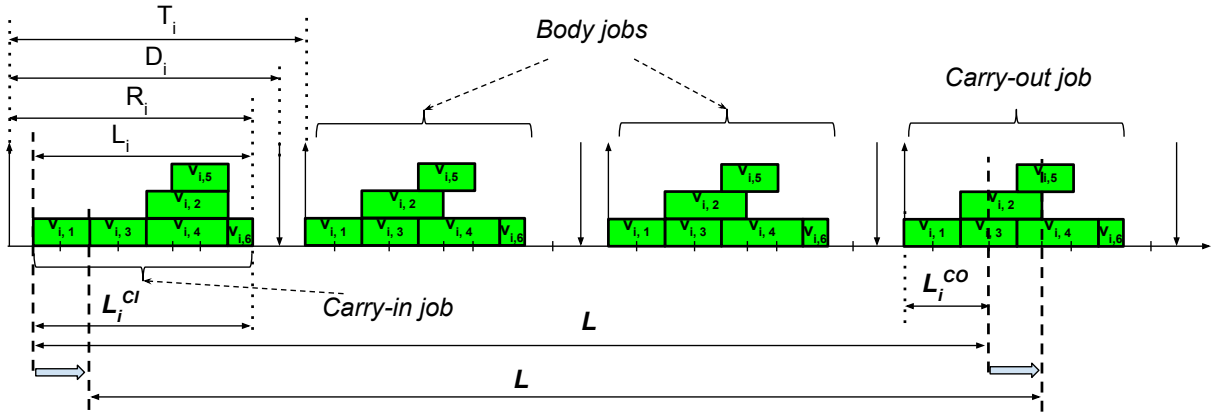


Figure 3: Workload generated by an interfering task τ_i in an interval of length L.

the carry-in job cannot complete in less than L_i time units. Thus for this case the workload of the body jobs is at most $\left(\left\lfloor \frac{L-L_i+R_i}{T_i} \right\rfloor - 1\right)C_i$.

Shifting the problem window to the left or right can change the workload contributed by the carry-in and carry-out job but does not increase the number of the body jobs and their workload. The bound thus follows. □

Let *carry-in window* and *carry-out window* be the intervals within the problem window during which the carry-in job and the carry-out job are executed, respectively. Intuitively, the carry-in window spans from the start of the problem window to the completion time of the carry-in job; the carry-out window spans from the starting time of the carry-out job to the end of the problem window. We denote the lengths of the carry-in window and carry-out window for task τ_i by L_i^{CI} and L_i^{CO} respectively. The sum of L_i^{CI} and L_i^{CO} can be computed by:

$$L_i^{CI} + L_i^{CO} = \max\left\{L - \left\lfloor \frac{L - L_i + R_i}{T_i} \right\rfloor T_i + R_i, 0\right\}. \tag{4}$$

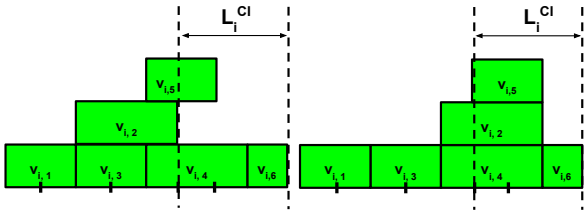
Let $W_i^{CI}(L^{CI})$ denote the maximum carry-in workload of τ_i for a carry-in window of length L^{CI}. Similarly, let $W_i^{CO}(L^{CO})$ denote the maximum carry-out workload of τ_i for a carry-out window of length L^{CO}. The maximum workload generated by τ_i in a problem window of length L, in which the carry-in window and carry-out window have length L_i^{CI} and L_i^{CO} respectively, can be computed by:

$$W_i^{CI}(L_i^{CI}) + \left(\left\lfloor \frac{L - L_i + R_i}{T_i} \right\rfloor - 1\right)C_i + W_i^{CO}(L_i^{CO}).$$

The maximum workload generated by τ_i in any problem window of length L can be found by taking the maximum for all L_i^{CI} and L_i^{CO} that satisfy Equation 4:

$$W_i(L) = \max_{L_i^{CI}, L_i^{CO}} \left\{W_i^{CI}(L_i^{CI}) + \left(\left\lfloor \frac{L - L_i + R_i}{T_i} \right\rfloor - 1\right)C_i + W_i^{CO}(L_i^{CO})\right\}. \tag{5}$$

Therefore if we can upper-bound $W_i^{CI}(L_i^{CI})$ and $W_i^{CO}(L_i^{CO})$, we can bound the inter-task interference of τ_i on τ_k and thus the response-time of τ_k.



(a) Subtasks execute as soon as possible **(b) Subtasks execute as late as possible**

Figure 4: Example for carry-in workload when subtasks execute as soon and as late as possible.

5.3 Comparison With The State-of-the-art

In [21], the authors bounded the carry-in workload by assuming that the carry-in job is executed in parallel on all m processors for $\frac{C_i}{m}$ time units. With this assumption, the makespan for the carry-in job is minimized, and thus more workload from the carry-in job can be included in the problem window. Similarly, for the carry-out workload, they assumed that it is executed on all m processors during the carry-out window. However, the assumption that DAG tasks have such abundant parallelism is likely unrealistic and thus makes the analysis pessimistic. Fonseca et al. [11] improved the bounds for carry-in and carry-out workloads by explicitly considering the DAGs of the tasks. The bound for the carry-in workload was calculated based on a schedule for the carry-in job in which the carry-in job can use as many processors as it needs to fully exploit its parallelism, and every subtask executes as soon as it is ready and for its full WCET.

However, such a schedule does not always create the worst-case carry-in workload. Instead, subtasks should delay their executions as late as possible — but the job still gets as many processors as it needs — so that more work is included in the carry-in window. Figure 4 shows an example for the two cases with a task τ_i presented in Figure 1. The length of the carry-in window is 3 time units. In Figure 4a, subtasks of the carry-in job execute as soon as possible. The carry-in workload in this case is 4 time units. Figure 4b

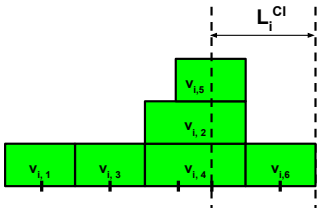


Figure 5: Example for carry-in workload when subtasks execute for their WCETs.

presents the case when subtasks of the carry-in job execute as late as possible. The carry-in workload in this case is 7 time units, larger than when subtasks execute as soon as possible. Another observation is that executing subtasks for their full WCETs also does not always generate the worst-case carry-in workload. Consider a task τ_i which has the same DAG as the one in Figure 1, except that subtask $v_{i,6}$ has WCET $C_{i,6} = 2$. Figure 5 shows the case when all subtasks execute for their WCETs. The carry-in workload is then 5 time units. If instead, $v_{i,6}$ executes for less than its WCET, say 1 time unit, the carry-in workload would be the same as in Figure 4b, which is 7 time units. This is because by completing a part of the task with lower parallelism earlier, parts with higher parallelism can be included and thus the carry-in workload increases. The same observation holds for the carry-out workload, but subtasks must execute as soon as possible in order to attain the worst-case workload.

In this work, we propose a method to compute the bounds for carry-in and carry-out workloads with the above observations kept in mind. We transform the problem of finding the bounds as optimization problems, which can be solved effectively by current solvers such as the IBM CPLEX Optimizer [15] or Gurobi [14]. The solutions then serve as safe and tight upper-bounds for carry-in and carry-out workloads. We present the analysis for carry-out workloads first in Section 6 since it is more straightforward. The analysis for carry-in workloads is presented later in Section 7.

6 BOUND FOR CARRY-OUT WORKLOAD

In this section we propose a method to bound the carry-out workload that can be generated by any job of a task τ_i by constructing an *integer linear program* (ILP) for which the solution is an upper bound of the carry-out workload.

Consider a carry-out job of task τ_i which is scheduled with an unrestricted number of processors, meaning that it can use as many processors as it requires. Subtasks of the carry-out job execute as soon as they are ready. We call such a schedule for the carry-out job $\mathcal{SCHE}^{CO}(\tau_i)$. For each subtask $v_{i,a}$ of the carry-out job of an interfering task τ_i, we define two integer variables $X_{i,a}$ and $W_{i,a}$. $X_{i,a}$ represents the actual execution time of subtask $v_{i,a}$ in the carry-out job and $W_{i,a}$ represents the contribution of subtask $v_{i,a}$ to the carry-out workload. Let $\mathcal{L}^{CO}$ be an integer constant denoting the length of the carry-out window. Then the carry-out workload is the sum of the contributions of all subtasks in $\mathcal{SCHE}^{CO}(\tau_i)$, which is upper-bounded by the maximum of the following *optimization objective function*:

$$obj(\tau_i, \mathcal{L}^{CO}) \triangleq \sum_{v_{i,a} \in V_i} W_{i,a}. \tag{6}$$

We now construct a set of constraints on the contribution of each subtask in $\mathcal{SCHE}^{CO}(\tau_i)$ to the carry-out workload. From the definitions of $X_{i,a}$ and $W_{i,a}$, we have the following constraints.

Constraint 1. *For any interfering task τ_i:*

$$\forall v_{i,a} \in V_i : 1 \leq X_{i,a} \leq C_{i,a}.$$

Constraint 2. *For any interfering task τ_i:*

$$\forall v_{i,a} \in V_i : 0 \leq W_{i,a} \leq X_{i,a}.$$

These two constraints come from the fact that the actual execution time of subtask $v_{i,a}$ cannot exceed its WCET, and each subtask can contribute at most its whole execution time to the carry-out workload. Let $S_{i,a}$ be the starting time of $v_{i,a}$ in $\mathcal{SCHE}^{CO}(\tau_i)$ assuming that the carry-out job starts at time instant 0. For simplicity of exposition, we assume that the DAG G_i has exactly one source vertex and one sink vertex. If this is not the case, we can always add a couple of dummy vertices, $v_{i,source}$ and $v_{i,sink}$, with zero WCETs for source and sink vertices, respectively. Then we add edges from $v_{i,source}$ to all vertices with no predecessors in the original DAG G_i, and edges from all vertices with no successors in G_i to $v_{i,sink}$. Without loss of generality, we assume that $v_{i,1}$ and v_{i,n_i} are the source vertex and sink vertex of G_i, respectively. Let $\sigma_{i,a}^p$ denote a path from the source $v_{i,1}$ to $v_{i,a}$: $\sigma_{i,a}^p \triangleq (v_{i,j_1}, ..., v_{i,j_p})$, where $j_1 = 1$, $j_p = a$, and $(v_{i,j_x}, v_{i,j_{x+1}})$ is an edge in G_i $\forall 1 \leq x < p$. Let $\mathcal{P}(v_{i,a})$ denote the set of all paths from $v_{i,1}$ to $v_{i,a}$ in G_i: $\mathcal{P}(v_{i,a}) \triangleq \{\sigma_{i,a}^p\}$. $\mathcal{P}(v_{i,a})$ can be constructed by a graph traversal algorithm with memoization to avoid redundant computation.

For a particular path $\sigma_{i,a}^p$, the sum of execution times of all subtasks in this path, excluding $v_{i,a}$ is called the *distance* to $v_{i,a}$ with respect to this path. We let $D_{i,a}^p$ be a variable denoting the distance to $v_{i,a}$ in path $\sigma_{i,a}^p$. We impose the following two straightforward constraints on $D_{i,a}^p$ based on its definition.

Constraint 3. *For any interfering task τ_i:*

$$\forall v_{i,a} \in V_i, \forall \sigma_{i,a}^p \in \mathcal{P}(v_{i,a}) : D_{i,a}^p \leq \sum_{v_{i,j_x} \in \{\sigma_{i,a}^p \setminus v_{i,a}\}} X_{i,a}.$$

Constraint 4. *For any interfering task τ_i:*

$$\forall v_{i,a} \in V_i, \forall \sigma_{i,a}^p \in \mathcal{P}(v_{i,a}) : D_{i,a}^p \geq \sum_{v_{i,j_x} \in \{\sigma_{i,a}^p \setminus v_{i,a}\}} X_{i,a}.$$

In the schedule $\mathcal{SCHE}^{CO}(\tau_i)$, the starting time $S_{i,a}$ of a subtask $v_{i,a}$ cannot be smaller than the distance to $v_{i,a}$ in any path $\sigma_{i,a}^p$. We prove this as follows.

Lemma 6.1. *In the schedule $\mathcal{SCHE}^{CO}(\tau_i)$ of any interfering task τ_i:*

$$\forall v_{i,a} \in V_i, \forall \sigma_{i,a}^p \in \mathcal{P}(v_{i,a}) : S_{i,a} \geq D_{i,a}^p.$$

Proof. We prove by contradiction. Let $\sigma_{i,a}^{p*}$ be a path so that the starting time $S_{i,a}$ is smaller than $D_{i,a}^{p*}$. Subtask $v_{i,a}$ must be ready, meaning all of its predecessors must finish, at time $S_{i,a}$ to start execution. Since $S_{i,a} < D_{i,a}^{p*}$, there must be a subtask $v_{i,j_x} \in \{\sigma_{i,a}^{p*} \setminus v_{i,a}\}$ executing (and thus has not finished) at time $S_{i,a}$. Then $v_{i,a}$ cannot be ready at time $S_{i,a}$ since it depends on v_{i,j_x}. This contradicts the assumption that $v_{i,a}$ is ready at $S_{i,a}$ and the lemma follows. □

In fact, in the schedule $\mathcal{SCHE}^{CO}(\tau_i)$ the starting time $S_{i,a}$ of $v_{i,a}$ is equal to the longest distance among all paths to it.

Lemma 6.2. *In the schedule $\mathcal{SCHE}^{CO}(\tau_i)$ of any interfering task τ_i:*

$$\forall v_{i,a} \in V_i : S_{i,a} = \max_{\sigma_{i,a}^p \in \mathcal{P}(v_{i,a})} D_{i,a}^p.$$

Proof. Consider a path $\sigma_{i,a}^{p*}$ constructed as follows. First we take a last-completing predecessor of $v_{i,a}$, say v_{i,j_x}. Since $v_{i,a}$ executes as soon as it is ready, it executes immediately after v_{i,j_x} finishes. We recursively trace back a list of last-completing predecessors in that way until we reach the source vertex $v_{i,1}$. Path $\sigma_{i,a}^{p*}$ is then constructed by chaining the last-completing predecessors together with $v_{i,a}$. We note that any subtask v_{i,j_x} in $\sigma_{i,a}^{p*}$ executes as soon as its immediately preceding subtask finishes since no other predecessors of v_{i,j_x} finish later than it does. Therefore, $S_{i,a} = D_{i,a}^{p*}$. From Lemma 6.1, $\sigma_{i,a}^{p*}$ must have the longest distance to $v_{i,a}$ among all paths in $\mathcal{P}(v_{i,a})$. Thus the lemma follows. □

Based on Lemmas 6.1 and 6.2, we now impose a constraint for the starting time of $v_{i,a}$.

Constraint 5. *For any interfering task τ_i:*

$$\forall v_{i,a} \in V_i, \forall \sigma_{i,a}^p \in \mathcal{P}(v_{i,a}) : S_{i,a} \geq D_{i,a}^p.$$

Proof. We prove that this constraint requires that $S_{i,a}$ of every subtask $v_{i,a}$ for which $\max_{\sigma_{i,a}^p \in \mathcal{P}(v_{i,a})} D_{i,a}^p < \mathcal{L}^{CO}$ receives a value that satisfies Lemma 6.2, that is $S_{i,a} = \max_{\sigma_{i,a}^p \in \mathcal{P}(v_{i,a})} D_{i,a}^p$. (Recall that $\mathcal{L}^{CO}$ is a constant for the carry-out window's length.) In other words, we prove that it requires that every subtask $v_{i,a}$, which would start executing within the carry-out window in a unrestricted-processor schedule $\mathcal{SCHE}^{CO}(\tau_i)$, gets exactly the same starting time from the solution to the optimization problem. Let Q_i denote the collection of such subtasks — the ones which would start executing within the carry-out window in $\mathcal{SCHE}^{CO}(\tau_i)$.

Let π^* be the solution to the optimization problem and $S_{i,a}^*$ be the corresponding value for the starting time of any subtask $v_{i,a} \in Q_i$ in the solution π^*. Obviously $S_{i,a}^* \geq \max_{\sigma_{i,a}^p \in \mathcal{P}(v_{i,a})} D_{i,a}^p$ for any $v_{i,a}$ since any solution to the optimization problem satisfies this constraint. If $S_{i,a}^* = \max_{\sigma_{i,a}^p \in \mathcal{P}(v_{i,a})} D_{i,a}^p$ for any $v_{i,a} \in Q_i$, then we are done. Suppose instead that $S_{i,a}^* = \max_{\sigma_{i,a}^p \in \mathcal{P}(v_{i,a})} D_{i,a}^p + \epsilon_{i,a}$, $\epsilon_{i,a} > 0$ for some $v_{i,a} \in Q_i$. Let Q'_i denote the set of such subtasks. We construct a solution π' to the optimization problem from π^* as follows. Consider a first subtask $v_{i,a} \in Q'_i$ in time. We reduce its starting time by $\epsilon_{i,a}$: $S'_{i,a} = S_{i,a}^* - \epsilon_{i,a}$. Since $v_{i,a}$ is the first delayed subtask, doing this does not violate the precedence constraints for other subtasks. We iteratively perform that operation for other subtasks in Q'_i in increasing time order. The solution π' constructed in this way yields a larger carry-out workload since more workload from individual subtasks can fit in the carry-out window. Therefore π' is a better solution, which contradicts to the assumption that π^* is an optimal solution. □

The workload contributed by a subtask $v_{i,a}$ can be computed as follows: $W_{i,a} = \min\left\{\max\{\mathcal{L}^{CO} - S_{i,a}, 0\}, X_{i,a}\right\}$. The second part of the outer minimization has been taken care of by Constraint 2. We now construct constraints to impose the first part of the minimization. Let $M_{i,a}$ be a variable denoting the expression $\max\{\mathcal{L}^{CO} - S_{i,a}, 0\}$. Let $A_{i,a}$ be a binary variable which takes value either 0 or 1. We have the following simple constraints.

Constraint 6. *For any interfering task τ_i:*

$$\forall v_{i,a} \in V_i : W_{i,a} \leq M_{i,a}.$$

Constraint 7. *For any interfering task τ_i:*

$$\forall v_{i,a} \in V_i : M_{i,a} \geq 0.$$

Constraint 8. *For any interfering task τ_i:*

$$\forall v_{i,a} \in V_i : M_{i,a} \leq (\mathcal{L}^{CO} - S_{i,a}) A_{i,a}.$$

Constraints 7 and 8 bound the value for $M_{i,a}$ and Constraint 6 enforces another upper bound for the workload $W_{i,a}$. If $\mathcal{L}^{CO} < S_{i,a}$, $A_{i,a}$ can only be 0 in order to satisfy both Contraints 7 and 8. If $\mathcal{L}^{CO} = S_{i,a}$, the value of $A_{i,a}$ does not matter. In both of these cases, these three constraints together with Constraint 2 bound $W_{i,a}$ to zero contribution of $v_{i,a}$ to the carry-out workload. If $\mathcal{L}^{CO} < S_{i,a}$, the maximizing process enforces that $A_{i,a}$ takes value 1. Therefore in any case Constraints 2, 6, 7, and 8 enforce a correct value for the workload contribution $W_{i,a}$ of $v_{i,a}$.

We have constructed an ILP with a quadratic constraint (Constraint 8) for each $v_{i,a}$, for which the solution is an upper bound for the carry-out workload. The carry-out workload of τ_i in a carry-out window of length $\mathcal{L}^{CO}$ can also be upper-bounded by the following straightforward lemma.

Lemma 6.3. *The carry-out workload of an interfering task τ_i scheduled by G-FP in a carry-out window of length $\mathcal{L}^{CO}$ is upper-bounded by $m\mathcal{L}^{CO}$.*

Lemma 6.3 follows directly from the fact that the carry-out job can execute at most on all m processors of the system during the carry-out window. Since the carry-out workload of τ_i is upper-bounded by both the maximum value returned for the optimization problem and Lemma 6.3, it is upper-bounded by the minimum of the two upper bounds.

Theorem 6.4. *The carry-out workload of an interfering task τ_i scheduled by G-FP in a carry-out window of length $\mathcal{L}^{CO}$ is upper-bounded by: $\min\left\{\mathcal{OBJ}, m\mathcal{L}^{CO}\right\}$, where $\mathcal{OBJ}$ is the maximum value returned for the maximization problem (Equation 6).*

7 BOUND FOR CARRY-IN WORKLOAD

As discussed in Section 5, the carry-in workload is maximized for a carry-in job when its subtasks execute as late as possible within the carry-in window — recall that the finishing time of the carry-in job must be the same as the end of the carry-in window in order to maximize the carry-in workload. Chwa et al. [8] proposed a method to compute the carry-in workload in which subtasks' executions are delayed as late as possible. However they assumed that every subtask executes for its whole WCET which does not always yield the worst-case carry-in workload as we discussed before. In this section we propose a method to bound carry-in workload that is similar to the one used for bounding carry-out workload — we construct an ILP for which the value of an optimal solution is an upper bound. As with bounding carry-out workload, this approach considers all possible values for the execution time of each subtask.

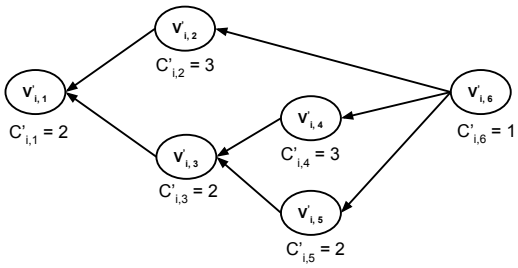
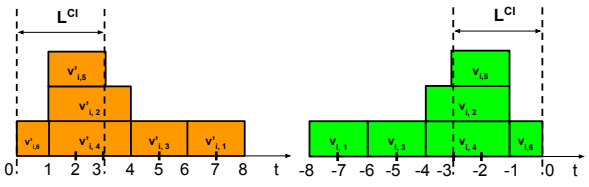


Figure 6: Example DAG of a transposed task τ_i' of τ_i.

Again let τ_i be an interfering task for which we aim to bound its carry-in workload in a carry-in window of length $\mathcal{L}^{CI}$. From τ_i we construct a transposed DAG task τ_i' of τ_i as follows. Task τ_i' has the same set of subtasks as in τ_i, meaning that it has the same number of subtasks and each subtask has the same WCET as its corresponding subtask in τ_i. Let $G_i' = (V_i', E_i')$ denote the DAG of τ_i', where V_i' and E_i' are the sets of vertices and edges respectively. Each subtask $v_{i,a}' \in V_i'$ corresponds to a subtask $v_{i,a} \in V_i$. For each edge $(v_{i,a}, v_{i,b}) \in E_i$, there is an edge $(v_{i,b}', v_{i,a}') \in E_i'$. In other words, G_i' is a copy of G_i except that the edges are reversed. Figure 6 shows the DAG for a transposed task τ_i' of the task τ_i presented in Figure 1. For any job of τ_i there always exists a job of τ_i' whose subtasks have exactly the same execution times as the ones from the job of τ_i. We call one the *mirror* of the other, and vice versa.

Consider a carry-in job J_i of τ_i. The carry-in window has length $\mathcal{L}^{CI}$. Let J_i' be a mirror job of J_i. Let $\mathcal{SCHE}^{CO}(\tau_i')$ denote an unrestricted-processor schedule for J_i', in which its subtasks execute as soon as possible like for the carry-out job (hence we use superscript CO here). From $\mathcal{SCHE}^{CO}(\tau_i')$ we can construct an unrestricted-processor schedule for J_i for which we prove that subtasks of J_i execute as late as possible and thus J_i's carry-in workload is maximized. Figures 7a and 7b show an example for a schedule of J_i' and the constructed schedule of J_i respectively. For simplicity of exposition and without loss of generality, we assume that J_i' starts at time 0 and J_i finishes at time 0. Let $s_{i,a}$ and $f_{i,a}$ denote



(a) A schedule $\mathcal{SCHE}^{CO}(\tau_i')$ for τ_i' **(b) A schedule for τ_i constructed from $\mathcal{SCHE}^{CO}(\tau_i')$**

Figure 7: Example unrestricted-processor schedules of τ_i' and τ_i.

the starting time and finishing time of $v_{i,a} \in V_i$, respectively. Similarly, let $s_{i,a}'$ and $f_{i,a}'$ denote the starting time and finishing time of $v_{i,a}' \in V_i'$. We construct a schedule for J_i in which the starting time of each subtask $v_{i,a}$ is assigned as follows.

$$\forall v_{i,a} \in V_i, s_{i,a} := -f_{i,a}'$$

For example, in Figure 7 $s_{i,6} := -f_{i,6}' = -1$, $s_{i,4} := -f_{i,4}' = -4$. We call the schedule constructed for J_i presented above $\mathcal{SCHE}^{CI}(\tau_i)$.

Now we prove that the schedule $\mathcal{SCHE}^{CI}(\tau_i)$ is a valid schedule of τ_i and the subtasks in $\mathcal{SCHE}^{CI}(\tau_i)$ execute as late as possible.

Lemma 7.1. *A schedule $\mathcal{SCHE}^{CI}(\tau_i)$ is a valid schedule of τ_i.*

Proof. Suppose there exists a subtask $v_{i,a}$ which executes before a predecessor $v_{i,b}$ finishes in $\mathcal{SCHE}^{CI}(\tau_i)$. Then:

$$s_{i,a} < f_{i,b} \Rightarrow s_{i,a} < s_{i,b} + x_{i,b},$$

where $x_{i,b}$ is the actual execution time of $v_{i,b}$ in $\mathcal{SCHE}^{CI}(\tau_i)$.

$$\Rightarrow -f_{i,a}' < -f_{i,b}' + x_{i,b} \Rightarrow f_{i,a}' > f_{i,b}' - x_{i,b}.$$

$$\Rightarrow f_{i,a}' > s_{i,b}'.$$

This means $v_{i,b}'$ starts before $v_{i,a}'$ finishes, which contradicts to the fact that $v_{i,a}'$ is a predecessor of $v_{i,b}'$ and $\mathcal{SCHE}^{CO}(\tau_i')$ is a valid schedule of τ_i'. The lemma follows. □

Lemma 7.2. *In a schedule $\mathcal{SCHE}^{CI}(\tau_i)$ of an interfering task τ_i, no subtask can start later than its current starting time in $\mathcal{SCHE}^{CI}(\tau_i)$.*

Proof. Suppose there exists a subtask $v_{i,a}$ that could have delayed its execution $\epsilon > 0$ time units in $\mathcal{SCHE}^{CI}(\tau_i)$. Without loss of generality, assume that $v_{i,a}$ is the latest subtask in time order whose execution can be delayed more. Let $v_{i,b}$ be a successor of $v_{i,a}$ which starts earliest in the schedule $\mathcal{SCHE}^{CI}(\tau_i)$ among all $v_{i,a}$'s successors. Then ϵ must have been the length of the interval from $f_{i,a}$ to $s_{i,b}$. We have:

$$s_{i,b} = f_{i,a} + \epsilon \Rightarrow s_{i,b} = s_{i,a} + x_{i,a} + \epsilon,$$

where $1 \le x_{i,a} \le C_{i,a}$ is the actual execution time of $v_{i,a}$.

$$\Rightarrow -f_{i,b}' = -f_{i,a}' + x_{i,a} + \epsilon.$$

$$\Rightarrow f_{i,b}' = f_{i,a}' - x_{i,a} - \epsilon.$$

$$\Rightarrow f_{i,b}' = s_{i,a}' - \epsilon \Rightarrow s_{i,a}' = f_{i,b}' + \epsilon. \quad (7)$$

By construction, since $v_{i,b}$ is a successor of $v_{i,a}$ with the earliest starting time, $v'_{i,b}$ must be a predecessor of $v'_{i,a}$ with the latest finishing time. Since subtasks execute as soon as possible in the schedule $\mathcal{SCHE}^{CO}(\tau'_i)$, we must have $s'_{i,a} = f'_{i,b}$, which contradicts (7). The lemma thus follows. □

Lemmas 7.1 and 7.2 show that the schedule $\mathcal{SCHE}^{CI}(\tau_i)$ constructed above for carry-in job of τ_i yields maximum carry-in workload. Moreover, the carry-in workload generated by $\mathcal{SCHE}^{CI}(\tau_i)$ in a carry-in window of length $\mathcal{L}^{CI}$ is equal to the carry-out workload generated by $\mathcal{SCHE}^{CO}(\tau'_i)$ in a carry-out window of the same length, $\mathcal{L}^{CI}$ (see Figure 7 for example). Therefore we can bound the carry-in workload for τ_i in a carry-in window of length $\mathcal{L}^{CI}$ by the maximum of the objective function $obj(\tau'_i, \mathcal{L}^{CI})$ in Equation 6 with the same set of constraints presented in Section 6.

Theorem 7.3. *The carry-in workload of an interfering task τ_i scheduled by G-FP in a carry-in window of length $\mathcal{L}^{CI}$ is upper-bounded by:* $\min\left\{\mathcal{OBJ}, m\mathcal{L}^{CI}\right\}$, *where $\mathcal{OBJ}$ is the maximum value returned for the maximization problem: $obj(\tau'_i, \mathcal{L}^{CI})$ (Equation 6) and τ'_i is the transposed task of τ_i.*

8 RESPONSE-TIME-BASED SCHEDULABILITY TEST

From the above calculations for the bounds of intra-task interference and inter-task interference on τ_k, we have the following theorem for the response-time bound of τ_k.

Theorem 8.1. *A constrained-deadline task τ_k scheduled by a global fixed-priority algorithm has response-time upper-bounded by the smallest R_k^{ub} that satisfies the following fixed-point iteration:*

$$R_k^{ub} \leftarrow L_k + \frac{1}{m}(C_k - L_k) + \frac{1}{m}\sum_{\tau_i \in hp(\tau_k)} W_i(R_k^{ub}).$$

Proof. This follows from Lemma 5.1 and the fact that the inter-task interference of τ_i on τ_k is bounded by the workload generated by τ_i (Equation 3). □

In the above fixed-point iteration, $W_i(R_k^{ub})$ is calculated using Equation 5, in which the carry-in workload and carry-out workload are bounded as discussed in Sections 7 and 6 for all carry-in window and carry-out window satisfy constraint 4. The lengths for carry-in window L_i^{CI} and carry-out window L_i^{CO} can be varied as follows. Let Δ denote the right-hand side of Equation 4. First L_i^{CI} takes its largest value: $L_i^{CI} \leftarrow \min\{\Delta, L_i\}$, and L_i^{CO} takes the remaining sum: $L_i^{CO} \leftarrow \min\{\Delta - L_i^{CI}, L_i\}$. Then in each subsequent step L_i^{CI} is decreased and L_i^{CO} is increased until L_i^{CO} takes its largest value and L_i^{CI} takes the remaining value. We note that if at the first step both L_i^{CI} and L_i^{CO} are equal to L_i, the carry-in workload and carry-out workload are both simply bounded by C_i and we do not have to vary L_i^{CI} and L_i^{CO} to find the maximum workload $W_i(L)$ (Equation 5). Similarly, if the sum of L_i^{CI} and L_i^{CO} is 0 in Equation 4, both the carry-in workload and the carry-out workload are 0. We also note that for the highest priority task, there is no interference from any other task, and thus its response-time bound can be computed simply by: $R_k^{ub} \leftarrow \left(L_k + \frac{1}{m}(C_k - L_k)\right)$.

Algorithm 1 A Schedulability Test Using Response-Time Bounds

```
1:  procedure SchedulabilityTest(τ)          ▷ Without loss of generality, assuming tasks are sorted in decreasing order of priorities
2:      for Each τ_k ∈ τ do                  ▷ Initialize the values for response-time bounds
3:          R_k^ub ← L_k + (1/m)(C_k − L_k)
4:      end for
5:      if R_1^ub > D_1 then
6:          Return Unschedulable
7:      end if
8:      for τ_k from τ_2 to τ_n do
9:          Calculate R_k^ub in Theorem 8.1
10:         if R_k^ub > D_k then
11:             Return Unschedulable
12:         end if
13:     end for
14:     Return Schedulable
15: end procedure
```

Using the above response-time bound, we derive a schedulability test in Algorithm 1. The algorithm first initializes the response-time bounds for the tasks to be $\left(L_k + \frac{1}{m}(C_k - L_k)\right)$. If the response-time bound for the highest-priority task, i.e., task τ_1, is larger than its relative deadline, the algorithm stops and deems the task set unschedulable. Otherwise it calculates the response-time bound for each task in decreasing order of priority using the fixed-point iteration in Theorem 8.1. If for any task, the response-time bound is larger than its relative deadline, the task set is deemed unschedulable. Otherwise, it is deemed schedulable.

9 SLACK-BASED SCHEDULABILITY TEST

The analysis for the workload of an interfering task presented in the previous sections can also be used to construct another schedulability test. The high level idea of this workload-based schedulability test is that (i) first we derive a necessary condition for a job of τ_k to miss its deadline, then (ii) we negate that condition to obtain a sufficient test for all jobs of τ_k to meet their deadlines. Chwa et al. [8] utilized this technique to propose a workload-based schedulability test for sporadic DAG tasks scheduled with G-EDF. We can also use this technique for G-FP.

Let J_k^* be the first job of τ_k that misses its deadline. A necessary condition for this to happen is:

$$len(\lambda_k^*) + I_k(r_k^*, d_k^*) > D_k, \tag{8}$$

where λ_k^* is a critical chain of K_k^* and $I_k(r_k^*, d_k^*)$ is the critical interference of J_k^* in interval $[r_k^*, d_k^*)$. We have

$$I_k(r_k^*, d_k^*) = \frac{1}{m}\sum_{\tau_i \in \{\tau_k \cup hp(\tau_k)\}} I_{i,k}(r_k^*, d_k^*)$$

(from Equation 1), where $hp(\tau_k)$ is the set of tasks with higher priorities than τ_k.

$$\Rightarrow I_k(r_k^*, d_k^*) = \frac{1}{m}\left(I_{k,k}(r_k^*, d_k^*) + \sum_{\tau_i \in hp(\tau_k)} I_{i,k}(r_k^*, d_k^*)\right).$$

$$\Rightarrow len(\lambda_k^*) + I_k(r_k^*, d_k^*) = \left(len(\lambda_k^*) + \frac{1}{m} I_{k,k}(r_k^*, d_k^*)\right) + \frac{1}{m} \sum_{\tau_i \in hp(\tau_k)} I_{i,k}(r_k^*, d_k^*). \quad (9)$$

Similar to Lemma 5.1, we can prove that

$$len(\lambda_k^*) + \frac{1}{m} I_{k,k}(r_k^*, d_k^*) \leq L_k + \frac{1}{m}(C_k - L_k).$$

Combined with $I_{i,k}(r_k^*, d_k^*) \leq W_i(D_k)$ (from Inequality 3), we have:

$$len(\lambda_k^*) + I_k(r_k^*, d_k^*) \leq L_k + \frac{1}{m}(C_k - L_k) + \frac{1}{m} \sum_{\tau_i \in hp(\tau_k)} W_i(D_k).$$

Therefore, if we can show that

$$L_k + \frac{1}{m}(C_k - L_k) + \frac{1}{m} \sum_{\tau_i \in hp(\tau_k)} W_i(D_k) \leq D_k,$$

then no job of τ_k can miss deadline, since the necessary condition (8) for the first deadline miss to happen cannot be satisfied for all jobs of τ_k. A schedulability test can be done by checking this condition for each task $\tau_k \in \tau$. We summarize this in the following theorem (similar to Theorem 1 in [8]).

Theorem 9.1. *A task set τ is schedulable under G-FP if for each task $\tau_k \in \tau$, the following condition holds:*

$$L_k + \frac{1}{m}(C_k - L_k) + \frac{1}{m} \sum_{\tau_i \in hp(\tau_k)} W_i(D_k) \leq D_k.$$

We note that in Theorem 9.1, the sufficient condition for each task τ_k to be schedulable implicitly assumes that each interfering task τ_i of τ_k finishes by its deadline — this is because the response-time bound for τ_i is unknown. Thus in the computation of $W_i(D_k)$ in Theorem 9.1, the carry-in job of τ_i is assumed to finish at its deadline (recall that the carry-in job must execute as late as possible to generate maximum interfering workload on τ_k). However, if the carry-in job finishes before its deadline, its slack value can be used to improve its interference on τ_k. Thus the idea is to employ a lower bound on the slack of each interfering task τ_i to reduce its interference on τ_k and hence improve the schedulability of τ_k. We also note that the calculation for carry-out workloads does not use information about slack since carry-out jobs execute as soon as possible. The idea of employing slack values was proposed by Bertogna et al. [6] for sequential tasks scheduled under G-EDF and G-FP. Chwa et al. [8] later extended the idea for parallel DAG tasks scheduled under G-EDF. A major difference between the idea of using slack for the schedulability test in Theorem 9.1 and the response-time analysis presented in the previous section is that the response-time analysis aggressively assumes maximum slack for all tasks initially and gradually reduces the slack as the fixed-point iterations proceed, whereas this idea assumes zero slack for all tasks at first, then updates them iteratively.

Let $\mathcal{S}_k^{lb}$ be a lower bound for the slack of τ_k. From Theorem 9.1 $\mathcal{S}_k^{lb}$ can be computed by:

$$\mathcal{S}_k^{lb} = D_k - L_k - \frac{1}{m}(C_k - L_k) - \left\lceil \frac{1}{m} \sum_{\tau_i \in hp(\tau_k)} W_i(D_k) \right\rceil,$$

when the result is positive. This formula still does not exploit the slacks of the interfering tasks of τ_k (in the terms $W_i(D_k)$). Let $W_i(D_k, \mathcal{S}_i^{lb})$ be the maximum workload generated by τ_i in an interval of length D_k with slack $\mathcal{S}_i^{lb}$ considered. Then we have:

$$\mathcal{S}_k^{lb} = D_k - L_k - \frac{1}{m}(C_k - L_k) - \left\lceil \frac{1}{m} \sum_{\tau_i \in hp(\tau_k)} W_i(D_k, \mathcal{S}_i^{lb}) \right\rceil, \quad (10)$$

when it is positive.

With the calculation of the slack's lower bound, we can derive an iterative schedulability test similar to those presented in Figure 5 in [6] and Algorithm 2 in [8] (they are also similar except that they compute slack differently since the former works for sequential tasks and the latter works for DAG tasks under G-EDF), except that the slack is computed using Equation 10 shown above. To the best of our knowledge, no one has compared the empirical performance of these two schedulability tests for DAG tasks — namely, the response-time analysis and the slack-based iterative test. In the future, we plan to investigate their respective performances for randomly generated DAG tasks.

10 CONCLUSION

In this paper we propose a method to bound carry-in and carry-out workloads of interfering tasks by converting the calculation of the bounds to optimization problems, for which efficient solvers exist. The optimal values for the optimization problems serve as safe upper bounds for carry-in and carry-out workloads. Our method considers all possible execution times of each subtask and thus covers the cases when maximum workloads are obtained when some subtasks execute for less than their WCETs. We present two schedulability tests for G-FP based on the proposed workload bounds: response-time analysis and slack-based schedulability test. In the future, we would like to extend this technique for bounding carry-in and carry-out workloads in which carry-in and carry-out jobs execute on m processors instead of unrestricted processors. We also would like to conduct empirical experiments to compare our analysis with the existing analyses for G-FP and G-EDF. It also would be interesting to see how the slack-based schedulability test performs compared to the response-time-based schedulability test.

REFERENCES

[1] Philip Axer, Sophie Quinton, Moritz Neukirchner, Rolf Ernst, Björn Döbel, and Hermann Härtig. 2013. Response-time analysis of parallel fork-join workloads with real-time constraints. In *25th Euromicro Conference on Real-Time Systems, 2013*. IEEE, 215–224.

[2] Theodore Baker. 2003. Multiprocessor EDF and deadline monotonic schedulability analysis. In *24th Real-Time Systems Symposium, 2003*. IEEE, 120–129.

[3] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. 2012. A generalized parallel task model for recurrent real-time processes. In *33rd Real-Time Systems Symposium, 2012*. IEEE, 63–72.

[4] Marko Bertogna and Michele Cirinei. 2007. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *28th Real-Time Systems Symposium, 2007*. IEEE, 149–160.

[5] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. 2005. Improved schedulability analysis of EDF on multiprocessor platforms. In *17th Euromicro Conference on Real-Time Systems, 2005.* IEEE, 209–218.

[6] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. 2009. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on parallel and distributed systems* 20, 4 (2009), 553–566.

[7] Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. 2013. Feasibility analysis in the sporadic dag task model. In *25th Euromicro Conference on Real-Time Systems, 2013.* IEEE, 225–233.

[8] Hoon Sung Chwa, Jinkyu Lee, Jiyeon Lee, Kiew-My Phan, Arvind Easwaran, and Insik Shin. 2017. Global edf schedulability analysis for parallel tasks on multi-core platforms. *IEEE Transactions on Parallel and Distributed Systems* 28, 5 (2017), 1331–1345.

[9] Hoon Sung Chwa, Jinkyu Lee, Kieu-My Phan, Arvind Easwaran, and Insik Shin. 2013. Global edf schedulability analysis for synchronous parallel tasks on multi-core platforms. In *25th Euromicro Conference on Real-Time Systems, 2013.* IEEE, 25–34.

[10] David Ferry, Gregory Bunting, Amin Maghareh, Arun Prakash, Shirley Dyke, Kunal Agrawal, Chris Gill, and Chenyang Lu. 2014. Real-time system support for hybrid structural simulation. In *Proceedings of the 14th International Conference on Embedded Software.* ACM, 1–10.

[11] José Fonseca, Geoffrey Nelissen, and Vincent Nélis. 2017. Improved response time analysis of sporadic DAG tasks for global FP scheduling. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems.* ACM, 28–37.

[12] Matteo Frigo, Charles E Leiserson, and Keith H Randall. 1998. The implementation of the Cilk-5 multithreaded language. *ACM Sigplan Notices* 33, 5, 212–223.

[13] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. 2009. New response time bounds for fixed priority multiprocessor scheduling. In *30th Real-Time Systems Symposium, 2009.* IEEE, 387–397.

[14] Gurobi. 2018. Gurobi Optimization. http://www.gurobi.com/index. Last accessed: 2018-07-07.

[15] IBM. 2018. IBM ILOG CPLEX Optimizer. https://www.ibm.com/analytics/cplex-optimizer. Last accessed: 2018-07-07.

[16] Intel. 2018. Intel Cilk Plus. https://www.cilkplus.org/. Last accessed: 2018-07-07.

[17] Intel. 2018. Intel Threading Building Blocks. https://www.threadingbuildingblocks.org/. Last accessed: 2018-07-07.

[18] Junsung Kim, Hyoseung Kim, Karthik Lakshmanan, and Ragunathan Raj Rajkumar. 2013. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems.* ACM, 31–40.

[19] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Raj Rajkumar. 2010. Scheduling parallel real-time tasks on multi-core processors. In *31st IEEE Real-Time Systems Symposium, 2010.* IEEE, 259–268.

[20] Cláudio Maia, Marko Bertogna, Luís Nogueira, and Luis Miguel Pinho. 2014. Response-time analysis of synchronous parallel tasks in multiprocessor systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems.* ACM, 3.

[21] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C Buttazzo. 2015. Response-time analysis of conditional dag tasks in multiprocessor systems. In *27th Euromicro Conference on Real-Time Systems, 2015.* IEEE, 211–221.

[22] OpenMP Architecture Review Board. 2018. OpenMP API Specification. https://www.openmp.org/. Last accessed: 2018-07-07.

[23] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. 2013. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems* 49, 4 (2013), 404–435.