Budgeting Processing Graphs Under Restricted Parallelism

Zelin Tong
Department of Computer Science
University of North Carolina at Chapel Hill
ztong@cs.unc.edu

James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill
anderson@cs.unc.edu

***Abstract*—Certifying graph-based workloads on multicore systems requires valid worst-case execution time (WCET) estimates, which are challenging to obtain. This warrants a method to enforce execution budgets at runtime. Allowing a node in a processing graph to overrun its budget can delay future invocations of that node due to data dependencies. Conversely, preventing overruns may lead to high rates of graph invocation aborts. This paper presents a budget-enforcement method that allows nodes to overrun with limited effect on future node invocations. Additionally, analysis is presented for bounding the abort probability of each graph invocation. Experimental results are given to demonstrate the efficacy of the presented method.**

Keywords—**Real-time systems, DAG scheduling, probabilistic analysis, restricted parallelism, budget enforcement**

I. Introduction

Complex AI and robotics applications are often modeled as processing graphs due to their dataflow dependencies. These graphs are often large and computationally demanding, requiring multicore machines for timely execution. In safety-critical contexts, schedulability assessments are needed to validate timing constraints, which requires that task worst-case execution times (WCETs) be known. Unfortunately, the complexity of modern multicore machines has led to the consensus that *static* timing analysis, where the WCET of a program is computed from its code structure, will likely never be viable [8]. The only other alternative, *measurement-based* timing analysis, may not capture the true WCET of a program.

Budget enforcement. Accurate WCETs can be obtained by enforcing execution budgets. However, such an approach requires a policy to handle budget *overruns*. Tong et al. proposed a budget-enforcement policy for processing graphs that allows a node that overruns its budget to continue execution by consuming the budget of "downstream" nodes in the same graph invocation [7]. The graph invocation is only aborted when there is insufficient budget in the graph's remaining nodes, aiming to reduce the overall graph abort rate.

The need for restricted parallelism. Unfortunately, [7] assumes that successive graph invocations can run in parallel without restrictions, which is unrealistic for many real-world applications. For example, in object tracking, processing the i^{th} frame of video may require data acquired from processing the $(i-\rho)^{\text{th}}$ frame. This restricts graph parallelism, as at most ρ frames can be processed in parallel. This paper presents a new holistic graph budget-management policy that improves upon [7] by supporting limited graph parallelism. We begin by examining the issues arising from limited parallelism.

Work was supported by NSF grants CPS 2038960, CPS 2038855, CNS 2151829, and CPS 2333120

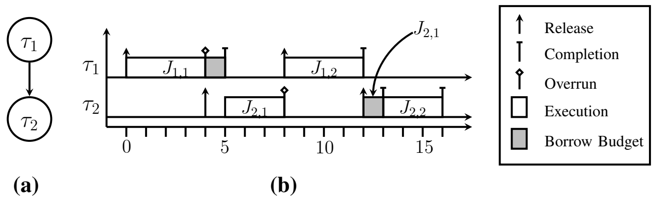

Fig. 1: (a) A processing graph G, where each node has a budget of 4.0 time units. (b) A schedule of G demonstrating budget reallocations both within the same graph invocation and across different invocations. $J_{i,j}$ denotes the j^{th} job of τ_i.

Cross-invocation overruns. When the parallelism of graph nodes is limited, a *cross-invocation* overrun can occur, where an invocation of a node overruns its budget and delays future invocations of the same node that cannot execute in parallel. To promptly execute these future node invocations, node invocations must be allowed to execute using budget from future graph invocations. (This is different from [7], which only moves budget from one node to another within the same graph invocation.) To illustrate this issue, consider the following example pertaining to the graph G depicted in Fig. 1.

Example 1. *At $t = 0$, the node τ_1 releases its first invocation, or* job, *$J_{1,1}$. At $t = 4$, $J_{1,1}$ overruns its budget, and utilizes the budget of $J_{2,1}$ (the first job of τ_2) to execute until completion at $t = 5$ using the policy in [7]. When $J_{1,1}$ completes, $J_{2,1}$ can begin execution. However, due to reallocating part of its budget to $J_{1,1}$, $J_{2,1}$ overruns its budget at $t = 8$. Assuming that only one invocation of each node can execute in parallel, $J_{2,1}$ must complete before $J_{2,2}$ can begin execution. Therefore, to ensure the timely execution of $J_{2,2}$, $J_{2,1}$ must use the budget of $J_{2,2}$ to complete its execution.*

Overrun cascades. From Ex. 1, we see that cross-invocation overruns cause nodes to drain budgets from future node invocations, thus increasing the overrun probabilities of future node invocations. Consequently, node overrun probabilities will steadily increase from one graph invocation to the next if cross-invocation overruns occur. This phenomenon, which we call an *overrun cascade*, is another implication of restricting processing graph parallelism. To limit the length of an overrun cascade, the feedback loop of ever-increasing cross-invocation overruns must be broken. While this can be done naïvely by *strictly* enforcing node budgets (i.e., any overrunning node

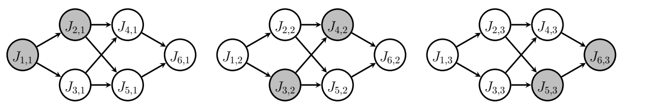

Fig. 2: Successive invocations of a graph G, where in each invocation the two shaded nodes have strictly enforced budgets.

invocation would be aborted), such a solution is far from ideal. This is because strictly enforcing node budgets disallows overrunning jobs from completing their execution by consuming the budget of "downstream" nodes, invalidating a key method in [7] to reduce the overall graph abort rate.

Mitigating overrun cascades. Fortunately, we can limit overrun cascades without strictly enforcing all node budgets. This can be done by strictly enforcing node budgets in a cyclic fashion across graph invocations as shown in Fig. 2. Since each node's budget is strictly enforced once every few graph invocations, overrun cascades can be broken.

Slowing down overrun probability increases. While the length of overrun cascades can be limited by cycling through some strict enforcement pattern, for large processing graphs, such a cycle may contain many graph invocations. Consequently, overrun cascades may be long, resulting in high per-node overrun rates. This, in turn, can greatly increase the graph abort rate. Thus, to minimize the abort rate of a processing graph, our budget management policy must also limit the increase in overrun probability across graph invocations.

Contributions. This paper presents a server-based graph-level budgeting policy that supports limited parallelism and addresses the issues raised above through three contributions. First, we propose three mechanisms that form our budgeting policy. The first two limit the increase in overrun probabilities due to cross-invocation overruns. We do this via **(i)** a slack-reallocation policy that allows node invocations to execute early using the leftover budget from past invocations, and **(ii)** an overrun-management policy that allows overrunning node invocations to execute simultaneously using the budgets of multiple downstream nodes. The third mechanism limits the length of overrun cascades through selective strict budget enforcement as discussed above. Second, we provide analysis to upper-bound the abort probability of each graph invocation. Finally, we present results from experimental evaluations that demonstrate the efficacy of our budget-management strategy.

Organization. In the following sections, we provide necessary background information (Sec. II), present our budgeting policy for graphs with restricted parallelism (Sec. III), provide graph abort-rate analysis for this policy (Sec. IV), present an experimental evaluation (Sec. V), and conclude (Sec. VI).

II. Task Model and Background

We consider a task system Γ consisting of N processing graphs scheduled on m identical processors. Each graph $G \in \Gamma$ is characterized by a set of n nodes, $\{\tau_1, \tau_2, ..., \tau_n\}$, representing the tasks of G and a set of directed edges between these nodes. An edge from τ_i to τ_j denotes a precedence constraint between the *predecessor* node of τ_j (τ_i) and *successor* node of τ_i (τ_j). We denote the predecessors of τ_i as $pred(\tau_i)$.

Graph structure assumptions. For simplicity, we assume that τ_1 (resp. τ_n) is a unique source (resp. sink) node for graph G. However, graphs with multiple sources/sinks can be supported by adding a "dummy" source/sink with zero execution time. We further assume that no cycles exist in each $G \in \Gamma$, making each graph we consider a *directed acyclic graph* (DAG).

Node parameters. Each node $\tau_i \in G$ releases a sequence of *job*s, $J_{i,1}, J_{i,2}, ...$ according to the following rule.

> **Job Release Rule.** $J_{1,j}$ is released periodically with an interarrival time of T. For $i \neq 1$, $J_{i,j}$ is immediately released after every $J_{k,j}$ such that $\tau_k \in pred(\tau_i)$ completes.

The Job Release Rule ensures that **(i)** DAG invocations are released periodically, and **(ii)** precedence constraints are enforced between nodes. Each node has a *relative deadline* of T, with deadlines being soft, i.e., deadline misses are acceptable. The *response time* of graph G is the maximum difference between the completion time of $J_{n,j}$ and the release time of $J_{1,j}$ across all job index j. Additionally, an upper bound on each job $J_{i,j}$'s execution-time distribution is given by its *probabilistic WCET* (pWCET), characterized by the *discrete* random variable (RV) $e_{i,j}$.

Restricted parallelism. Each graph G has a *parallelization level* ρ, representing the maximum number of concurrently executing jobs per node, similar to the notion of *restricted parallelism* introduced in [2]. Thus, for each node τ_i, $J_{i,j}$, can only begin execution when $J_{i,j-\rho}$ completes. In light of this, a job $J_{i,j}$ is *ready* (to be scheduled) when **(i)** it is released and not complete, and **(ii)** $J_{i,j-\rho}$ is complete. As $J_{i,j}$ *depends* on the completion of other jobs before it can become ready, we let $dep(J_{i,j})$ denote the set of jobs on which $J_{i,j}$ depends.

Server-based graph budgeting. In a server-based graph budgeting approach such as [7], each node $\tau_i \in G$ is assigned a *reservation server* S_i with a relative deadline of T and an execution budget of C_i. Each server S_i releases a sequence of server jobs $S_{i,1}, S_{i,2}$.... We assume these server jobs are scheduled by a global earliest-deadline-first (G-EDF) scheduler with the following Tie-Breaking Rule.

> **Tie-Breaking Rule.** If $S_{i,x}$ and $S_{k,y}$ have the same deadline, then $S_{i,x}$ has a higher priority if $C_i < C_k$, or if $C_i = C_k$, and $i < k$.

We denote the release time of $S_{i,j}$ as $r_{i,j}$. These server jobs serve as "containers" on which nodes' jobs are scheduled. The budget of server jobs is managed according to the following.

> **Consumption Rule.** $S_{i,j}$ begins with C_i budget and consumes one unit of budget per unit of time scheduled. $S_{i,j}$ completes when its budget is exhausted.

The *response time* of each server job $S_{i,j}$, $R_{i,j}$, is the difference between its completion and release time. The response time of a reservation server S_i, R_i, is the maximum response time of any of its jobs. Server-based graph budgeting uses server jobs to emulate an "idealized system" where the jobs of each node τ_i execute for exactly C_i time units. Within this schedule, each server job $S_{i,j}$ then schedules jobs of nodes in G. We say that a job $J_{i,j}$ *overruns* (resp. *underruns*) its budget when $S_{i,j}$ (resp. $J_{i,j}$) is complete, but $J_{i,j}$ (resp. $S_{i,j}$) is not. Since server jobs mimic the schedule of jobs in an idealized system, server jobs must also respect the parallelism restriction of jobs. This can be done by transforming the reservation servers of G into an *rp-sporadic* task set.

Transforming servers into an rp-sporadic task set. Prior work has shown how a periodic DAG task can be converted to an rp-sporadic task set [5], [1]. Using such a method, each server S_i can be considered as an rp-sporadic task, i.e., a task with an execution cost C_i, a relative deadline and period of T, and a parallelization level ρ. In the rp-sporadic task model, we can use existing analysis in [2] to compute an upper bound on the response time of each server S_i, denoted as R_i^{max}.

Precedence constraints between two servers can be eliminated through the use of a *release offset* $\mathcal{O}_i$ for each server S_i that specifies the difference between $r_{i,j}$ and $r_{1,j}$. The release offset of each server is assigned according to the following rule. Then, using release offsets, server jobs are released according to the Server Release Rule.

> **Offset Rule.** $\mathcal{O}_1 = 0$. For $i \neq 1$, $\mathcal{O}_i = \max(\{\mathcal{O}_k + R_k^{max} \mid \tau_k \in pred(\tau_i)\})$.
> **Server Release Rule.** $S_{1,j}$ is released when $J_{1,j}$ is released. For $i \neq 1$, $S_{i,j}$ is released at time $r_{1,j} + \mathcal{O}_i$.

The Offset Rule ensures Prop. 1 and the Server Release Rule, along with the Job Release Rule, guarantees Prop. 2.

Property 1. *For $i \neq 1$, a server job $S_{i,j}$ is released after every $S_{k,j}$ where $\tau_k \in pred(\tau_i)$ completes.*

Property 2. *Server jobs of each node are released periodically with an interarrival time of T.*

Due to the parallelism level of the rp-sporadic task model, a released server job $S_{i,j}$ is not *ready* to be scheduled until $S_{i,j-\rho}$ completes. Therefore, similarly to the case for jobs of nodes, we say that a server job $S_{i,j}$ is ready when **(i)** it is released and not complete, and **(ii)** $S_{i,j-\rho}$ is complete.

Example 2. *Consider the DAG G in Fig. 3(a) with $\rho = 1$ and $T = 3$ scheduled on a two-core machine. Suppose that after transforming the reservation servers of G into an rp-sporadic task set, using the method in [2] to compute response time bounds, $R_1^{max} = 4$, $R_2^{max} = 4$, $R_3^{max} = 5$, and $R_4^{max} = 4$. Due to the Offset Rule, $\mathcal{O}_1 = 0$, $\mathcal{O}_2 = 4$, $\mathcal{O}_3 = 4$, and $\mathcal{O}_4 = 9$. Fig. 3(b) demonstrates the Server Release Rule. Assuming that $J_{1,1}$ is released at $t = 0$, due to the Server Release Rule, $S_{1,1}$ is released at $t = 0$, $S_{2,1}$ and $S_{3,1}$ are released at $t = 4$, and $S_{4,1}$ is released at $t = 9$. Note that at $t = 3$, due to the nature of rp-sporadic tasks, and $\rho = 1$, $S_{1,2}$ must wait for the completion of $S_{1,1}$ before it becomes ready at $t = 4$.*

Node priority. We say that a node τ_i has *higher* (resp. *lower*) priority than τ_k when $S_{i,j}$ has higher (resp. lower) priority than $S_{k,j}$ for each job index j. We can compare the priorities of τ_i and τ_k in G by examining $\mathcal{O}_i$ and $\mathcal{O}_k$. If $\mathcal{O}_i < \mathcal{O}_k$, then due to the Server Release Rule and Prop. 2, $S_{i,j}$ is released earlier than $S_{k,j}$ for each j. Since all server jobs have the same relative deadline of T, τ_i has higher priority than τ_k under G-EDF. Additionally, we say that τ_i *tops* τ_j's priority when $S_{i,j}$ has higher priority than $S_{k,j+\rho}$ for each j. Whether τ_i tops τ_k's priority can be determined by examining node offsets. If $\mathcal{O}_i < \mathcal{O}_k + \rho T$, then by the Server Release Rule, $S_{i,j}$ is released ρT time units earlier than $S_{k,j}$. Then, by Prop. 2, $S_{i,j}$ is released earlier than $S_{k,j+\rho}$, implying τ_i tops τ_k's priority.

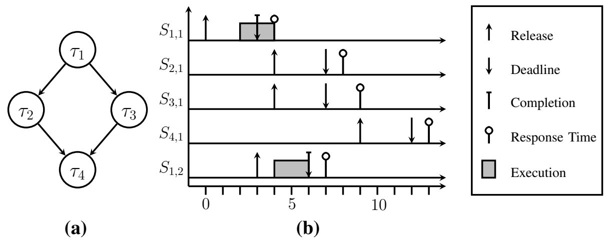

Fig. 3: (a) A processing graph G, and (b) a timeline showing server job releases of G after being converted into an rp-sporadic task set.

III. DAG Budgeting

This section presents our budget-management policy for DAG tasks with restricted parallelism. Similar to the scheme in [7], our budget-management policy schedules regular jobs within the budget of server jobs. The budget of each server job $S_{i,j}$ can be split into three states: **(i)** before $J_{i,j}$ becomes ready, **(ii)** when $J_{i,j}$ is ready, and **(iii)** after $J_{i,j}$ is complete. State **(i)** occurs when jobs in $dep(J_{i,j})$ overrun, preventing $J_{i,j}$ from becoming ready. To quickly progress from **(i)** to **(ii)**, we propose rules for managing overrunning jobs in Sec. III-B. When in **(ii)**, $S_{i,j}$ can progress to state **(iii)** through Rule R1.

> **R1.** If $S_{i,j}$ is scheduled and $J_{i,j}$ is ready at time t, then $J_{i,j}$ is scheduled on $S_{i,j}$.

Finally, **(iii)** occurs when $J_{i,j}$ underruns. In this state, we propose rules in Sec. III-A that allow $S_{i,j}$ to use the slack from $J_{i,j}$'s underrun to schedule other jobs. Rules in both Sec. III-A and Sec. III-B aim to limit the increase in job overrun probabilities due to cross-invocation overruns. Based on these ideas, in Sec. III-C, we propose a DAG abort scheme that limits the length of overrun cascades.

A. Cross Invocation Slack Reallocation

When a job $J_{i,j}$ underruns, slack is generated in the server job $S_{i,j}$. In certain scenarios, this slack, or remaining budget of $S_{i,j}$, can be used to execute $J_{i,j+\rho}$ before the server job $S_{i,j+\rho}$ is released. Doing so reduces the probability of $J_{i,j+\rho}$ overrunning, thus slowing down the increase in job overrun probabilities due to cross-invocation overruns. In Ex. 3, we illustrate how the slack of $S_{i,j}$ can be used to execute $J_{i,j+\rho}$.

Example 3. *Consider the DAG G in Fig. 4(a) with a period of $T = 14$. τ_1 is a dummy source node with a server budget of 0, and all other nodes in G have a server budget of 6. Fig. 4(b) gives an example schedule of G with $\rho = 1$ on a*

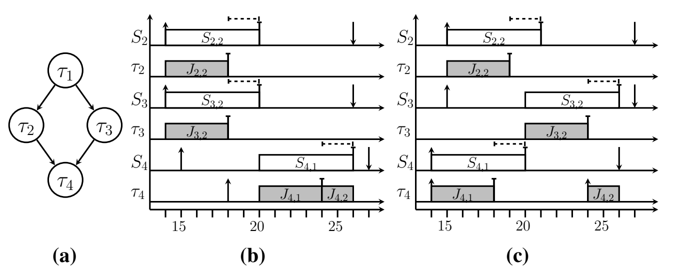

Fig. 4: (a) A processing graph G. (b) A schedule of G that shows how $J_{i,j+\rho}$ can be scheduled on the slack of $S_{i,j}$. (c) An alternate schedule of G that shows how $J_{i,j+\rho}$ can instead be scheduled on the slack of $S_{k,j+\rho}$ where $\tau_k \in pred(\tau_i)$. Job executions are shaded, and dotted lines are used to indicate the slack of server jobs.

two-core machine. $S_{4,1}$ has a lower priority than $S_{2,2}$ and $S_{3,2}$, so $S_{2,2}$ and $S_{3,2}$ are scheduled from $t = 14$ to 20. This allows $J_{2,2}$ (resp. $J_{3,2}$) to be scheduled on $S_{2,2}$ (resp. $S_{3,2}$) via Rule R1 and complete at $t = 18$. By the Job Release Rule, $J_{4,2}$ is therefore released at $t = 18$. $J_{4,1}$, which was released prior to $t = 14$, completes at $t = 24$. This makes $J_{4,2}$ ready at $t = 24$ due to the definition of job readiness. Additionally, since $J_{4,1}$ completes at $t = 24$, 2 units of slack are created in $S_{4,1}$ from $t = 24$ to 26. This slack can be used to schedule the ready job $J_{4,2}$ from $t = 24$ to 26.

From Ex. 3, we see that if a server job $S_{i,j}$ ($S_{4,1}$ in the example) has a lower priority than server jobs in $\{S_{k,j+\rho} \mid \tau_k \in pred(\tau_i)\}$ ($S_{2,2}$ and $S_{3,2}$), then $S_{i,j}$ can use its slack to schedule $J_{i,j+\rho}$ ($J_{4,2}$). However, what if $S_{i,j}$ has a higher priority than server jobs in $\{S_{k,j+\rho} \mid \tau_k \in pred(\tau_i)\}$? We consider this alternative scenario in the following example.

Example 4. *The DAG G in Fig. 4(a) has a new period of $T = 15$, and $S_{4,1}$ is now released first at $t = 14$, followed by the release of $S_{2,2}$ and $S_{3,2}$ at $t = 15$. Fig. 4(c) depicts the new schedule of G. $S_{3,2}$ now has a lower priority than $S_{4,2}$ and $S_{2,2}$, resulting in the slack of $S_{4,1}$ to be from the time interval $t = 18$ to 20. Meanwhile, due to the Job Release Rule, $J_{4,2}$ can only become released (and therefore ready) at $t = 24$ when both $J_{2,2}$ and $J_{3,2}$ are complete. Thus, $J_{4,2}$ cannot take advantage of the slack of $S_{4,1}$. However, $J_{4,2}$ can take advantage of the slack of $S_{3,2}$ from $t = 24$ to 26.*

From Ex. 4, we see that if $S_{i,j}$ ($S_{4,1}$ in the example) has a higher priority than server jobs in $\{S_{k,j+\rho} \mid \tau_k \in pred(\tau_i)\}$ ($S_{2,2}$ and $S_{3,2}$), then $J_{i,j+\rho}$ ($J_{4,2}$) can be scheduled in the slack of the server job $S_{\ell,j+\rho}$ ($S_{3,2}$) with the lowest priority under G-EDF in $\{S_{k,j+\rho} \mid \tau_k \in pred(\tau_i)\}$.

Slack reallocation rules. Ex. 3 exhibits a scenario where a server job, $S_{4,1}$, uses its slack to schedule a future job of the same task, $J_{4,2}$. Meanwhile, in Ex. 4, a server job, $S_{3,2}$, uses its slack to schedule a job of a successor node, $J_{4,2}$. To guarantee the progress of jobs in our analysis, the job executed using the slack of server jobs must be determined offline. We therefore allow a server job to prioritize jobs of certain nodes through the use of *preferred successors* as introduced in [7]. Each node has only one preferred successor, so jobs of

Algorithm 1 Procedure for assigning preferred successors.

```
 1: procedure PREFSUCC($G$)
 2:     for $\tau_i \in V$ do
 3:         $pref(\tau_i) := \tau_i$
 4:         for $\tau_k \in pred(\tau_i)$ do
 5:             if $\tau_i$ tops $\tau_k$'s priority then
 6:                 $pref(\tau_i) := \emptyset$
 7:     for $\tau_i \in G - \tau_1$ do
 8:         Let $\tau_k$ be the lowest priority node in $pred(\tau_i)$
 9:         if $pref(\tau_k) = \emptyset$ then
10:             $pref(\tau_k) := \tau_i$
```

the preferred successor node will naturally be prioritized. We denote the preferred successor of a node τ_i as $pref(\tau_i)$. Using preferred successors, we can schedule jobs as follows.

> **R2.1.** $pref(\tau_i) = \tau_i$: if $S_{i,j}$ is scheduled, $J_{i,j}$ is complete, and $J_{i,j+\rho}$ is ready, then $S_{i,j}$ schedules $J_{i,j+\rho}$.
>
> **R2.2.** $pref(\tau_k) = \tau_i$, where $i \neq k$: if $S_{k,j}$ is scheduled, $J_{k,j}$ is complete, and $J_{i,j}$ is ready, then $S_{k,j}$ schedules $J_{i,j}$.

Assigning preferred successors. Rule R2.1 is intended to correspond to the scenario in Ex. 3, where a server job $S_{i,j}$ uses its slack to schedule $J_{i,j+\rho}$ if $S_{i,j}$ has lower priority than server jobs in $\{S_{k,j+\rho} \mid \tau_k \in pred(\tau_i)\}$. This intention is realized through the preferred successor assignment in Alg. 1.

Property 3. *If $pref(\tau_i) = \tau_i$, then $S_{i,j}$ has lower priority than each server job in $\{S_{x,j+\rho} \mid \tau_x \in pred(\tau_i)\}$*

Proof. Consider the assignment of $pref(\tau_i)$ in Alg. 1. After line 3, $pref(\tau_i) = \tau_i$. After the loop in lines 4 to 6, $pref(\tau_i) = \tau_i$ only if τ_i is lower priority than each $\tau_x \in pred(\tau_i)$ across invocations. Thus, $S_{i,j}$ is lower priority than each $S_{x,j+\rho}$. $\square$

Meanwhile, Rule R2.2 corresponds to the scenario in Ex. 4, where an overrunning job $J_{i,j+\rho}$ is scheduled in the slack of $S_{\ell,j+\rho}$, the lowest-priority server job in $\{S_{k,j+\rho} \mid \tau_k \in pred(\tau_i)\} \cup \{S_{i,j}\}$. Alg. 1 guarantees this through the following property, which in turn implies Cor. 1.

Property 4. *If $pref(\tau_k) = \tau_i$, where $i \neq k$, then $S_{k,j}$ is the lowest-priority server job in $\{S_{x,j} \mid \tau_x \in pred(\tau_i)\} \cup \{S_{i,j-\rho}\}$.*

Proof. Consider the assignment of τ_i to $pref(\tau_k)$ in Alg. 1. Due to line 8, τ_k is the lowest-priority node in $pred(\tau_i)$. This implies that $S_{k,j}$ is the lowest-priority job in $\{S_{x,j} \mid \tau_x \in pred(\tau_i)\}$. Additionally, in the **if** on line 9, τ_i is only assigned to $pref(\tau_k)$ if $pref(\tau_k) = \emptyset$. This can only occur in the **if** statement in line 5, which implies that $pref(\tau_k) = \tau_i$ only when τ_k does not top τ_x's priority for each $\tau_x \in pred(\tau_k)$. This implies that $S_{i,j-\rho}$ has higher priority than $S_{k,j}$. $\square$

Corollary 1. *If $pref(\tau_k) = \tau_i$ and $i \neq k$, then $\tau_k \in pred(\tau_i)$*

B. Cross Invocation Overrun Management

When jobs in $dep(J_{i,j})$ overrun, the time $J_{i,j}$ becomes ready can be delayed. In our overrun-management scheme, we aim to reduce the effect this delay has on the ready time of $J_{i,j}$. We provide the intuition of our approach through an example.

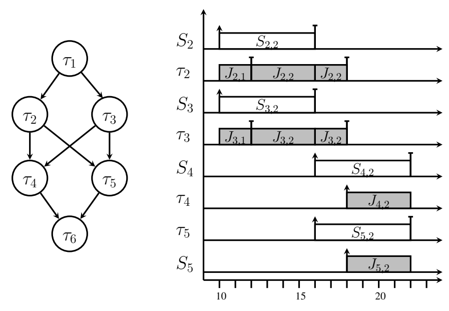

Fig. 5: (a) A processing graph G. (b) A schedule of G where server jobs schedule overrunning jobs in parallel.

Example 5. *Consider the DAG G in Fig. 5(a) where τ_1 and τ_6 are dummy nodes, and all other nodes have a server budget of 6. Additionally, $T = 10$, $\rho = 1$, and the offset of nodes 1 through 3 is 0, while the offset of nodes 4 and 5 is 6. Fig. 5(b) gives an example schedule of G on a two-core machine. Suppose $J_{2,1}$ and $J_{3,1}$ overrun by 2 time units before $t = 10$. Then $J_{2,2}$ and $J_{3,2}$ are not ready to be scheduled by Rule R1 at $t = 10$. Thus, $S_{2,2}$ (resp. $S_{3,2}$) must first complete $J_{2,1}$ (resp. $J_{3,1}$) before $J_{2,2}$ (resp. $J_{3,2}$) becomes ready. This cross-invocation overrun causes both $J_{2,2}$ and $J_{3,2}$ to overrun by 2 time units. Due to the Job Release Rule, both $J_{2,2}$ and $J_{3,2}$ must complete before $J_{4,2}$ and $J_{5,2}$ can be released. Since $S_{4,2}$ and $S_{5,2}$ are scheduled in parallel at $t = 16$, $J_{2,2}$ and $J_{3,2}$ can be completed in parallel on these two server jobs. This minimizes the time for both $J_{4,2}$ and $J_{5,2}$ to become ready, reducing the effects of the earlier cross-invocation overrun on the overrun of $J_{4,2}$ and $J_{5,2}$.*

Observe from Ex. 5 that when a job $J_{i,j-\rho}$ ($J_{2,1}$ in the example) prevents $J_{i,j}$ ($J_{2,2}$) from being ready, the server job $S_{i,j}$ ($S_{2,2}$) can ensure that $J_{i,j}$ *makes progress* towards being ready by ensuring that $J_{i,j-\rho}$ *makes progress* towards its completion. We formally define making progress as follows.

Definition 1. *A job $J_{i,j}$ makes progress on server job $S_{k,\ell}$ if a job in $dep(J_{i,j}) \cup \{J_{i,j}\}$ is executing and $S_{k,\ell}$ is scheduled.*

From Def. 1, we see that a job $J_{i,j}$ makes progress on $S_{k,\ell}$ when $S_{k,\ell}$ schedules a ready job in $dep(J_{i,j}) \cup \{J_{i,j}\}$. Thus, we can formalize our first observation of Ex. 5 into the first overrun-management rule.

> **R3.1.** If $S_{i,j}$ is scheduled, and $J_{i,j}$ is released but not ready, then $S_{i,j}$ schedules a ready job in $dep(J_{i,j-\rho}) \cup \{J_{i,j-\rho}\}$.

Parallel scheduling of overrunning jobs. Additionally, observe from Ex. 5 that when the release of a job $J_{i,j}$ ($J_{5,2}$ in the example) is delayed due to overrunning jobs in $dep(J_{i,j})$ ($J_{2,2}$ and $J_{3,2}$), these overrunning jobs can make progress in parallel on server jobs scheduled in parallel ($S_{4,2}$ and $S_{5,2}$). This parallel scheduling of jobs can quickly complete overrunning jobs in $dep(J_{i,j})$, greatly expediting $J_{i,j}$'s release. To take advantage of this observation, we first identify server jobs that can execute in parallel. Logically, server jobs of the same graph with the same release can execute in parallel.

Definition 2. *For two nodes in G, τ_i and τ_k, τ_k is in the* parallel set *of τ_i if $r_{k,j} = r_{i,j}$ for all job indices j. We denote the parallel set of τ_i as $PS(\tau_i)$.*

Due to the Offset Rule and the Server Release Rule, it is not uncommon for a parallel set to contain more than one node. For instance, in the DAG in Fig. 5(a), both τ_4 and τ_5 share the same predecessor nodes, resulting in $\mathcal{O}_4 = \mathcal{O}_5$ from the Offset Rule. Thus, by the Server Release Rule, $r_{4,j} = r_{5,j}$ for all job indices j, implying $PS(\tau_5) = \{\tau_4, \tau_5\}$. However, due to the Tie-Breaking Rule, a node such as τ_4 can have higher priority than another node (τ_5) in the same parallel set. This, along with the following property can allow jobs from nodes of the same parallel set to execute in parallel.

Property 5. *If a server job $S_{k,y}$ has higher priority than the server job $S_{i,x}$, and $S_{i,x}$ is scheduled for t time units after $S_{k,y}$ becomes ready, then $S_{k,y}$ must be scheduled for at least $\min(t, C_k)$ time units.*

Proof. With G-EDF, if $S_{i,x}$ is scheduled after $S_{k,y}$ becomes ready, then the higher-priority $S_{k,y}$ is also scheduled unless it is complete. Thus, if $S_{i,x}$ is scheduled for t time units after $S_{k,y}$ becomes ready, then either $S_{k,y}$ is scheduled for at least t time units, or is complete (scheduled for C_k time units). □

Due to Prop. 5, it is helpful to introduce the following.

Definition 3. *HPS(τ_i) is the set of nodes in $PS(\tau_i)$ with priority at least that of τ_i.*

Helping set. Once we have identified a set of server jobs that can execute in parallel, we can use these server jobs to allow overrunning jobs to make progress in parallel. We see this in Fig. 5(b) at $t = 16$ to 18, where a job of τ_2 (resp. τ_3) makes progress on a server job of τ_4 (resp. τ_5). In such a case, we say that τ_4 (resp. τ_5) *helps* τ_2 (resp. τ_3). Formally, we have:

Definition 4. *We say that a node τ_i* helps *a node τ_k if $S_{i,j}$ must ensure that the overrunning job $J_{k,j}$ makes progress on $S_{i,j}$. We let the* helping set *of τ_i, denoted as $\mathbb{H}_i$, be the set of nodes that τ_i must help.*

With careful assignment of nodes to helping sets, we can use the following rule to ensure that overrunning jobs in $dep(J_{i,j})$ make progress in parallel, greatly expediting $J_{i,j}$'s release. We demonstrate this in Ex. 6

> **R3.2.** If $S_{i,j}$ is scheduled, but $J_{i,j}$ is not released, then $S_{i,j}$ schedules a ready job in $dep(J_{x,j}) \cup \{J_{x,j}\}$ where $\tau_x \in \mathbb{H}_i$.

Example 6. *Continuing Ex. 5, we have $pred(\tau_5) = \{\tau_2, \tau_3\}$, and PS$(\tau_5) = \{\tau_4, \tau_5\}$. If we assign $\mathbb{H}_4 = \{\tau_2\}$, $\mathbb{H}_5 = \{\tau_3\}$, then by Def. 4, $S_{4,2}$ (resp. $S_{5,2}$) must ensure that $J_{2,2}$ (resp. $J_{3,2}$) makes progress on $S_{4,2}$ (resp. $S_{5,2}$). This is realized during $t = 16$ to 18 of the schedule in Fig. 5(b), where $S_{4,2}$ and $S_{5,2}$ are scheduled, but $J_{4,2}$ is not released. By Rule R3.2, $S_{4,2}$ (resp. $S_{5,2}$) schedules the ready job $J_{2,2}$ (resp. $J_{3,2}$) where $\tau_2 \in \mathbb{H}_4$ (resp. $\tau_3 \in \mathbb{H}_5$).*

From Ex. 6, we see that $S_{4,2}$ (resp. $S_{5,2}$) completes $J_{2,2}$ (resp. $J_{3,2}$), thus allowing $J_{5,2}$ to release. However, what if $S_{4,2}$ completes before $J_{2,2}$ completes? In that case, $S_{5,2}$ must complete $J_{2,2}$ along with $J_{3,2}$ before $J_{5,2}$ can release. This motivates our final overrun management rule.

R3.3. If $S_{i,j}$ is scheduled, but $J_{i,j}$ is not released and no job can be scheduled using Rule R3.2, then $S_{i,j}$ schedules a ready job in $dep(J_{x,j}) \cup \{J_{x,j}\}$ where $\tau_x \in pred(\tau_i)$.

Helping set assignment. Observe from Ex. 6 that the assignment $\mathbb{H}_4 = \{\tau_2\}$ and $\mathbb{H}_5 = \{\tau_3\}$ can be achieved by *evenly partitioning* the common predecessors of nodes in $PS(\tau_5)$ (τ_2 and τ_3) across the helping sets of nodes in $PS(\tau_5)$ ($\mathbb{H}_4$, $\mathbb{H}_5$). This assignment allowed the overrunning jobs $J_{2,2}$ and $J_{3,2}$ to execute in parallel. Therefore, following the intuition in Ex. 6, we wish to evenly partition the common predecessors of nodes in $PS(\tau_i)$ across $\mathbb{H}_k$ for $\tau_k \in PS(\tau_i)$. To do so, we first formally define the concept of an *even partitioning*.

Definition 5. *Let $\mathbb{P}^s(X) = \{\mathbb{P}^s_1(X), \mathbb{P}^s_2(X), ..., \mathbb{P}^s_s(X)\}$ be an s-partition of set X. $\mathbb{P}^s(X)$ is an* even partitioning *of X across s sets if $\lfloor \frac{|X|}{s} \rfloor \leq |\mathbb{P}^s_i(X)| \leq \lceil \frac{|X|}{s} \rceil$ for each $\mathbb{P}^s_i(X) \in \mathbb{P}^s(X)$.*

Using Def. 5, we can express our helping set assignment using the Assignment Rule.

Assignment Rule. For each node τ_i, let (k) be the index of the k^{th} node in $PS(\tau_i)$. The helping set of each $\tau_{(k)}$ is assigned according to the following equation.

$$\mathbb{H}_{(k)} = \mathbb{P}_k^{|PS(\tau_i)|}\left(\bigcap_{\tau_x \in PS(\tau_i)} pred(\tau_x)\right)$$

In the Assignment Rule, the intersection represents the common predecessors of nodes in $PS(\tau_i)$. These common predecessors are evenly partitioned into $|PS(\tau_i)|$ sets using the $\mathbb{P}_k^{|PS(\tau_i)|}$ term. Each of these sets is then assigned to $\mathbb{H}_{(k)}$ for some $\tau_{(k)} \in PS(\tau_i)$. Overall, this results in the common predecessors of nodes in $PS(\tau_i)$ being evenly partitioned across $\mathbb{H}_k$ for $\tau_k \in PS(\tau_i)$. We can show that the Assignment Rule ensures Prop. 6. Using this property, we can show in Prop. 7 that a node must help the nodes in its helping set.

Property 6. *For each node τ_i, $\bigcup_{\tau_k \in HPS(\tau_i)} \mathbb{H}_k \subseteq pred(\tau_i)$*

Proof. Let $X = \bigcup_{\tau_k \in HPS(\tau_i)} \mathbb{H}_k$, $Y = \bigcup_{\tau_k \in PS(\tau_i)} \mathbb{H}_k$, and $Z = \bigcap_{\tau_k \in PS(\tau_i)} pred(\tau_k)$. The choice of Z implies $Z \subseteq pred(\tau_i)$. Since the Assignment Rule evenly partitions nodes in Z across $\mathbb{H}_k$ for each $\tau_k \in PS(\tau_i)$, we have $Z = Y$. This results in $Y \subseteq pred(\tau_i)$. Due to Def. 3, $HPS(\tau_i) \subseteq PS(\tau_i)$, which implies that $X \subseteq Y$. This results in $X \subseteq pred(\tau_i)$. □

Property 7. *If $S_{i,j}$ is scheduled, and there exist an incomplete job in $\{J_{k,j} \mid \tau_k \in \mathbb{H}_i\}$, then a job in $\{J_{k,j} \mid \tau_k \in \mathbb{H}_i\}$ makes progress on $S_{i,j}$.*

Proof. Since $\tau_i \in HPS(\tau_i)$, Prop. 6 implies that $\mathbb{H}_i \subseteq pred(\tau_i)$. Therefore, when a job in $\{J_{k,j} \mid \tau_k \in \mathbb{H}_i\}$ is incomplete, that incomplete job must also be in $\{J_{k,j} \mid \tau_k \in pred(\tau_i)\}$. By the Job Release Rule, this implies that $J_{i,j}$ is not released. Therefore, by Rule R3.2, a job in $\{J_{k,j} \mid \tau_k \in \mathbb{H}_i\}$ makes progress on $S_{i,j}$. □

C. Graph Abort Condition

An invocation of a graph G may be aborted for two reasons. First, when a job of the sink node $J_{n,j}$ overruns, the invocation is aborted so that G has a known response-time bound. Since $S_{n,j}$ is released at time $r_{1,j} + \mathcal{O}_n$, it is guaranteed to complete by $r_{1,j} + \mathcal{O}_n + R_n^{max}$. By aborting $J_{n,j}$ when it overruns, either G completes within $\mathcal{O}_n + R_n^{max}$ time units, or it is aborted. This forms the first graph abort condition.

R4.1. If $J_{n,j}$ is not complete when $S_{n,j}$ exhausts its budget, then the j^{th} graph invocation is aborted.

The second reason to abort a graph invocation is to limit overrun cascades. Due to the definition of job readiness, a job $J_{i,j}$ can overrun and delay the ready time of $J_{i,j+\rho}$. This can then cause $J_{i,j+\rho}$ to overrun and further delay the completion of $J_{i,j+2\rho}$. This forms an *overrun cascade* on τ_i.

Definition 6. *An overrun cascade of length L exists on τ_i when $J_{i,j+x\rho}$ for each $x \in [1, L]$ is allowed to overrun.*

We can see that as the length of an overrun cascade increases, the chance of a job overrun also increases. Fortunately, we can place a limit, $\mathcal{L}$, on the overrun cascade length of each node in G by periodically ensuring that jobs cannot overrun. We do this by cycling through nodes that cannot experience budget overruns on different graph invocations. An example of this cycle is given in Fig. 2. When a job $J_{i,j}$ cannot overrun, it may not be able to complete, thereby preventing the release of other jobs due to the Job Release Rule. This necessitates aborting the entire j^{th} graph invocation, giving rise to the following abort condition. Due to the complex nature of Rule R4.2, we give a demonstrative example.

R4.2. The j^{th} graph invocation is aborted if $J_{i,j}$ is not complete when $S_{i,j}$ exhausts its budget and $i \in \left[\frac{n}{\mathcal{L}}\mathcal{P} + 1, \frac{n}{\mathcal{L}}(\mathcal{P} + 1)\right]$, where

$$\mathcal{P} = \left(\left\lfloor \frac{j-1}{\rho} \right\rfloor \mod \mathcal{L}\right).$$

Example 7. *Consider the graph G in Fig. 5(a). Suppose $\rho = 2$ and $\mathcal{L} = 2$. For the first graph invocation, $\mathcal{P}$ evaluates to 0. This results in a different value each time a new set of nodes can cause a graph invocation to abort. $\frac{n}{\mathcal{L}}$ evaluates to 3. This is the number of nodes that can cause each graph invocation to abort. According to Rule R4.2, if $J_{i,1}$ overruns and $i \in [1, 3]$, then the 1^{st} graph invocation is aborted. This includes $J_{1,1}$, $J_{2,1}$, and $J_{3,1}$. For the second graph invocation, $\mathcal{P}$ remains 0, so the graph invocation is aborted if $J_{i,2}$ overruns and $i \in [1, 3]$. For the third (resp. fourth) graph invocations, $\mathcal{P}$ evaluates to 1, so the graph invocation is aborted if $J_{i,3}$ (resp. $J_{i,4}$) overrun for $i \in [4, 6]$. Finally, for the 5^{th} graph invocation, the mod operator restarts the cycle, resulting in a graph invocation abort when $J_{1,5}$, $J_{2,5}$, or $J_{3,5}$ overrun.*

IV. Drop-Rate Analysis

In this section, we provide analysis to upper bound the abort probability of each graph instance. In Sec. IV-A, we analyze the effects of our slack reallocation technique introduced in Sec. III-A, and in Sec. IV-B, we analyze the effects of our overrun management technique introduced in Sec. III-B. Finally, in Sec. IV-C, we combine the effects of both to obtain an upper bound on the abort probability of each graph instance.

Tracking job progress with demand. We can track how much progress a job needs to make before its completion with the concept of *demand*, formally defined below.

Definition 7. *The* demand *of $J_{i,j}$ is the number of time units for which $J_{i,j}$ makes progress on $S_{i,j}$ or server jobs released after $S_{i,j}$ before $J_{i,j}$ completes.*

Definition 8. *Let the RV $\delta_{i,j}$ equal a value that upper-bounds the demand of $J_{i,j}$.*

From our scheduling rules in Sec. III, we can show that the job $J_{i,j}$ makes progress whenever $S_{i,j}$ is scheduled.

Lemma 1. *If a server job $S_{i,j}$ is scheduled and $J_{i,j}$ is incomplete, then $J_{i,j}$ makes progress on $S_{i,j}$*

Proof. If $J_{i,j}$ is ready, then $S_{i,j}$ schedules $J_{i,j}$ by Rule R1, implying $J_{i,j}$ makes progress on $S_{i,j}$ by Def. 1. If $J_{i,j}$ is not ready, then by the Job Release Rule and the definition of job readiness, $J_{i,j}$ depends on the completion of jobs in $X = \{J_{k,j} \mid \tau_k \in pred(\tau_i)\} \cup \{J_{i,j-\rho}\}$. If we show that when $S_{i,j}$ is scheduled, Rules R3.1 to R3.3 cause some job in X to make progress on $S_{i,j}$, then $J_{i,j}$ also makes progress on $S_{i,j}$ due to Def. 1. We consider two cases depending on whether $J_{i,j}$ has been released when $S_{i,j}$ is scheduled.

Case 1. $J_{i,j}$ has not been released. Rule R3.2 ensures that a job in $\{J_{k,j} \mid \tau_k \in \mathbb{H}_i\}$ makes progress on $S_{i,j}$. Since $\mathbb{H}_i \subseteq pred(\tau_i)$ by Prop. 6, a job in $\{J_{k,j} \mid \tau_k \in pred(\tau_i)\}$ makes progress on $S_{i,j}$. Rule R3.3 also ensures that a job in $\{J_{k,j} \mid \tau_k \in pred(\tau_i)\}$ makes progress on $S_{i,j}$.

Case 2. $J_{i,j}$ has been released, but is not ready. Rule R3.1 ensures that $J_{i,j-\rho}$ makes progress on $S_{i,j}$ by Def. 1. □

From Def. 7 and Lem. 1, we see that when the demand of a job $J_{i,j}$ is greater than its server budget C_i, $J_{i,j}$ will overrun. Since $\delta_{i,j}$ upper-bounds the demand of $J_{i,j}$, $P(\delta_{i,j} > C_i)$ upper-bounds the probability $J_{i,j}$ overruns. Therefore, because Rules R4.1 and R4.2 can cause graph invocations to abort when certain jobs overrun, we can compute the graph drop rate by analyzing how $\delta_{i,j}$ is affected by our scheduling rules.

Comparisons between RVs. In the following analysis, we say that for RVs X and Y, $X \geq_0 Y$ when the value of X is always at least that of Y.

A. Effects of Slack Reallocation

Here, we examine how our slack-reallocation rules affect each $\delta_{i,j}$. We do this using the RV $\Psi_{i,j}$, which represents a lower bound on the duration $J_{i,j}$ is scheduled due to R2.1 and R2.2. To reflect that Rules R2.1 and R2.2 only apply to preferred successor nodes, we have the following.

Definition 9. *If τ_i is not a preferred successor node, then $\Psi_{i,j} = 0$ for each job $J_{i,j}$.*

Additionally, $J_{i,j}$ cannot be scheduled on $S_{i,j-\rho}$ by Rule R2.1 if $j - \rho < 1$ ($S_{i,j-\rho}$ does not exist).

Definition 10. *If $j - \rho < 1$, then $\Psi_{i,j} = 0$ for each job $J_{i,j}$.*

For the rest of the analysis, we assume that $j-\rho \geq 1$. If τ_i is a preferred successor node due to Alg. 1, either $\tau_i = pref(\tau_k)$ where $i \neq k$ or $\tau_i = pref(\tau_i)$. In the former case, $\Psi_{i,j}$ is defined as follows.

Definition 11. *If $pref(\tau_k) = \tau_i$ and $k \neq i$, then $\Psi_{i,j} = C_k - \max(\{s_{x,j} \mid \tau_x \in pred(\tau_i)\} \cup \{s_{i,j-\rho} + C_i - C_k\})$, where*

$$s_{i,j} = \begin{cases} \delta_{i,j} & \delta_{i,j} < C_i \\ \infty & \textit{otherwise.} \end{cases} \tag{1}$$

To understand this definition, first note that $J_{i,j}$ can only be scheduled on $S_{k,j}$ via Rule R2.2 if $J_{i,j}$ is ready and $J_{k,j}$ is complete, which occurs when all jobs in $\{J_{x,j} \mid \tau_x \in pred(\tau_i)\} \cup \{J_{i,j-\rho}\}$ complete. Thus, by examining the remaining budget of $S_{k,j}$ when these jobs complete, we can compute $\Psi_{i,j}$. To do this, we first show that when $S_{k,j}$ is scheduled, these jobs are scheduled in parallel on server jobs in $\{S_{x,j} \mid \tau_x \in pred(\tau_i)\} \cup \{S_{i,j-\rho}\}$ (hence the max).

Lemma 2. *If $pref(\tau_k) = \tau_i$ and $k \neq i$, then $S_{x,j}$ where $\tau_x \in pred(\tau_i)$ becomes ready no later than $S_{k,j}$'s release.*

Proof. This lemma follows from the definition of job readiness if **(i)** $S_{x,j}$ is released no later than $S_{k,j}$ and **(ii)** $S_{x,j-\rho}$ completes no later than $S_{k,j}$'s release. From Prop. 4, $S_{k,j}$ has lower priority than $S_{i,j-\rho}$ and $S_{x,j}$. Thus, by the definition of G-EDF, $r_{k,j} \geq r_{i,j-\rho}$ and $r_{k,j} \geq r_{x,j}$. $r_{k,j} \geq r_{x,j}$ implies **(i)**. Meanwhile, by Prop. 1, $S_{x,j-\rho}$ completes no later than $S_{i,j-\rho}$'s release. Since $r_{k,j} \geq r_{i,j-\rho}$, $S_{x,j-\rho}$ completes no later than $S_{x,j}$'s release as well, satisfying **(ii)**. □

Lemma 3. *If $pref(\tau_k) = \tau_i$ where $k \neq i$, and $S_{k,j}$ has been scheduled for t time units, then $S_{x,j}$ where $\tau_x \in pred(\tau_i)$ has been scheduled for at least* $\min(t, C_x)$ *time units.*

Proof. The lemma follows from Prop. 5 if **(i)** $S_{k,j}$ has lower priority than $S_{x,j}$ and **(ii)** $S_{x,j}$ becomes ready no later than $S_{k,j}$'s release. Prop. 4 implies **(i)** and Lem. 2 implies **(ii)**. □

To lower-bound the execution of $S_{i,j-\rho}$ when $S_{k,j}$ has executed for t time units, we require the following theorem of rp-sporadic task systems proved in [2].

Theorem 1. *There exists a constant x such that the response-time bound R_i^{max} of any server S_i under G-EDF scheduling is at most $x + T + C_i$.*

Lemma 4. *If $pref(\tau_k) = \tau_i$ and $k \neq i$, then $S_{i,j-\rho}$ becomes ready at most $C_i - C_k$ time units after $S_{k,j}$'s release.*

Proof. This lemma follows from the definition of job readiness if **(i)** $S_{i,j-\rho}$ is released no later than $S_{k,j}$'s release and **(ii)** $S_{i,j-2\rho}$ (if it exists) completes at most $C_i - C_k$ time units after $S_{k,j}$'s release. From Prop. 4, $S_{k,j}$ has lower priority than $S_{i,j-\rho}$. Thus, by the definition of G-EDF, $r_{k,j} \geq r_{i,j-\rho}$, implying **(i)**. Meanwhile, because of Cor. 1, $\tau_k \in pred(\tau_i)$. Thus, by the Offset Rule and the Server Release Rule, we have $r_{i,j-\rho} \geq r_{k,j-\rho} + R_k^{max}$. Since $r_{k,j} \geq r_{i,j-\rho}$, we have $r_{k,j} \geq r_{k,j-\rho} + R_k^{max}$. This implies by Prop. 2 that $\rho T \geq R_k^{max}$. From Thm. 1, we have $R_k^{max} = x + T + C_k$ and $R_i^{max} = x + T + C_i$, which imply $R_k^{max} = R_i^{max} - C_i + C_k$. By substitution into $\rho T \geq R_k^{max}$, we have $\rho T + C_i - C_k \geq R_i^{max}$. This means that $S_{i,j-2\rho}$, if it exists, completes no later than $t = r_{i,j-2\rho} + \rho T + C_i - C_k$. By Prop. 2, $r_{i,j-\rho} = r_{i,j-2\rho} + \rho T$, so we have $t = r_{i,j-\rho} + C_i - C_k$. Since $r_{k,j} \geq r_{i,j-\rho}$, we have $t \leq r_{k,j} + C_i - C_k$. This means that $S_{i,j-2\rho}$ completes at most $C_i - C_k$ time units after $S_{k,j}$'s release, satisfying **(ii)**. □

Lemma 5. *If $pref(\tau_k) = \tau_i$ where $k \neq i$ and $S_{k,j}$ has been scheduled for t time units, then $S_{i,j-\rho}$ has been scheduled for at least $\min(t - C_i + C_k, C_i)$ time units.*

Proof. The lemma follows from Prop. 5 if **(i)** $S_{k,j}$ has lower priority than $S_{x,j}$ and **(ii)** $S_{k,j}$ has been scheduled for at most $C_i - C_k$ time units before $S_{i,j-\rho}$ becomes ready. Prop. 4 implies **(i)** and Lem. 4 implies **(ii)**. □

We now use Lems. 3 and 5, to show that $\Psi_{i,j}$ is indeed a lower-bound on the time $J_{i,j}$ is scheduled due to Rule R2.2.

Lemma 6. *If $pref(\tau_k) = \tau_i$ and $k \neq i$ and $\Psi_{i,j} > 0$, then $S_{k,j}$ executes $J_{i,j}$ for at least $\Psi_{k,j}$ time units*

Proof. For brevity, let $X = \max(\{\delta_{x,j} \mid \tau_x \in pred(\tau_i)\} \cup \{\delta_{i,j-\rho} + C_i - C_k\})$. From Def. 11, we can verify that if $\Psi_{i,j} > 0$, then $C_k - X > 0$. Thus, we can let t be the time instant when $S_{k,j}$ has executed for X time units. By Lem. 3 (resp. Lem. 5), $S_{x,j}$ for each $\tau_x \in pred(\tau_i)$ (resp. $S_{i,j-\rho}$) must have executed for at least $\min(X, C_x)$ (resp. $\min(X - C_i + C_k, C_i)$) time units by t. Due to our choice of X, we have $X \geq_0 \delta_{x,j}$ and $X \geq_0 \delta_{i,j-\rho} + C_i - C_k$. This implies that by t, each $S_{x,j}$ (resp. $S_{i,j-\rho}$) has executed for at least $\min(\delta_{x,j}, C_x)$ (resp. $\min(\delta_{i,j-\rho}, C_i)$) time units. Because $\Psi_{i,j} > 0$, by Def. 11, $\delta_{x,j} < C_x$ for each τ_x and $\delta_{i,j-\rho} < C_i$. This implies that at t, each $S_{x,j}$ and $S_{i,j-\rho}$ is not complete. Thus, by Lem. 1, each $J_{x,j}$ (resp. $J_{i,j-\rho}$) has made progress on $S_{x,j}$ (resp. $S_{i,j-\rho}$) for at least $\delta_{x,j}$ (resp. $\delta_{i,j-\rho}$) time units by t. Defs. 7 and 8 then imply that $J_{i,j-\rho}$ and each $J_{x,j}$ are complete at t. Thus, by the Job Release Rule and the definition of job readiness $J_{i,j}$ is ready at t. Additionally, due to Cor. 1, $\tau_k \in pred(\tau_i)$. This implies that since each $J_{x,j}$ is complete at t, $J_{k,j}$ is also complete at t. Since $J_{i,j}$ is ready and $J_{k,j}$ is complete, by Rule R2.2, $J_{i,j}$ is scheduled on $S_{k,j}$ from t until $S_{k,j}$ completes. Since $S_{i,j-\rho}$ has already expended X units of budget by t. $J_{i,j}$ is scheduled on $S_{k,j}$ for $\Psi_{i,j}$ time units. □

Next, we define $\Psi_{i,j}$ for the case $\tau_i = pref(\tau_i)$. Note the similarities between Defs. 11 and 12.

Definition 12. *If $pref(\tau_i) = \tau_i$, then $\Psi_{i,j} = C_i - \max(\{s_{x,j} \mid \tau_x \in pred(\tau_i)\} \cup \{s_{i,j-\rho}\})$, where $s_{i,j}$ is given by (1)*

Lemma 7. *If $pref(\tau_i) = \tau_i$ and $\Psi_{i,j} > 0$, then $S_{i,j-\rho}$ executes $J_{i,j}$ for at least $\Psi_{i,j}$ time units.*

We can use a similar argument to the one in Lem. 6 to prove Lem. 7 by examining the amount of time server jobs in $\{S_{x,j} \mid \tau_x \in pred(\tau_i)\} \cup \{S_{i,j-\rho}\}$ has executed along-side $S_{i,j-\rho}$ instead of $S_{k,j}$. Using Lems. 6 and Lem. 7, we now show that $\delta_{i,j}$ is a function of $\Psi_{i,j}$.

Lemma 8. *If $\Psi_{i,j} > 0$, then $\delta_{i,j} = e_{i,j} - \Psi_{i,j}$*

Proof. If $\Psi_{i,j} > 0$, then by Lems. 7 and 6, $J_{i,j}$ executes for at least $\Psi_{i,j}$ time units on either $S_{i,j-\rho}$ or $S_{k,j}$ where $pref(\tau_k) = \tau_i$ (depending on whether it is the preferred successor of its own node or some other node). From Cor. 1, $\tau_k \in pred(\tau_i)$. By Prop. 1 and the definition of server job readiness, $S_{i,j}$ becomes ready after each $S_{k,j}$ and $S_{i,j-\rho}$ complete. Therefore, $J_{i,j}$ is scheduled for $\Psi_{i,j}$ time units before $S_{i,j}$ becomes ready. Since $J_{i,j}$ can execute for at most $e_{i,j}$ time units (its pWCET) before it completes, it can execute for at most $e_{i,j} - \Psi_{i,j}$ time units on $S_{i,j}$ and other server jobs before its completion. The lemma follows by Defs. 7 and 8. □

B. Effects of Overrun Management

In this section, we quantify the effects of our overrun management rules R3.1 to R3.3 on each $\delta_{i,j}$. Recall from Def. 7 that the demand of $J_{i,j}$ is the number of time units $J_{i,j}$ makes progress before its completion on $S_{i,j}$ and server jobs released after $S_{i,j}$. To aid in our analysis, we break the times that $J_{i,j}$ makes progress into distinct phases.

First phase. The first phase begins after the time instant $S_{i,j}$ becomes ready, which we denote as t_0. In this phase, jobs that prevent the release of $J_{i,j}$ can make progress in parallel due to Rule R3.2. This is because, during this phase, server jobs can execute alongside $S_{i,j}$. To prove this, we use the following property of the Tie-Breaking Rule.

Property 8. *For $\tau_k \in HPS(\tau_i)$, $C_i \geq C_k$.*

Lemma 9. *If $S_{i,j}$ has been scheduled for t time units, then $S_{k,j}$ where $\tau_k \in HPS(\tau_i)$ has been scheduled for at least $\min(t, C_k)$ time units.*

Proof. We prove this lemma via induction on job indices. For the base case, we first show that the lemma holds for each job index $x \leq \rho$. Since $x \leq \rho$, $S_{k,x-\rho}$ and $S_{i,x-\rho}$ do not exist, implying that $S_{k,x}$ (resp. $S_{i,x}$) is ready immediately after $r_{k,x}$ (resp. $r_{i,x}$) due to the definition of server job readiness. Since $S_{k,x}$ and $S_{i,x}$ are ready at the same time (due to having the same release time by Defs. 2 and 3), and $S_{k,x}$ has higher priority (as $\tau_k \in HPS(\tau_i)$), the lemma is true for $S_{k,x}$ and $S_{i,x}$ by Prop. 5. For the inductive step, we show that if the lemma is true for some job index $j-\rho$, it is true for job index j. Since the lemma is true for $S_{k,j-\rho}$ and $S_{i,j-\rho}$, when $S_{i,j-\rho}$ completes (scheduled for C_i time), $S_{k,j-\rho}$ has been scheduled for at least $\min(C_i, C_k)$ time units. Due to Prop. 8, $\min(C_i, C_k) = C_k$, implying that $S_{k,j-\rho}$ is also complete. Since $S_{i,j}$ and $S_{k,j}$ have the same release, and $S_{k,j-\rho}$ is complete when $S_{i,j-\rho}$ is complete, then by the definition of server job readiness, $S_{k,j}$ is ready when $S_{i,j}$ is ready. Since $S_{k,j}$ also has higher priority than $S_{i,j}$, the lemma holds for $S_{i,j}$ and $S_{k,j}$ by Prop. 5. $\square$

While each server job $S_{k,j}$ is scheduled, it can execute overrunning jobs of nodes in its helping set by Rule R3.2. We can upper-bound the time each of these overrunning jobs can make progress before their completion.

Lemma 10. *After each server job $S_{i,j}$ completes, $J_{i,j}$ makes progress on server jobs for at most $\max(0, \delta_{i,j} - C_i)$ time units before its completion.*

Proof. By Defs. 7 and 8, $\delta_{i,j}$ gives an upper bound on the time $J_{i,j}$ makes progress on $S_{i,j}$ and servers released after $S_{i,j}$'s completion. As a result, if $\delta_{i,j} < C_i$, $J_{i,j}$ completes on $S_{i,j}$ due to Lem. 1, and can make progress for at most $0 = \max(0, \delta_{i,j} - C_i)$ time units after $S_{i,j}$'s completion. On the other hand, if $\delta_{i,j} \geq C_i$, $J_{i,j}$ makes progress on $S_{i,j}$ for C_i time units, and thus can make progress for at most $\max(0, \delta_{i,j} - C_i)$ time units after $S_{i,j}$'s completion. $\square$

Lem. 10 then implies the following.

Lemma 11. *After $S_{i,j}$ becomes ready, jobs in $\{J_{x,j} \mid \tau_x \in \mathbb{H}_i\}$ make progress on server jobs for at most a total of $\sum_{\tau_x \in \mathbb{H}_i} \max(0, \delta_{x,j} - C_x)$ time units before they complete.*

Proof. By Prop. 1, when $S_{i,j}$ is ready (and hence released), all server jobs in $\{S_{x,j} \mid \tau_x \in \mathit{pred}(\tau_i)\}$ are complete. Since Prop. 6 implies $\mathbb{H}_i \subseteq \mathit{pred}(\tau_i)$, when $S_{i,j}$ is ready, all server jobs in $\{S_{x,j} \mid \tau_x \in \mathbb{H}_i\}$ are complete. Thus, by Lem. 10, each job in $\{J_{x,j} \mid \tau_x \in \mathbb{H}_i\}$ can make progress on server jobs for at most $\max(0, \delta_{x,j} - C_x)$ time units after $S_{i,j}$ is ready. $\square$

We denote the end of the first phase as t_1. This occurs when $S_{i,j}$ has been scheduled for $\Phi^1_{i,j}$ time units, as defined next.

Definition 13. *For each job $J_{i,j}$, let the RV $\Phi^1_{i,j} = \max(\{\min(C_k, O_{k,j}) \mid \tau_k \in \mathit{HPS}(\tau_i)\})$ where*

$$O_{k,j} = \sum_{\tau_x \in \mathbb{H}_k} \max(0, \delta_{x,j} - C_x). \tag{2}$$

Note that due to Prop. 8, $\Phi^1_{i,j} \leq C_i$ by Def. 13. This implies that t_1 is at or before $S_{i,j}$'s completion. Since $S_{i,j}$ is scheduled for $\Phi^1_{i,j}$ time units between t_0 (when it became ready) and t_1, we have the following due to Lem. 1.

Corollary 2. *Between t_0 and t_1, $J_{i,j}$ makes progress on $S_{i,j}$ for $\Phi^1_{i,j}$ time units.*

Due to the parallel execution of server jobs for nodes in helping sets, we can infer the following after t_1.

Lemma 12. *After t_1, for each $\tau_k \in \mathit{HPS}(\tau_i)$, jobs in $\{J_{x,j} \mid \tau_x \in \mathbb{H}_k\}$ make progress for at most a total of $\max(0, O_{k,j} - C_k)$ time units before they complete.*

Proof. Since $S_{i,j}$ is scheduled for $\Phi^1_{i,j}$ time units by t_1, Lem. 9 implies that $S_{k,j}$ for each $\tau_k \in \mathit{HPS}(\tau_i)$ is scheduled for at least $\min(\Phi^1_{i,j}, C_k)$ time units by t_1. From Def. 13, $\Phi^1_{i,j} \geq \min(C_k, O_{k,j})$, implying that $\min(\Phi^1_{i,j}, C_k) \geq \min(C_k, O_{k,j})$ for each $\tau_k \in \mathit{HPS}(\tau_i)$. This implies that each $S_{k,j}$ is scheduled for at least $\min(C_k, O_{k,j})$ time units by t_1. By Prop. 7, this implies that for each $S_{k,j}$, jobs in $\{J_{x,j} \mid \tau_x \in \mathbb{H}_k\}$ have made progress for at least $\min(C_k, O_{k,j})$ time units on $S_{k,j}$ by t_1. Since Lem. 11 states that jobs in $\{J_{x,j} \mid \tau_x \in \mathbb{H}_k\}$ can make progress for at most a total of $O_{k,j}$ time units before they complete, the lemma follows because $O_{k,j} - \min(C_k, O_{k,j}) = \max(0, O_{k,j} - C_k)$. $\square$

Second phase. The second phase begins at t_1, and ends when $J_{i,j}$ is ready. To see how much progress $J_{i,j}$ can make on server jobs in this phase, we examine how much progress jobs in $\mathit{dep}(J_{i,j})$ can make before $J_{i,j}$ becomes ready.

Definition 14. *For each job $J_{i,j}$, let the RV*

$$\Phi^2_{i,j} = \max(0, \delta_{i,j-\rho} - C_i) + \sum_{\tau_k \in \mathit{HPS}(\tau_i)} \max(0, O_{k,j} - C_i)$$
$$+ \sum_{\tau_x \in \mathit{pred}(\tau_i) - \bigcup_{\tau_k \in \mathit{HPS}(\tau_i)} \mathbb{H}_k} \max(0, \delta_{x,j} - C_x) \tag{3}$$

where each $O_{k,j}$ is given by (2) and $\delta_{i,j'} = C_i$ where $j' < 1$.

Lemma 13. *After t_1, jobs in $\{J_{x,j} \mid \tau_x \in \mathit{pred}(\tau_i)\} \cup \{J_{i,j-\rho}\}$ can make progress on server jobs for at most a total of $\Phi^2_{i,j}$ time units before they complete.*

Proof. For brevity, we let $X = \bigcup_{\tau_k \in \mathit{HPS}(\tau_i)} \mathbb{H}_k$. Since $S_{i,j}$ is ready at t_0, by the definition of server job readiness and Prop. 1, server jobs in $\{S_{x,j} \mid \tau_x \in \mathit{pred}(\tau_i)\} \cup \{S_{i,j-\rho}\}$ are complete at t_0, implying that these server jobs are also complete at t_1. Due to Prop. 6, we can partition $\{S_{x,j} \mid \tau_x \in \mathit{pred}(\tau_i)\} \cup \{S_{i,j-\rho}\}$ into three sets of server jobs that are complete at t_1: $\{S_{i,j-\rho}\}$, $\{S_{x,j} \mid \tau_x \in X\}$, and $\{S_{x,j} \mid \tau_x \in \mathit{pred}(\tau_i) - X\}$. By Lem. 10, the first term in (3) upper-bounds the amount of progress $J_{i,j-\rho}$ (if it exists) makes on server jobs after t_1. Similarly, by Lem. 10, the third term in (3) upper-bounds the amount of progress jobs in $\{J_{x,j} \mid \tau_x \in \mathit{pred}(\tau_i) - X\}$ make on server jobs after t_1. Finally, by Lem. 12, the second term in (3) upper-bounds the amount of progress jobs in $\{J_{x,j} \mid \tau_x \in X\}$ make on server jobs after t_1. $\square$

Lem. 13 allows us to upper-bound the progress $J_{i,j}$ makes during the second phase.

Lemma 14. *Between t_1 and t_2, $J_{i,j}$ make progress on server jobs for at most $\Phi^2_{i,j}$ time units.*

Proof. For brevity, let $X = \{J_{x,j} \mid \tau_x \in \mathit{pred}(\tau_i)\} \cup \{J_{i,j-\rho}\}$. If $J_{i,j}$ makes progress before t_2, then Def. 1 implies that some job $J \in \mathit{dep}(J_{i,j})$ is scheduled. This implies that J must complete before $J_{i,j}$ is ready, which by the definition of job readiness and the Job Release Rule means that $J \in \mathit{dep}(J_{k,\ell}) \cup \{J_{k,\ell}\}$ for some $J_{k,\ell} \in X$. Therefore, by Def. 1 $J_{i,j}$ can only make progress when a job in X makes progress. Since jobs in X can make progress for at most $\Phi^2_{i,j}$ time units after t_1 due to Lem. 13, the lemma follows. $\square$

Upper-bounding demand. After t_2, $J_{i,j}$ is ready, and therefore can make progress for at most its pWCET, $e_{i,j}$ time units, before it completes. This, combined with the progress that $J_{i,j}$ made through the first two phases gives us the following.

Lemma 15. *For each job $J_{i,j}$, $\delta_{i,j} = \Phi^1_{i,j} + \Phi^2_{i,j} + e_{i,j}$.*

Proof. From Cor. 2 $J_{i,j}$ makes progress for at most $\Phi^1_{i,j}$ time units on $S_{i,j}$. Also, due to Lem. 14, $J_{i,j}$ makes progress for at most $\Phi^2_{i,j}$ time units before $J_{i,j}$ becomes ready at t_2. After $J_{i,j}$ is ready, it can make progress for at most $e_{i,j}$ time units before it completes. The lemma follows by Defs. 7 and 8. □

C. Combining

Lem. 15 allows us to compute an upper-bound on the demand of each job $J_{i,j}$ ($\delta_{i,j}$) in the general case. Lem. 8 also allows us to compute a tighter upper-bound on $\delta_{i,j}$ if $\Psi_{i,j} > 0$. We can utilize both lemmas to form a single equation for $\delta_{i,j}$.

Theorem 2. *For each job $J_{i,j}$, $\delta_{i,j} = \Delta_{i,j} + e_{i,j}$ where*

$$\Delta_{i,j} = \begin{cases} -\Psi_{i,j} & \text{if } \Psi_{i,j} > 0 \\ \Phi^1_{i,j} + \Phi^2_{i,j} & \text{otherwise.} \end{cases} \tag{4}$$

Proof. Follows directly from Lems. 8 and 15. □

Observe from Defs. 12–14 that for a job $J_{i,j}$, $\Psi_{i,j}$ and $\Phi_{i,j}$ are functions of $\delta_{i,j-\rho}$ and $\delta_{k,j}$ for each $\tau_k \in pred(\tau_i)$. As a result, Thm. 2 forms a recurrence relation where $\delta_{i,j}$, can be computed from $\delta_{i,j-\rho}$ and each $\delta_{k,j}$. Therefore, the demand upper bound of each node can be computed using the recurrence relation in Eq. 4.

Computing demand upper bound for large job indices. While it is possible to compute $\delta_{i,j}$ using the recurrence relation in Eq. 4, this method can become computationally intensive for large job indices. Fortunately, we can shorten the recursive depth required to compute $\delta_{i,j}$ using the following.

Lemma 16. *If the overrun of a job $J_{i,j}$ results in a graph invocation abort due to Rules R4.1 or R4.2, then $\delta_{i,j} = C_i$.*

Proof. If the overrun of $J_{i,j}$ causes a graph-invocation abort, then $J_{i,j}$ cannot make progress on server jobs after $S_{i,j}$ completes. Therefore, $J_{i,j}$ can only make progress on $S_{i,j}$ for at most C_i time units. The lemma follows by Def. 8. □

Computing the probability distribution of $\delta_{i,j}$. Notice from Thm. 2 and Defs. 12–14 that the recurrence relation in (4) consists of a sequence of operations on RVs. These operations modify the probability distribution functions (PDFs) of RVs according to known rules [6], [7]. Thus, by applying these rules, we can compute the PDF of $\delta_{i,j}$.

Theorem 3. *Let A be the set of jobs whose overrun can cause the j^{th} invocation of G to abort due to Rules R4.1 and R4.2. The abort probability of the j^{th} invocation of G is at most*

$$\sum_{J_{i,j} \in A} P(\delta_{i,j} > C_i).$$

Proof. This theorem follows from the fact that the j^{th} graph invocation is aborted when a job in A overruns its budget. □

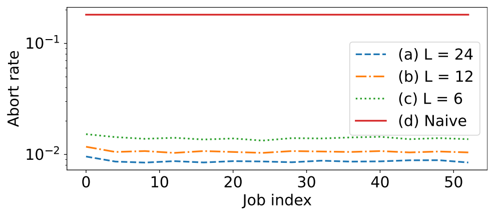

Fig. 6: Plot of the maximum analytical abort-rate upper bound of 1,000 random DAGs across 50 job invocations. Four scenarios are shown, (a) overrun cascade lengths are limited to $\mathcal{L} = 24$, (b) $\mathcal{L} = 12$, (c) $\mathcal{L} = 6$, and (d) each node's budget is strictly enforced.

V. Experimental Evaluation

We conducted experiments to demonstrate two attributes of our budget-enforcement method. First, our method ensures that the abort rate of the j^{th} invocation of a graph does not increase unboundedly as j^{th} increases. Second, our method produces a lower graph abort-rate upper bound than the naïve method where all node budgets are strictly enforced. We do not compare our method against that in [7] as it does not support graphs with limited preemption. We demonstrate these attributes by evaluating the maximum analytical abort-rate upper bound of 1,000 randomly generated DAGs for the first 50 job invocations. Each generated DAG has 200 nodes and is assigned a random value of ρ between 1 and 4.

DAG generation. To generate an n-node DAG, we first generated nodes with the following parameters. The pWCET of each job follows a type-1 Gumbel distribution with a mean of 5 ms and a standard deviation of 2 ms. (The Gumbel distribution is often used to represent measurement-based pWCETs [4].) We then set each node's budget as the 99.9^{th} percentile value of our Gumbel distribution. Since our focus is DAG abort rate and not schedulability, we set the period T to a large value of $50n$ ms so that the DAG is trivially schedulable. We then generated the structure of each DAG using a modified version of the *Erdös-Rényi method* [3] used in [7].

Fig. 6 gives our evaluation results. We see that our budget management method is successful in preventing an unbounded increase in graph abort-rates across invocations. Additionally, our method gives an analytical abort-rate upper bound that is an order of magnitude improvement over the naive method.

VI. Conclusion

We have presented a budget-management method for DAGs that supports restricted parallelism. Our policy limits the length of overrun cascades by strictly enforcing node budgets in a cyclic fashion, and limits increases in abort probabilities across graph invocations by allowing overrunning jobs to be scheduled in parallel. We also presented a probabilistic abort-rate analysis of our method. Finally, we presented experiments that demonstrate that our method does indeed limit graph abort-rate increases across graph invocations, and is superior to the naïve approach of strict per-node budget enforcement.

In future work, we plan to introduce locking-protocol usage by graph nodes that access shared resources. We also plan to evaluate implementation tradeoffs by implementing our method within an actual real-time OS.

References

[1] S. Ahmed and J. Anderson, "Exact response-time bounds of periodic DAG tasks under server-based global scheduling," in *RTSS*, 2022, pp. 447–459.

[2] T. Amert, S. Voronov, and J. Anderson, "OpenVX and real-time certification: The troublesome history," in *RTSS*, 2019, pp. 312–325.

[3] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *SIMUTools*, 2010.

[4] R. Davis and L. Cucu-Grosjean, "A survey of probabilistic timing analysis techniques for real-time systems," *Leibniz Trans. Embedded Syst.*, vol. 6, no. 1, pp. 03:1–03:60, 2019.

[5] C. Liu and J. Anderson, "Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss," in *RTSS*, 2010, pp. 3–13.

[6] F. Marković, P. Roux, S. Bozhko, A. Papadopoulos, and B. Brandenburg, "Cta: A correlation-tolerant analysis of the deadline-failure probability of dependent tasks," in *RTSS*, 12 2023, pp. 317–330.

[7] Z. Tong, S. Ahmend, and J. Anderson, "Holistically budgeting processing graphs," in *RTSS*, 2023, pp. 27–39.

[8] R. Wilhelm, "Real time spent on real time," in *RTSS*, 2020, pp. 1–2.