Impact of Non-stationary Workload on Resource Reservation Based Slack Reclamation

Technical Report CU-CS-1053-09
May 2009

Wang-ting Lin and Gary Nutt
Department of Computer Science
University of Colorado
Boulder, Colorado 80309
Email: {linwt, nutt}@colorado.edu

***Abstract*—In open real-time systems with a non-stationary workload, tasks can be dynamically mapped to servers. If a server-based, slack reclamation scheduler is used, each task reserves a fraction of the bandwidth based on its resource estimation. However, the slack time scheduler makes no guarantee to a task beyond its processor capacity reserve, i.e., the scheduler cannot guarantee that a soft real-time task will receive a higher quality of service than is specified by its resource reservation. Nevertheless users often assume that a desired *quality of service* (QoS) can be achieved using an optimistic resource reservation. Even though existing slack time schedulers perform well in a closed real-time system (with well-behaved applications, stationary workloads, or excessive processor speed), their performance can drop significantly when a task changes its workload. We identify resource underbooking and overbooking problems in non-stationary workloads, resulting in a *denial of service* (DoS) phenomenon. We solve these problems by preventing DoS attack on the slack time scheduler; we also allow any server to donate slack time even when it is in an idle state. Finally, we introduce a forward donation algorithm that helps existing slack time schedulers reclaim extra slack time. By increasing the robustness of the scheduler, it can be used with broader spectrum of applications in an open real-time system.**

I. Introduction

Our work focuses on systems such as cyber phyiscal systems (CPS) that support a hybrid workload composed of hard real-time (HRT), soft real-time (SRT) and best effort (BE) tasks. When a CPS relies on resource adaptive computation, maps a BE task to its own server, or obtains SRT applications from various vendors, its workload rarely remains stationary. Reserving a fixed, periodic budget to process non-stationary workloads imposes new challenges to the secondary slack time schedulers.

In this paper we evaluate the performance of a non-stationary workload by viewing the workload in phases. When each task has a stationary workload at the same interval, the system workload reaches the stationary state and the slack time supply and demand reach an equilibrium state which supports a constant level of QoS. By using the same statistical characteristics of each workload in the short stationary timeframe, a longer trace of workload with the same characteristics can be generated for performance evaluation. If any task changes its workload, the system enters a new stationary state providing new QoS levels.

When a task overbooks resources, i.e. submits a lighter workload, its less intense overruns lead to a low deadline miss ratio (DMR) and a high QoS. When the slack time scheduler produces a relatively high QoS system because of a light system workload, it is difficult to identify any improvement to the scheduler. I.e., scheduling slack time more effectively and generating more slack time does not increase the QoS of the system much when ample slack time is generated or tasks are not asking for slack time.

We observe that when a non-stationary task changes its workload by overloading its server, it does not change the resource reservation of the other tasks but it may reduce the QoS of all SRT tasks in the system. Since the slack time scheduler tries to help SRT tasks beyond their QoS resource reservations, such schedulers encourage SRT tasks to underbook their resources in order to allow more tasks to be admitted to the system. When a non-stationary task overloads its server, it breaks the equilibrium state and decreases the QoS of the system. We argue that any excess slack request due to a significant resource underbooking should be controlled. If our system detects that a task underbooks, it will not allocate slack time to the task (although it still assures the reserve budget) whose performance is excluded from system performance. Otherwise, the performance comparison will be skewed.

Our experiments demonstrate cases where the previous slack reclamation schedulers suffer from significant performance drop due to a resource underbooking task; our *DoS prevention* algorithm mitigates the performance degradation. We also show that our *forward donation* algorithm reclaims additional slack time compared to the existing slack reclamation algorithm. As a result, when a task overbooks its resource, forward donation can bring the system into an equilibrium state with an increase in system QoS. The contribution of this paper is to allow slack time algorithms to schedule non-stationary workloads with a better performance by controlling the damage caused by any task that exploits the fixed priority

This is a longer version of a paper submitted for publication.

vulnerability of the slack scheduler. By doing so, the scheduler can be used in a broader range of systems.

II. Related Works

Mercer et al. used the computation rate ρ, expressed as computation time C per duration of time T, to specify the utilization, i.e. processor capacity reserve, instead of $\frac{WCET}{P}$ [1]. Each multimedia task specifies a range of computation rates and a target rate, and the scheduler can vary its period duration and/or its computation time as long as it maintains the same utilization. As a result, the constraint of using WCET can be removed. When a continuous media job overruns, its task must wait until its next period for the budget replenishment.

In Deadline Deferrable Server (DSS), after a server's budget is exhausted it becomes idle until its deadline for replenishment [2]. The bandwidth reservation of the Constant Bandwidth Server (CBS) is similar to DSS but the budget can be immediately replenished when it is exhausted [3]. Without waiting for the budget replenishment like DSS, a CBS server is more likely to meet the deadlines of the overrun jobs than DSS. The resource isolation property allows CBS to deal with both periodic HRT and aperiodic tasks in the same system. CBS is similar to Total Bandwidth Server (TBS) except TBS requires the exact knowledge of WCET for the correct behavior [4].

The Idle-time Reclaiming Improved Server(IRIS) [5] improved CBS to allow the spare bandwidth to be distributed more fairly among the SRT servers when the system is overloaded. Best-effort Bandwidth Server (BEBS) supports best-effort tasks by processing them on an aperiodic server [6]. It assigns priority to the legacy best-effort applications based on run-time behavior of the tasks to make them more responsive. BEBS assigns different period and utilization parameters to the best-effort tasks to transform them into periodic tasks based on whether they are I/O or computation bound tasks. Whenever a slack time is generated, it will be used by the any task that wants to use it. Greedy Reclamation of Unused Bandwidth (GRUB) also dynamically allocates excess capacity to the needy servers in direct proportion to their reservations while maintaining a lower context switch count than CBS [7]. Since it requires very fine granularity of time while doing the computation, the algorithm itself has large overhead.

HisReWri allocates spare CPU capacity (gain time) from HRT and SRT tasks to other SRT tasks by monitoring the past execution history and retroactively allocating the gain time to tasks that executed using the fixed priority RM algorithm [8].

The BandWidth Inheritance (BWI) algorithm allows lower priority tasks to inherit the priority from a higher priority task which is blocked [9]. The CFA algorithm improves BWI by monitoring the slack usage and allows the stolen slack time to be recovered [10].

CASH extends CBS to allow slack time to be reclaimed into the CASH queue where slack time are ordered by their deadlines [11]. When an earliest deadline server is picked to run, its deadline will be compared to the deadline of the slack time in the CASH queue. The budget with the earliest deadline will then be used first. The disadvantage is that a task which needs more time to complete will have its deadline extended before it completes and the task having slack time in the CASH queue cannot release its next job until the slack time is consumed. The Bandwidth Sharing (BASH) improved CASH by reclaiming the slack time into the global BASH queue [12]. The slack can be donated to other tasks as early as possible, e.g., to the job with the earliest deadline. The slack time in the BASH queue is more resilient against idle intervals and is independent of an idle interval length.

Except for the CBS style budget borrowing, BACKSLASH adopts the back donation concept designed by HisReWri but it integrates the back donation with EDF instead of RM [13]. BACKSLASH is similar in concept to CFA, but unlike CFA it allows a SRT task to use a value lower than its WCET, e.g. the mean execution time, for its resource reservation.

The four **slack competition priority categories** of BACKSLASH, in a **descending priority ordering**, are briefly reviewed as follows:

1) Donate to a server that has borrowed from its future budget to finish its previous job and has the earliest **virtual** (original) deadline in the back donation queue.
2) Donate to a server that has borrowed from its future budget to process its current job and has the earliest **virtual** (original) deadline.
3) If no server has borrowed from its future budget, donate to a server that has the earliest deadline.
4) If no server is ready to run, donate to the idle task.

Caixue Lin designed SMASH that simultaneously supports the hybrid workload by using Rate-Based Earliest Deadline (RBED) [14] enhanced through Taxed-based Resource Allocation Policy (TRAP) and EDF-based scheduling enhanced through SMASH slack reclamation [15]. SMASH improves the performance of BACKSLASH by using slack preservation algorithm of BASH. For easier discussion, the irrelevant modules like the BASH slack preservation algorithm, TRAP, and the Rate-Based mapping of the best-effort tasks are separated from SMASH.

III. DoS Attack to the Slack Scheduler

We call the preemptive earliest deadline first (EDF) algorithm safeguarded by an admission manager the **primary scheduling policy**, which guarantees *resource isolation* between servers [16]. In order to conserve the server budget for jobs that arrive later than the periodic server activation points, both BASH and BACKSLASH change the periodic behavior of the primary scheduler causing the server period to drift. They allow *server activation* to be delayed to the next *job release* in order to provide a full budget to the job.

If a server's reserved budget is not used by its job, slack time is generated and it can be donated to other servers according to a **secondary, slack, scheduling policy** that is independent of the primary scheduling policy. The secondary policy is activated only when slack time is generated by the primary policy, hence it cannot change the feasible EDF schedule.

Using a self-reserved future budget changes the deadline and the borrowing state of a server, so it can influence the

decisions of the secondary scheduler. But, consuming slack time (donated by peers) does not have the same side-effect. Therefore, we distinguish budget borrowing from slack time scheduling policy by identifying the former as the **pacing scheduling policy**. The pacing policy allocates extra budget to a server when the primary scheduler finds that it does not have enough budget to finish its computation and the secondary policy fails to provide it enough slack time from other servers.

There are two root causes of the DoS problem that can occur in BACKSLASH. First, a server may constantly show the need for slack time because its task underbooks the resource. Second, when a server borrows from its future budget the slack scheduler boosts its priority into a higher category. A task can force its server into a higher priority state indefinitely by submitting large jobs; we call this the **fixed priority problem**. A server in a lower category always wins the slack time competition against all servers in the higher categories. In our earlier work in progress paper, we identified DoS attacks [17]. We refine those ideas by defining three levels of DoS attacks on BACKSLASH in a descending priority ordering as follows:

Definition 1 (First Level DoS Attack)**.** *This attack is launched by a server that enters into the BACKSLASH back donation queue after it borrows from its future budget to complete a job. It takes advantage of the first priority category and has the strongest attacking power over the other slack reclamation principles.*

Definition 2 (Second Level DoS Attack)**.** *The second level attack is waged by a server whose current job requires enormous service time and the server continuously borrows from its future budget in an attempt to complete the job. Even though the second level DoS attack is not as strong as the first level, it is the easiest to launch and it is very effective.*

Definition 3. *[Third Level DoS Attack] The third level attack is waged by a server that tries to maximize its opportunities to obtain slack donation in the third priority category.*

A. First Level DoS Attack Example

Suppose three tasks, T_1, T_2, and T_3, are submitted at time $t = 0$ and three servers, S_1, S_2, and S_3, with $u_1 = \frac{8}{48}$, $u_2 = \frac{33}{99}$, and $u_3 = \frac{5}{10}$ resource reservations are created to represent the tasks respectively in figure 1. All jobs are released at the beginning of each server period except that the second job of T_1, $job_{1,2}$, is released at time $r_{1,2} = 85$. The first job of T_1, $job_{1,1}$, overruns which needs 16 time units to complete, so it causes the first level DoS attack to BACKSLASH. The first job of T_2 needs 35 units to complete, so $job_{2,1}$ relies on 2 units of slack time to meet its job deadline at $d_{2,1} = 99$. Each of the first four jobs released by T_3 requires all 5 units of time to complete, but each of the following jobs, from $job_{3,5}$ to $job_{3,9}$, requires only 4 units of reservation to complete.

Since S_3 has the earliest server deadline $d_{S_3} = 10$, it executes $job_{3,1}$ first. After $job_{3,1}$ is completed, S_1 executes $job_{1,1}$ for 5 time units until it is preempted by the release of $job_{3,2}$ on S_3. When $job_{3,2}$ completes at 15, S_3 is set to the idle

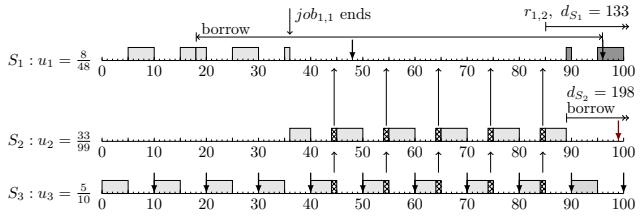


Fig. 1. DoS by back donation

state. Since $d_{S_1} = 48 < d_{S_2} = 99$, S_1 executes $job_{1,1}$ until its budget is depleted at 18. Because $job_{1,1}$ is not completed, S_1 borrows from its future budget with d_{S_1} extended to 96. The primary scheduler executes as usual until $job_{1,1}$ is completed at $t = 36$. S_1 enters the back donation queue because it has borrowed from its future budget to complete $job_{1,1}$. After S_2 executes $job_{2,1}$ from 36 to 40, $job_{3,5}$ arrives to S_3 at 40 and it requires only 4 out of 5 units of budget to complete. When S_3 generates one unit slack time at 44, the secondary scheduler kicks in, finds S_1 in the back donation queue and initiates the back donation process. From 44 to 45, both S_2 and S_3 spend one unit of budget each to run $job_{2,1}$ for one unit, so the other unit of time becomes slack time for back donation to S_1. Similar back donation continues until $job_{1,2}$ arrives at $t = 85$. The cross-hatched blocks mark the slack back donations. The release of $job_{1,2}$ at 85 activates S_1 server with 8 units of budget and the server deadline is set to $d_{S_1} = 133$. After S_2 executes $job_{2,1}$ from 85 to 89 and depletes its budget, the pacing policy allows it to borrow from its future budget and extends its server deadline d_{S_2} to 198. Since $d_{S_1} = 133 < d_{S_2} = 198$, $job_{1,2}$ runs from 89 to 90 and is preempted by the release of $job_{3,10}$ at 90. After $job_{3,10}$ completes at 95, $job_{1,2}$ runs until 100 causing $job_{2,1}$ to miss its deadline $d_{2,1}$ at 99. Please note that the server deadline $d_{S_2} = 198$ is different from $job_{2,1}$'s deadline $d_{2,1} = 99$.

Even though S_1 has no job to run from $t = 36$ to $t = 85$, it still consumes slack time via back donation. If the back donation does not occur, S_2 could have obtained two units of slack time from S_1 to complete $job_{2,1}$ before $d_{2,1} = 99$.

B. Second Level DoS Attack Example

Figure 2 is an example of a second level DoS attack on the secondary scheduler of BACKSLASH where an enormous job places its server in the second priority category for an enormous duration. Suppose three tasks, T_1, T_2, and T_3, are submitted at time $t = 0$ and three servers, S_1, S_2, and S_3, with $\frac{1}{10}$, $\frac{15}{50}$, and $\frac{60}{100}$ resource reservations are created to represent the tasks respectively. Further suppose that the DoS attacker T_1 requires 50 time units to complete each job. It may be triggered by a programming error, a resource estimation error, or a malicious client. Each job released by T_2 requires 10 units to complete, so it is a slack generator. $Job_{3,1}$ slightly overruns by 2 units and requires 62 time units to complete.

Since $d_{S_1} = 10$ is the earliest deadline, S_1 runs first until it is preempted at time $t = 1$ where its reserved budget is

depleted. S_1 borrows 1 unit of its future budget and extends its server deadline d_{S_1} to 20, so its original deadline $d_{S_1}^{orig} = 10$ will be used for the EVDF (earliest virtual deadline first) slack competition. Since $d_{S_1} = 20 < d_{S_2} = 50 < d_{S_3} = 100$, S_1 uses its future budget from 1 to 2. The budget is depleted and refilled as usual until $t = 5$ where S_1's server deadline d_{S_1} is extended to 60. Since $d_{S_2} = 50 < d_{S_1} = 60 < d_{S_3} = 100$, S_2 executes $job_{2,1}$ from 5 to 15. When $job_{2,1}$ is completed at 15, S_2 generates 5 units of slack time and is set to the idle state. The cross-hatched blocks represent the slack donations. S_1 wins the slack donation because it has the earliest original deadline $d_{S_1}^{orig} = 10$ and there is no server in the back donation queue to compete with it. Thus, S_1 runs from $t = 15$ to 20 using the slack donation. Similarly, S_1 dominates the slack time from 60 to 65.

At $t = 100$, S_3 has executed $job_{3,1}$ for 60 time units using its reserved budget. Even though 10 units of slack time is generated, S_3 does not receive any slack time because of the DoS attack launched by S_1. If the secondary scheduler can prevent S_1 from stealing the whole 10 units of slack time and give 2 units to S_3, S_3 could finish $job_{3,1}$ by its deadline at 100. BACKSLASH can boost the priority of $job_{3,2}$ in obtaining slack time through back donation after $job_{3,1}$ completes, but the help comes too late for $job_{3,1}$.

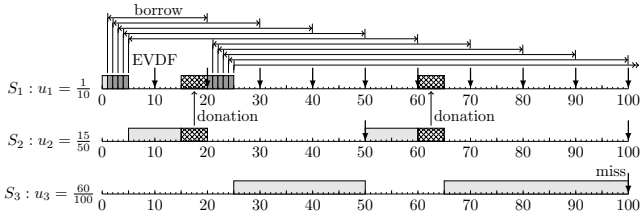


Fig. 2. Second level DoS before back donation takes place

Once a server borrows from its future budget, it can stay in the second priority category as long as its job does not end. Its virtual deadline remains at the fixed point and is very likely to become the earliest if the server stays in the borrowing state longer. Once its original deadline becomes the earliest, it only loses the slack time competition to the servers in the back donation queue. If a second level attacker finishes its job, it enters into the back donation queue which gives it the highest first level DoS attacking power.

C. Third Level DoS Attack Example

By reserving a relatively small period, a server's deadline can be relatively close to the time when the secondary scheduler kicks in. If it has the earliest deadline and there are no servers in the first two slack competition priority categories, it obtains the slack donation. It is difficult for a server to stay at the pure third DoS attack level without entering the higher priority levels, i.e. use more than the reserved budget without borrowing. S_2 of figure 3 represents an example of the third level DoS attacker.

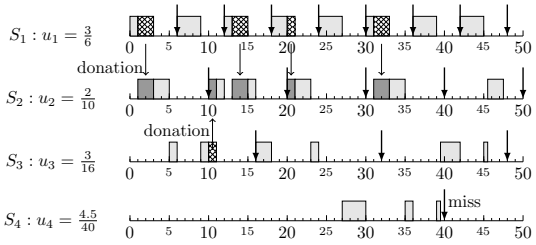


Fig. 3. Third level DoS before back donation takes place

Even though this level of attack is not as strong as the first two levels, this more subtle form of attack is more difficult to identify and can remain stealthy. It is not easy to manipulate the jobs to grab as much slack time as possible without ending up borrowing from the task's own future budget, i.e. remaining as a pure third DoS level attacker. When a server borrows from its future budget, its attacking power is upgraded to the first and second level causing a mixed attack.

For simplicity, our example focuses only on the pure third level DoS attack in figure 3 by carefully selecting service times. Suppose four tasks, labeled T_1, T_2, T_3, and T_4, are submitted at time $t = 0$ and four servers, S_1, S_2, S_3, and S_4, with $\frac{3}{6}$, $\frac{2}{10}$, $\frac{3}{16}$, and $\frac{4.5}{40}$ resource reservations are created to represent the tasks respectively. Further suppose that $job_{1,1}$, $job_{1,3}$, $job_{1,4}$, and $job_{1,5}$ donate 2, 2, 1, and 2 units of slack time respectively, and $job_{3,1}$ generates 1 unit of slack time. To illustrate the pure third level DoS attack, T_2 has a service time pattern as 4, 5, 3, 4, and 3 units of time for $job_{2,1}$ to $job_{2,5}$. A 5 units of service time means a 150% overrun. T_4 reserves 4.5 units of budget for $job_{4,1}$, but it requires 5 units of time to complete. Since we already discuss this in our WiP paper, we omit the detailed execution description of this example.

S_2 is a *pure* third level attacker that steals slack time from tasks with both shorter and longer periods, like S_1 and S_3. The cross-hatched blocks mark the slack donations, and the dark gray blocks mark the slack consumptions. If S_4 needs 5 units to complete its job before 40, it needs only 0.5 units of slack time. Even though there are 8 units of slack time generated before 40, S_4 does not get any slack time because of the third level DoS attack launched by S_2.

D. DoS Prevention

We repeat the definition of estimation error from [17]:

Definition 4 (Estimation Error (EE))**.**

$$EE = \frac{real\ service\ time\ of\ a\ job}{estimated\ service\ time\ of\ a\ job} - 1$$

where "estimated service time" is used to reserve each task's budget per period.

A zero EE implies that the task has made a perfect service time estimation. A positive EE indicates that the task is underbooking the processor for this job, and thus is a potential DoS attacker. A negative EE implies that the task is overbooking processor time for this job.

A simple and effective solution to the slack scheduler DoS attack is to prevent any job whose EE exceeds a threshold from receiving slack time. During our experiments, we find that a lookahead evaluation prevents a job from continuously using the slack time after its EE exceeds the threshold. For example, if a job's EE exceeds the threshold after consuming 2 out of 10 units of slack time, a one-shot timer must be set at 2 units from the beginning of the donation, otherwise the job can consume all 10 units of slack time if there is no other event that triggers the scheduler and preempts its execution. The one-shot timer provides all other tasks a fair opportunity to be reevaluated for the remaining 8 units of slack time. If other events occurs before the timer expires, the superfluous timer will be canceled.

We impose the following algorithm on the four slack competition priority categories of BACKSLASH in order to prevent the DoS attacks on its secondary scheduler.

1) For the first priority category, there is no job associated with a server in the back donation queue, so EE is calculated from a server's previously completed job. The slack time received through the back donation is counted as the additional execution time of the previous job. Find a server, whose EE is lower than the system specified threshold, according to the EVDF ordering using the first principle to receive the slack donation. If there is no qualified server, try to find a server using the second priority category.
2) For the second priority category, find a server whose EE is lower than the specified threshold, according to the EVDF ordering to receive the slack donation. If there is no qualified server, try to find a server using the third priority category.
3) For the third priority category, find a server, whose EE is lower than the specified threshold, according to the earliest deadline first ordering using the third category to receive the slack donation.
4) If there is no qualified server in the above priority categories, try to find the earliest deadline first server which can be an idle server if there is no other waiting server in the system.

IV. Forward Donation

Definition 5 (Earliest Activation Time (EAT))**.** *For a server S_i in the idle state, its EAT is defined as the earliest time for the server's **lag** — defined in RBED paper [14] — to become zero or positive. If a server has a zero or positive lag, it can be immediately assigned its full budget ($c_{S_i} = R_i$ with deadline one period away) and moved to the waiting state, where c_{S_i} is the server budget and R_i is its resource reservation.*

If a server, S_i, is activated at its EAT by a timer and it has no job to execute, its budget will be used to help other servers as slack time. It is not a problem if the next job, which requires R_i units of budget, is released on S_i when the server's budget is replenished back to its full bandwidth. However, any job which is released on S_i and requires more than the residual budget of S_i after its budget is partially consumed by other servers will miss the job deadline if there is no slack time to help the job. Both BASH and BACKSLASH avoid the problem by activating a server from the idle state when its job is released.

Activating a server by using its job release solves the problem of wasting budget, but it introduces two other problems. First, when a server S_i receives a lighter workload than its reservation and it is not activated, its budget cannot be donated from the idle state. Second, the server may have consumed slack back donation that moved its EAT backward to EAT' but $job_{i,j+1}$ does not arrive before EAT' which prevents the slack time from being used to help other servers. Thus, using a job release to activate its server prevents the server from donating slack time or allows the server to waste precious slack time through back donation in the idle state that can otherwise be reclaimed to help other waiting servers.

We propose a forward donation algorithm that can be used with slack reclamation algorithms, like BASH or BACKSLASH, to mitigate the resource overbooking problem. It allows the reserved, but unused, budget to be donated when its server stays in the idle state, without sacrificing the full bandwidth reservation for a new job which is released later.

Definition 6 (Forward Donation (FD))**.** *When a server enters the idle state, a one-shot timer is set to its EAT for a pseudo server activation. (If a back donation is used later to move its EAT backward to EAT', the timer is reset to EAT' which is no later than the current time.) If a forward donation timer goes off and its associated server has no job to execute, the server is inserted into the **Forward Donation Queue** (FDQ) instead of the waiting queue and marked as a **forward donation server**. The server is removed from the back donation queue, if it had borrowed budget to complete its previous job.*

When a new job arrives, the server is activated based on the chosen activation algorithm which assigns the budget and deadline to the server.

1) *If the new job arrives before the timer goes off, BACKSLASH enqueues the new job to the sever and delays the server activation until the timer goes off. (The FD timer is also used as the server activation delay timer.)*
2) *If the new job arrives after the timer goes off, i.e. the server was enqueued into the FDQ, BACKSLASH removes the server from the FDQ and then activates the server immediately with $c_{S_i} = R_i$ and d_{S_i} set to one period from the job release time. Please note that each server in the FDQ has its full reserved bandwidth available at any moment.*

The primary scheduler selects the earliest server from the waiting queue for execution. If it is a slack server, the secondary scheduler is triggered. After the server executes from t^{start} to t^{end}, whether it is a regular or slack server, its server budget is deducted only by

$$l - \sum_{S_i \in FDQ} f_i\,, \quad \textit{where } l = t^{end} - t^{start},\ f_i = u_i * l$$

If a new job arrives at a forward-donating server at $r_{i,j}$, the server is activated into the waiting queue immediately at $r_{i,j}$. The server is removed from the FDQ, and its budget and deadline are set to $c_{S_i} = R_i$ and $d_{S_i} = r_{i,j} + P_i$ respectively.

Theorem IV.1. *Given a feasible EDF schedule and valid secondary and pacing policies, adding forward donation policy does not invalidate the schedule.*

Proof: Even though a server has enough budget to be activated at its EAT, BACKSLASH leaves it in the idle state in order to conserve its budget for its up coming job. Since it is valid to activate a server at its EAT, a one-shot timer in the FD algorithm provides a fake workload to activated an idle state server at its EAT.

Suppose that $(n-1)$ forward donation servers are submitted to the system at t^{start}, where the primary scheduler can select a server for execution from the combination of the waiting and forward donation queue until an interrupt occurs at t^{end}. Figure 4 depicts our assumption where S_1 to S_{n-1} represent all servers in the forward donation queue, S_n to S_{n+m} represent all servers in the waiting queue, and S_n has the earliest deadline among all waiting servers. We call this the initial state of a forward donation process. We will prove that the system will return to this state after each execution and any budget spent during an execution interval $[t^{start}, t^{end}]$ will not invalidate the given feasible schedule.

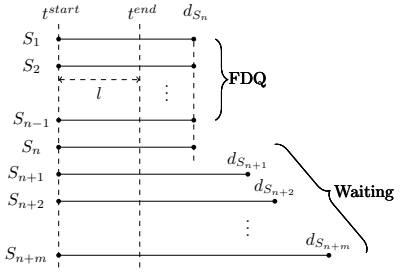


Fig. 4. FDQ

Theorems in the RBED paper allow any server, S_i, in the idle state to be activated at its EAT, where $lag_{S_i} = 0$, with its full reserved bandwidth $u_i = \frac{R_i}{P_i}$ or $\frac{R_i/h}{P_i/h}$, where $0 < h < \infty$, without invalidating a feasible schedule [14]. For each server in the FDQ, its deadline is set to the deadline d_{S_n} of the earliest waiting server S_n and its budget is set to $R_i * \frac{d_{S_n} - t^{start}}{P_i}$ by a prorated amount.

When the primary scheduler is triggered to select the earliest server to run at t^{start}, any server with the same d_{S_n} deadline can be scheduled first. The execution may be interrupted at t^{end} by a new arriving forward-donation server, a job release on any forward-donating server, the end of job on S_n or other events. Even though an unlimited number of feasible scheduling sequences can happen in $[t^{start}, t^{end}]$, we list three valid scheduling sequences here.

1) S_n is scheduled to run first and a timer is set by the primary scheduler to prevent it from using more than its current budget c_{S_n}. Thus, it is impossible to overload S_n when it is interrupted at t^{end} where $c_{S_n} \geq l = t^{end} - t^{start}$. S_i in the FDQ can be scheduled after t^{end}.
2) Instead of letting S_n execute all the way through, the processor sharing concept can be used to let S_i spend its R_i/h budget in very small intervals P_i/h, where $1 \leq i \leq n-1$ and $h \to \infty$ in $[t^{start}, t^{end}]$. The budget spent by a forward-donating server in $[t^{start}, t^{end}]$ is
$$\int_{t^{start}}^{t^{end}} \frac{R_i/h}{P_i/h} dx = \frac{R_i}{P_i} * x \mid_{x=t^{start}}^{x=t^{end}} = u_i * l$$
where $l = t^{end} - t^{start}$. S_n executes only when those forward-donating servers are not executing in $[t^{start}, t^{end}]$, so the budget spent by S_n is $l - \sum u_i * l$.
3) Instead of matching d_{S_n} at t^{start}, each forward-donating server can set its deadline to $d_{S_i} = t^{end}$ and their budget will be prorated to $c_{S_i} = \frac{R_i}{P_i} * l$. Since $d_{S_1} = \ldots = d_{S_{n-1}} = t^{end} < d_{S_n}$, S_1 to S_{n-1} can be scheduled in sequence. After each forward-donating server spends $u_i * l$ budget, S_n can be scheduled to run from $(t^{start} + \sum u_i * l)$ to t^{end}. Thus, S_n spends $l - \sum u_i * l$ budget in $[t^{start}, t^{end}]$.

The first scheduling sequence represents the traditional EDF scheduling and budget management that deducts l from c_{S_n} after S_i executes from t^{start} to t^{end}. In contrast, the second and third scheduling sequences execute the forward donation servers for $u_i * l$ units of time and only execute S_n for $l - \sum u_i * l$ units of time in $[t^{start}, t^{end}]$. Since the forward donation servers do not have their own jobs for execution in $[t^{start}, t^{end}]$, our FD algorithm uses their budgets to execute the job on S_n. From the scheduler's point of view, there is no difference in running the idle process or the job of S_n. Consequently, S_i receives slack donation from those forward donation servers.

Since all forward donation servers only spend budget at their reserved rate, they can be considered leaving the system and re-entering the system at t^{end}, where $lag_{S_i \in FDQ} = 0$, using theorems in the RBED paper [14]. Upon their reentrance, the feasible schedule is maintained as long as their bandwidth reservation $u_i = \frac{R_i}{P_i}$ or $\frac{R_i/h}{P_i/h}$ remains unchanged, where $0 < h < \infty$. Of course, a forward donated server S_i activated into the waiting queue by its new job at t^{end} is assigned $c_{S_i} = R_i$ and $d_{S_i} = t^{end} + P_i$ and does not reenter the FDQ at t^{end}. In other words, a new job released on a forward donation server will be given its full reserved budget to use with the deadline one period away from the release time. The number of servers in either forward donation or waiting queue vary from time to time, but each t^{end} can be considered as a new t^{start}. Thus, the system returns to the initial state of the forward donation process. When the number of servers in the FDQ reaches zero, the FD performance degrades to the traditional EDF budget consumption.

From the second and the third scheduling sequences, we find that it is not necessary to assign server budget and deadline to the servers in the FDQ because their deadlines can be redefined

to any value based on a fixed utilization reservation. Our FD algorithm controls the execution rate of each forward donation server at its reservation rate, so it neither overloads the primary scheduler nor jeopardizes any new job released on a forward donation server. ■

Theorem IV.2. *Given a feasible EDF schedule and valid secondary and pacing policies, a server that forward donates its budget is guaranteed to immediately provide its new released job a full reserved bandwidth. (See proof for theorem IV.1.)*

V. Experiments

A. Performance Metrics

We use the same deadline miss ratio (DMR) and tardiness (TRD) performance metrics as defined in BACKSLASH [13]. Suppose there are $miss_i$ deadline misses among n_i completed jobs, and the accumulated lateness resulting from those deadline misses is $late_i$, DMR and TRD are calculated as

$$DMR(T_i) = \frac{miss_i}{n_i}; \quad TRD(T_i) = \frac{late_i}{n_i * P_i}$$

The system wide deadline misses and tardiness for all m SRT tasks can be calculated using average deadline miss ratio (ADMR) and average tardiness (ATRD) defined as

$$ADMR = \frac{\sum_1^m DMR(T_i)}{m}; \quad ATRD = \frac{\sum_1^m TRD(T_i)}{m}$$

Except using exact WCET in each period, two BACKSLASH formulas are used to generate synthetic workloads for our experiments.

$$NW(\mu) = \begin{cases} \frac{1}{\sqrt{2\pi}} \exp^{-\frac{(x-\mu)^2}{2\sigma^2}} & , 0 < x \leq \mu \\ 0 & , x \leq 0 \text{ or } x > \mu \end{cases}$$

$$NA(\mu) = \begin{cases} \frac{1}{\sqrt{2\pi}} \exp^{-\frac{(x-\mu)^2}{2\sigma^2}} & , 0 < x \\ 0 & , x \leq 0 \end{cases}$$

Both $NW(\mu)$ and $NA(\mu)$ use a normal distribution with mean μ and standard deviation $\sigma = 0.1\mu$.

A trace of stationary job service times is generated for each task using a random number generator. By changing seeds, fifty workloads with the same statistical characteristics are generated and scheduled with or without DoS prevention and forward donation algorithms for two million time steps. After the performance metrics, like DMR and TRD, are calculated for each workload, the mean and the standard error of each performance metrics is calculated from fifty workloads. As a result, each data point in our experiment figures represents the average of fifty workloads with the same statistical characteristics with the error bars.

B. Effectiveness of DoS Prevention on Various Slack Amount

When adequate slack time is generated, BACKSLASH outperforms other existing algorithms. However, when the ratio of supply to demand is low, its performance may drop sharply. Furthermore, any task can mislead the slack scheduler by claiming that it needs the slack time the most. Despite the difficulty of obtaining an accurate or appropriate resource estimation the scheduler should not just ignore the problem.

The first experiment mimics one of BACKSLASH's workloads with one additional small utilization task, ATK4, to demonstrate the limitation of BACKSLASH and the effectiveness of our DoS prevention algorithm. Each experiment consists of two periodic HRT tasks, one periodic SRT, and one potential attacker as shown in table I. When a task changes its parameters in a new experiment run, its corresponding server's parameters are modified accordingly, i.e. $R_{S_i} = \bar{e}_{T_i}$. To clarify, when HRT2 increases its $\bar{e}$ by 7 and reserves 2% more utilization, SRT3 decreases its $\bar{e}$ by 9 and compensates for this 2% increase in utilization. All periods remain fixed as listed in table I.

Task	Server Parameters			Task Parameters		Δ Parameter	
	R	P_S	u_S	$e = f(\bar{e})$	P_T	$\Delta\bar{e}$	Δu_S
HRT1	234	600	39%	234	600	0	0
HRT2	207	450	46%	NW(207)	450	-9	-2%
SRT3	49	350	14%	NA(49)	350	$+7$	$+2\%$
ATK4	3	300	1%	3 or 30	300	3 or 30	0

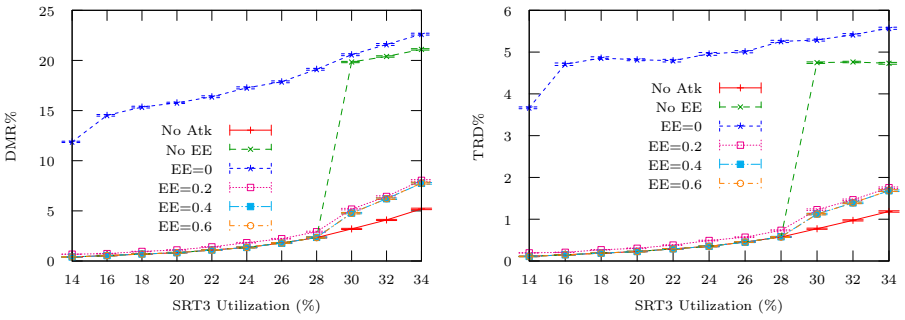
TABLE I
Workload

Each job submitted by HRT1 consumes its WCET exactly which is 234. The workload submitted by HRT2 is generated by NW($\bar{e}$) formula that is guaranteed to produce the service times e no larger than $\bar{e}$; so HRT2 is a regular slack time donor. SRT3 generates its workload using NA($\bar{e}$) formula that produces the service times e around a mean value $\bar{e}$, so the average workload of SRT3 matches its server reservation. The workload submitted by ATK4 depends on its role, e.g. attacker or non-attacker. It requires its $WCET = 3$ to complete each job as a regular HRT task, but it requires $10 * WCET$ to complete each job as an attacker.

By adjusting the utilization of HRT2 and SRT3, eleven sets of workloads are generated. When they are scheduled by BACKSLASH with a non-attacking ATK4, the results are marked as "No Atk" in figure 5. ATK4 becomes an attacker in all other workloads that are scheduled with various EE thresholds. When there is no constraints on EE, i.e. $EE = \infty$, plain BACKSLASH is used and its results are marked as "No EE". When an EE threshold x is used to protect the system, the results are marked as "$EE = x$".

Because 50% of service times generated for SRT3 are greater than its reservation, the primary scheduler without the secondary and the pacing policies guarantees to meet only 50% of SRT's deadlines. The "No Atk" case has the best performance, because ATK4 behaves like a HRT task and does not oversubscribe its server. Therefore, all slack time generated by HRT2 can be used by SRT3. This case also demonstrates the effectiveness of BACKSLASH which drops the DMR from 50% guaranteed by the primary policy to less than 5.2%.

When ATK4 becomes an attacker, it does not affect either HRT1 or HRT2 because their reservations are guaranteed by the primary policy. However, ATK4 competes with SRT3 for slack time and changes the decision of the slack scheduler



(a) DMR as a Function of Load

(b) TRD as a Function of Load

Fig. 5. Effectiveness of DoS prevention on various slack amount

by underbooking resources. The "No EE" line depicts the performance of BACKSLASH which does a very good job in meeting SRT3's demand before its load becomes greater than 28%. After the utilization of SRT3 becomes greater than 28%, HRT2 with a lower utilization cannot generate adequate amounts of slack time to meet the demand of SRT3 and ATK4.

When limited amounts of slack time are available and ATK4 dominates the slack consumption, BACKSLASH significantly drops the performance of SRT3. When an EE threshold is used to protect the secondary scheduler, $EE = 0$ overprotects the system, strips down the effectiveness of the slack reclamation, and produces the worst performance. Even though SRT3 can consume slack time before its job executes longer than its reserved budget, the job is not eligible for slack consumption in the first three steps of the DoS prevention algorithm after its elapsed execution time exceeds its reserved budget. When the EE threshold equals 0.2, 0.4 or 0.6, i.e. limiting the slack consumption of ART4, the performance of SRT3 is restored back to very close to the "No Atk" case. For example, when SRT3 has a 30% load, its DMR is 3.2% without an attacker which increases to 19.8% when BACKSLASH is under attack. When an $EE = 0.4$ protection is used, the DMR of SRT3 is significantly reduced to 4.8% which is very close to 3.2%.

Since the purpose of the EE threshold is to make each task responsible for its resource estimation, it does not make sense to set a very high value. If a task decides to extremely underbook its resource, it can rely on its future budget and wait for the leftover slack time.

C. Impact of DoS Attack to Multiple SRT Tasks

This section demonstrates the impact of the slack DoS attack on several SRT tasks in the system and how effectively our DoS prevention algorithm can protect the system against the attack. The workload consists of two HRT tasks, four SRT tasks, and one all time attacker ATK7 as summarized in table II. HRT1 and HRT2 are all time slack donors. Any SRT task can overload is server by about 2%. For example, an average workload $\frac{mean(NA(33))}{150}$ is equivalent to 102% of SRT3's reserved server utilization $\frac{30}{150}$. In each experiment, only one of the four SRT tasks submits an average workload which exceeds its reserved server utilization. When an experiment contains a SRTn that overruns for $m\%$ is scheduled with various EE thresholds, the results of the experiment are drawn as a line marked as "SRTn $m\%$" in figure 6.

Task	Server Parameters			Task Parameters		
	R	P	u	$e = f(\bar{e})$	P	$\Delta\bar{e}(\Delta u_{Task})$
HRT1	38	200	19%	NW(38)	200	0
HRT2	91	910	10%	NW(91)	910	0
SRT3	30	150	20%	NA(30)	150	3(2%)
SRT4	65	260	25%	NA(65)	260	5(1.92%)
SRT5	81	540	15%	NA(81)	540	11(2.03%)
SRT6	71	710	10%	NA(71)	710	14(1.97%)
ATK7	3	300	1%	30	300	0

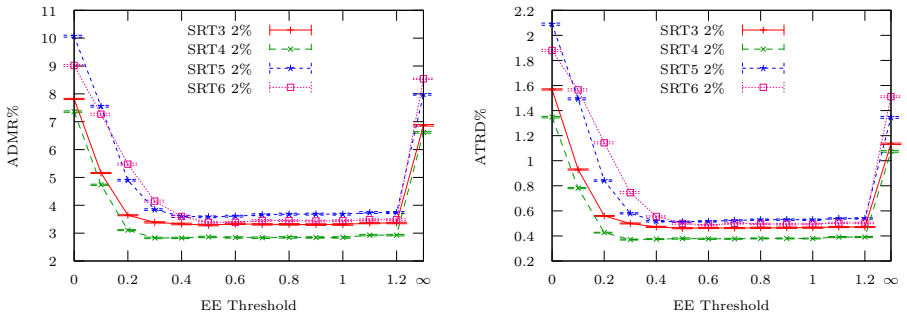
TABLE II
WORKLOAD FOR SYSTEM-WIDE SRT TASKS PERFORMANCE

Since HRT1 and HRT2 do not miss any deadline, their results are omitted. When the system is protected by $EE = 0.4$ to $EE = 1.2$, all SRT tasks perform the best compared to other thresholds. Within the reasonable range of the EE threshold, say 0.4 to 1.2, the ADMR decreases for about 8%, and the ATRD decreases for about 0.7% compared to $EE = \infty$, i.e. BACKSLASH. If the EE threshold is lower than 0.2, the functionality of the slack reclamation is restricted so the ADMR are ATRD are increases. If the overloaded SRT task is excluded in the performance measurement, the ADMR can further decrease.

D. Forward Donation on Various Slack Amount

This experiment demonstrates that by increasing the utilization ratio of one HRT and two SRT tasks, BACKSLASH increases the performance of each individual SRT task because more slack time is reclaimed from the HRT task and the SRT tasks overrun less intensely. For a HRT task that releases less than one job per period, we show that forward donation can reclaim additional slack time from the task in the idle state.

This experiment consists of one periodic HRT task, two periodic SRT tasks, and one potential attacker as shown in table III. Besides submitting only HRT jobs, ATK4 further reduces its workload by submitting one job in three periods. The results are plotted in figure 7. We intentionally overload each SRT server by about one percent to show that the additional slack time reclaimed by the forward donation algorithm can further improved the performance of BACKSLASH. I.e., the



(a) ADMR (b) ATRD

Fig. 6. Average performance when SRTn overloads by 2%

benefit of the extra slack time can be distinguished only when the slack time reclaimed by BACKSLASH is not sufficient to meet the requirements of the SRT servers.

Task	Server Parameters			Task Parameters		Δ Parameter	
	R	P_S	u_S	$e = f(\bar{e})$	P_T	R	Δu_S
HRT1	120	600	20%	120	600	0	0
SRT2	90	450	20%	NA($R+4$)	450	+9	+2%
SRT3	70	350	20%	NA($R+3$)	350	+7	+2%
ATK4	120	300	40%	NW(120)	300	-12	−4%

TABLE III
WORKLOAD

ATK4 reserves $\frac{120}{300}$ utilization and SRT2 and SRT3 reserve $\frac{90}{450}$ and $\frac{70}{350}$ utilization respectively to start with, where $R_{SRT2} = 90$ and $R_{SRT3} = 70$. After fifty sets of job traces are scheduled with and without forward donation, each SRT task increases its server utilization by 2% in each of the successive runs by increasing its reserved budget R_{SRT2} and R_{SRT3} while keeping its server periods unchanged. In order to keep the total server utilization fixed at the theoretical upper bound 100%, the attacker ATK4 decreases its utilization by 4% when two SRT tasks together increase 4% utilization.

Each job released by HRT1 requires exactly its WCET through out this experiment to provide some noise to the system and to check for any violation of the server schedulability. SRT2 and SRT3 use $R_{SRT2}+4$ and $R_{SRT3}+3$ respectively in NW($\bar{e}$) formula to generate their service times. Consequently, their average service times are slightly larger than their server reservations. Since the service time of ATK4 never exceeds its WCET, it is a regular slack time donor. We only plot the DMR and TRD of SRT2 and SRT3 in figure 7, because our forward donation algorithm does not invalidate the feasible schedule of BACKSLASH and HRT1 and ATK4 miss no deadline. When the system is scheduled by BACKSLASH, the results for SRT2 and SRT3 are marked as "T2" and "T3" respectively. When the system is scheduled with forward donation, the results are marked as "T2Fd" and "T3Fd" respectively.

When ATK4 increases its utilization by reserving more budget, its job service times generated by NW() formula provides more slack time to help SRT2 and SRT3. Moreover, SRT2 and SRT3 overrun less intensively when they reserve less budget and use the smaller values in NA() formula to generate their job service times. Not surprisingly, the result conforms to a similar experiment of BACKSLASH. When the utilization of ATK4 increases from 16 to 40 percent in the BACKSLASH scheduler, the DMR drops from about 15% to about 7% in figure 7(a) and their TRD drops to about 1.5% in figure 7(b).

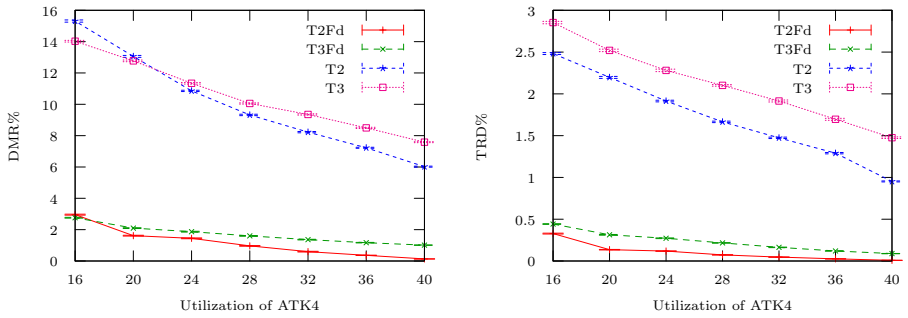
We next demonstrate the effectiveness of forward donation on top of BACKSLASH. Even though ATK4 releases only one job in every three periods, BACKSLASH cannot donate slack time from the idle state. With forward donation, those budgets can help SRT2 and SRT3 without jeopardizing any forthcoming jobs of ATK4 or any WCET jobs of HRT1. I.e., any job released by ATK4 or HRT1 will be immediately granted a full bandwidth when it is released. In figure 7(a), forward donation decreases the DMR of SRT2 and SRT3 from about 14% to about 3% when the utilization of ATK4 equals 16%. Their DMR further decreases to very close to 0 when the utilization of ATK4 increases. In figure 7(b), forward donation decreases the TRD of SRT2 and SRT3 from about 2.7% to about 0.4% when the utilization of ATK4 equals to 16%. Their TRDs further decrease when the utilization of ATK4 increases.

By changing the utilization ratio of one HRT task and two SRT tasks while keeping their periods unchanged, we demonstrate the effectiveness of BACKSLASH in scheduling slack time collected from the residual budget of the non-idle server ATK4. Using it as a benchmark, we demonstrate that forward donation collects additional slack time from ATK4 in the idle state.

E. Impact of Forward Donation on Multiple SRT Tasks

Four sets of experiments shown in table IV demonstrate that forward donation improves the system-wide performance of all SRT tasks in various workload compositions and overload situations. The total server utilization always remains at 100%. Submitting a job in ∞ periods means that the task never submits after the first job is done. Even numbered periods are skipped simply to cut the simulation runtime. Since HRT1 and ATK7 do not miss any deadline, we only plot the ADMR and ATRD of all SRT tasks in figure 8.

We arbitrarily pick one HRT task, four SRT tasks and one potential resource overbooking attacker ATK6 in experiment



(a) (b)

Fig. 7. Effectiveness of forward donation on various slack amount (ATK4 releases one job in every three server periods)

Exp	Task	Server Parameters			Task Parameters		
	ID	R	P	u	$e = f(\bar{e})$	P	Period Per Job
G1	HRT1	42	350	12%	42	350	1
	SRT2	54	360	15%	NA(57)	360	1
	SRT3	72	450	16%	NA(75)	450	1
	SRT4	90	500	18%	NA(93)	500	1
	SRT5	114	600	19%	NA(117)	600	1
	ATK6	126	630	20%	NW(126)	630	1,3,5,7,9, or ∞
G2	HRT1	81	675	12%	81	675	1
	SRT2	96	640	15%	NA(99)	640	1
	SRT3	96	600	16%	NA(99)	600	1
	SRT4	99	550	18%	NA(102)	550	1
	SRT5	95	500	19%	NA(98)	500	1
	ATK6	90	450	20%	NW(90)	450	1,3,5,7,9, or ∞
G3	HRT1	42	350	12%	42	350	1
	SRT2	90	360	25%	NA(93)	360	1
	SRT3	72	450	16%	NA(75)	450	1
	SRT4	90	500	18%	NA(93)	500	1
	SRT5	114	600	19%	NA(117)	600	1
	ATK6	63	630	10%	NW(63)	630	1,3,5,7,9, or ∞
G4	HRT1	42	350	12%	42	350	1
	SRT2	90	360	25%	NA(93)	360	1
	SRT3	72	450	16%	NA(76)	450	1
	SRT4	115	500	23%	NA(120)	500	1
	SRT5	114	600	19%	NA(120)	600	1
	ATK6	15	300	5%	NW(15)	300	1,3,5,7,9, or ∞

TABLE IV
WORKLOADS

G1. The 20% server utilization of ATK6 generates plenty of slack time when it releases less than one job per period. We will demonstrate that the slack time cannot be reclaimed by BACKSLASH from ATK6 when it is in the idle state, but forward donation can. Experiment G2 keeps the same server utilization for each server from G1, but it changes the period length from ascending to descending ordering. G3 increases the server utilization ratio of SRT2 and ATK6 from G1, so ATK6 produces less slack time. In addition to increasing the workload of SRT4 and SRT5, G4 further increases the server utilization of SRT2 and ATk6.

When ATK6 submits a lighter workload in G1, G2, and G3, our forward donation reduces ADMR by about 5% to 8% across the board. In some cases, the ADMR is very close to 0. If the overloaded SRT tasks are excluded from the ADMR calculation, the ADMR will be lower than those reported results. Since ATK6 reserves less utilization and the SRT tasks overload more intensively, the ADMR is very high at 16% to 18%. Our forward donation algorithm reduces the ADMR by 8% to 11%. We see a similar improvement for the ATRD.

F. Non-stationary Workload

Finally, we demonstrate how non-stationary loads influence maximum QoS level in local steady states. Table V represents 11 stationary stages in a non-stationary system. Figure 9 shows 100 service time samples of SRT5 in each stationary stage and the ADMR for each stage. The first 5 experiments use DoS prevention with EE threshold at 0.5, the next 5 experiments use forward donation, and the last experiment uses both algorithms. The results are compared with BACKSLASH results drawn in a dashed line.

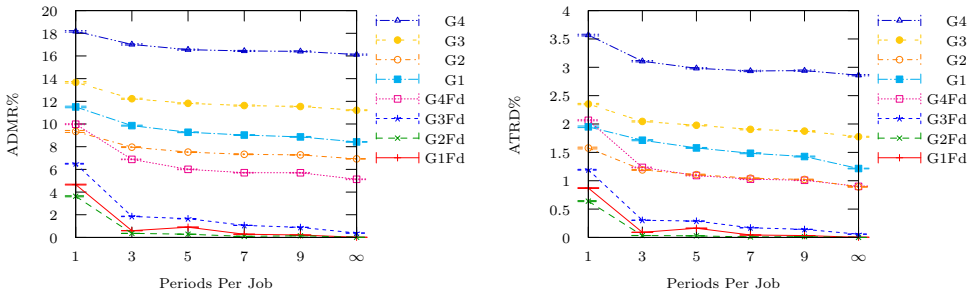
In Exp1 (stage 1), the average workload of each task equals its resource reservation. Since there is no significant resource over- or under- booking, the ADMR is about 1%. Both SRT4 and SRT5 reduce their average workload but ATK7 increases its average workload in Exp2 (stage 2). AMDR increases to 2% in BACKSLASH, because it allows ATK7 to dominate the slack time. With DoS prevention, ADMR drops to 0.2%. SRT4 and SRT5 gradually increase their workloads from Exp3 to Exp5, so ADMR continues to increase with or without DoS prevention. However, the DoS prevention algorithm always produces a lower ADMR than does BACKSLASH.

In Exp6, ATK7 stops its resource underbooking attack while the servers of SRT4 and SRT5 remain overloaded. The system is less overloaded than Exp5, so the ADMR drops with or without DoS prevention. ATK7 stops submitting jobs from Exp7 to Exp10. From Exp8 to Exp10, HRT1 reduces its workload by submitting 1 job in every 2 periods and SRT4 and SRT5 gradually reduce their average workloads. Since forward donation reclaims extra slack time from ATK7 and HRT1, it outperforms BACKSLASH from Exp6 to Exp10.

In Exp11, HRT1 overbooks its resource by submitting 1 job in every 3 periods but ATK7 underbooks its resource by asking for 10 times of its WCET to complete a job. In the mean time, SRT4 and SRT5 overloads their server. With the combination of DoS prevention at $EE = 0.5$ and forward donation, it's AMDR is 7.7% lower than BACKSLASH's.

VI. CONCLUSION

In general, we show that when the system workload reaches a stationary state and the slack time supply and demand reach



(a) (b)

Fig. 8. Effectiveness of forward donation when various slack time is reclaimed from the idle state

Task	Server			Task service time generation										
ID	R	P	u	Exp1	Exp2	Exp3	Exp4	Exp5	Exp6	Exp7	Exp8	Exp9	Exp10	Exp11
HRT1	38	200	19%	NW(R)							1 Job/(2 Periods)			1J/3P
HRT2	91	910	10%	NW(R)										
SRT3	30	150	20%	NA(R)										
SRT4	65	260	25%	NA(R)	NA(R-5)	NA(R)		NA(R+5)				NA(R)		NA(R+10)
SRT5	81	540	15%	NA(R)	NA(R-10)	NA(R)		NA(R+11)					NA(R)	NA(R+11)
SRT6	71	710	10%	NA(R)										
ATK7	3	300	1%	R	R*10				R	Stop job release				R*10
Attack Type				-	Underbooking				Overbooking					Both

TABLE V
DOS PREVENTION AND FORWARD DONATION)

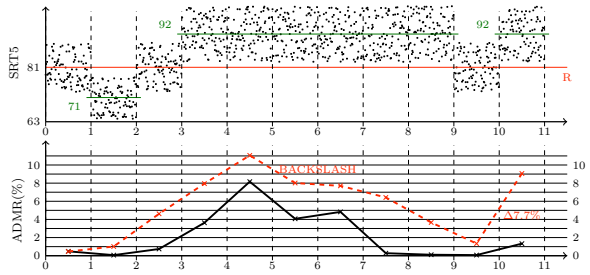


Fig. 9. Sample workload for SRT5 and system performance results

equilibrium, the system supports a specific level of QoS. In a non-stationary open system, any task can exploit its vulnerability and dominate slack consumption. Our work demonstrated that when the secondary scheduler uses an EE threshold, the secondary scheduler can alleviate the performance degradation caused by a resource underbooking task. Both BASH and BACKSLASH intentionally use a job release to activate its server, so the server budget is not lost at its EAT. Our Forward Donation effectively reclaims the slack time from any full bandwidth server from the idle state without jeopardizing any upcoming jobs that require WCET to complete on a forward donated server. It can cooperate with the DoS prevention to stabilize the QoS of a non-stationary system as demonstrated by the flatter curve of ADMR in figure 9.

REFERENCES

[1] C. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *In Proceedings of the IEEE ICMCS*, 1994, pp. 90–99.

[2] T. M. Ghazalie and T. P. Baker, "Aperiodic servers in a deadline scheduling environment," *Real-Time Systems*, vol. 9, no. 1, pp. 31–67, July 1995.

[3] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," *The 19th IEEE RTSS*, pp. 4 – 13, 1998.

[4] M. Spuri, G. C. Buttazzo, and F. Sensini, "Robust aperiodic scheduling under dynamic priority systems," in *IEEE RTSS*, 1995, pp. 210–221.

[5] G. Marzario, L.and Lipari, P. Balbastre, and A. Crespo, "IRIS: a new reclaiming algorithm for server-based real-time systems," *The 10th RTAS*, pp. 211 – 218, 2004.

[6] S. Banachowski, T. Bisson, and S. A. Brandt, "Integrating best-effort scheduling into a real-time system," *25th IEEE RTSS*, December 2004.

[7] G. Lipari and S. Baruah, "Greedy reclamation of unused bandwidth in constant-bandwidth servers," *The 12th ECRTS*, pp. 193–200, 2000.

[8] G. Bernat, I. Broster, and A. Burns, "Rewriting history to exploit gain time," *25th IEEE Real-Time Systems Symposium*, pp. 328 – 335, 2004.

[9] G. Lamastra, G. Lipari, and L. Abeni, "A bandwidth inheritance algorithm for real-time task synchronization in open systems," *22nd IEEE Real-Time Systems Symposium*, pp. 151–160, 2001.

[10] R. Santos, G. Lipari, and J. Santos, "Scheduling open dynamic systems: The clearing fund algorithm," *The 10th RTCSA*, 2004.

[11] M. Caccamo, G. Buttazzo, and L. Sha, "Capacity sharing for overrun control," *The 21st IEEE RTSS*, pp. 295 – 304, Nov 2000.

[12] M. Caccamo, G. C. Buttazzo, and D. C. Thomas, "Efficient reclaiming in reservation-based real-time systems with variable execution times," *IEEE Transactions on Computers*, vol. 54, no. 2, pp. 198–213, 2005.

[13] C. Lin and S. A. Brandt, "Improving soft real-time performance through better slack reclaiming," in *The 26th IEEERTSS*, 2005, pp. 410–421.

[14] S. Brandt, S. Banachowski, C. Lin, and T. Bisson, "Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes," *The 24th IEEE RTSS*, pp. 396 – 407, 2003.

[15] C. Lin, "Unified and effective soft real-time processing in integrated systems," Ph.D. dissertation, University of California at Santa Cruz, 2006.

[16] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[17] W. Lin and G. Nutt, "Detecting and preventing dos attacks in slack scheduling," *28th IEEE RTSS, WiP*, December 2007.