Building on the BACKSLASH Algorithm

Wang-ting Lin and Gary Nutt
Department of Computer Science
University of Colorado
Technical Report Number CU-CS-1022-06

{linwt, nutt}@colorado.edu

Abstract

Contemporary computer systems often mix real-time and non real-time (best effort) work, due to the increasing range of applications. Contemporary scheduling algorithms in such an environment may use earliest deadline first scheduling along with slack time scheduling. This paper describes some additions made to the BACKSLASH slack time scheduling algorithm. It introduces the idea of early release of work under certain circumstances. The paper also introduces a refined measure of the deadline attainment performance of tasks in such a system. Finally, we address a problem in which tasks that dramatically under book their actual processor needs can produce the effect of a denial of service attack on the other tasks in the system.

1. Introduction

Contemporary desktop computing systems support increasingly diverse types of applications, processing diverse types of information. A dozen years ago, these systems focused on document publishing, decision support tools, rudimentary web browsers (no applets), etc. Since 1995, these desktop computers have increasingly supported applications that playback streaming media data, enabling users to listen to audio data (such a MP3 data) and view audio/video streams (such as MPEG data).

This changing character of the applications' demand for various system resources has influenced OS resource management policies. Whereas conventional applications can use the computer's resources under the traditional *best effort* (BE) resource allocation policies, many of the new applications rely on certain assurances of service rates for resource usage. BE resource policies are intended to optimize allocation based on one or more traditional performance metrics such as throughput, turnaround, equity, or utilization, but not to provide any assured *rate* of service.

As long as the computer is operating with excess resource capacity, applications can deliver acceptable behavior even if the system does not meet all of the applications' rate-based resource requirements. However, when the system approaches saturation, rate-based applications will begin to fail. *Real-time* systems are designed with resource allocation policies that accommodate rate-based resource usage. In a strict sense, *hard real-time* (HRT) tasks require that the associated jobs *never* underestimate their resource needs, and that the system *always* be able to fulfill the specified needs. *Soft real-time* (SRT) applications also pre specify their resource needs, but they are able to tolerate a variety of situations in which the system is unable to meet some of the rate-based resource requirements; SRT tasks may also underestimate their resource requirements. That is, the SRT strategy is an instance of *quality of service* (QoS) computing strategies. In QoS systems, the OS makes an assurance regarding the rate that it can provide resources to a task over several different jobs within the task. The details of the assurance may differ across the spectrum of QoS algorithms.

Today many computers support SRT and BE applications (usually with only best effort assurances for the SRT applications, meaning that all applications are managed under a BE policy). However there is also a class of computers that simultaneously support HRT, SRT, and BE tasks. For example a desktop computer may have an HRT task that monitors and controls the power usage in a residence; it may have SRT tasks for playing back movies; and it may have BE tasks such as spreadsheet and publishing programs. The work described in this paper focuses on resource strategies for such systems.

Since approximately 1990, people have studied various resource management strategies to support combined HRT, SRT, and BE tasks, particularly for CPU scheduling. Various camps have formed to focus on one approach or another, based on different criteria for evaluating acceptable performance. One camp has adopted the earliest deadline first (EDF) scheduling policy, with supplementary mechanisms to address cases where one or more of the SRT application tasks exceeds their service time estimates within particular periods and thereby fail to complete the work for that period. In 1992, Lehoczky and Ramos-Thuel introduced the idea of using *slack time* – or time that was reserved for another job, but which was not used to fulfill its service time in a period – to handle these tasks overruns [20]. Since that time many others have refined this basic idea, including [1, 6, 10, 11, 15, 31].

Within this framework, Brandt, et al. showed how BE tasks could utilize slack time created by HRT and SRT jobs scheduled using EDF [4]. Lin and Brandt refined the technique to enable SRT jobs to utilize the slack time in their SRAND, SLAD, SLASH, and BACKSLASH algorithms. The research described in this paper builds on their work. They describe four principles for managing slack time [23]:

1. Allocate slack as early as possible, with the priority of the donating job.
2. Allocate slack to the job with the highest priority (earliest **original** deadline).
3. Allow tasks to borrow against their own future resource reservations to complete their current job.
4. Retroactively allocate slack to jobs that have borrowed from their current budget to complete a previous job.

Our work first reexamines the performance metrics used to evaluate algorithms, and then adds two new scheduling principles for this class of scheduling algorithms:

5. Reevaluate job EDF priorities at the moment the slack time becomes available.
6. Suppose that a job misses its deadline: then let the release time for job_{i+1} be redefined to be $R'_{i+1} = d_i + \Delta t$; i.e., R'_{i+1} is a pseudo release time for period i+1 when the job misses deadline d_i for Δt.

> Suppose that the scheduler attempts to satisfy both job_i and job_{i+1} service times prior to d_{i+1} (which is the same as R_{i+2}). If it succeeds, the task is back on schedule when job_{i+2} is released; if it fails, it can again adjust the release time to catch up in period i+2, etc.

In the next section we provide a more complete description of slack time scheduling, including a discussion of related work in the area. In Section 3 we describe our improvements to Lin and Brandt's work in detail. In the final section we summarize the work.

2 Background

Scheduling policies for contemporary systems may support a mixed workload of HRT, SRT, and BE tasks. The basic premise of the work is that HRT tasks are conservative and will reserve excessive CPU that can be used to accommodate other task executions (in every period). Service time estimates for the jobs in a HRT task are normally the worst case execution time (WCET) for any execution of the job in any period. Depending on the variance of the individual job service times from the WCET, it may be that the scheduling algorithm will reserve excessive amounts of time for this worst case, but actual execution will not use all of the reserved time. SRT jobs will also normally execute in less time than is reserved for their execution. However SRT tasks may also overrun their service time reservation, since their service time estimates are typically not as conservative as HRT estimates, i.e., when a job actually uses the worst case execution time, any reservation that is less than the WCET will be insufficient. There is an opportunity for a system to utilize unused but reserved time – slack time – to execute SRT jobs that overrun their reservation.

2.1 Processor Capacity Reserves

As mentioned in the introduction, SRT scheduling relaxes the requirements compared to those for HRT, e.g., by allowing a percentage of a computation's tasks to miss their deadlines, perhaps by a bounded amount of computation time. An interesting aspect of SRT is the spectrum of techniques that have been used to relax resource requirements. This, of course, leads to a spectrum of metrics for comparing different SRT approaches.

SRT began to grow in importance when general purpose computers began to support multiple media types, particularly streaming media. Besides the obvious focus on SRT, researchers began to consider strategies for supporting a mix of HRT, SRT, and BE applications. For example, Berkeley researchers saw the need to support continuous media applications in the DASH processor [2]; researchers at CMU began to consider ways to modify Mach so that it could accommodate mixed classes of applications, e.g., see [27, 37]; Fall and Pasquale described in-kernel modules to support multimedia playback [14]; Microsoft researchers developed Rialto for multimedia support [18]; and the SMART scheduler was designed to accommodate multimedia in a mixed system [28]. The need for mixed class scheduling was well established by 1997.

The processor capacity reserves work established a new model for thinking about mixed load environments [26]. Briefly, this approach employs a quality of service (QoS) approach for admitting tasks to the system. Each task executes on an abstract machine – a server – that expects to use a fractional amount of the physical processor. That is, each task uses a server that has a processor requirement, C, that represents the amount of time required to execute each of the task's jobs during a period of the computation, T. The fraction of time that the server requires is then

$$\rho = C/T$$

HRT tasks are theoretically characterized by a processor time estimate of C = WCET and T = period for each task. However, a fraction of the ongoing processor time for a HRT task can be specified using ρ and T (rather than C and T). For example a periodic task with a 50 msec period may require 20% of the processor. BE tasks that have no deadline can also be assigned to a server with some ρ, but without specifying T. If the BE task exceeds ρ over any specified time T', then the task has exceeded its reservation and it should be temporarily suspended until it replenishes its processor reservation by the passage of time.

2.2 EDF versus RM Scheduling

Liu and Layland established admission requirements for rate monotonic (RM) and earliest deadline first (EDF) scheduling. They showed that RM tasks uses static priorities determined as a function of 1/T – the smaller the value of T, the higher the task's priority [21]. Besides showing that RM & EDF algorithms produce optimal schedules, they showed that a system can use RM to assure that a collection of n tasks receive service provided that

$$\Sigma_{i=1}^{n} C_i / T_i \leq n(2^{1/n} - 1)$$

As n increases indefinitely, the bound approaches 0.693, meaning that the admitted tasks can be scheduled with RM provided that they do not reserve more than 69.3% of the processor. EDF uses dynamic priorities – the nearness of a task's deadline; this allows the admission bound to be 100%.

Mercer, et al., consider some pros and cons of RM versus EDF algorithms. Although EDF allows for a higher admission bound, the requirement for managing dynamic priorities can introduce enough scheduling complexity to offset its value. In general, most real-time system developed from 1973 to the late 1990s use RM because of its known admission criteria and its simple implementation.

By the 1990s, researchers began to reconsider EDF because of the possibility of higher processor utilization in saturated systems. In the 1970s, EDF was defined in terms of nonpreemptable tasks, meaning that:

- If the job finished earlier than its reserved execution time in a period, the server (and processor) became idle and no other jobs in other tasks were able to run within that reserved time frame.
- If the task overran its reserved time, it used the original deadline to compete with other tasks which caused a "domino effect" whereby the task that missed its deadline continued to have high priority, thereby continuing to use the CPU at the expense of all subsequent tasks – frequently causing all subsequent jobs within a task to miss their deadlines.

Buttazzo conducted a careful comparison of RM and EDF in 2003 [9]. He observed that RM only guarantees that the highest priority (highest rate) task will never miss the deadline, but make no guarantee on all the other tasks in the system. On the other hand, the tasks start to behave like they are submitting jobs at a lower rate when EDF is overloaded. This suggests that RM is not predictable for all tasks, only the one with highest priority. Another problem with RM is that the processor utilization tends to be lower than that of EDF. Buttazzo argued that EDF perform no worse than RM in many aspects while EDF yields higher processor utilization.

2.3 Handling Job Overruns

In SRT systems, including ones that use processor reserves, SRT jobs will sometimes attempt to use more than their server reservation amount: this is referred to as an *overrun* situation. Gardner and Liu identified two general strategies for handling overruns (and processor overloading) [15]. First, then identified a class of algorithms that are optimal and have no particular mechanism for addressing overloads, e.g., deadline monotonic (DM), RM, and EDF, is used as the baseline for comparison. This class of algorithms is used to establish a benchmark for comparing the two new strategies. The first strategy for handling overrun is (1) to detect an overrun when it occurs, and (2) to reschedule the remaining work of the overrunning job on a distinct server that has capacity reserved without concern for deadlines – an aperiodic server. These algorithms are said to use the *Overrun Server Method* (OSM). The second strategy – the *Isolation Server Method* (ISM) – detects a server reserve overrun when it occurs, but reschedules the remaining work using that server's future processor reservations. For example, the algorithm might slip the deadline of all subsequent jobs in the task by one period, thereby allowing the overrunning job to use the budgeted processor reserve originally intended for the next job in the task. As one might expect, the relative behavior of the approaches is influenced by the nature of the workload.

This led to a spectrum of OSM refinements: for example, Sprunt described a Sporadic Server (SS) rather than an aperiodic overrun server [33]. In SS the overrun portion of the job is given a static priority when it is assigned to the overrun server, thereby completing the overrun processing in a predictable amount of time in a server environment that admits sporadic jobs. CUS [13] uses the idea of an OSM server, but consider dynamic priority algorithms for scheduling the work on the overrun server. Ghazalie and Baker refined the SS work by using EDF (rather than RM) [16].

2.4 Exploiting Slack Time Donation

The idea of using slack time to address SRT overruns stimulated considerable work in SRT scheduling. In systems that support HRT and SRT, some of the tasks must behave stably over time, e.g., physical system components must be scheduled by HRT scheduling algorithm to assure their correct control. Because of the computational complexity, the HRT portion of the workload is simplified into periodic tasks using WCET. This, in turn, is likely to introduce slack time that can be used to handle SRT task overruns. Several systems provide innovative ways of using slack time: within the same task (e.g., see [29, 34, 35]), across tasks (e.g., see [3, 4, 6, 17, 19, 24, 25, 31]), and across servers (e.g., see [10, 12]).

Recently Lin and Brandt described the BACKSLASH algorithm [23] that uses the idea of a task donating slack time backward to jobs that have already missed their deadlines [6, 31]. When a job overruns, it will be processed on the same server with the replenished budget and the extended deadline (the ISM technique). It also allows the overrun to be processed by using an OSM -like slack time donation. However, rather than using WCET, BACKSLASH uses mean execution time for the SRT resource reservation. As noted in Section 1, the BACKSLASH work establishes the foundation used for the work described in this paper. Specifically, we presume the four principles for managing slack time [23]:

1. Allocate slack as early as possible, with the priority of the donating job.
2. Allocate slack to the job with the highest priority (earliest **original** deadline).
3. Allow tasks to borrow against their own future resource reservations to complete their current job.
4. Retroactively allocate slack to jobs that have borrowed from their current budget to complete a previous job.

Our early release and slack time refinements are built on these principles, suggesting the need for refinements in performance metrics.

3 Early Release

BACKSLASH represents the state-of-the-art in ISM EDF scheduling [23]. We observed that BACKSLASH's workload does not allow the SRT jobs to be released early, which lowers the potential throughput. In this section we propose an algorithm in which jobs may be released early, thereby enabling a job that misses a deadline to recover, possibly by the time that the next job in the task completes.

Suppose that job_1 in a SRT task misses its deadline at the end of phase p_1 (see Figure 1). Further suppose that job_1 is able to finish its overrun processing soon after p_2 begins – in sufficient time for job_2 to start after its normal release time, yet still complete its processing before its deadline (and the release of job_3). That is, after a deadline is missed, the successive job should have a chance to be released as early as the finish time of its previous job within period p_2. The extreme case of the early release is that all jobs in the system do not overrun, so they can be released at the period where they were originally intended. We argue that releasing the next job early after a deadline miss improves the overall system throughput.

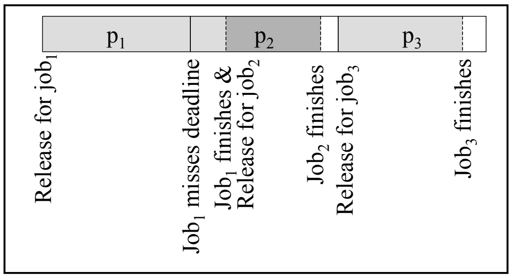

Figure 1: Early Release of a Job

However, there are several cases to consider prior to adopting this early release strategy. First, should the job always be released as early as possible? Second, if the early release causes a deadline miss again, should the next job be released as early as possible and create the effect of period shifting? Third, what is the best choice when a system is overloaded and the task itself is overloaded? Fourth, the early released job may win the competition over other jobs for the slack time donation, which causes those jobs to have higher potential to miss the deadline.

Here is one early release strategy:

- Give the option of early release to the user who can make the decision depending on the application requirements.
- A job is released early only after an earlier job has missed a deadline, but has completed prior to the next deadline. The early released job keeps the deadline defined by the execution time replenishment (i.e., the end of the next period) as well as the leftover execution time not used by its predecessor. This prevents a task from processing jobs faster than its predefined pace, e.g., it prevents such behavior as playing a ten minutes movie in thirty seconds.
- When a task is admitted, a portion of CPU time is reserved; even so, it may actually overload itself by overrunning often. If more than one SRT job overruns at the same time, the system may be overloaded and we can do nothing to remedy the situation. However, the early released job may receive the slack donation or have a shorter execution time to complete the job in time and catch up with its original pace. After a few deadline misses the application may start to show a reservation deficit, i.e., the application may need to consider dropping a job or lowering the task's quality of service.
- The early release may cause more deadline misses than without early release. However, the deadline misses alone do not really reflect the performance of the system. Without early release, there is no chance at all to complete all possible jobs in time. All tasks can still receive their reservation with early release enabled, but the early release significantly prevents the processor from idling by not sleeping voluntarily. As a result, all jobs can be completed relatively closer to the predefined period and the overall system throughput can be higher. Even though it will not interfere with other RT tasks' resource reservation, it may affect the amount of slack time that can be freely reallocated to help other tasks. Temporary system overload is allowed and cannot be avoided if we want to raise the system throughput/utilization, but we must prevent the system from long term overloading and minimize the impact of the short term overloading.

Some tasks need to process external-event driven workload generated by periodic interrupts from input devices. The device determines the period, and controls the job release times. When the next period starts, the device interrupts the processor to notify it the job arrival. If the previous job is not yet completed, it misses the deadline. For HRT systems, the deadline miss ordinarily triggers a failure recovery routine. A simple example would be using a robotic arm to assemble products on a transport belt in a factory. If the arm misses the target, the transport belt may be stopped by the recovery routine, waiting for a person to remove the product and restart the system. For SRT systems, double or triple buffers are often used to store the input data when deadlines are missed.

It is very likely that the interrupt handler will use one of the following methods to deal with an overrun. First, it may continue to process the current job and queue the new job. The current job can be handled on the same task server (ISM) or separate overrun server (OSM) (see Section 2.3). Second, it continues to process the current job and skips the new job. Third, it abandons the current job to process the new job. However, abandoning the current job or skipping the next job may not be the best choice because it may leave the system in an inconsistent state. Thus, the strategy choice depends on the nature of application, so we advocate giving the user the option to deal with the dilemma.

Another type of SRT task is that the application defines its own period and processes data according to that period to provide a certain level of QoS. For example a media player, like open source MPlayer, should play audio and video at the proper pace on machines with different speed and workload. If thirty frames per second are required, a job should be processed within one thirtieth of a second. However, the duration of decoding each frame varies. A scene cut or a key frame may take a longer time to decode, and skipping the current big frame may cause longer rippling effect on subsequent update frames. To solve the previous problem, it is better to try to finish processing the key frame using ISM even though it missed the deadline instead of abandoning it. To migrate the overrun job to an OSM server may delay completion of the job even further. After a deadline is missed, the new data should be buffered and wait for decoding. Even though the next job arrives at the time a deadline was missed, the media player usually deals with the problem by sleeping until the beginning of the next period or simply skipping the forthcoming job. A `wait_for_next_release()` system call can be provided to the regular real-time applications, so the developer does not need to check the deadline miss and to calculate the duration of sleep. They can simply call `wait_for_next_release()` and let the scheduler determine if an early release can be issued.

3.1 Preliminary Experimentation with Early Release

Lin and Brandt modified the Linux kernel to implement the BACKSLASH scheduler, and then tested it with an experimental workload [23]. We used a similar workload that reserves 2% of the processor cycles for BE tasks, leaving up to 98% to run real-time jobs. Three SRT jobs with utilization $u_1 = e_1/p_1 = {}^{160}/_{400} = 0.4$, $u_2 = e_2/p_2 = {}^{150}/_{500} = 0.3$, and $u_3 = e_3/p_3 = {}^{168}/_{600} = 0.38$ respectively are executed concurrently in the system for about seventeen seconds. For this workload, $u_1 = {}^{160}/_{400}$ means that the period is 400µs and the execution time is 160µs. Since SRT tasks are used, the $e_i <$ WCET values were used for the execution time reservation. We generate the normally distributed execution time using N(e, 0.1e), where mean = $e_1 = 160$ for $u_1 = e_1/p_1 = {}^{160}/_{400}$ case, and use u_1 for resource reservation. The same method is used for u_2 and u_3 reservations. Thus those tasks will overrun about 50% of the time.

The detailed traces of the execution of the above workload using pure BACKSLASH are shown in Tables 1-3, and the results of using BACKSLASH with early release are shown in Tables 4-6. In these experiments, the period i = 0 started at release time = 0, and the "release", "deadline", and "finish" fields in each table represent the relative time to the beginning of the first period. The "exe_time" represents the execution time generated by the normal distribution as described above, and the "x" in the status filed marks a deadline miss. Since the same workload uses the same seed for generating the execution time, we can use the same exe_time trace for comparison.

If we compare the first three periods of Tables 2 and 5, we observe that early release actually finished the second job (i = 1) within the second period corresponding to the example shown in Figure 1. A similar situation happens again at i = 31, so only 41 out of 43 jobs period are completed without early release as shown in Table 1. Table 4 shows that the system completes 43 jobs in 43 periods with early release. Table 3 shows that i = 34 completed the job whose execution time was 150790.30, but the same job can be completed at i = 30 in the corresponding early situation (see Table 6). We will provide additional interpretation of this data after introducing additional metrics in Section 4.

i	Release	Period	Deadline	Finish	Exe_time	Status
0	0	399852	399852	494417	176018	**X**
1	-	-	-	-	-	**X**
2	799704	399852	1199555	954039	152930	**.**
3	1199555	399852	1599407	1367958	168032	**.**
4	1599407	399852	1999259	1871471	168554	**.**
5	1999259	399852	2399111	2146608	147121	**.**
6	2399111	399852	2798963	2577074	176761	**.**
7	2798963	399852	3198814	3001548	131172	**.**
8	3198814	399852	3598666	3518419	165013	**.**
9	3598666	399852	3998518	3751712	152063	**.**
10	3998518	399852	4398370	4194220	194530	**.**
11	4398370	399852	4798222	4521656	122416	**.**
12	4798222	399852	5198074	4961035	161966	**.**
13	5198074	399852	5597925	5435848	172123	**.**
14	5597925	399852	5997777	5820645	180372	**.**
15	5997777	399852	6397629	6146472	142044	**.**
16	6397629	399852	6797481	6677230	163282	**.**
17	6797481	399852	7197333	6951701	152726	**.**
18	7197333	399852	7597184	7360912	163158	**.**
19	7597184	399852	7997036	7739315	140757	**.**
20	7997036	399852	8396888	8150649	153310	**.**
21	8396888	399852	8796740	8557288	159152	**.**
22	8796740	399852	9196592	9017870	178108	**.**
23	9196592	399852	9596443	9515151	171782	**.**
24	9596443	399852	9996295	9731681	134167	**.**
25	9996295	399852	10396147	10340849	193630	**.**
26	10396147	399852	10795999	10677197	164621	**.**
27	10795999	399852	11195851	10955202	158298	**.**
28	11195851	399852	11595702	11401133	165889	**.**
29	11595702	399852	11995554	11742982	146523	**.**
30	11995554	399852	12395406	12486380	173934	**X**
31	-	-	-	-	-	**X**
32	12795258	399852	13195110	12963798	***160843***	**.**
33	13195110	399852	13594961	13380421	183055	**.**
34	13594961	399852	13994813	13779302	154789	**.**
35	13994813	399852	14394665	14167769	172571	**.**
36	14394665	399852	14794517	14532401	137358	**.**
37	14794517	399852	15194369	15035532	144072	**.**
38	15194369	399852	15594221	15392946	159258	**.**
39	15594221	399852	15994072	15740103	144731	**.**
40	15994072	399852	16393924	16136795	141577	**.**
41	16393924	399852	16793776	16616445	167322	**.**
42	16793776	399852	17193628	16965667	170864	**.**

Table 1: No Early Release: Service Time = 160, Period = 400, and Utilization = 0.4

i	Release	Period	Deadline	Finish	Exe_time	Status
0	0	499815	499815	504262	165017	X
1	-	-	-	-	-	X
2	999630	499815	1499444	1144187	143372	.
3	1499444	499815	1999259	1861673	157530	.
4	1999259	499815	2499074	2350282	158019	.
5	2499074	499815	2998889	2864314	137926	.
6	2998889	499815	3498703	3508163	165713	X
7	-	-	-	-	-	X
8	3998518	499815	4498333	4312100	122974	.
9	4498333	499815	4998148	4763508	154699	.
10	4998148	499815	5497962	5258449	142559	.
11	5497962	499815	5997777	5999036	182372	X
12	-	-	-	-	-	X
13	6497592	499815	6997407	6612686	114765	.
14	6997407	499815	7497221	7149519	151843	.
15	7497221	499815	7997036	7839932	161365	.
16	7997036	499815	8496851	8352984	169099	.
17	8496851	499815	8996666	8834487	133166	.
18	8996666	499815	9496480	9166467	153077	.
19	9496480	499815	9996295	9788433	143181	.
20	9996295	499815	10496110	10666757	152960	X
21	-	-	-	-	-	X
22	10995925	499815	11495739	11229993	131960	.
23	11495739	499815	11995554	11847233	143728	.
24	11995554	499815	12495369	12309153	149205	.
25	12495369	499815	12995184	12663860	166976	.
26	12995184	499815	13494999	13173078	161046	.
27	13494999	499815	13994813	13836473	125782	.
28	13994813	499815	14494628	14344398	181528	.
29	14494628	499815	14994443	14885071	154332	.
30	14994443	499815	15494258	15178984	148405	.
31	15494258	499815	15994072	15812678	155521	.
32	15994072	499815	16493887	16269008	137365	.
33	16493887	499815	16993702	16775055	163063	.
34	16993702	499815	17493517	17270217	***150790***	.

Table 2: No Early Release: Service Time = 150, Period = 500, and Utilization = 0.3

i	Release	Period	Deadline	Fnish	Exe_time	Status
0	0	599778	599778	523698	184819	.
1	599778	599778	1199555	762245	160577	.
2	1199555	599778	1799333	1542641	176433	.
3	1799333	599778	2399111	2194442	176981	.
4	2399111	599778	2998889	2729002	154477	.
5	2998889	599778	3598666	3503000	185599	.
6	3598666	599778	4198444	3886919	137731	.
7	4198444	599778	4798222	4611054	173263	.
8	4798222	599778	5397999	5118171	159666	.
9	5397999	599778	5997777	5797611	204256	.
10	5997777	599778	6597555	6272533	128537	.
11	6597555	599778	7197333	6998217	170065	.
12	7197333	599778	7797110	7539528	180729	.
13	7797110	599778	8396888	8185278	189391	.
14	8396888	599778	8996666	8703932	149146	.
15	8996666	599778	9596443	9502701	171446	.
16	9596443	599778	10196221	9951648	160362	.
17	10196221	599778	10795999	10678752	171315	.
18	10795999	599778	11395777	11100477	147795	.
19	11395777	599778	11995554	11559659	160975	.
20	11995554	599778	12595332	12481800	167109	.
21	12595332	599778	13195110	13014324	187013	.
22	13195110	599778	13794887	13558450	180372	.
23	13794887	599778	14394665	13980737	140876	.
24	14394665	599778	14994443	14733357	203312	.
25	14994443	599778	15594221	15513892	172852	.
26	15594221	599778	16193998	15981692	166213	.
27	16193998	599778	16793776	16606447	174184	.
28	16793776	599778	17393554	17120401	153849	.

Table 3: No Early Release: Service Time = 168, Period = 600, and Utilization = 0.28

i	Release	Period	Deadline	Fnish	Exe_time	Status
0	0	399852	399852	494394	176018	X
1	399852	399852	799704	647281	***152930***	.
2	799704	399852	1199555	1305054	168032	X
3	1199555	399852	1599407	1649524	168554	X
4	1599407	399852	1999259	1947192	147121	.
5	1999259	399852	2399111	2445739	176761	X
6	2399111	399852	2798963	2576883	131172	.
7	2798963	399852	3198814	3073384	165013	.
8	3198814	399852	3598666	3390120	152063	.
9	3598666	399852	3998518	3884049	194530	.
10	3998518	399852	4398370	4145427	122416	.
11	4398370	399852	4798222	4624632	161966	.
12	4798222	399852	5198074	4979267	172123	.
13	5198074	399852	5597925	5435863	180372	.
14	5597925	399852	5997777	5741665	142044	.
15	5997777	399852	6397629	6162686	163282	.
16	6397629	399852	6797481	6605725	152726	.
17	6797481	399852	7197333	6969967	163158	.
18	7197333	399852	7597184	7478474	140757	.
19	7597184	399852	7997036	7781403	153310	.
20	7997036	399852	8396888	8156512	159152	.
21	8396888	399852	8796740	8576252	178108	.
22	8796740	399852	9196592	9051344	171782	.
23	9196592	399852	9596443	9371293	134167	.
24	9596443	399852	9996295	9838901	193630	.
25	9996295	399852	10396147	10165134	164621	.
26	10396147	399852	10795999	10555427	158298	.
27	10795999	399852	11195851	10977960	165889	.
28	11195851	399852	11595702	11434959	146523	.
29	11595702	399852	11995554	11895670	173934	.
30	11995554	399852	12395406	12157134	***160843***	.
31	12395406	399852	12795258	12830206	183055	X
32	12795258	399852	13195110	13310752	154789	X
33	13195110	399852	13594961	13656303	172571	X
34	13594961	399852	13994813	13943532	137358	.
35	13994813	399852	14394665	14140300	144072	.
36	14394665	399852	14794517	14563885	159258	.
37	14794517	399852	15194369	15062654	144731	.
38	15194369	399852	15594221	15368936	141577	.
39	15594221	399852	15994072	15882111	167322	.
40	15994072	399852	16393924	16219350	170864	.
41	16393924	399852	16793776	16693467	153574	.
42	16793776	399852	17193628	17013549	157298	.

Table 4: Early Release: Service Time = 160, Period = 400, and Utilization = 0.4

i	Release	Period	Deadline	Finish	Exe_time	Status
0	0	499815	499815	659331	165017	X
1	499815	499815	999630	802669	143372	.
2	999630	499815	1499444	1795180	157530	X
3	1499444	499815	1999259	2284816	158019	X
4	1999259	499815	2499074	2422748	137926	.
5	2499074	499815	2998889	2901940	165713	.
6	2998889	499815	3498703	3191836	122974	.
7	3498703	499815	3998518	3844326	154699	.
8	3998518	499815	4498333	4283248	142559	.
9	4498333	499815	4998148	4801947	182372	.
10	4998148	499815	5497962	5249919	114765	.
11	5497962	499815	5997777	5928833	151843	.
12	5997777	499815	6497592	6447506	161365	.
13	6497592	499815	6997407	6770006	169099	.
14	6997407	499815	7497221	7237860	133166	.
15	7497221	499815	7997036	7867069	153077	.
16	7997036	499815	8496851	8359739	143181	.
17	8496851	499815	8996666	8873316	152960	.
18	8996666	499815	9496480	9178557	131960	.
19	9496480	499815	9996295	9800067	143728	.
20	9996295	499815	10496110	10309254	149205	.
21	10496110	499815	10995925	10806878	166976	.
22	10995925	499815	11495739	11282866	161046	.
23	11495739	499815	11995554	11876536	125782	.
24	11995554	499815	12495369	12660548	181528	X
25	12495369	499815	12995184	13142633	154332	X
26	12995184	499815	13494999	13450884	148405	.
27	13494999	499815	13994813	14261437	155521	X
28	13994813	499815	14494628	14399205	137365	.
29	14494628	499815	14994443	15070570	163063	X
30	14994443	499815	15494258	15222063	***150790***	.
31	15494258	499815	15994072	15869352	171614	.
32	15994072	499815	16493887	16359277	145115	.
33	16493887	499815	16993702	16850182	161785	.
34	16993702	499815	17493517	17297022	128773	.

Table 5: Early Release: Service Time = 150, Period = 500, and Utilization = 0.3

i	Release	Period	Deadline	Fnish	Exe_time	Status
0	0	599778	599778	982564	184819	X
1	599778	599778	1199555	1293182	160577	X
2	1199555	599778	1799333	1629452	176433	.
3	1799333	599778	2399111	2584015	176981	X
4	2399111	599778	2998889	2738848	154477	.
5	2998889	599778	3598666	3532184	185599	.
6	3598666	599778	4198444	4020071	137731	.
7	4198444	599778	4798222	4620028	173263	.
8	4798222	599778	5397999	5136809	159666	.
9	5397999	599778	5997777	5779615	204256	.
10	5997777	599778	6597555	6446708	128537	.
11	6597555	599778	7197333	7106534	170065	.
12	7197333	599778	7797110	7562815	180729	.
13	7797110	599778	8396888	8218267	189391	.
14	8396888	599778	8996666	8722978	149146	.
15	8996666	599778	9596443	9487070	171446	.
16	9596443	599778	10196221	9997504	160362	.
17	10196221	599778	10795999	10642320	171315	.
18	10795999	599778	11395777	11123377	147795	.
19	11395777	599778	11995554	11593458	160975	.
20	11995554	599778	12595332	12481698	167109	.
21	12595332	599778	13195110	13473396	187013	X
22	13195110	599778	13794887	13973610	180372	X
23	13794887	599778	14394665	14258474	140876	.
24	14394665	599778	14994443	14914610	203312	.
25	14994443	599778	15594221	15539140	172852	.
26	15594221	599778	16193998	16045798	166213	.
27	16193998	599778	16793776	16537151	174184	.
28	16793776	599778	17393554	17168931	153849	.

Table 6: Early Release: Service Time = 168, Period = 600, and Utilization = 0.28

4 Refining the Deadline Miss Ratio Metric

The deadline miss ratio is often used to evaluate performance. However it uses a metric that may not be suitable for some workloads, including ones that incorporate early release. In many SRT algorithms, once a job completes a task it sleeps until the beginning of the next period. When the next period begins, the task's next job is released whether or not the earlier job missed its deadline. This approach is also used in other conventional software, e.g. the open source MPlayer has a function named `usec_sleep()` that takes a similar action. The idea is that when a job misses its deadline, the task is blocked until its next release. This means that subsequent jobs may be delayed for a full period. This situation is illustrated in Figure 2, which shows three periods (p_1, p_2, and p_3), where each period would normally have a job released at the beginning of the period. However, during p_2 no new job will be released because the task sleeps after its previous job overruns its service time during p_1, thereby forcing the job that would normally execute during p_2 to be released during p_3. Traditionally the *deadline miss ratio* (DMR) is calculated as

number of deadlines missed / jobs released

where a job that spans n periods counts for n − 1 deadline misses. As suggested by Figure 2, one could also interpret this situation as two deadline misses because the second job also missed its deadline because of the first job missed its deadline. Similarly, if a job spans three periods and delays its successor to the forth period, it could be counted as three misses. In other word, the number of the deadline misses could be calculated as the number of period a job spans if the task postpones the next job release by sleeping until a later period, i.e., a job spans n periods counts for n deadline misses instead of n − 1, where n > 1.

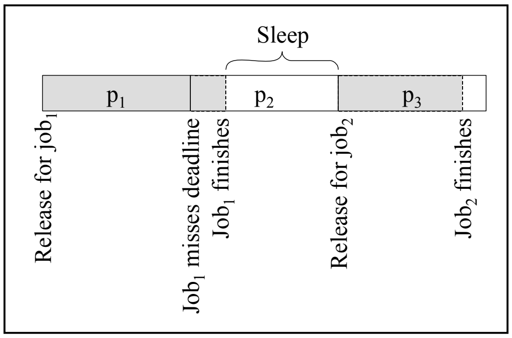

Figure 2: Traditional Deadline Miss Count

Another problem with the traditional DMR is that it uses the number of jobs actually released, instead of the number of possible jobs released, in the denominator. After a deadline miss, a task releases fewer jobs than it would have otherwise, and the metric reduces the DMR. For example, if a scheduler failed to schedule a task for 10 periods after it finished its first job prior to the deadline, it should have a DMR = $^{10}/_{11}$ instead of $^{0}/_{1} = 0$.

Further, suppose that an algorithm allows an SRT job to be released early: then there is no clear definition of traditional DMR. Notice that the DMR could be refined so that an additional metric such as throughput is used in conjunction with DMR. Then if early release of a job is allowed, the traditional DMR cannot be used as a metric. The traditional DMR in Figure 3 is $^{1}/_{3}$, but the DMR values for Figure 4 and Figure 5 are not defined. If the early release is considered as a normal release at the previously predefined time, the DMR for Figure 4 and Figure 5 may be calculated as 2/4 and 2/3 respectively. Since $^{1}/_{3} < ^{2}/_{4} < ^{2}/_{3}$, so the conclusion would then be that Figure 3 is better than Figure 4 which in turn is better than Figure 5.

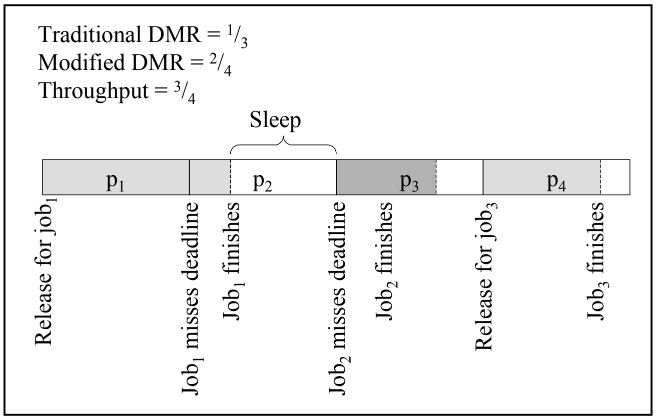

Figure 3: Deadline Miss Count without Early Release

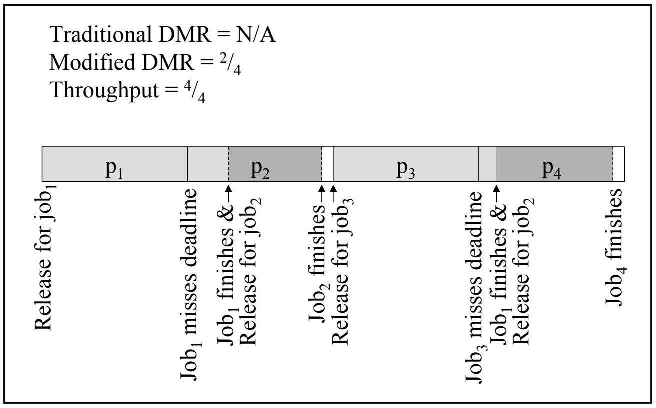

Figure 4: Deadline Miss Count with Early Release

How can the case shown in Figure 4 (which completed four jobs in four periods) be worse than Figure 3 that completed only three in the same amount of time? Based on this reasoning, we define the *Improved DMR* (IDMR) as

IDMR = deadline misses / maximum possible jobs

and use it together with the throughput for a more comprehensive comparison. The deadline misses are defined so that when a jobs spans n periods, there are n deadline misses instead of $n - 1$, where $n > 1$. The throughput is defined as

Throughput = jobs completed / maximum possible jobs

The three examples have the same IDMR = $^{2}/_{4}$, so throughput is used for the further comparison. The example in Figure 4 completed four jobs in four periods, so it is better than the other two that completed three jobs in four periods.

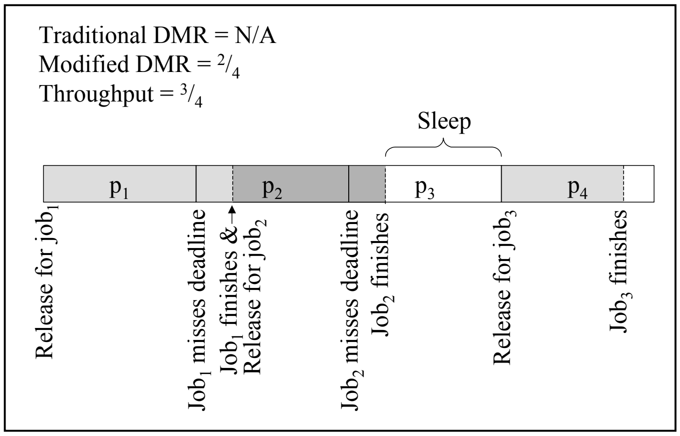

Figure 5: Deadline Miss Count with Early Release

In Table 7 IDMR is used to highlight that fact that early release results in improved performance. The table indicates the alternative measure of deadline misses as well as noting that the throughput of the individual task is higher (because early release does not waste CPU cycles by sleeping). The jobs are completed very close to the predefined pace without skipping or fast-forwarding with early release and the overall system throughput reaches 100% which is 5.66% better than no early release.

Measure	No Early Release			Early Release		
Period (μs)	400	500	600	400	500	600
Budget (μs)	160	150	168	160	150	168
Utilization (%)	40	30	38	40	30	38
Max Possible Jobs	43	35	28	43	35	28
Jobs Completed	41	31	28	43	35	28
Deadline Misses	4	8	0	7	7	5
IDMR (%)	9.3	22.85	0	16.28	20	17.86
Throughput (%)	95.35	85.57	100	100	100	100
Total Throughput	94.34			100		

Table 7: Early Release Increases the Throughput

Having higher overall IDMR is not really a bad thing from the throughput point of view. We ran several experiments with different mixture of tasks, and found that the result of early release depends strongly on the workload; we will investigate this phenomenon further in future work. If a system contains only HRT tasks, the early release will not change the behavior of EDF scheduling because no job can overrun by definition. When there are many SRT tasks with no HRT or far less HRT tasks, the system can be either long-term or transiently overloaded. For long-term overload, all real-time tasks can still get their reserved resource and the overloaded SRT task may behave like submitting jobs in a lower rate.

5 Refining Slack Time Scheduling Policies

In the BACKSLASH algorithm, slack time donation is based solely on the earliest original deadline [23]. In this section we demonstrate that in some cases performance can be improved by using slightly more general criteria for managing slack time. This points out the need to consider other importance/urgency criteria to generalize the approach.

Figure 6 illustrates the problems of both poor utilization estimation and an unfair slack time competition that can occur. The beginning of the shaded block marks the beginning of a period and the deadline of the previous period, and a gray block reflects the service time reserved for the task in a period.

The reserved service time is guaranteed within any period, but the actual usage depends on its dynamic priority. If the first job of $task_1$ requires much longer service time as illustrated by the first rectangle in Figure 6, it cannot finish the job at the original deadline, $d_{1,2}$, $d_{1,3}$, $d_{1,4}$, and so on. After the budget is exhausted in a period, it will be replenished with an extended deadline. If the extended deadline has highest priority, the job can be scheduled to run. Otherwise, it will wait for the slack donation from other tasks. Thus, its priority will eventually get lower and lower by deadline extensions and still desperately needs slack time. Its original deadline remains unchanged and is used to potentially compete with $d_{2,1}$, $d_{2,2}$, and $d_{2,3}$ of $task_2$ and $d_{3,1}$ of $task_3$ when slack is generated. The cross-hatched block pattern marked as "Slack" in Figure 6 represents the potential slack consumption. $Task_1$ will always win the slack time competition using BACKSLASH algorithm, because it has the earliest original deadline, $d_{1,1}$. In this situation, one rationale is for $task_1$ to be responsible for its own poor estimation of the service time (or intentional denial of service – DoS – attack) and should not prevent other tasks from fair slack time competition. For example, a task can submit the resource requirement of 5/100 (WCET/P), enter an infinite loop after a job release, and always obtain the slack donation because of its earliest original deadline.

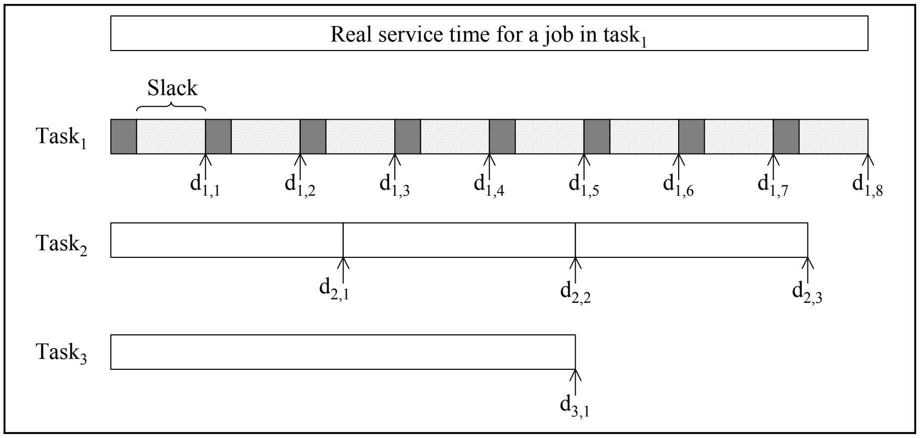

Figure 6: Non-fair Competition of Task 1 S Marks the Slack Time Competition Pattern

This can be addressed by monitoring task execution, and then choosing the best task for slack time donation using more information than the earliest original deadline. Besides providing the functionality of early release, the `wait_for_next_release()` system call is used to identify the boundary between jobs released by the same task. Because the user application calls the `wait_for_next_release()` after a job is completed, it does not need to manually calculate the sleep time and call the sleep function. Once the boundary of the jobs can be identified, the ratio

real service time of a job / est. service time of a job

can be calculated to identify extreme underbooking, i.e. a DoS attack. The higher ratio tells us that the task may be underbooking processor time. Even though its job has the earliest original deadline, the slack time will not be donated to it unless there is nothing else waiting for slack time. Another purpose of this API is for gathering the statistics, because the periodic server has no idea that the real-time job is enqueued by a new release or coming back from preemption without this API.

The problem can be viewed from another angle. If the slack is available and two jobs are waiting for the slack, we may choose the one that pays more money for the service or the one that is more likely to finish on time. The BACKSLASH algorithm schedules the overrun jobs using earliest original deadline as the dynamic priority for slack time, so it tries to meet the earliest deadline while giving the jobs with later deadlines more chances to compete for the slack time donation.

There are still problems with this approach (which we expect to address in future work). For example, since this approach does not use the earliest original deadline, there is no assurance that the scheduling algorithm is optimal.

6 Conclusion

This work builds on the work of Brandt, et al. relating to slack time scheduling. The technical report describes our first results in this extended work.

Early release is intended to take advantage of variation in the amount of time required to execute jobs within a task, thereby allowing a job that overruns to temporarily propagate missed deadlines within the task, but to also give the task an opportunity to get jobs back onto their original schedule. Our preliminary results on early release are encouraging.

The early release study indicates that the traditional deadline miss ratio does not necessarily measure the complete performance of the set of tasks. The IDMR measure, along with throughput, sheds additional light on early release performance. Like the ongoing work on early release, the work on performance metrics will also evolve.

Slack time scheduling could use arbitrary policies. Of course as policies become more complex, they are less attractive for scheduling due to their overhead. This work is inspired by the idea of generality, and by the problem of a task underbooking its reservation – effectively launching a denial of service attack on the other tasks in the system. This leads us to consider policies of measuring the amount of underbooking, and for penalizing tasks that dramatically underbook their reservation. This too is ongoing work.

Acknowledgements

We wish to thank Scott Brandt and Caixue Lin for their friendly cooperation in encouraging us to work in this area. They have provided us with their insight, results, unpublished code, and papers that have been invaluable in enabling us to do this work.

References

1. Abeni, Luca and Giorgio Buttazzo, “Integrating Multimedia Applications in Hard Real-time System,” *19th IEEE Real-time Systems Symposium*, 1998, pp. 4-13.
2. Anderson, David P., Shin-Yuan Tzou, Robert Wahbe, Ramesh Govindan, and Martin Andrews, “Support for Continuous Media in the DASH System, “ EECS Department, University of California at Berkeley Technical Report No. UCB/CSD-89-537, 1989.
3. Banachowski, Scott, Timothy Bisson, and Scott A. Brandt. Integrating best-effort scheduling into a real-time system. 25th IEEE Real-Time Systems Symposium, December 2004.
4. Brandt,S. A., S. Banachowski, C. Lin, and T. Bisson, “Dynamic Integrated Scheduling of Hard Real-time, Soft Real-time and Non-real-time Processes,” *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, Dec. 2003, pp. 396-407.
5. Banachowski, Scott and Scott Brandt, "The BEST Scheduler for Integrated Processing of Best-effort and Soft Real-time Processes," *Multimedia Computing and Networking 2002*, 2002, pp. 46–60.
6. Bernat, G., I Broster, and A. Burns, “Rewriting History to Exploit Gain Time,” *25th IEEE Real-time Systems Symposium*, 2004, pp. 328-335.
7. Brandt, Scott A, S. Banachowski, Casixue Lin and T. Bisson. “Dynamic Integrated Scheduling of Hard Real-time, Soft Real-time, and Non-real-time Processes,” *24th IEEE Real-time Systems Symposium*, 2003, pp. 396-407.
8. Brandt, Scott and Gary Nutt, "Flexible Soft Real-Time Processing in Middleware," *Real-Time Systems Journal, Special Issue on Flexible Scheduling in Real-Time Systems*, 22,1/2 (January-March 2002), pp. 77–118.
9. Buttazzo, Giorgio C., “Rate Monotonic vs. EDF: Judgment Day,” *Real-Time Systems*, 29, 1 (2005), pp 5–26.
10. Caccamo, Marco, Giorgio Buttazzo, and Lui Sha, “Capacity Sharing for Overrun Control,” *21st IEEE Real-time Systems Symposium*, 2000, pp. 295-304.
11. Caccamo, Marco, Giorgio Buttazzo, and Lui Sha, “Handling Execution Overruns in Hard Real-time Control Systems,” *IEEE Transactions on Computing*, 51, 7 (2002), pp. 835-849.
12. Caccamo, Marco, Giorgio C. Buttazzo, and Deepu C. Thomas, “Efficient Reclaiming in Reservation-based Real-time Systems with Variable Execution Times,” *IEEE Transactions on Computing*, 54, 2 (2005), pp. 198-213.
13. Deng, Z., J.W.-S Liu, and J. Sun, “A Scheme for Scheduling Hard Real-time Applications in Open System Environments,” *Ninth Euromicro Workshop on Real-Time Systems*, June 1997, pp 191-199.

14. Fall, K., and Pasquale, J., "Improving Continuous-Media Playback Performance With In-Kernel Data Paths", *Proceedings of the IEEE International Conference on Multimedia Computing and Systems* (ICMCS), pp. 100-109, Boston, MA, June 1994.
15. Gardner, M. K and J. W. S. Liu, "Perofrmance of Algorithms for Scheduling Real-time Systems with Overrun and Overload." *11th Euromicro Conference on Real-time Systems*, June 1999, pp. 287-296.
16. Ghazalie, T. M. and T. P. Baker, "Aperiodic Servers in a Deadline Scheduling Environment," *Real-time Systems*, Vol. 9, No. 1 (July 1995), pp. 31–67.
17. Goddard, S. and Xin Liu. A variable rate execution model. 16th Euromicro Conference on Real-Time Systems, pages 135 – 143, 2004.
18. Jones, M., Rosu, D., Rosu, M., "CPU Reservations and Time Constraints: Efficient Predictable Scheduling of Independent Activities", *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October, 1997.
19. Lamastra, G., G. Lipari, and L. Abeni, "A Bandwidth Inheritance Algorithm for Real-time Task Synchronization in Open Systems," *22nd IEEE Real-Time Systems Symposium*, 2001, pp. 151–160.
20. Lehoczky, J.P. and Ramos-Thuel, S., "An Optimal Algorithm for Scheduling Soft Aperiodic Tasks in Fixed-Priority Preemptive Systems," *IEEE Real-time Systems Symposium*, 1992, pp. 110-123.
21. Liu, C. L. and James W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment," *Journal of the ACM*, 20, 1 (1973), pp. 46-61.
22. Liu, Jane W. S., *Real-time Systems*, Pearson Education, Inc., Boston, MA, 2000.
23. Lin, Caixue and Scott A. Brandt, "Improving Soft Real-time Performance Through Better Slack Reclaiming," *26th IEEE Real-time Systems Symposium*, 2005, pp. 410.421.
24. Lipari, G. and S. Baruah, "Greedy Reclamation of Unused Bandwidth in Constant-bandwidth Servers," *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, June 2000, pp. 193–200.
25. Marzario, Luca, Guiseppe Lipari, Patricia Balbastre, and Alfons Crespo, "IRIS: A New Reclaiming Algorithm for Server-based Real-time Systems," *10th IEEE Real-time and Embedded Applications Symposium*, 2004, pp 211-218.
26. Mercer, Clifford W., Stefan Savage, and Hideyuki Tokuda, "Processor Capacity Reserves for Multimedia Operating Systems," School of Computer Science, Carnegie Mellon University Technical Report No. CMU-CS-93-157, May, 1993.
27. Mercer, Clifford W., Stefan Savage, and Hideyuki Tokuda, Processor Capacity Reserves: Operating System Support for Multimedia Applications, *Proceedings of the First IEEE International Conference on Multimedia Computing and Systems*, pp. 90-99, Boston, MA, May 1994.
28. Nieh, J., and Lam, M., "The design, implementation and evaluation of SMART: A Scheduler for Multimedia Applications", *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principle*s, October, 1997.
29. Rajkumar, R., K. Juvva, A. Molano, and S. Oikawa, "Resource Kernels: A Resource-centric Approach to Real-time and Multimedia Systems," *SPIE/ACM Conference on Multimedia Computing and Networking*, January, 1998.
30. Rodrigo, Santos, Guiseppe Lipari, and Jorge Santos, "Scheduling Open Dynamic Systems: The Clearing Fund Algorithm," *10th IEEE Real-time and Embedded Applications Symposium*, 2004, pp. 211-218.
31. Santos, Rodrigo, Giuseppe Lipari, and Jorge Santos, "Scheduling Open Dynamic Systems: The Clearing Fund Algorithm," *Proc. 10th International Conference on Real-Time Computing Systems and Applications*, 2004.
32. Siewert, Sam, Gary Nutt, and Elaine Hansen, "The Real-Time Execution Performance Agent: An Approach for Balancing Hard and Soft Real-Time Execution for Space Applications," *International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, June, 1999.
33. Sprunt, Brinkley, Lui Sha, and John Lehoczky, "Aperiodic Task Scheduling for Hard Real-time Systems," *Real-Time Systems*, Vol. 1, No. 1 (June 1989), pp.27–60.
34. Spuri, Marc and Giorgio C. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *Real-time Systems*, 10, 2 (1996).
35. Spuri, Marc, Giorgio C. Buttazzo, and Fabrizio Sensini, "Robust Aperiodic Scheduling Under Dynamic Priority Systems," *16th IEEE Real-time Systems Symposium*, 1995, pp. 210-221.
36. Stankovic, John A., Marc Spuri, Marco Di Natale, and Giorgio G. Buttazzo, "Implications of Classical Scheduling Results for Real-time Systems," *IEEE Computer*, 28, 6 (June 1995), pp. 16-25.

37. Tokuda, Hideyuki, Tatsuo Nakajima, and Prithvi Rao. "Real-Time Mach: Towards Predictable Real-Time Systems," *Proceedings of the USENIX 1990 Mach Workshop*, October 1990.