GROUP-EDF: A NEW APPROACH AND AN EFFICIENT NON-PREEMPTIVE ALGORITHM FOR SOFT REAL-TIME SYSTEMS

Wenming Li, B.S., M.S.

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

August 2006

Li, Wenming, Group-EDF: A new approach and an efficient non-preemptive algorithm for soft real-time systems. Doctor of Philosophy (Computer Science), August 2006, 124 pp., 7 tables, 49 illustrations, references, 48 titles.

Hard real-time systems in robotics, space and military missions, and control devices are specified with stringent and critical time constraints. On the other hand, soft real-time applications arising from multimedia, telecommunications, Internet web services, and games are specified with more lenient constraints. Real-time systems can also be distinguished in terms of their implementation into preemptive and non-preemptive systems. In preemptive systems, tasks are often preempted by higher priority tasks. Non-preemptive systems are gaining interest for implementing soft-real applications on multithreaded platforms.

In this dissertation, I propose a new algorithm that uses a two-level scheduling strategy for scheduling non-preemptive soft real-time tasks. Our goal is to improve the success ratios of the well-known earliest deadline first (EDF) approach when the load on the system is very high and to improve the overall performance in both underloaded and overloaded conditions. Our approach, known as group-EDF (gEDF), is based on dynamic grouping of tasks with deadlines that are very close to each other, and using a shortest job first (SJF) technique to schedule tasks within the group. I believe that grouping tasks dynamically with similar deadlines and utilizing secondary criteria, such as minimizing the total execution time can lead to new and more efficient real-time scheduling algorithms. I present results comparing gEDF with other real-time algorithms including, EDF, best-effort, and guarantee scheme, by using randomly generated tasks with varying execution times, release times, deadlines and tolerances to missing deadlines, under varying workloads. Furthermore, I implemented the gEDF algorithm in the Linux kernel and evaluated gEDF for scheduling real applications.

ACKNOWLEDGEMENTS

First, I am deeply grateful to Dr. Kavi, my major professor, for his consistent assistance and brilliant supervision since 2002. Without which, I would not have achieved significant progress. He enlightened me to choose a new approach that I initially used in developing a real-time extension of the Scheduled Dataflow Architecture. He encouraged me to develop the innovative idea into a complete real-time algorithm.

I would like to thank Dr. Akl for providing the modeling and simulation tools, and his participation in weekly research meetings for the last three years. I would like to thank Dr. Akl and Dr. Sweany for their suggestions and supervision on my doctoral study and dissertation.

I truly appreciate all the professors, staff, and students in the Department of Computer Science and Engineering at the University of North Texas, who have given me much help since 1999.

Last but not least, I owe my parents, my wife, and my daughter so much. I particularly thank them for their strong support during these years.

LIST OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The earliest deadline first (EDF) algorithm is the most widely used scheduling algorithm for real-time systems on uniprocessors and multiprocessors [1, 2]. All the discussions in this dissertation will focus on uniprocessors instead of multiprocessors, although our approach can be extended to multiprocessors. Real-time applications can be characterized as hard real-time or soft real-time systems. Hard real-time applications require that all time constraints be met, while soft real-time systems permit some tolerance in meeting time constraints. Real-time systems are also distinguished based on their implementation. In preemptive systems, tasks may be preempted by higher priority tasks, while non-preemptive systems do not permit preemption. It is easier to design preemptive scheduling algorithms for real-time systems. It is our contention, however, that non-preemptive scheduling is more efficient, particularly for soft real-time applications and applications designed for multithreaded systems, than the preemptive approach since the non-preemptive model reduces the overhead needed for switching among tasks or threads [3, 4]. For a set of preemptive tasks (be periodic, aperiodic, or sporadic), EDF will find a schedule if a schedule is possible [5]. The application of EDF for non-preemptive tasks is not as widely studied. EDF is optimal for sporadic non-preemptive tasks, but EDF may not find an optimal schedule for periodic and aperiodic non-preemptive tasks. It has been shown that scheduling periodic and aperiodic non-preemptive tasks is NP-hard (Non-deterministic Polynomial-time hard) [6, 7, 8]. However, non-preemptive EDF techniques have produced near optimal schedules for periodic and aperiodic tasks, particularly when the system is lightly loaded. When the system is overloaded, it has been shown that the EDF approach leads to very poor performance (i.e., low success rates) [9]. In this dissertation, a system load or utilization is used to refer to the ratio of the sum of the execution times of pending tasks and the time available to complete the tasks. The poor performance of EDF is due to the fact that, as tasks that are scheduled based on their deadlines miss their deadlines, other tasks waiting for their turn are likely to miss their deadlines also – an outcome sometimes known as the domino effect. It should also be remembered that worst-case execution time (WCET) estimates for tasks are used in most real-time systems. We believe that, in practice, WCET estimates are very conservative, and more aggressive scheduling scheme based on average execution times for soft real-time systems using either EDF or hybrid algorithms can lead to higher resource utilizations.

The major contributions of my research are described here. I propose a new approach for scheduling soft real-time systems. To our knowledge we are the first research team to propose grouping of task dynamically into deadline groups and then use a two level scheduling to schedule tasks. I use EDF for scheduling groups and use a different technique (e.g., shortest job first) for scheduling tasks within a group.

While investigating scheduling algorithms, I have analyzed a variation of EDF that can improve the success ratio (that is, the number of tasks that have been successfully scheduled to meet their deadlines), particularly in overloaded conditions. The new algorithm can also decrease the average response time for tasks. We call the algorithm group-EDF, or gEDF, where the tasks with "similar" deadlines are grouped together (i.e., deadlines that are very close to one another), and the shortest job first (SJF) algorithm is used for scheduling tasks within a group. It should be noted that our approach is different from adaptive schemes that switch between different scheduling strategies based on system load; gEDF is used in overloaded as well as underloaded conditions. The computational complexity of gEDF is approximately the same as that of EDF. In this dissertation, I will evaluate the performance of gEDF using randomly generated tasks with varying execution times, release times, deadlines and tolerances to missing deadlines, under varying loads. I have also implemented the gEDF algorithm in the Linux kernel. I will present performance results of our implementation for some real-time benchmarks.

We believe that gEDF is particularly useful for soft real-time systems as well as applications known as "anytime algorithms" and "approximate algorithms," where applications generate more accurate results or rewards with increased execution times [10, 11]. Examples of such applications include search algorithms, neural-net based learning in AI (Artificial Intelligence), FFT (Fast Fourier Transform) and block-recursive filters used for audio and image processing. I model such applications using a tolerance parameter that describes by how much a task can miss its deadline, or by how much the task's execution time can be truncated when the deadline is approaching.

This dissertation is organized as follows. In chapter 2, I present related work. In chapter 3, I present the real-time system model and a formal description of the gEDF algorithm. Extensive experiments and numerical results for evaluating the performance of gEDF are presented in chapter 4. The implementation of the gEDF algorithm in Linux operating system is presented in chapter 5. Conclusions are given in chapter 6. In appendices A and B, I provide guidelines on how to implement new real-time scheduling algorithms in the modified Linux kernel.

CHAPTER 2

RELATED WORK

2.1 Overview

The earliest deadline first (EDF) algorithm schedules real-time tasks based on their deadlines. Because of its optimality for periodic, aperiodic, and sporadic preemptive tasks, its optimality for sporadic non-preemptive tasks, and its acceptable performance for periodic and aperiodic non-preemptive tasks, EDF is widely studied as a dynamic priority-driven scheduling scheme [6]. EDF is more efficient than many other scheduling algorithms, including the static rate-monotonic (RM) scheduling algorithm [12]. For preemptive tasks, when the system is lightly loaded, EDF is able to reach the maximum possible processor utilization. Although finding an optimal schedule for periodic and aperiodic non-preemptive tasks is NP-hard [7, 8], our experiments have shown that EDF can achieve very good utilization even for non-preemptive tasks when the system is lightly loaded. However, when the processor is over-loaded (i.e., the combined requirements of pending tasks exceed the capabilities of the system) EDF performs poorly. Researchers have proposed several adaptive techniques for handling heavily loaded situations, but they require the detection of the overload condition.

A best-effort algorithm [9] is based on the assumption that the arrival probability of a high value-density task is low. The value-density is defined by *V*/*C*, where *V* is the value of a task and *C* is its worst-case execution time. Given a set of tasks with defined values if completed successfully, it can be shown that a sequence of tasks in decreasing order by value-density will produce the maximum value when compared to any other scheduling technique. The best-effort algorithm admits tasks based on their value-densities and schedules them using the EDF policy. When higher value tasks are admitted, some lower value tasks may be deleted from the schedule or delayed until no other tasks with higher value exist. One key consideration in implementing such a policy is the estimation of current workload, which is either very difficult or very inaccurate in most practical systems that utilize worst-case execution time (WCET) estimations. WCET estimation requires complex analysis of tasks [13, 14], and the estimates are significantly larger than average execution times of tasks. Thus the best-effort algorithm that uses WCET to estimate loads may lead to sub-optimal value realization. Best-effort has been used as an overload control strategy for EDF: that is, EDF is used when a system is underloaded but best-effort is used when the overload condition is detected. One integrated real-time scheduler including best-effort strategy for general-purpose operating systems has been proposed in [15]. However, this approach relies on preemptive scheduling and uses best-effort as an overload control strategy.

Other approaches for detecting overload and rejecting tasks were reported in [16, 17]. In the guarantee scheme [16], the load on the processor is controlled by acceptance tests on new tasks entering the system. If the new task is found schedulable under worst-case assumptions, it is accepted; otherwise, the arriving task is rejected. In the Robust scheme [17], if the system is underloaded, the acceptance test is based on EDF; if the system is overloaded, one or more tasks may be rejected based on their importance. Because the guarantee and Robust algorithms also rely on computing the schedules of tasks that are based on worst-case estimates, they usually lead to underutilization of resources. Thus best-effort, guarantee, or Robust scheduling algorithms are not good for soft real-time systems or applications that are generally referred to as “anytime” or “approximate” algorithms [11]. For these algorithms, the quality of results (or accuracy) improves when more computation time is allowed.

The combination of SJF and EDF, referred to as SCAN-EDF for disk scheduling, was proposed in [18]. In this work, SJF is only used to break a tie between tasks with identical deadlines. The research reported in [19, 20] is very closely related to our idea of groups. This approach quantizes deadlines into deadline bins and places tasks into these bins. However, tasks within a bin (or group) are scheduled using first come first served (FCFS) policy. The gEDF groups that I use are created dynamically instead of statically as done in [19, 20].

In the following sections, I will introduce the milestone events and the key algorithms in the development of real-time systems, especially those that are related to our approach. In section 2.7, I will introduce the development of real-time systems. In section 2.8, I will introduce real-time operating systems.

2.2 FIFO/FCFS, RR, SJF: Basic Real-Time Scheduling

First come first served (FCFS) scheduling uses a simple “first in, first out” queue. It is simple to implement but it has several deficiencies. Its average wait times are typically long. It is a non-preemptive scheduling technique, and it is subject to the negative effect if there are many I/O bound processes mixed with a few CPU bound processes. In such cases, there can be large amounts of idle times as the I/O bound processes sit idle waiting for the CPU bound process to complete.

Round robin (RR) scheduling is similar to FCFS scheduling but preemption is added, so that on each time quantum a new process receives access to the system resources. This way, each process gets a share of the system resources without having to wait for all processes ahead of it to run to completion. The average waiting time is typically long and its performance is proportional to the size of the time quantum. Round robin scheduling is the degenerative case of priority scheduling when all priorities are equal.

Shortest job first (SJF) scheduling is probably optimal but requires clairvoyance, profiling, or expected execution time to fully implement. SJF can be implemented either preemptively or non-preemptively. SJF has low average waiting time. In fact, SJF is optimal with respect to average waiting time.

It is very easy to prove this claim by comparing it with other real-time algorithms. A formal proof can be found in [21, 22].

Although FIFO/FCFS, RR, and SJF are very basic real-time scheduling schemes, they are widely implemented in real-time systems.

2.3 Static Priority Scheduling: Rate-Monotonic

Classical scheduling theory deals with static scheduling. Static scheduling refers to the fact that the scheduling algorithm has complete information regarding the task set including knowledge of deadlines, execution times, precedence constraints, and release times.

In rate-monotonic (RM) scheduling, the shorter the period of a task, the higher is its priority. If there is a set of n independent periodic tasks, and a task τ_i ($1 \leq i \leq n$) is characterized by a period P_i ($1 \leq i \leq n$) and a worst-case execution time e_i, we have the following result.

A set of n independent, periodic jobs can be scheduled by the rate monotonic policy if $e_1/P_1 + e_2/P_2 + \ldots + e_n/P_n \leq n\ (2^{1/n} - 1)$. However, by this formula, the upper bound on utilization is only ln2 = 0.69 as n approaches infinity.

The detailed description and proof can be found in [5]. A better result was provided in [23], which claims that the upper bound on utilization of RM is 88%.

In deadline-monotonic (DM) scheduling, the shorter the relative deadline (i.e. the difference between the deadline and the current time, also known as the laxity) of a task, the higher is its priority. This approach investigates schedulability tests for sets of periodic tasks whose deadlines are permitted to be less than their period. Such a relaxation enables sporadic tasks to be directly incorporated with periodic tasks [24, 25]. For arbitrary relative deadlines, DM outperforms RM in terms of utilization.

I use dynamic scheduling approaches in my work. However, I also include a description of static scheduling methods for the sake of completeness.

2.4 Dynamic Priority Scheduling: EDF

Earliest deadline first (EDF) scheduling is one of the first dynamic priority-driven scheduling algorithms proposed. As the name implies, tasks are selected for execution in the order of their deadlines. It provides the basis for many of the real-time algorithms. EDF suffers significantly when the system is overloaded. Compared to static priority driven scheduling such as RM with approximate 69% utilization, EDF can approach 100% utilization for periodic jobs.

In the convention of the scheduling theory [22], I first give the specification of scheduling problems. Classes of scheduling problems are specified in terms of a three tuple: α | β | γ where α specifies the machine environment, β specifies the job characteristics, and γ denotes the objective criterion. The value is either = 1, for uniprocessor environment or P, for a multiprocessor environment with P processors. The job characteristics are specified by β containing the elements such as pmtn (i.e. preemptive scheduling) or nonpmtn (i.e. non-preemptive scheduling), release time r_i (i.e. arriving time of a job), independent tasks, precedence constrained tasks, etc. The objective criteria can be minimize maximum lateness L_{max} = max {L_i | i = 1, …, n} = max {C_i - d_i | i = 1, …, n}, where, max is the function of maximum, d_i is defined as a due date or deadline of job τ_i, C_i is defined as completion time of job τ_i, or minimizing the makespan max{C_i | i = 1, …, n}.

Suppose that there are n independent jobs. The problem: 1 | nonpmtn | L_{max}, (where, 1 stands for a uniprocessor, nonpmtn stands for non-preemptive scheduling, and L_{max} stands for the objective that is to minimize L_{max} = max {L_i | i = 1, …, n}) has a solution: Any sequence of jobs in nondecreasing order of due dates d_i, results in an optimal schedule [26].

Another important result is that if r_i is the release time of a job, then the scheduling problem {1 | nonpmtn, r_i | L_{max}} is NP-hard [27].

This result is important to our approach because the gEDF algorithm deals with non-preemptive scheduling of tasks based on release times. I will introduce the gEDF algorithm in detail in a later chapter. Since generating an optimal schedule is NP-hard, I conducted extensive experiments to analyze the performance of heuristic gEDF algorithm.

If job preemption is allowed at any instant, the problem: {1 | pmtn, r_i | L_{max}} has a polynomial solution. Any technique that at any instant schedules a job with the earliest deadline among all the eligible jobs (i.e. those with release time less than or equal to the current times) is optimal with respect to minimizing maximum lateness [29], since a currently running task can be preempted in favor of scheduling another task with a shorter relative deadline.

This result is the basis of preemptive EDF and least laxity first (LLF) algorithms. That is because the value of minimized maximum lateness can be any value. We can let $L_{max} = \max \{C_i - d_i \mid i = 1, \ldots, n\} = 0$, that is, all deadlines of tasks must be met. According to the result, it implies that there is always an optimal schedule. In fact, specifically, for a set of n independent periodic processes, EDF scheduling shows that 100% processor utilization is achievable if and only if $e_1/P_1 + e_2/P_2 + \ldots + e_n/P_n = 1$ [5].

EDF can be applied to periodic, aperiodic, and sporadic real-time jobs. However, EDF performs poorly in overloaded conditions [28]. LLF behaves similar to EDF except that LLF needs to compute execution times of jobs when scheduling.

It should be noted that dynamic scheduling doesn't mean online scheduling. An online scheduling algorithm has only complete knowledge of the currently active set of tasks, and no knowledge of any new arriving tasks. Likewise, offline scheduling is not the same as static scheduling. Offline includes pre-analysis of scheduling regardless of whether the runtime algorithm is static or dynamic. Our gEDF algorithm is a dynamic one, which can be either online or offline, depending on the selection of the types of real-time jobs involved. Usually, offline scheduling has higher performance than online scheduling but may lead to poorer utilization.

2.5 Non-preemptive Scheduling

In many practical real-time scheduling problems such as I/O scheduling, properties of device hardware and software make preemption either impossible or prohibitively expensive. Non-preemptive scheduling algorithms are easier to implement than preemptive algorithms, and can exhibit dramatically lower overhead at run-time. The overhead of preemptive algorithms is more difficult to characterize and predict than that of non-preemptive algorithms. Non-preemptive scheduling on a uniprocessor naturally guarantees exclusive access to shared resources and data, thus eliminating both the need for synchronization and its associated overhead. The problem of scheduling all tasks without preemption forms the theoretical basis for more general tasking models that include sharing of resources.

EDF scheduling for aperiodic jobs is NP-complete. However, for sporadic jobs, non-preemptive EDF scheduling is optimal; for the scheduling of periodic jobs, it is NP-hard [6].

2.6 Real-Time Algorithm Metrics

The most important metric of a real-time system is the success ratio of system deadlines. By success ratio we mean the percentage of jobs completed before their deadlines. However, other metrics, such as the minimized total (or weighted sum) of the execution times of real-time jobs, the minimized average response time, the minimized maximum lateness or tardiness of real-time jobs, and the minimized number of processors required for real-time jobs, may be important for real-time systems, especially for soft real-time systems. Because missing a few deadlines is not critical in soft real-time systems, the overall performance becomes important. Using these metrics (in addition to success ratios) is often overlooked in many real-time systems. Minimizing total or average execution time has secondary importance in helping minimize resource requirements for a system. However, minimizing execution time does not directly address the fact that individual tasks have deadlines. In fact, there is no direct relation between preventing missing deadlines and maximizing or minimizing these values. For instance, minimizing the maximum lateness metrics can be useful at design time where resources can be continually added until the maximum lateness ≤ 0 (i.e., no deadline is missed). In this particular case, no tasks miss their deadlines. However, generally, the metric is not always useful because minimizing the maximum lateness doesn't necessarily prevent tasks from missing their deadlines. Some related work and algorithms can be found in [29].

Another concept that often appears in the real-time literature is the optimality of an algorithm. We say that a scheduling algorithm is optimal if no other scheduling algorithm can find a better solution for the same scheduling problem.

2.7 Real-Time Systems

In this section and section 2.8, I will introduce real-time systems and real-time operating systems (RTOS). Although I intend to focus only on scheduling in real-time systems, an overview of real-time systems and RTOS will be helpful to the implementation of a new scheduling algorithm in real-time systems and RTOS.

Before I discuss real-time operating systems, I first define a real-time system. According to the IEEE (Institute of Electrical and Electronics Engineers), a real-time system is a system whose correctness includes its response time as well as its functional correctness.

2.7.1 Hard Real-Time Systems

Hard real-time means that the system must be designed to guarantee all time constraints. Every resource management system such as the scheduler, I/O manager, and communications, must work in the correct order to meet time constraints. Military applications and space missions are typical instances of hard real-time systems.

Here are some applications with real-time requirements: telecom switching, car navigation, the medical instruments with the critical time constraints, rocket and satellite control, aircraft control and navigation, industrial automation and control, and robotics.

2.7.2 Soft Real-Time Systems

Soft real-time systems are similar to hard real-time systems in their infrastructure requirements, but it is not necessary that every time constraint be met. In other words, some time constraints are not strict but they are nonetheless important. Soft real-time is not equivalent to non-real-time, since the goal of the systems is still to meet as many deadlines as possible.

Here are some applications with soft real-time requirements: web services such as real-time query, call admittance in voice over IP and cell phone, digital TV transmissions, cable and digital TV set-top-boxes, video conferencing, TV broadcasting, games, and gaming equipment. Even in some typical hard real-time systems, some functions have soft real-time constraints. For instance, in Apollo 11 mission, there are two sets of real-time subsystems, one with hard real-time, and one with soft real-time deadlines.

2.7.3 Scheduling in Real-Time System

Figure 2.1 shows the taxonomy of real-time scheduling in real time systems. EDF/gEDF algorithms studied in this research are applicable to the shaded region (i.e., soft, dynamic, and non-preemptive).

Soft	Dynamic	Non-preemptive
		Preemptive
	Static	Non-preemptive
		Preemptive
Hard	Dynamic	Non-preemptive
		Preemptive
	Static	Non-preemptive
		Preemptive

Figure 2.1: The taxonomy of real-time scheduling. Our EDF/gEDF algorithm is applicable to the shaded region.

2.7.4 Priority in the EDF Scheduling

EDF usually is implemented as a dynamic priority driven scheduling scheme. Usually, the number of priority levels in real-time systems should be at least 32. As we know, EDF schedules real-time jobs based on deadlines. Because the number of different deadline values could be a large number, we cannot use a priority level to represent each different deadline. One solution is to use a base priority and several dynamic priorities. The latter is dependent on the deadline of a real-time job. The base priority can be assigned statically or dynamically by the programmer or the system. The deadline of a real-time job determines its priority among all the real-time jobs with the same level of base priority. Thus, actually, there are two kinds of priorities for EDF based for real-time jobs.

2.8 Real-Time Operating Systems

2.8.1 The Requirements of RTOS

A real-time operating system (RTOS) [30] is not simply a real-time system. It is the core part of any real-time system. A real-time system includes all the system elements such as hardware, middleware, applications, communications and I/O devices. All the elements are needed to meet the system requirements. However, RTOS provides sufficient functionality to enable a real-time application to meet its requirements. It is also important to distinguish between a fast operating system and a RTOS. Speed, although useful for meeting the overall requirements, by itself is not sufficient to determine whether a system meets the requirements for an RTOS.

2.8.2 POSIX 1003.1 for RTOS

The IEEE computer society's portable application standards committee (PASC) defined a standard for Portable Operating System Interface (POSIX) [31, 32]. This IEEE Standard 1003.1 includes IEEE Standard 1003.1a, IEEE Standard 1003.1b, and 1003.1c, IEEE Standard 1003.1d/j/q, and IEEE Standard 1003.13. IEEE Standard 1003.1a is the base for all the POSIX standards. IEEE Standard 1003.1b (formerly POSIX 1003.4) defines the needed real-time extensions. IEEE Standard 1003.1c defines the functionality of threads. These various standards have been combined by the Austin Group in producing IEEE standard 1003.1-2001. The latest version is now known as the IEEE 1003.1 2004 Edition.

POSIX 1003.1b provides the standard criteria for RTOS services and is designed to allow programmers to write applications that can easily be ported to any operating systems (OS) that is POSIX compliant. The basic RTOS services covered by POSIX 1003.1b include asynchronous I/O, synchronous I/O, memory locking, semaphores, shared memory, timers, inter-process communication (IPC), real-time files, real-time threads, and scheduling.

Real-time scheduling is the most important feature of a RTOS. POSIX 1003.1b specifies the following scheduling policies.

SCHED FIFO - Priority based preemptive scheduling, FIFO is used among tasks with the same priority.

SCHED RR - Processes with same priority use Round Robin policy. A process executes for a quantum of time; and then it is moved to the end of the queue corresponding to its priority level. Higher priority tasks can preempt tasks. The size of the quantum can be fixed, configurable, or specific for each priority level.

SCHED OTHER - Availability required but not defined by the standard. Usually SCHED OTHER is implemented as a classical time-sharing policy.

As additions to IEEE POSIX scheduling policies, I implemented SCHED EDF and SCHED gEDF policies for non-preemptive real-time tasks on the Linux kernel to evaluate our scheduling techniques.

2.8.3 RTOS Examples

a. Microsoft Windows CE – Non-Linux Based Commercial RTOS

Microsoft Windows CE is designed as a general-purpose and portable real-time operating system for small memory, 32-bit mobile devices. Windows CE slices CPU time among threads and provides 256 priority levels. To optimize performance, all threads are enabled to run in kernel mode. All non-preemptive portions of the kernel are broken into small sections reducing the duration of non-preemptive code. Windows CE incurs long latencies for tasks.

b. VxWorks – Commercial RTOS

VxWorks, by Wind River Systems, is a real-time operating system. It runs currently on its own kernel. However, its development is done on a host machine such as Linux or Windows. Its cross-compiled target software can be run on various target CPU architectures. VxWorks runs in supervisor mode, and does not use traps for system calls. VxWorks supports priority interrupt-driven preemption and optional round-robin time slicing. The micro kernel supports 256 priority levels. VxWorks supports some of the IEEE POSIX 1003.1 functions.

c. LynxOS – POSIX Compatible Commercial RTOS

LynxOS is a POSIX compatible, multithreaded OS designed for complex real-time applications that require fast, deterministic response. It is scalable from small, embedded products to large switching systems. The micro-kernel can schedule, dispatch interrupts, and synchronize tasks. It uses scheduling policies such as prioritized FIFO, dynamic deadline monotonic (DDM, the shorter the dynamic deadline, the higher is its priority) scheduling, time-slicing, etc. It has 512-priority levels and supports remote console and remote monitoring. For instance, LynxOS can be used as a hard real-time system for controlling gas levels in chemical plants remotely.

d. RTLinux – Open Source Linux-Based RTOS

RTLinux [33] is a hard real-time operating system that runs Linux as its lowest priority thread. The Linux thread is completely preemptible so that real-time threads and interrupt handlers are never delayed by non-real-time operations. Real-time applications can make use of all the powerful, non-real-time services of Linux. RTLinux scheduling policies supports EDF. RTLinux, originally developed at the New Mexico Institute of Technology, is an open-source product. RTLinux-specific components are released under the GNU (a recursive acronym for “GNU's Not UNIX”) General Public License (GPL), and Linux components are released under the standard Linux license. The source code is freely distributed. Non-GPL versions of the RTLinux components are available from FSMLabs [33].

e. RED-Linux – Open Source Linux-Based RTOS

RED-Linux [34] is an open-source real-time and embedded version of Linux version 2.2.14. In addition to the original Linux capability, it improves the real-time behaviors of the Linux kernel in many ways. RED-Linux supports a short kernel blocking time, a quick task response time and, modularized runtime general scheduler interface (GSI) so that different scheduling methods can be selected depending on the application.

f. KURT-Linux – Open Source Linux-Based RTOS

KU real-time Linux (KURT) [35] is a Linux system with real-time modifications to allows scheduling of real-time events at 10's of microseconds resolution. Rather than relying on priority based scheduling or strictly periodic schedules, KURT relies on application specified schedules. KURT can function in two modes: focused mode, where only real-time processes are allowed to run; and mixed mode, where the execution of real-time processes still takes precedence, but non-real-time processes are allowed to run when real-time tasks are not running. KURT was developed by the Information and Telecommunication Technology Center (ITTC) at the University of Kansas. KURT may be used and distributed according to the terms of the GNU Public License.

In this chapter, I reviewed research and technologies that are closely related to our research. In particular, I presented some well known real-time scheduling techniques, the underlying limitations of the algorithms, and practical real-time operating systems that are currently available.

In the next chapter, I will introduce our real-time scheduling algorithm.

CHAPTER 3

REAL-TIME SYSTEM MODEL

3.1 Definitions

A job in a real-time system, a thread in multithreading processing, or a task in single threaded systems, τ_i, is defined as $\tau_i = (r_i, e_i, D_i, P_i)$; where r_i is its release time (or its arrival time); e_i is either its estimated worst-case or its estimated average execution time; and D_i is its deadline. We also maintain a dynamic deadline d_i with an initial value $r_i + D_i$, which tracks the absolute time before the deadline expires. In other words D_i is the relative deadline of the job with respect to the arrival time and d_i is the absolute (wall clock) deadline.

If modeling periodic jobs, P_i defines a task's periodicity. Note that aperiodic and sporadic jobs can be modeled by setting P_i appropriately. For instance, an aperiodic job can be modeled by setting P_i to a variable; a sporadic job by setting P_i to a variable that is greater than a minimum value.

Figure 3.1 graphically shows the relationship among the various parameters used in our task model.

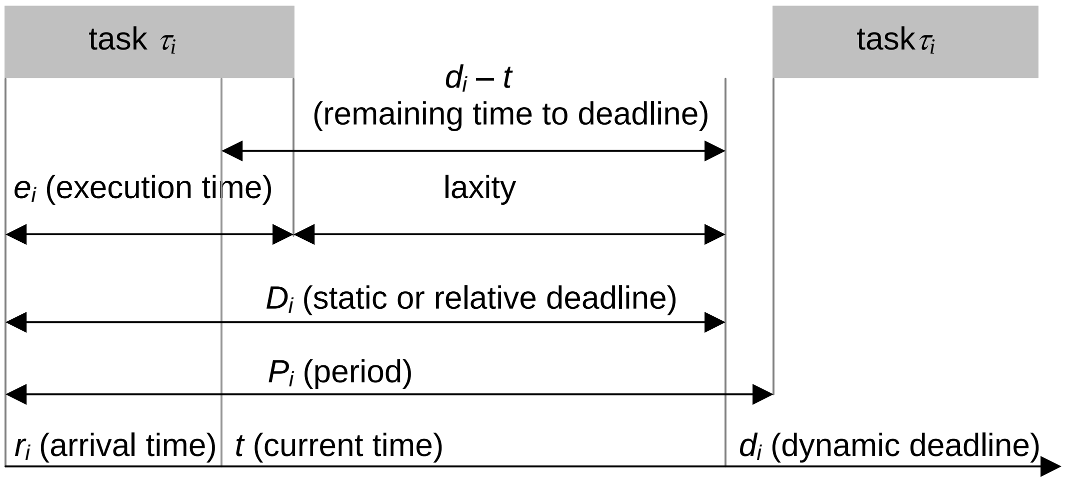

Figure 3.1: The relationship among the real-time task parameters.

For the experiments in this work, I have generated a fixed number (N) of jobs with randomly generated arrivals, execution times and deadlines. We assume that jobs are mutually independent. Each experiment is terminated when the predetermined experimental time T has expired. This permitted us to investigate the sensitivity of the various task parameters on the success rates (i.e. success ratios, we use these two terms interchangeably) of EDF and gEDF. I use random distributions available in MATLAB to generate the necessary parameters for tasks.

A group in the gEDF algorithm depends on a group range parameter Gr. A job τ_j belongs to the same group as job τ_i if $d_i \leq d_j \leq (d_i + Gr^*(d_i - t))$[1], where t is

[1] We are using the remaining time to a task deadline (called dynamic deadlines) in forming groups. We found that using static deadlines for defining groups did not significantly change the results.

the current time, $1 \leq i, j \leq N$. In other words, we group jobs with deadlines that are very close to each other. I schedule groups based on EDF (all jobs in a group with an earlier deadline will be considered for scheduling before jobs in a group with later deadlines), but schedule jobs within a group using shortest job first (SJF) approach. Since SJF results in more (albeit shorter) jobs completing, intuitively gEDF should lead to a higher success rate than pure EDF.

Let's take look at some examples.

Example 1:

There are four jobs $\tau_0 = (0, 5, 14, P_0)$, $\tau_1 = (0, 3, 14, P_1)$, $\tau_2 = (0, 6, 14, P_2)$, $\tau_3 = (0, 2, 14, P_3)$. They arrive at the same time 0 and have the same deadline, i.e. 14. In the following bars, the gray part represents expected execution time of a job. The clear part represents laxity[2] time of a job. P_0, P_1, P_2, and P_3 can be constant (for periodic job), variable (for aperiodic job), or variable with a minimum value (for sporadic job). To simplify the analysis, P_i, i = 0, 1, 2, and 3, will be ignored in this example and the following two examples. That is, we assume that there is only one instance for each task. Therefore, the four jobs become $\tau_0 = (0, 5, 14)$, $\tau_1 = (0, 3, 14)$, $\tau_2 = (0, 6, 14)$, $\tau_3 = (0, 2, 14)$. These four jobs are shown as four separate bars in Figure 3.2 (a). The length of the shaded part of each bar represents execution time. The length of each bar represents the deadline. The result scheduled by EDF and FIFO is shown in Figure 3.2 (b). The result scheduled by EDF and SJF is shown in Figure 3.2 (c).

[2] Laxity is the remaining time before the deadline expires.

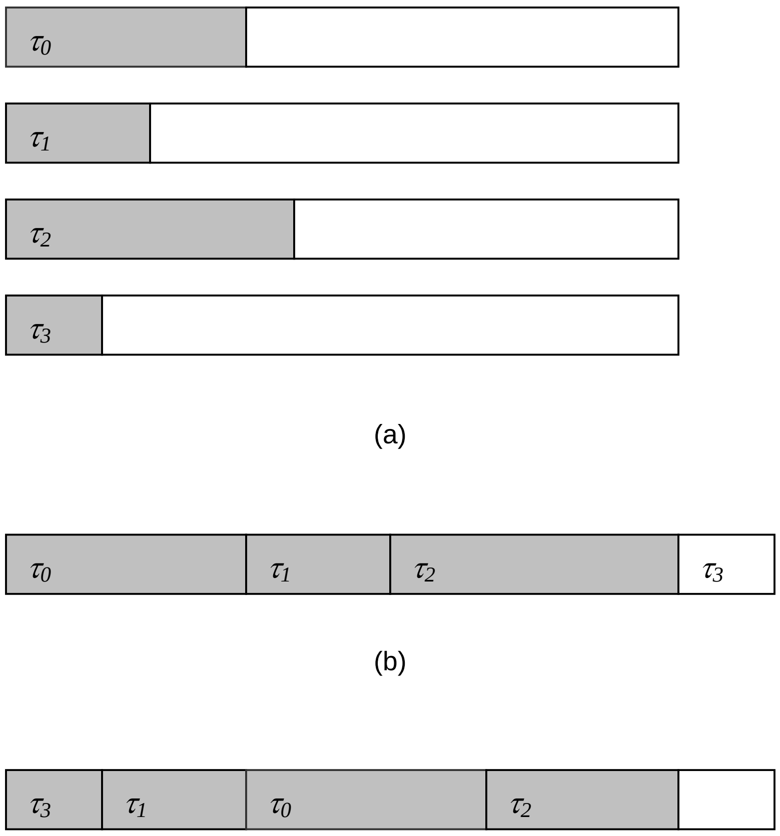

Figure 3.2: Example 1 – (a) Four jobs with the same deadlines. (b) EDF Scheduling using FIFO. (c) EDF Scheduling using SJF.

As in (a), the four jobs have the same deadlines. If using EDF with FIFO, success ratio is 3/4. The average response time of the completed jobs is ((0+5) + (5+3) + (8+6)) / 3 = 9. In (c), we use SJF with EDF and now the success ratio is 3/4. However, τ_2 completes partially before the deadline. If it is soft real-time system, which allows some tolerance of missing deadline, the success ratio could be 4/4. By comparison, the average response time of the first three completed jobs in (c) is ((0+2) + (2+3) + (5+6)) / 3 = 6. The average response time of all the completed jobs in (c) is ((0+2) + (2+3) + (5+6) + (10+6)) / 4 = 8.5.

Example 2:

In example 1, one can use shortest job first (SJF) scheduling instead of first-in-first-out (FIFO) scheduling since we can group all the four jobs in a single group as the deadlines are the same for the jobs. Often jobs fall into different groups since jobs have different deadlines. Consider the following set of τ_0 = (0, 5, 11), τ_1 = (0, 3, 10), τ_2 = (0, 6, 9), τ_3 = (0, 2, 12). These four jobs are shown as four separate bars in Figure 3.3 (a). Four jobs have different execution times as shown by the shaded areas. The four jobs also have different deadlines as shown by the length of the bars. The result of EDF scheduling using FIFO is shown in Figure 3.3 (b); the EDF scheduling using SJF is shown Figure 3.3 (c).

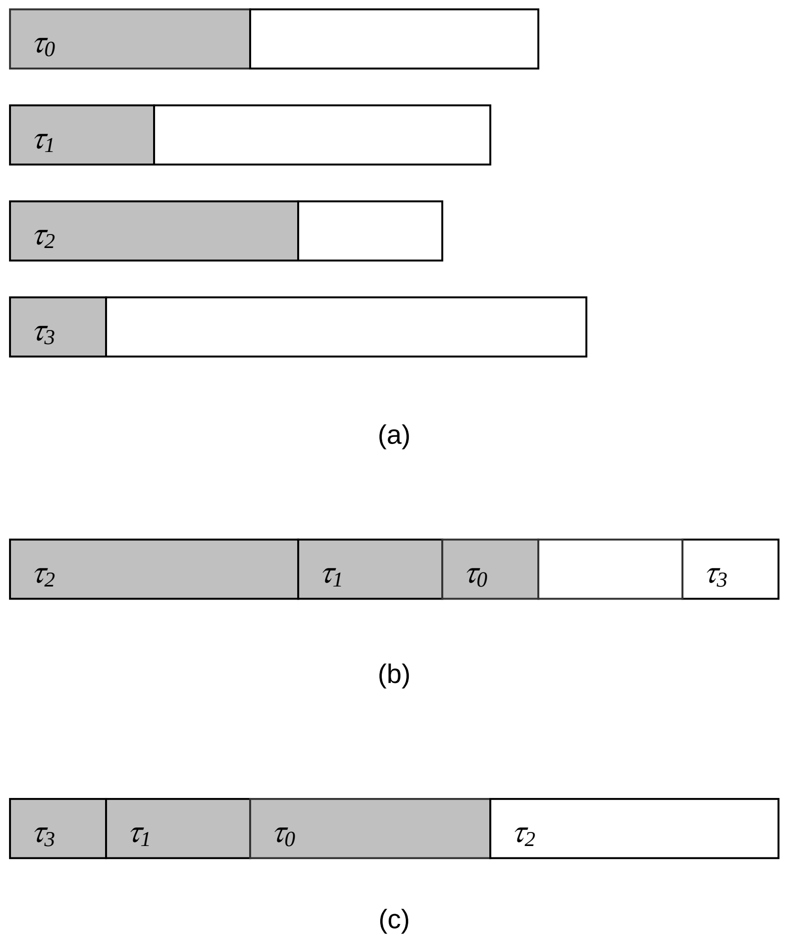
τ_0
τ_1
τ_2
τ_3
(a)
τ_2 τ_1 τ_0 τ_3
(b)
τ_3 τ_1 τ_0 τ_2
(c)

Figure 3.3: (a) Four jobs with different deadlines. (b) EDF Scheduling with FIFO. (c) EDF Scheduling with SJF.

As in (a), the four jobs have different deadlines. We still can apply the SJF scheme in EDF. For this example we will use deadline groups to schedule jobs with deadline that are very close to each other. In this example we will group all 4 jobs into a single group. If we use EDF scheduling using FIFO, as shown in (a) the success ratio is 2/4 and the average response time of the completed jobs is

((0+6) + (6+3))/2 = 7.5. If we use SJF for the jobs in a group, the success ratio is 3/4. By comparison, the average response time of the first two completed jobs in (c) is ((0+2) + (2+3)) / 2 = 3.5. The average response time of all the completed jobs in (c) is ((0+2) + (2+3) + (5+6)) / 3 = 6.

Example 3:

In the previous examples, we assumed that jobs have strict deadlines. In other words, if a task misses its deadline, the task is considered failed. However, for soft real-time jobs, deadlines are defined with some grace period and a task is allowed to miss its deadline as long as it can complete within its specified grace period. We use the term “deadline tolerance” to specify grace periods. As can be expected, success rates can increase with larger deadline tolerances. Then, consider the set of jobs τ_0 = (0, 5, 14), τ_1 = (0, 3, 13), τ_2 = (0, 6, 15), τ_3 = (0, 2, 15). Let us assume that the deadline tolerance is 20%. For example, τ_0 is allowed to complete within the deadline 14 + 2.8 = 16.8 time units from its arrival time. The four jobs are shown in Figure 3.4 (a). The result scheduled by EDF and FIFO is shown in Figure 3.4 (b). The result scheduled by EDF and SJF is shown in Figure 3.4 (c).

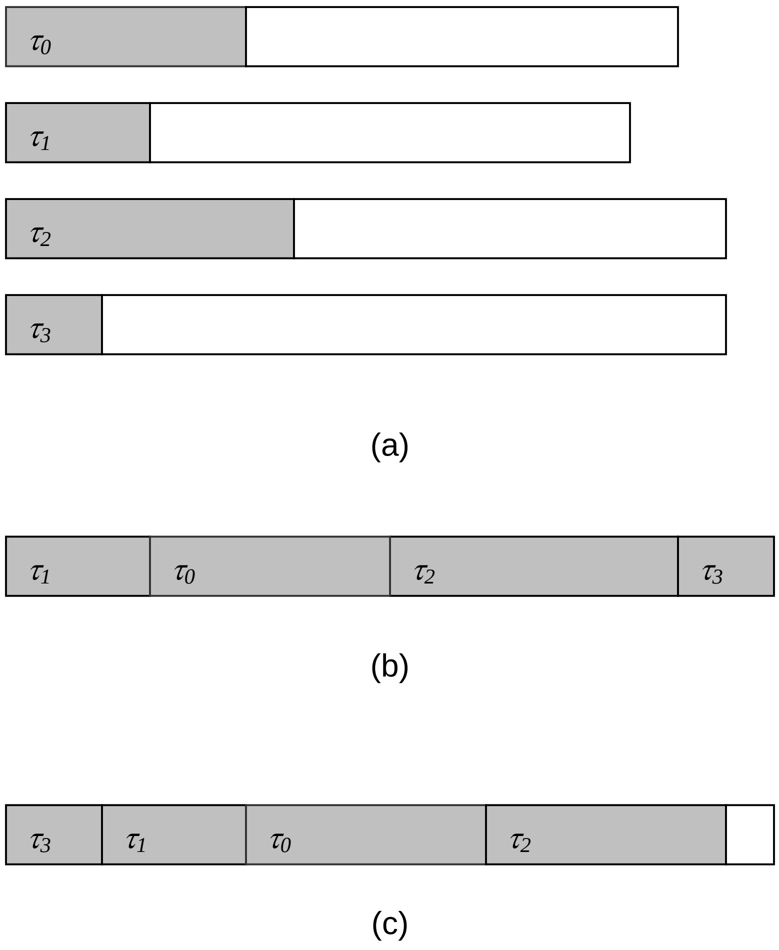
τ_0
τ_1
τ_2
τ_3
(a)
τ_1 τ_0 τ_2 τ_3
(b)
τ_3 τ_1 τ_0 τ_2
(c)

Figure 3.4: (a) Four jobs with different deadlines. (b) EDF Scheduling with FIFO when soft tolerance is allowed. (c) EDF Scheduling with SJF when soft tolerance is allowed.

As before we can group the tasks based on their deadlines. In part (b), with the EDF scheduling using FIFO only three jobs can be completed (It misses 1/2 of its deadline, and exceeds the 20% deadline tolerance allowed). Success ratio is thus 3/4. The average response time of the completed jobs is ((0+3) + (3+5) + (8+6) = 8.3. If we use SJF with a group of jobs, and using the 20% deadline tolerance, we can complete all 4 tasks, giving us a success ratio of 4/4. The average response time of the first three completed jobs in (c) is ((0+2) + (2+3) + (5+6)) /3 = 6. The average response time of all the completed jobs in (c) is ((0+2) + (2+3) + (5+6) + (10+6)) / 4 = 8.5.

Nomenclature:

We use the following notations for the various parameters and computed values:

ρ: is the utilization of the system, $\rho = \Sigma e_i / T$. This is also called the load.

γ: is the success ratio, γ = the number of jobs completed successfully / N.

Tr: is the deadline tolerance for soft real-time systems. A job τ is successful if τ finishes before the time $(1 + Tr) * D$, where $Tr \geq 0$.

μ_e: is used either as the average execution time or the worst-case execution time, and defines the expected value of the exponential distribution used for this purpose.

μ_r: is used to generate arrival times of jobs, and is the expected value of the exponential distribution used for this purpose.

μ_D: is the expected value of the random distribution used to generate task deadlines. We set this parameter as a multiple of μ_e. We use only the generated values if they are larger than the execution time of a job.

$\Re$: is the average response time of the jobs. This is a computed value.

∂: is the response-time ratio, $\partial = \Re / \mu_e$.

η_γ: is the success-ratio performance factor, $\eta_\gamma = \gamma_{gEDF} / \gamma_{EDF}$. This is used to compare gEDF with EDF.

η_∂: is the response-time performance factor, $\eta_\partial = \partial_{EDF} / \partial_{gEDF}$. This is used to compare gEDF with EDF.

3.2 gEDF Algorithm

3.2.1 Description and Pseudo Code

We assume a uniprocessor system. Q_{gEDF} is a queue for gEDF scheduling. The current time is represented by t. $|Q_{gEDF}|$ represents the length of the queue Q_{gEDF}. $\tau = (r, e, D, P)$ is the job at the head of the queue.

We define a group in our gEDF algorithm as follows:

gEDF Group = $\{\tau_k \mid \tau_k \in Q_{gEDF}, d_k - d_1 \leq D_1 * Gr, 1 \leq k \leq m$, where $m \leq |Q_{gEDF}|\}$, and D_1 is the deadline of the first job in a group.

Algorithm:

```
Function Enqueue ($Q_{gEDF}$, $\tau$)
    if ( $\tau$'s deadline $d > t$ ) then
        insert job $\tau$ into $Q_{gEDF}$ by earliest deadline first, i.e. $d_i \leq d_{i+1} \leq d_{i+2}$,
        where $\tau_i, \tau_{i+1}, \tau_{i+2} \in Q_{gEDF}$, $1 \leq i \leq |Q_{gEDF}| - 2$;
    end

Function Dequeue ($Q_{gEDF}$)
    if $Q_{gEDF} \neq \phi$ then
        find a job $\tau_{min}$ with $e_{min} = \min \{e_k \mid \tau_k \in Q_{gEDF}$,
        $d_k - d_1 \leq Gr*D_1$, $1 \leq k \leq m$, where
        $m \leq |Q_{gEDF}|\}$;
        run it and delete $\tau_{min}$ from $Q_{gEDF}$;
    end
```

Enqueue is invoked on job arrivals and Dequeue is called when the processor becomes idle.

3.2.2 Complexity of the gEDF Algorithm

Here I outline the complexity of gEDF. Assume that there are *n* jobs to be scheduled. Standard EDF needs O(*n*) to find a job schedule since it must find a job with the earliest deadline. Our gEDF effectively performs similar search within a group to find a job with shortest execution time. Note that, although the asymptotic complexity of gEDF is O(*n*), the number of jobs in a group is much smaller than *n*. Now I analyze the complexity of gEDF. Assume there are *n* jobs to be scheduled. EDF needs O(*n*) for one schedule. The gEDF needs to find the shortest job from *m* ($m \le n$) jobs within a group. Assuming that there are k groups, the overall complexity is given by

$$O(\sum_{i}^{k} m_i)$$

Since the total number of jobs in all groups is still *n*, the time complexity of gEDF is also O(*n*).

3.2.3 Analysis of the gEDF Algorithm

To argue that gEDF is better than EDF, we first define *w*, which represents a set of jobs $\tau_i \in w$ ($1 \le i \le |w|$) that are ready for scheduling at time *t* and are sharing the same deadline *d* (the jobs have to finish before the time *d*). In addition, δ represents the time needed for the scheduling and *v* represents an associated set of jobs $\tau_j \in v$ ($1 \le j \le |w|$) that share the deadlines window of $d + \delta$.

In addition, we define γ_w as the success ratio of |*w*| jobs in the time slot *w* with the deadline *d*. We assume that there is no interval between the scheduled jobs if the jobs are available. L_w represents the laxity time of all the jobs after scheduling in a time slot w. E_v represents the sum of the execution time of $|v|$ jobs with the largest deadline $d < d_v \leq d + \delta$. We can obtain the following result.

If, i) there are $|w|$ jobs sharing the same deadline d in a time slot $w = d - t$ (starting from the time current time t and ends at the time d); and ii) there are $|v|$ jobs in the time δ that is to be added to the time slot w for scheduling, we can conclude that $\gamma_{w+\delta} \geq \gamma_w$ under any of the following situations.

1. $\delta = 0$

2. $Tr \geq \delta$

3. $L_w \geq E_v$

For case 1, if utilization $\rho \leq 1$ (or more precisely speaking $L_w \geq 0$ because we refer ρ as an average value), there will be no change in the success ratios because the sum of execution time that can be completed by the deadline will not change. If $\rho > 1$ or $L_w = 0$, because SJF places jobs with longer execution times closer to the end of queue, there should be fewer jobs rejected. For case 2, if we set the soft real-time tolerance Tr large enough, gEDF should be able schedule even longer jobs towards the end of the queue, provided the scheduling overhead δ time is smaller than the deadline tolerance for all jobs $Tr \geq \delta$. For case 3, if there is large enough laxity time left for the jobs of v without making the jobs of w unscheduled, gEDF success rate will be greater than EDF using FIFO.

Thus we argue that the performance of gEDF is equal to or better than that of EDF under the conditions listed above.

Furthermore, we analyze the performance of gEDF under the remaining situations, namely, the situation when $\delta \neq 0$, $Tr < \delta$, and $L_w < E_v$. First, we assume $w_2 \in w$ is the set of jobs unscheduled after the jobs of $v_1 \in v$ are scheduled in time $d - t$. The success ratio by EDF is $\gamma_w + \gamma_v$. The success ratio by gEDF is $\gamma_{w-w2+v1} + \gamma_{v+w2-v1}$. It is apparent that $\gamma_{w-w2+v1} \geq \gamma_w$ if SJF is used. In addition, by comparing γ_v with $\gamma_{v+w2-v1}$, the former should be larger than the latter. However, usually, this difference is not as large as that of $\gamma_{w-w2+v1}$ and γ_w. Therefore, we are able to conclude gEDF is better than EDF.

In the following experiments, conducted with various data distributions, we can observe the above features of gEDF. Moreover, we can see how much of a performance increase gEDF can obtain for the different parameters and the various data distributions. I will analyze each behavior of gEDF. In addition, average response time, another important metric of gEDF for real-time systems, will be analyzed.

CHAPTER 4

NUMERICAL RESULTS

MATLAB [36] is used to generate tasks based on various random distributions and the generated tasks are scheduled using EDF, gEDF, and other scheduling algorithms. For each chosen set of parameters, I have repeated each experiment 100 times (each time, generating *N* tasks using the random probability distributions and scheduling the generated tasks) and computed the average success rates. In what follows, I report the results and analyze the sensitivity of gEDF to the various parameters used in the experiments, the effects of the percentage of small jobs on the success ratios, and how well gEDF performs when compared to a best-effort algorithm. Note that I use the non-preemptive task model. Non-preemptive scheduling algorithms are easier to implement than preemptive algorithms, and can exhibit dramatically lower overhead at run-time resulting from fewer context switches.

4.1 Comparison of gEDF and EDF

First I will show that the gEDF algorithm achieves higher success rates in scheduling real-time tasks than the well-known EDF algorithm, particularly when the system is heavily loaded.

4.1.1 Experiment 1 – Effect of Deadline Tolerance (*Tr*)

Figures 4.1 - 4.3 show that gEDF achieves higher success rate than EDF when the deadline tolerance (i.e., soft real-time nature of the jobs) is varied from 20%, 50% to 100% (that is, a task can miss its deadline by 20%, 50% and 100%).

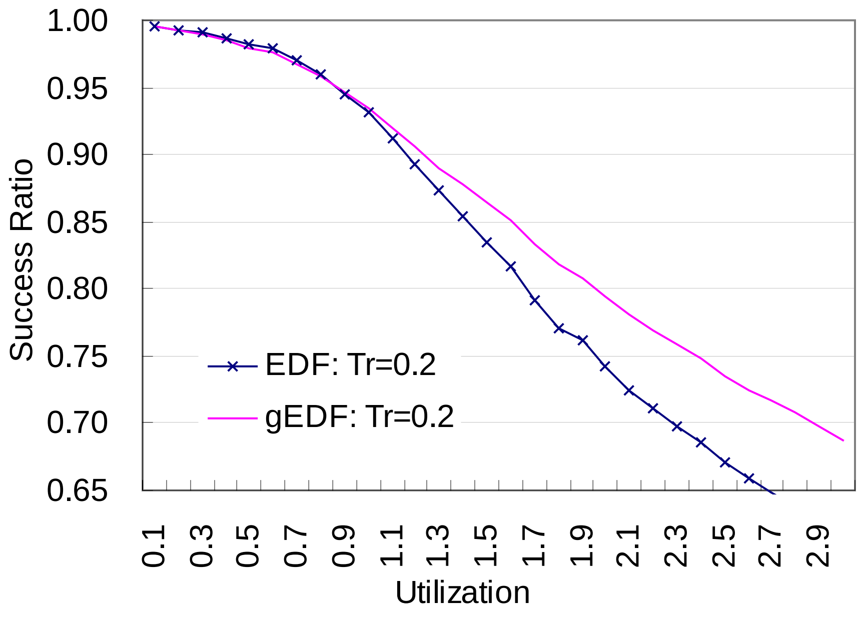

Figure 4.1: Success rates when deadline tolerance is 0.2.

With utilization $\rho \leq 1.0$, the difference in the success ratios of EDF and gEDF is not significant. When the load is very light, all the jobs can be scheduled by EDF and gEDF; when the load becomes heavier but is still less than 1.0, a few jobs could not be scheduled by EDF or gEDF. Therefore, the success ratios of EDF and gEDF become lower (as low as 0.93). When $\rho \leq 1.0$, gEDF doesn't perform any better than EDF because most jobs can be scheduled by either method. Since I use a non-preemptive real-time model, instead of preemptive periodic job model, the success ratio of EDF or gEDF will fall below 100% as ρ reaches 1.0. As this load is reached, the success ratios of EDF and gEDF start showing differences. The success ratio of EDF decreases more quickly than that of gEDF. For instance, the success ratio of EDF becomes 0.65 when ρ = 2.7; however, the success ratio of gEDF is still high at 0.72. In other words, the success ratio of gEDF is about 11% higher than that of EDF when ρ = 2.7. I experimented with utilization values between 0 and 3. The deadline tolerance in Figure 4.1 is 20%.

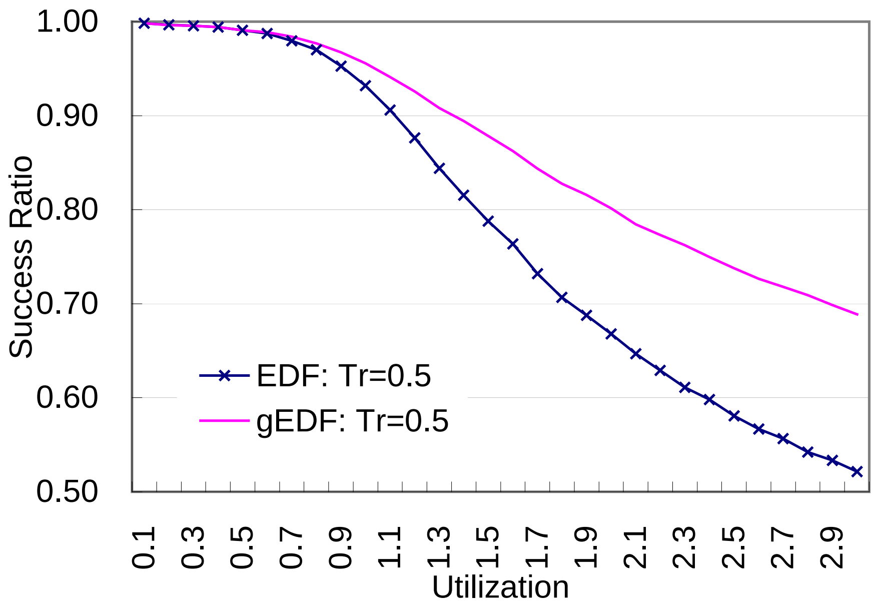

Figure 4.2: Success rates when deadline tolerance is 0.5.

When deadline tolerance increases to 50%, the success ratio of gEDF improves; while the success ratio of EDF worsens. Even when $\rho < 1.0$ (underloaded), the success ratio of gEDF is higher than that of EDF. For instance, when $\rho = 1.0$, the success ratio of gEDF is 0.97; and the success ratio of EDF is 0.91. In contrast, the success ratios of gEDF and EDF in Figure 4.1 (20% deadline tolerance) were both 0.93. EDF suffers “the domino effect” (since as jobs at the front of the queue miss their deadlines, so do other jobs in the queue) when $\rho > 1.0$. Since gEDF uses SJF, the jobs at the front of the queue are less likely to miss their deadlines, diminishing the domino effect.

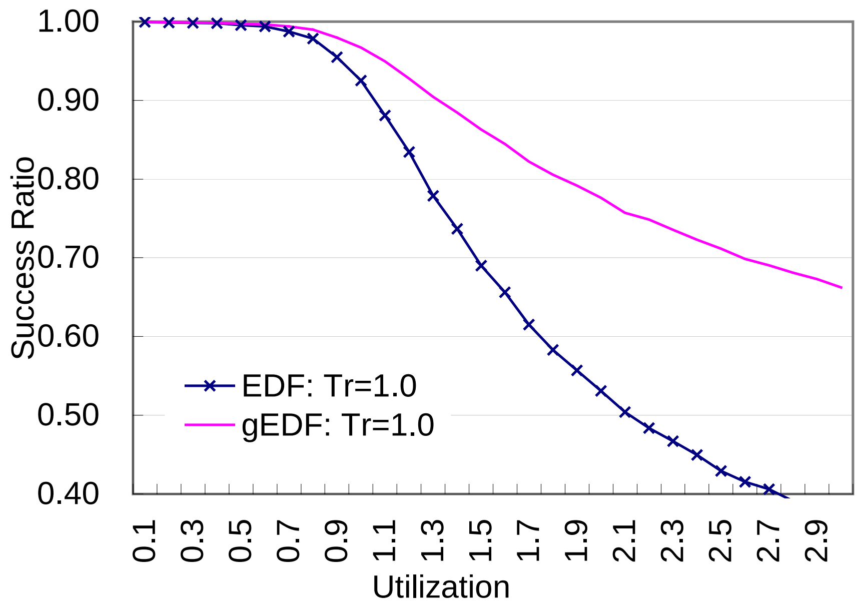

Figure 4.3: Success rates when deadline tolerance is 1.0.

When deadline tolerance increases to 100%, gEDF behaves even better; but EDF's success rates do not increase significantly. For instance, when $\rho < 1.0$, the success ratio of gEDF is close to 100%. In addition, when $\rho = 1.1$ (lightly overloaded), the success ratio of gEDF is still as high as 0.95. However, the success ratio of EDF worsens quickly.

For these experiments, I generated tasks by fixing expected execution times and deadline parameters based on probability distributions, but varied the arrival rate parameter to change the system load. The group range for these experiments is fixed at $Gr = 0.4$ (i.e., all jobs whose deadlines fall within 40% of the deadline of current job are in the same group). It should be noted that gEDF consistently performs as well as EDF under light loads (utilization less than 1), but outperforms EDF under heavy loads (utilization greater than 1; see the X-axis). Both EDF and gEDF achieve higher success rates when tasks are provided with greater deadline tolerance. The tolerance benefits gEDF more than EDF, particularly under heavy loads. Thus, gEDF is better suited for soft real-time tasks.

Figure 4.4 summarizes these results by showing the percent improvement in success ratios achieved by gEDF when compared to EDF. The Y-axis shows that higher success rates are achieved by gEDF when compared to EDF for different system loads and different deadline tolerance parameters.

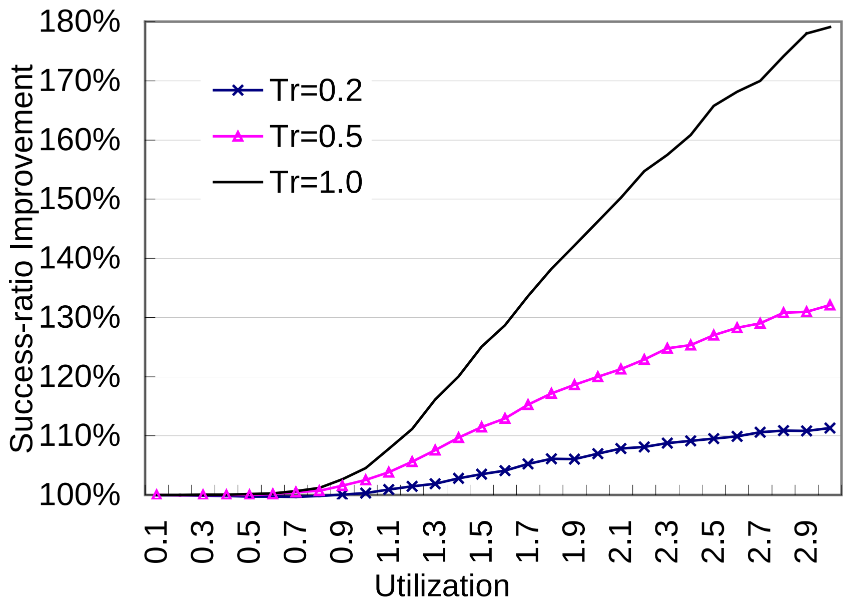

Figure 4.4: Success-ratio performance factor when *Tr* = 0.2, 0.5, and 1.0*.*

I use success-ratio performance factor, i.e., η_γ = γ_{gEDF} / γ_{EDF}, to express the performance increase. Individually, γ_{gEDF} is the success ratio of gEDF and γ_{EDF} is the success ratio of EDF. When ρ is varied from 0 to 3.0 and deadline tolerance is 20%, success-ratio performance factor η_γ is between 100% and 112%, indicating that gEDF can achieve 12% higher success rates. This ratio becomes even larger at higher system loads and larger deadline tolerance values.

Table 4.1: Success-ratio performance factor

Utilization	Success-ratio improvement (%)		
ρ	*Tr* = 0.2	*Tr* = 0.5	*Tr* = 1.0
0.1	100	100	100
0.2	100	100	100
0.3	100	100	100
0.4	100	100	100
0.5	100	100	100
0.6	100	100	100
0.7	100	100	101
0.8	100	101	101
0.9	100	102	103
1.0	100	103	105
1.1	101	104	108
1.2	101	106	111
1.3	102	108	116
1.4	103	110	120
1.5	104	111	125
1.6	104	113	129
1.7	105	115	134
1.8	106	117	138
1.9	106	119	142
2.0	107	120	146
2.1	108	121	150
2.2	108	123	155
2.3	109	125	157
2.4	109	125	161
2.5	110	127	166
2.6	110	128	168
2.7	111	129	170
2.8	111	131	174
2.9	111	131	178
3.0	112	132	179

Table 4.1 shows the values of success-ratio performance factor $\eta_\gamma = \gamma_{gEDF} / \gamma_{EDF}$ (i.e., success-ratio improvement) when ρ = 1.0 to 3.0.

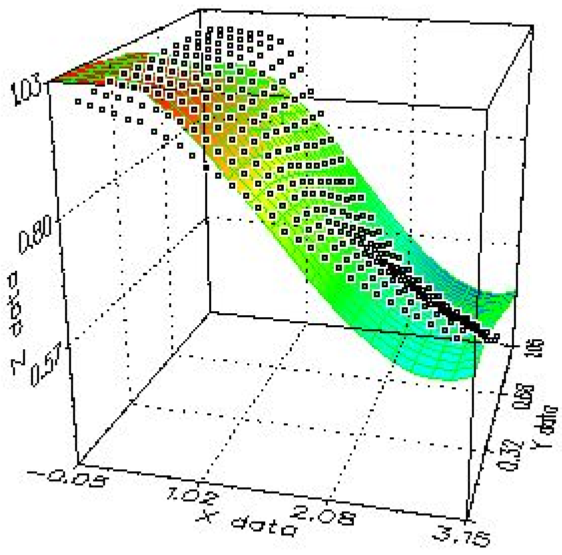

Figure 4.5: for EDF, X-axis is ρ, Y-axis is *Tr*, Z-axis is success ratio.

Figure 4.5 shows the relationship between utilization, deadline tolerance, and the success ratio of EDF; with X-axis representing utilization ρ, Y-axis representing deadline tolerance *Tr*, and Z-axis representing success ratio of EDF γ_{EDF}. It is interesting to note that as the tolerance increases, the success ratios do not show concomitant increases.

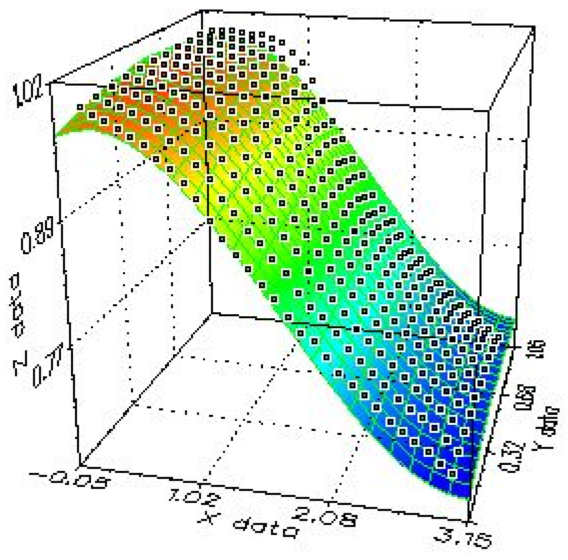

Figure 4.6: for gEDF, X-axis is ρ, Y-axis is *Tr*, Z-axis is success ratio.

Figure 4.6 shows the relation of utilization, deadline tolerance, and success ratio of gEDF. As before X-axis represents utilization ρ, Y-axis deadline tolerance *Tr*, and Z-axis represents success ratio of gEDF γ_{gEDF}, Unlike EDF, gEDF shows improved success ratios with increasing deadline tolerances.

4.1.2 Experiment 2 - Effect of Deadline on Success Rates (γ)

In the next experiment, I explore the performance of EDF and gEDF when the deadlines are very tight (deadline = execution time) and when the deadlines are loose (deadline = 5 * execution time). Note that the deadlines generated using an exponential distribution with mean values set to 1 and 5 times the mean execution time μ_e. I varied the soft real-time parameter (*Tr*, or tolerance to deadline) in these experiments also, but all other parameters are kept the same as in the previous experiment. As can be seen in Figures 4.7 and 4.8, all scheduling algorithms perform poorly for tight deadlines[3], except under extremely light loads. Even under very tight deadlines, as in Figure 4.8, the deadline tolerance favors gEDF more than EDF. With looser deadlines (or more laxity), as in Figures 4.9 and 4.10, both EDF and gEDF achieve better performance. However, gEDF outperforms EDF consistently for all values of the deadline tolerance, *Tr*.

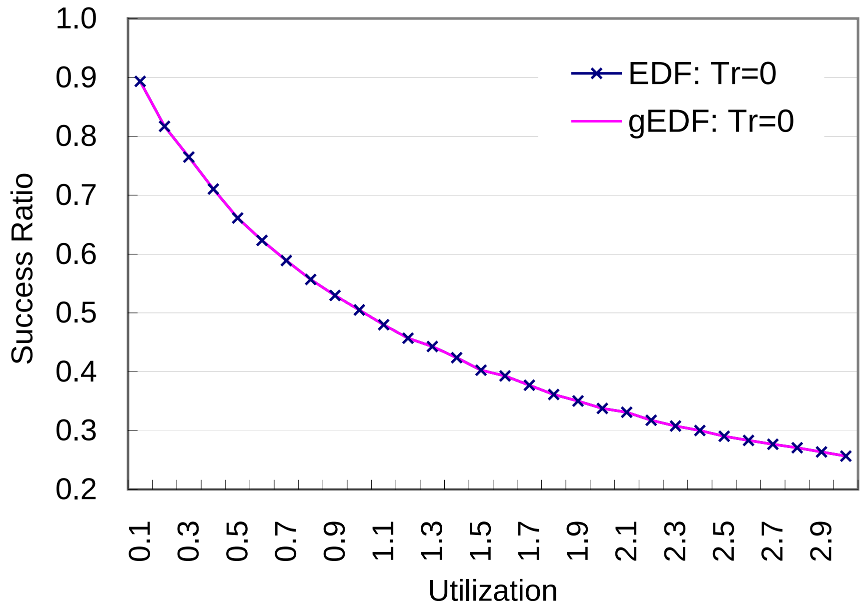

Figure 4.7: Tight deadline $\mu_D = \mu_e$ and $Tr = 0$.

[3] It should be noted that when $\mu_D = \mu_e$, any job should be scheduled immediately upon arrival, lest it misses its deadline. The impact of using least laxity first approach is indirectly reflected by EDF when the deadlines are very tight.

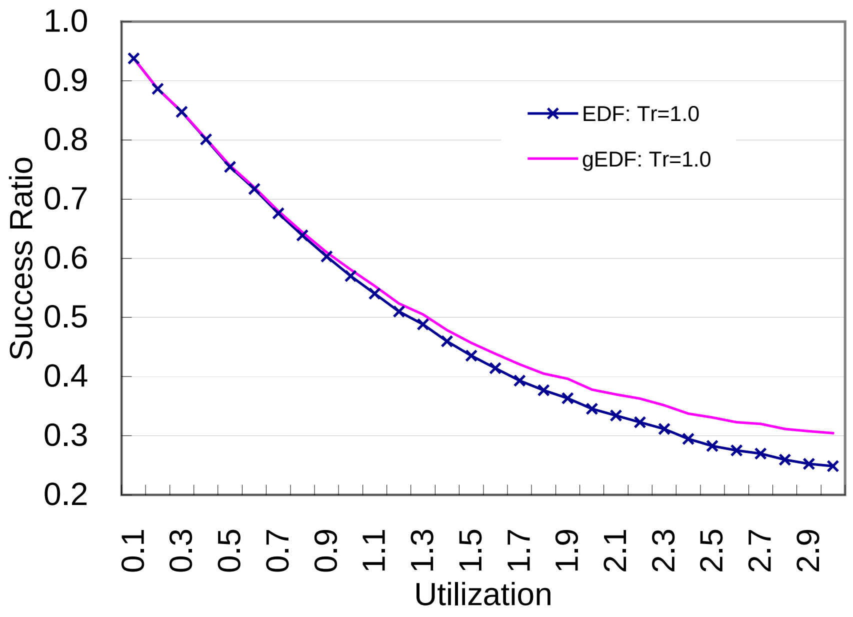

Figure 4.8: Tight deadline $\mu_D = \mu_e$ and *Tr* = 1.0.

When deadline tolerance *Tr* = 1.0, the gEDF scheduling algorithm can outperform the EDF scheduling algorithm under both underloaded and overloaded conditions, even with tight deadlines.

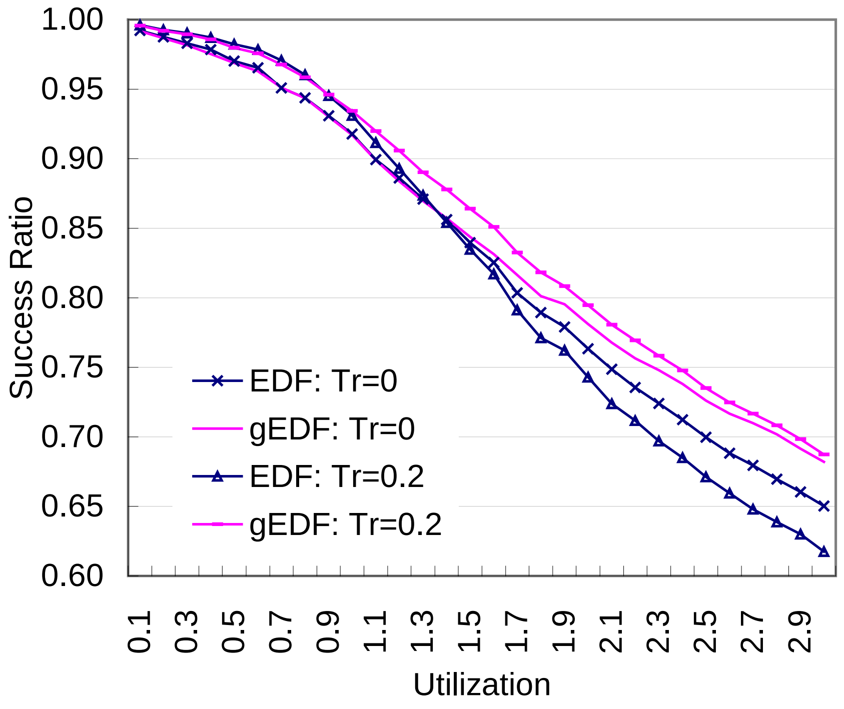

Figure 4.9: Looser deadline $\mu_D = 5\mu_e$ and *Tr* = 0 and 0.2.

With looser deadlines of $\mu_D = 5\mu_e$, for *Tr* = 0 or 0.2, gEDF performs slightly better than EDF. For instance, when $\rho = 1.0$ (underloaded) and *Tr* = 0.2, gEDF performs as well as EDF; when $\rho = 2.0$ (overloaded) and *Tr* = 0.2, gEDF outperforms EDF by 6%.

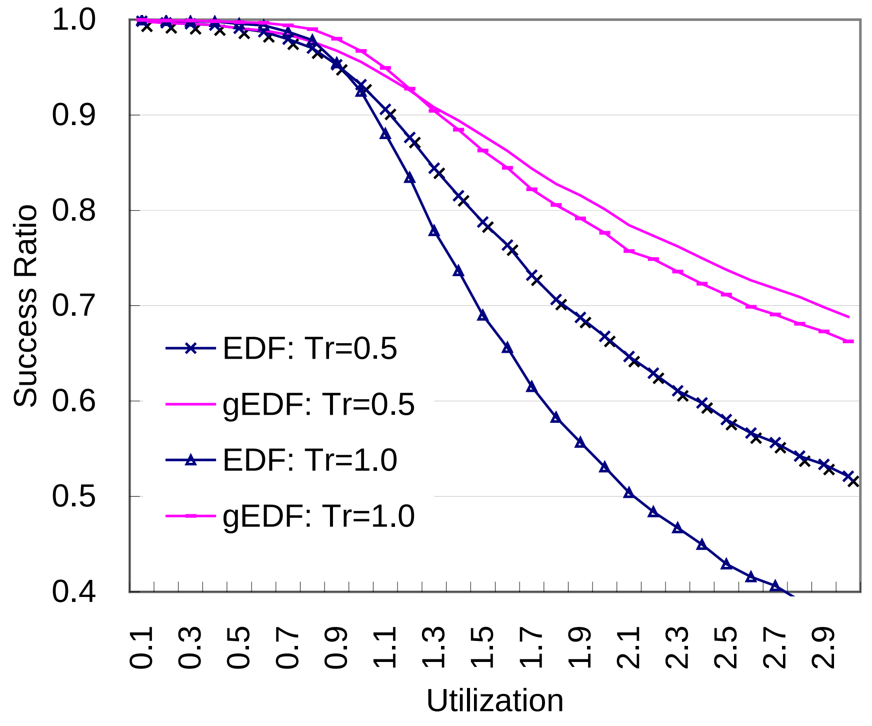

Figure 4.10: Looser deadline $\mu_D = 5\mu_e$ and Tr = 0.5 and 1.0.

With $\mu_D = 5\mu_e$, for higher deadline tolerance values of Tr, gEDF performs better than EDF. For instance, when ρ = 1.0 (underloaded) and Tr = 1.0, gEDF outperforms EDF by 4%; when ρ = 2.0 (overloaded) and Tr = 1.0, gEDF outperforms EDF by 47%.

Figures 4.11 and 4.12 respectively, highlight the effect of deadline laxities on both EDF and gEDF. To more clearly evaluate how these approaches perform when the deadlines are very tight and loose, I set the deadlines to 1, 2, 5, 10 and 15 times the execution time of a task. I set μ_e = 40, Tr = 0.2, (for gEDF Gr = 0.4).

When $\mu_D = \mu_e$ and $2\mu_e$, the success ratios of EDF and gEDF show no appreciable differences. However, when μ_D becomes reasonably large, such as $5\mu_e$, $10\mu_e$, and $15\mu_e$, the success ratio of gEDF is better than that of EDF.

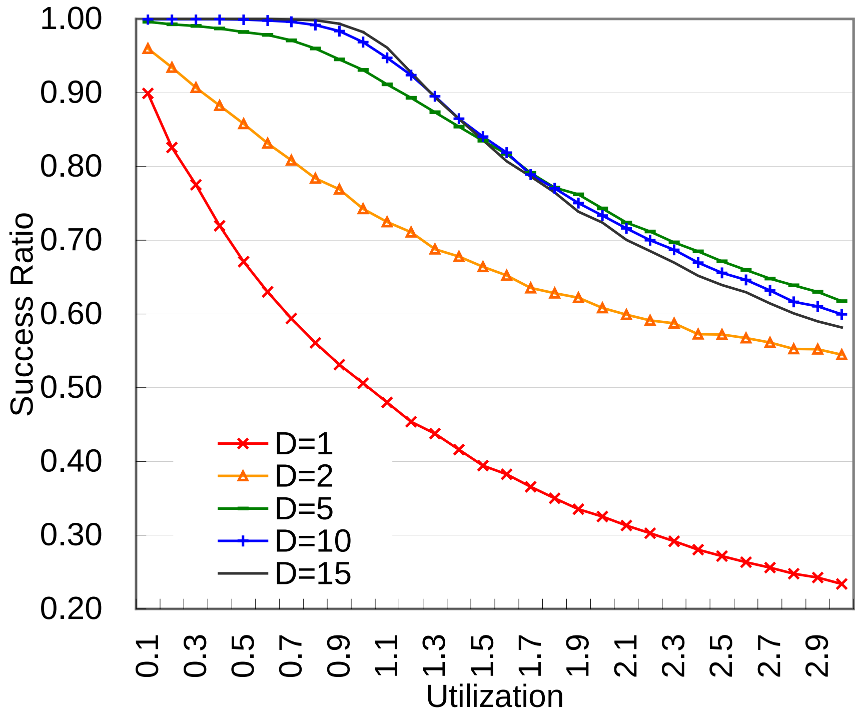

Figure 4.11: Success ratio of EDF when $\mu_D = \mu_e$, $2\mu_e$, $5\mu_e$, $10\mu_e$, and $15\mu_e$.

When μ_D changes from μ_e, $2\mu_e$, to $5\mu_e$, the success ratio of EDF increases sharply. For instance, when $\rho = 1.0$ and $\mu_D = \mu_e$, the success ratio of EDF is 0.51; when $\rho = 1.0$ and $\mu_D = 2\mu_e$, the success ratio of EDF is 0.74. However, when μ_D continues to increase to be $10\mu_e$, and $15\mu_e$, the success ratio of EDF doesn't see further improvements. For instance, when ρ < 1.5 (underloaded or lightly overloaded), the success ratio of EDF increases; however, when ρ > 1.8, the success ratio of EDF decreases slightly.

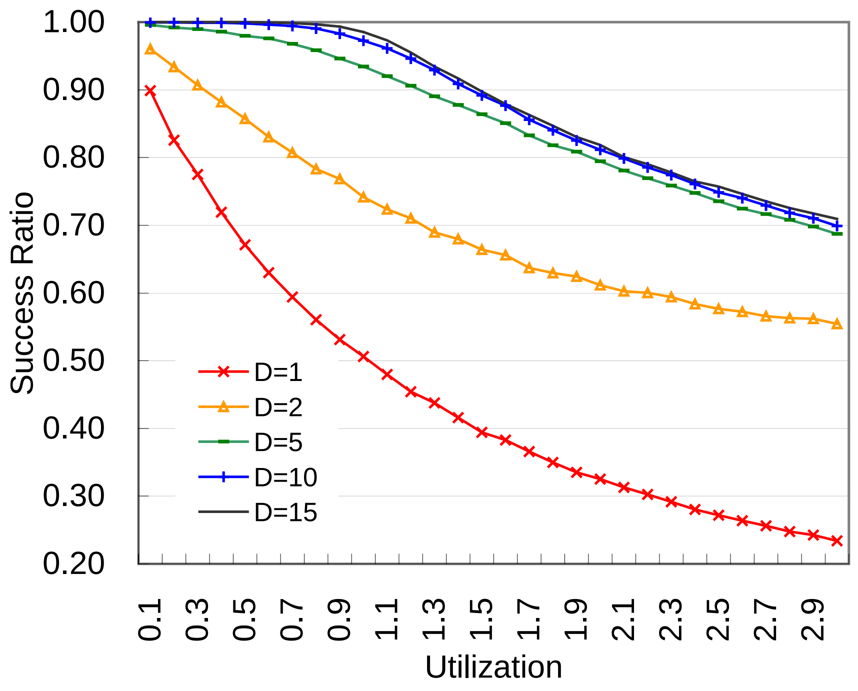

Figure 4.12: Success ratio of gEDF when $\mu_D = \mu_e, 2\mu_e, 5\mu_e, 10\mu_e$, and $15\mu_e$.

The gEDF scheduling algorithm exhibits similar behavior for different values of μ_D. However, gEDF shows improvements for a wider range of values of μ_D. When $\mu_D = \mu_e$, both gEDF and EDF exhibit almost the same performance. When $\mu_D = 2\mu_e$, $5\mu_e$, $10\mu_e$, and $15\mu_e$, gEDF performs better than EDF. For instance, when $\mu_D = 5\mu_e$ and $\rho = 3.0$, the success ratio of gEDF is 0.69; the success ratio of EDF is 0.62. As in the Figure 4.1.1.1-4, success-ratio performance factors, that is, $\eta_\gamma = \gamma_{gEDF} / \gamma_{\mathrm{EDF}}$ are shown in Figure 4.13 for $\mu_D = \mu_e$, $2\mu_e$, $5\mu_e$, $10\mu_e$, and $15\mu_e$.

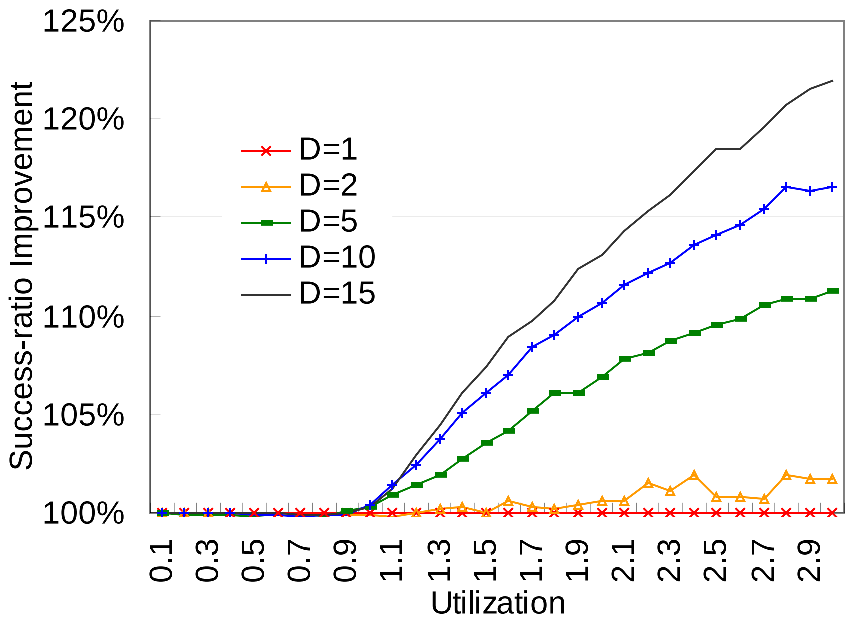

Figure 4.13: η_γ when $\mu_D = \mu_e$, $2\mu_e$, $5\mu_e$, $10\mu_e$, and $15\mu_e$.

When $\mu_D = 2\mu_e$ and ρ is large (heavily overloaded), Success-ratio performance factor η_γ increases slightly. The biggest jump occurs when μ_D changes from $2\mu_e$ to $5\mu_e$. When $\mu_D = 10\mu_e$ and $15\mu_e$, η_γ increases gradually. It should be noted that the laxity (as represented by μ_D) depends on the workload and application domain.

4.1.3 Experiment 3 - Effect of Group Range

In our third experiment, I vary the group range parameter *Gr* for grouping tasks into a single group. Note that in the following figures I do not include EDF data since the concepts of groups is not applicable to EDF. I set $\mu_D = 5\mu_e$ (Deadline = 5* Execution Time) and maintain the same values for other parameters as in the previous experiments. I set the deadline tolerance parameter *Tr* to 0.1 (10% tolerance in missing deadlines) in Figure 4.14, and to 0.5 (50% tolerance in missing deadlines) in Figure 4.15. The data shows that by increasing the size of a group, gEDF achieves higher success rates. In the limit, by setting the group range parameter to a very large value, gEDF behaves more like SJF; and by setting the group range value to zero, gEDF behaves like EDF. There is a threshold value for the group size for achieving optimal success rates and the threshold depends on the execution time, tightness of deadlines (or deadline laxity) and deadline tolerance parameters. For the experiments, I used a single exponential distribution for generating all task execution times. However, if we were to use a mix of tasks created using different exponential distributions with different mean values, thus creating tasks with widely varying execution times, the group range parameter will have more pronounced effect on the success rates (see Section 4.2).

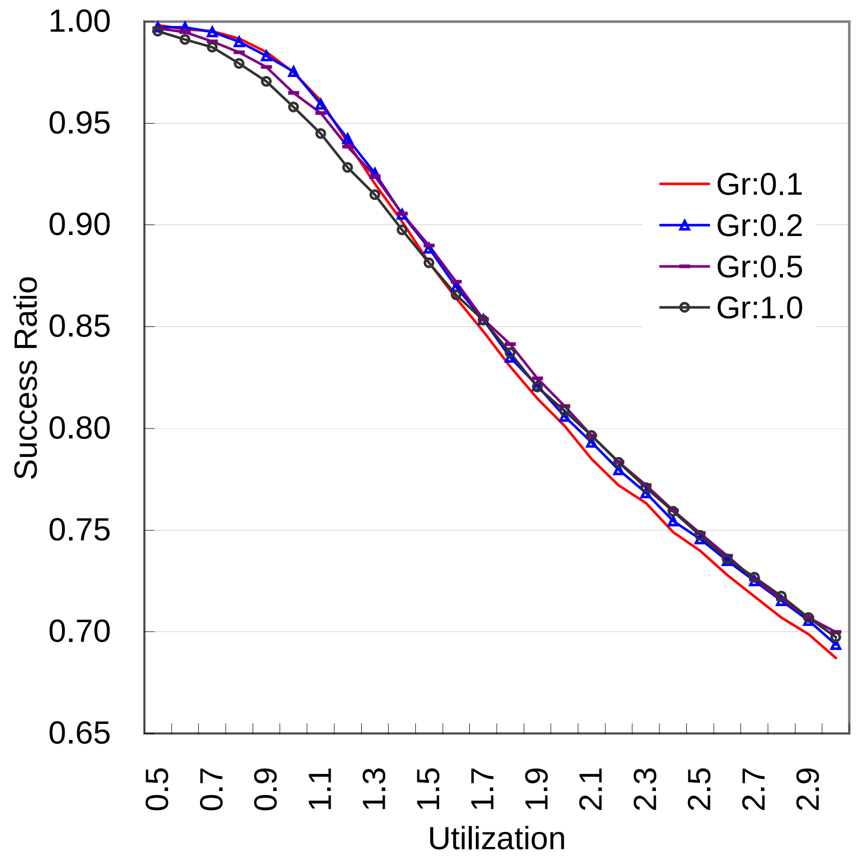

Figure 4.14: Group Range: *Gr* = 0.1, 0.2, 0.5, 1.0 (*Tr* = 0.1).

Success rates when *Gr* is varied from 0.2 to 1.0 are shown in Figure 4.14. The differences may not be significant here because I set the deadline tolerance to a small value (of 10%). The next figure (Figure 4.15) shows that success rates do show more significant differences when the tolerance is set to 0.4.

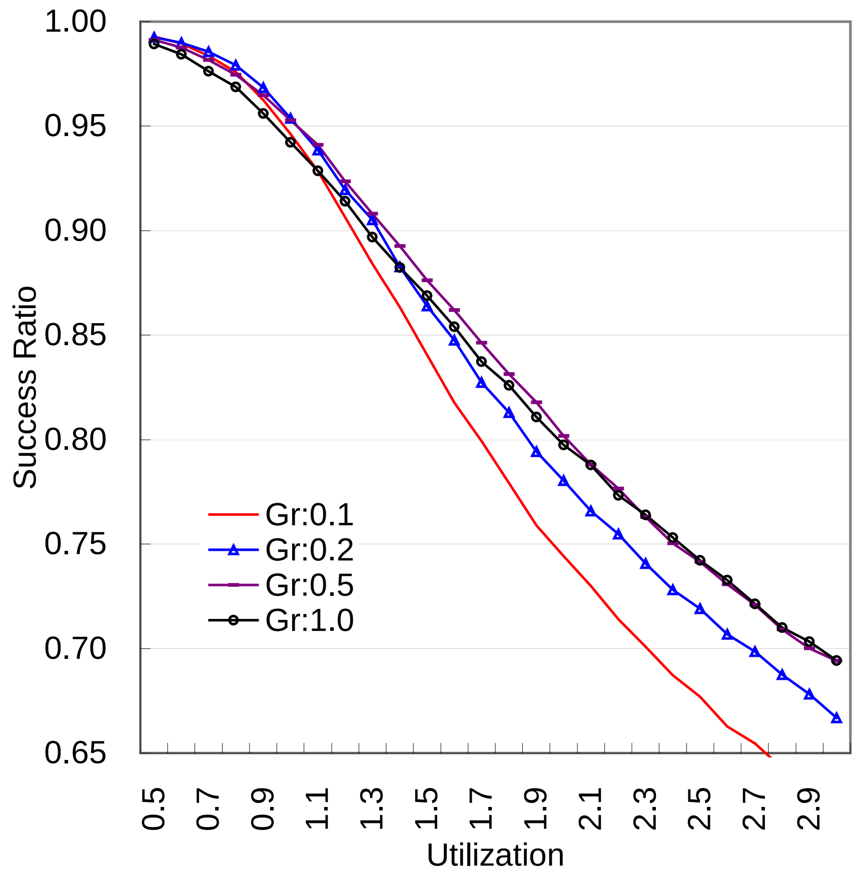

Figure 4.15: Group Range: *Gr* = 0.1, 0.2, 0.5, 1.0 (*Tr* = 0.5).

Instead of creating groups dynamically as jobs arrive, it is possible to define deadline bins and create groups based on these deadlines. Figure 4.16 shows the results based on statically defined groups, a window of $\mu_e/4$, while Figure 4.17 uses a window of $4*\mu_e$. These fixed windows do not produce success rates as high since the number of jobs in most of the windows will be very small. Creating groups dynamically allows us to create equal sized groups.

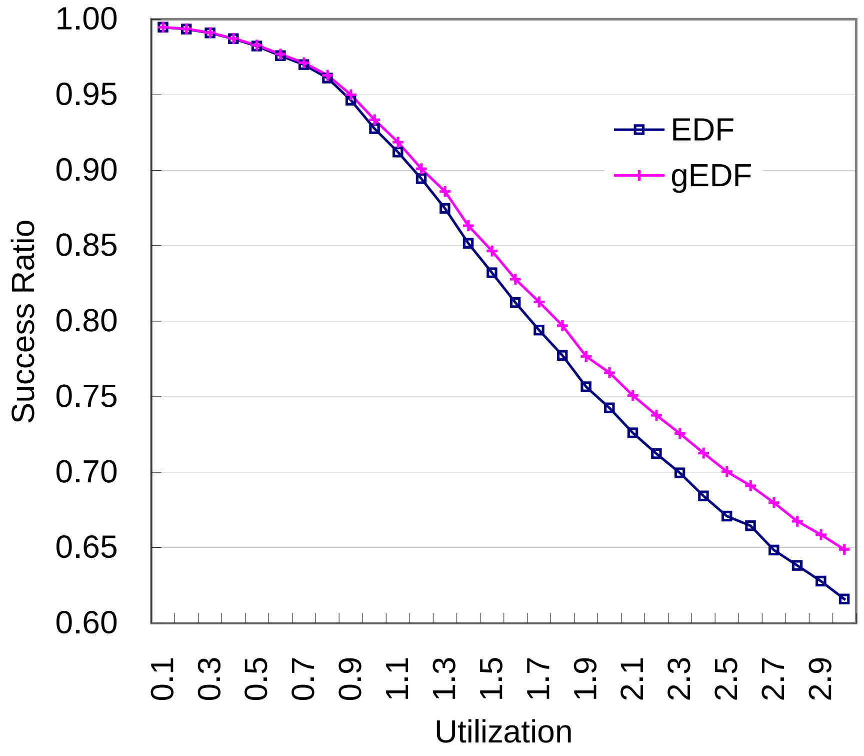

Figure 4.16: Group Window Size = μ_e /4.

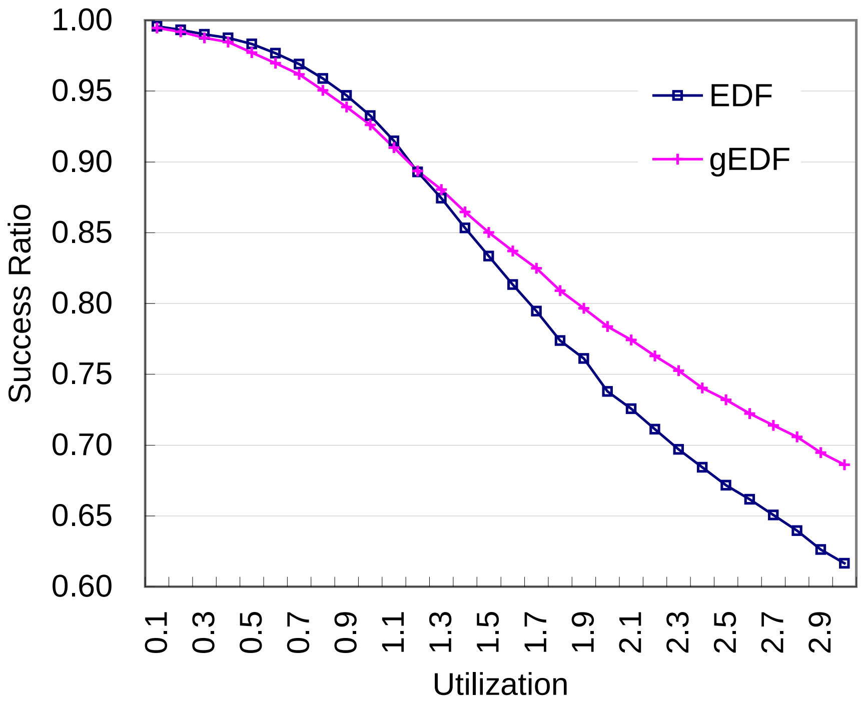

Figure 4.17: Group Window Size = 4*μ_e.

4.1.4 Experiment 4 – Effect of the Values of Single μ_e on γ

In this section I change the mix of tasks by using different execution times, generated using exponential distributions with means (i.e. μ_e). The following figures show the effect of different values of μ_e on the success rates achieved by gEDF and EDF. The values of μ_e are listed in absolute values.

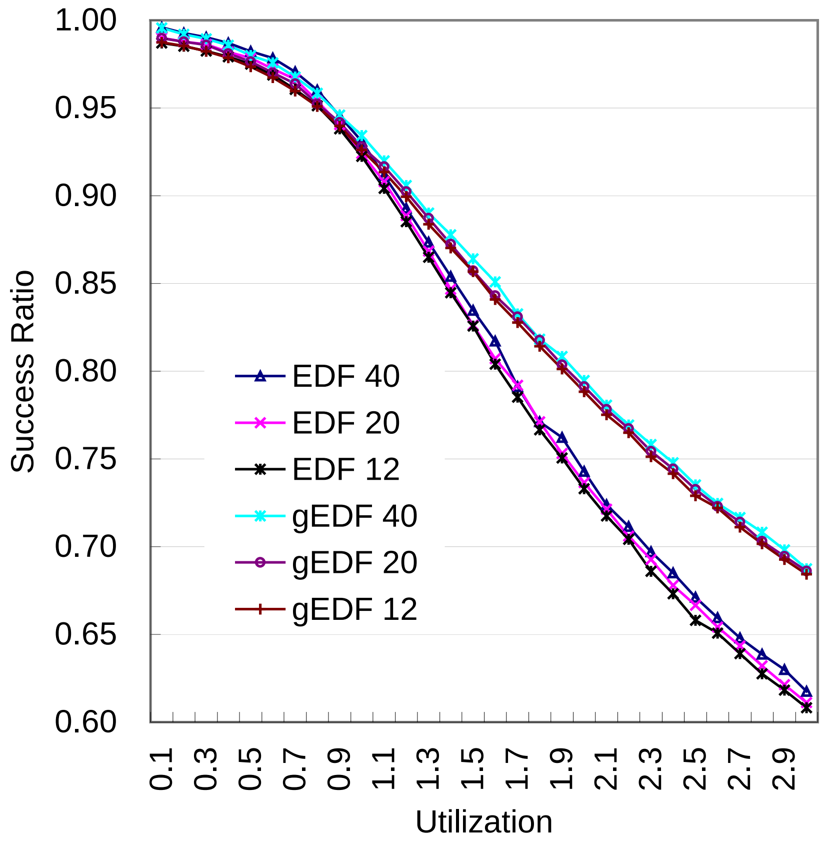

Figure 4.18: Success ratios of EDF and gEDF when μ_e = 40, 20, and 12.

In Figure 4.18, there is no apparent difference in the success ratios when μ_e changes from 40, 20, to 12 either for EDF or gEDF. The lower three curves represent data for the EDF algorithm; the upper three curves represent data for the gEDF algorithm.

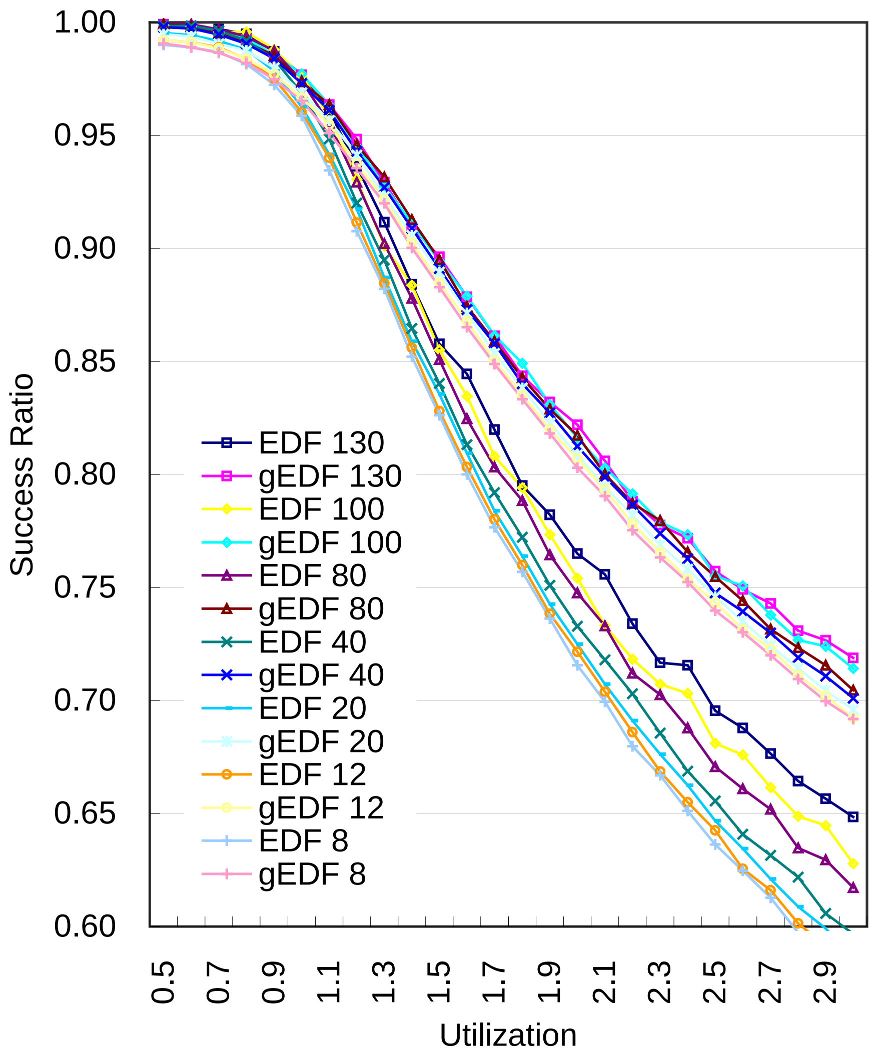

Figure 4.19: Success Ratio of EDF and gEDF when μ_e = 130, 100, 80, 40, 20, 12, and 8.

In Figure 4.19, there is no obvious difference in the success ratios of gEDF when μ_e changes from 130, 100, 80, 40, 20, 12, to 8. This data indicates that the success rates of EDF and gEDF are sensitive to the load which is given by (μ_r /μ_e) but not to the execution time parameter μ_e alone.

4.1.5 Experiment 5 – Effect of *Tr* on Response Time ($\Re$)

Thus far I have shown that gEDF results in higher success rates than EDF, particularly when the system is overloaded. Next, I will compare the average response times achieved using gEDF with those resulting from EDF. Intuitively, completing shorter jobs first should result in faster response times[4]. Our experiments support this. I set μ_e = 40, μ_D = 5μ_e, *Gr* = 0.4. Figures 4.20, 4.21, and 4.22 show that gEDF can yield faster response times than the response time when using EDF, and when soft real-time tolerance parameter *Tr* is changed from 0 to 0.5 to 1.0, respectively.

[4] Of course, this can be viewed as an unfair schedule since longer jobs will less likely be scheduled for execution.

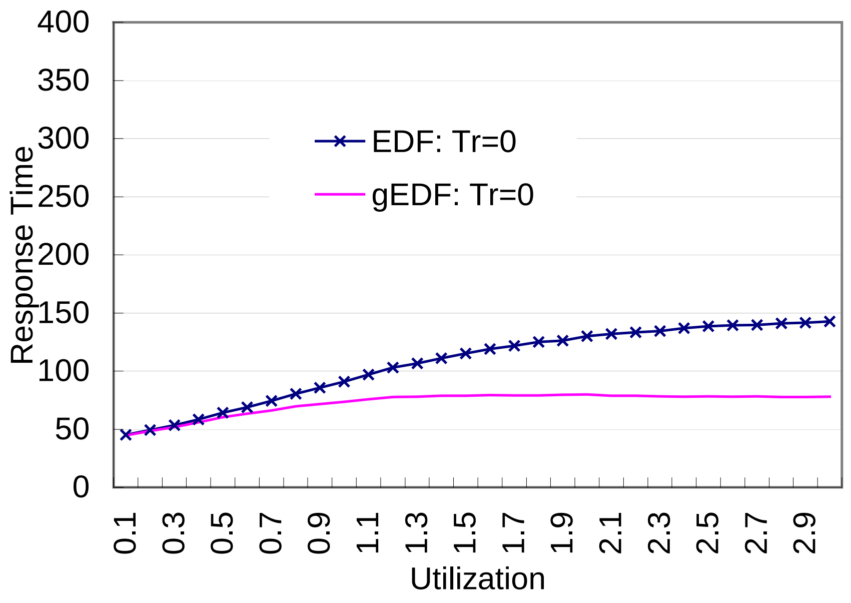

Figure 4.20: Response time when deadline tolerance *Tr* = 0.

As can be seen from these figures, gEDF outperforms EDF in terms of the response time, when the system is lightly loaded (about ρ = 0.5) and when the system is heavily loaded (ρ = 3.0). In this experiment, when deadline tolerance *Tr* is 0 (i.e. hard real-time), gEDF outperforms EDF. For instance, when ρ = 1.0, gEDF yields 24% faster response times when compared to the response times using EDF; when ρ = 2.0, gEDF yields 63% faster response times than EDF. Likewise, with a deadline tolerance *Tr* of 0.5, gEDF results in faster response time under both lightly loaded and heavily loaded situations. For example when ρ = 1.0, gEDF yields 33% faster response time than EDF; when ρ = 2.0, gEDF yields 59% faster response time than EDF. This is also the case when deadline tolerance *Tr* is 1.0. For instance, when ρ = 1.0, gEDF yields 20% faster response times than EDF; when ρ = 2.0, gEDF yields 35% faster response time than EDF. Simply increasing *Tr* cannot guarantee higher performance of gEDF. With larger values of tolerance *Tr*, as previously noted both gEDF and EDF achieve higher success rates. Thus the performance gains achieved by gEDF over that of EDF will become less pronounced for larger deadline tolerance parameters. In the limit, when *Tr* is set to a very large value, the system is no longer a real-time system and both EDF and gEDF can achieve 100% success rates, since deadlines are no longer meaningful. In such cases, gEDF does not show any performance improvements over that of EDF.

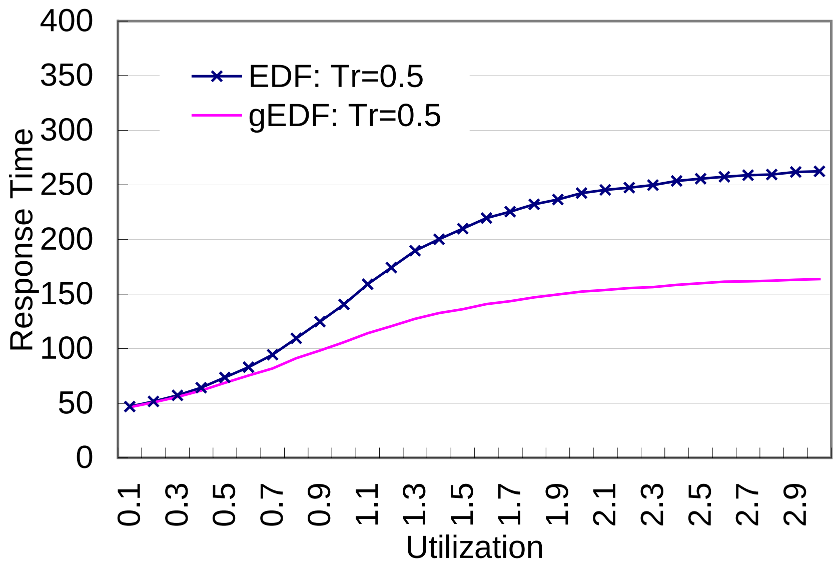

Figure 4.21: Response time when deadline tolerance *Tr* = 0.5.

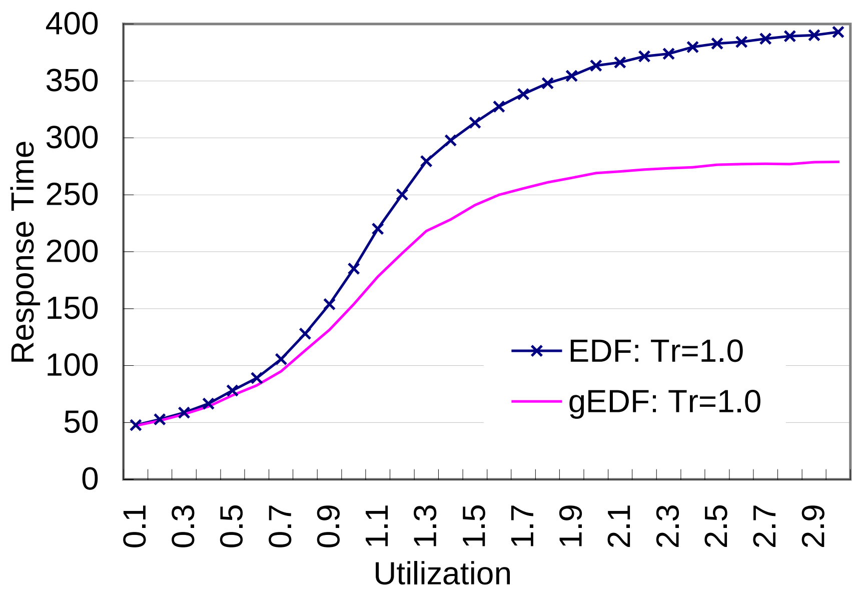

Figure 4.22: Response time when deadline tolerance *Tr* =1.0.

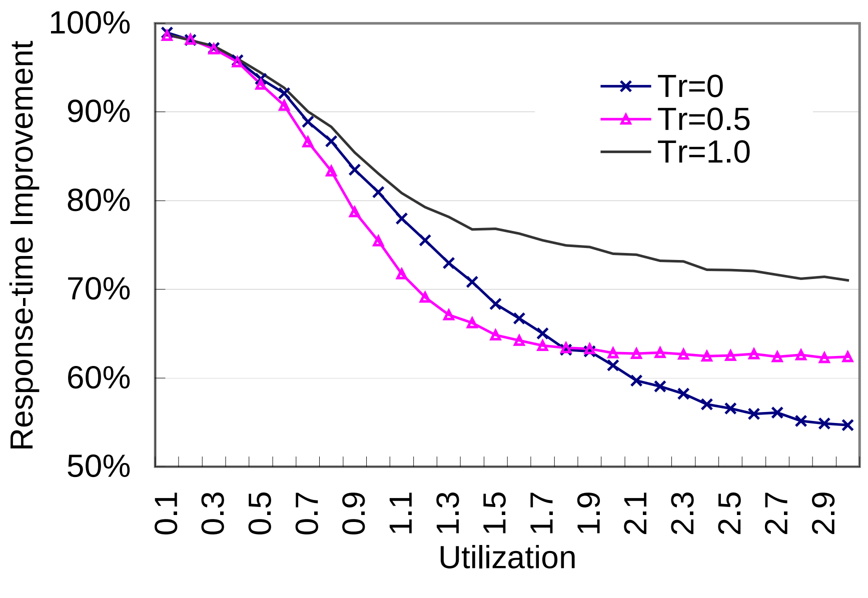

Figure 4.23: The ratio of response time of gEDF vs. response time of EDF.

Figure 4.23 above summarizes the response time improvements achieved by gEDF when compared with the response times of EDF. Note that that Y-axis shows the relative response times (and smaller numbers are better).

4.1.6 Experiment 6 - The Effect of Tight Deadlines on $\Re$

I set $\mu_r = \mu_e/\rho$, $\mu_e = 40$, $Gr = 0.4$, $Tr = 0.1$. Figures 4.24 and 4.25 show the change in response time of EDF and gEDF when μ_D is varied from μ_e, $2\mu_e$, $5\mu_e$, and $10\mu_e$. Like the success ratios of EDF and gEDF, when μ_D is very small such as μ_e and $2\mu_e$, there is no difference between EDF and gEDF. This is because both EDF and gEDF will have very low success rates. However, as μ_D is increased, gEDF results in faster response times when compared with EDF.

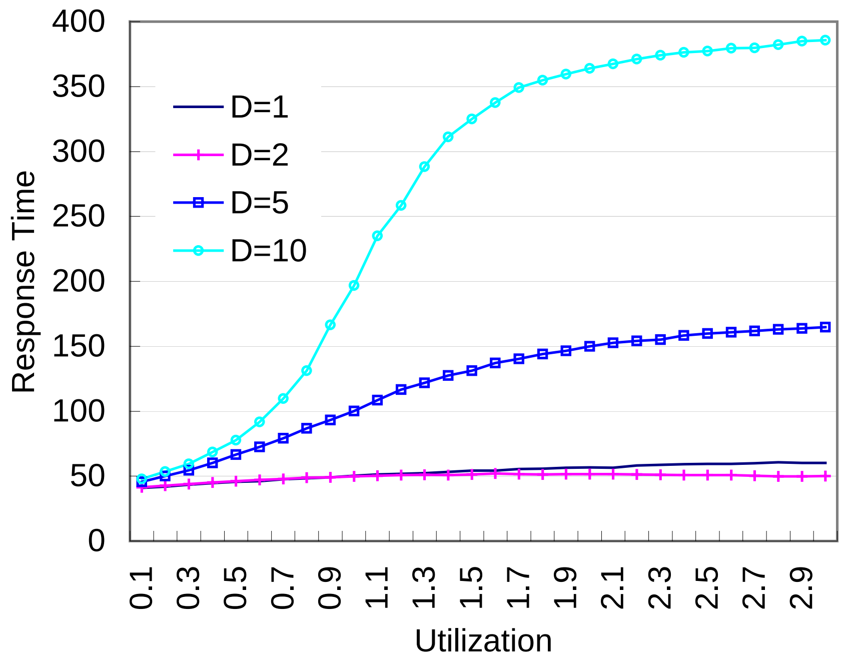

Figure 4.24: Response time of EDF when $\mu_D = \mu_e$, $2\mu_e$, $5\mu_e$, and $10\mu_e$.

The response times of EDF do not show significant differences for $\mu_D = \mu_e$ and $2\mu_e$. However, as μ_D is set to $5\mu_e$ and $10\mu_e$, EDF shows improved response times.

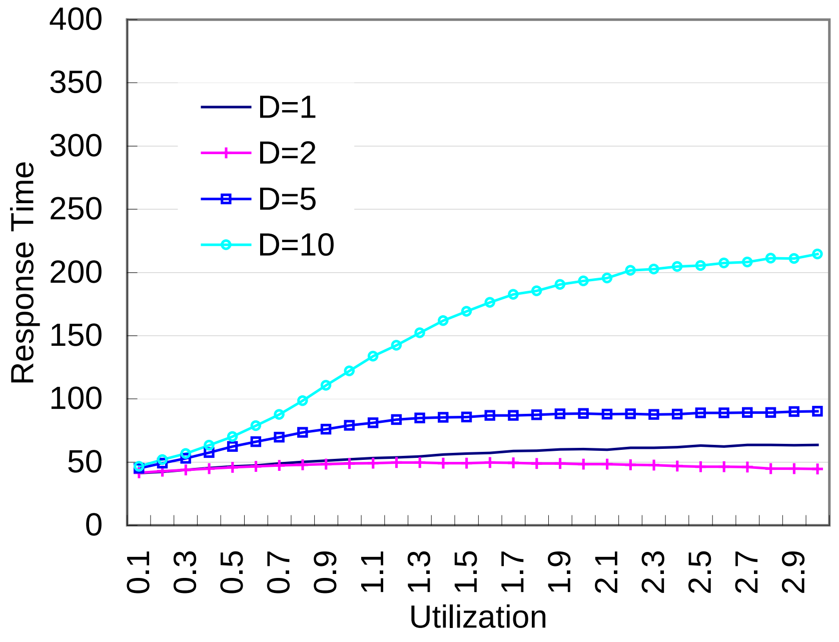

Figure 4.25: Response time of gEDF when μ_D = μ_e, 2μ_e, 5μ_e, and 10μ_e.

Similar behavior can be observed with gEDF; the response times when μ_D = μ_e and 2μ_e are about the same, but the response times show more dramatic improvements for μ_D = 5μ_e and 10μ_e.

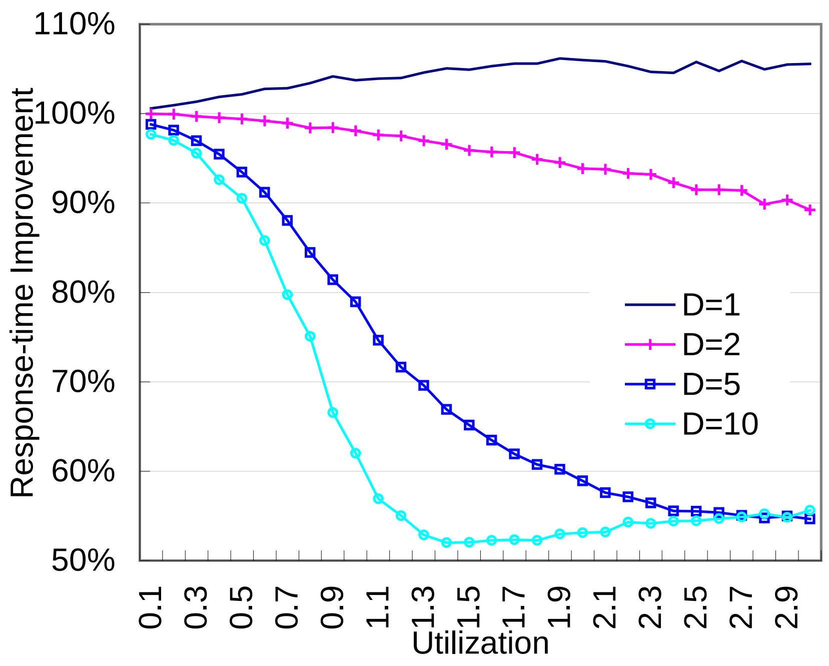

Figure 4.26: The ratio of response time of gEDF vs. response time of EDF when $\mu_D = \mu_e$, $2\mu_e$, $5\mu_e$, and $10\mu_e$.

Figure 4.26 above summarizes the improvements in response time achieved by gEDF when compared with EDF. Note that the Y-axis shows the relative response times (and smaller numbers are better).

4.1.7 Experiment 7 - The Effect of Single μ_e on $\Re$

To understand the properties of response time $\Re$, at some specific ρ, I defined response-time ratio ∂ = average response Time / μ_e. The following figures show the result when μ_e = 40, 20, and 12 for EDF (Figure 4.27) and for gEDF (for Figure 4.28). The results show that μ_e has little effect on response-time ratio ∂. The system load plays more critical role on response times.

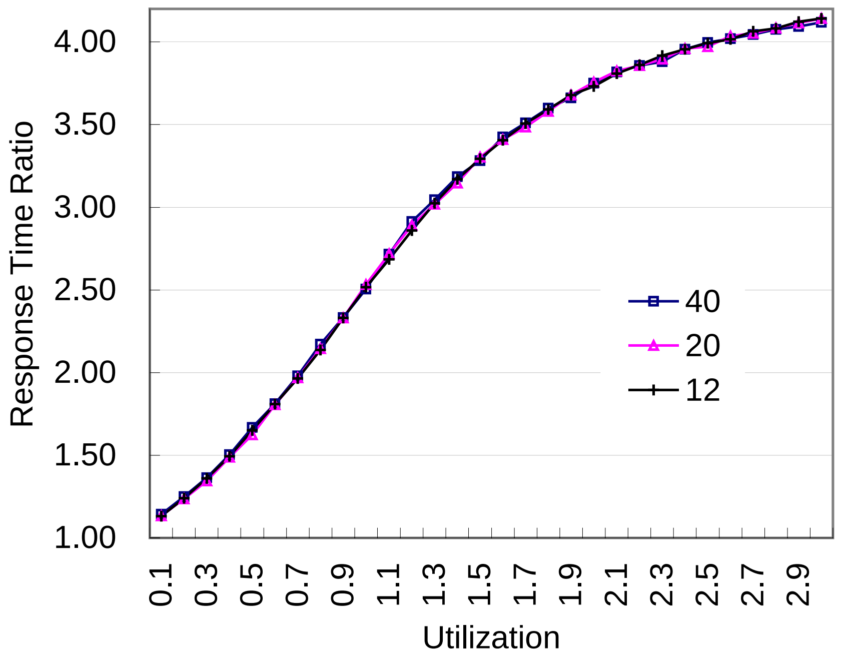

Figure 4.27: Response-time Ratio of EDF when μ_e = 40, 20, and 12.

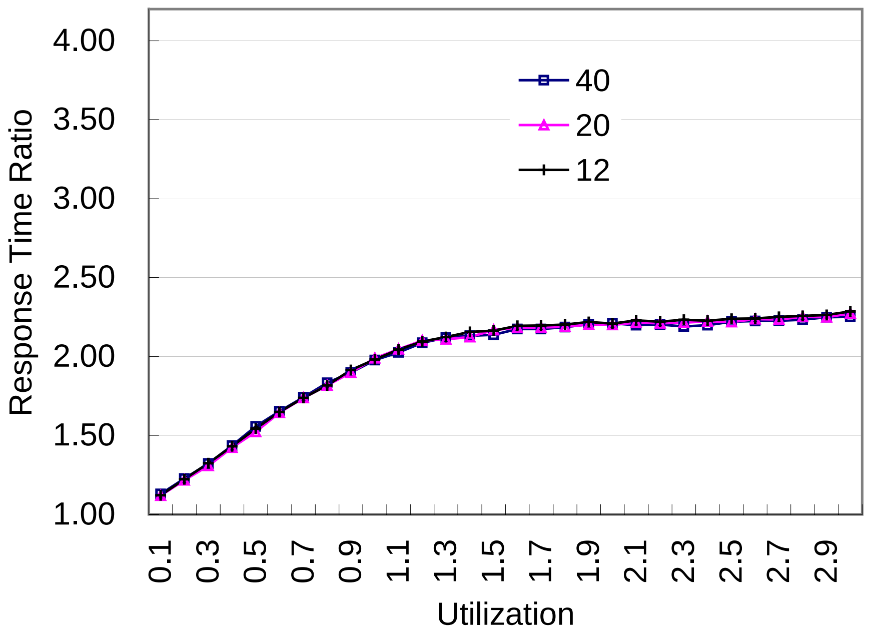

Figure 4.28: Response-time Ratio of gEDF when μ_e = 40, 20, and 12.

4.2 The Effect of Multiple Expected Execution Times

4.2.1 Experiment 8 – The Effect of Multiple μ_es on γ

The jobs generated for all the experiments thus far were generated using a single exponential distribution. To evaluate the impact of the case when jobs come from different classes, I generated tasks using different exponential distributions with different mean values.

I designate job classes using (*m*, *n*) where *m* represent the mean value of the distribution used to generate execution times of tasks, and *n* represents the fraction of all jobs (out of *N*) that are generated with the mean *m*.

Set-1: This is the base line consisting of jobs drawn from a single exponential distribution. I generate *N* jobs using an exponential distribution with a mean μ_e to represent average (or worst case expected) execution time. I will designate this set of jobs as (μ_e, *N*).

Set-2: Here we have two types of jobs, one generated using a mean of (1/2)*μ_e, and the second with a mean of μ_e. Sixty-six percent of the jobs have a mean execution time of (1/2)μ_e. This set is designated by (1/2μ_e, 2/3*N*) and (μ_e, 1/3*N*).

Set-3: This set contains three classes of jobs generated using mean execution times of 1/4μ_e, 1/2μ_e, and μ_e. I designate this set as (1/4μ_e, 4/7*N*), (1/2μ_e, 2/7*N*), and (μ_e, 1/7*N*). Remember that the second number in each tuple represents the fraction of total number of jobs of each class.

Figure 4.29 shows that, when *Tr* is 0 (hard real-time), the different classes of jobs, even when there are more small jobs do not improve the success ratios. The difference in the performance among the different sets of job classes is less than 1%.

However, when dealing with soft real-time jobs (with a deadline tolerance *Tr* of 0.2 and 0.5), job classes do impact performance gains of gEDF as shown in Figures 4.30 and Figure 4.31. Note that Set 2 and Set 3 have more small jobs than Set 1. As expected gEDF results in higher success rates over EDF when there are more small jobs. For example for *Tr* = 0.2 and ρ = 2.0, Set-3 can result in 3% performance improvement than Set-2; Set-2 can result in 3% more performance improvement than Set-1. When *Tr* = 0.5 and, when ρ = 2.0, Set-3 can result in 15% performance improvement than Set-2; Set-2 can result in 15% more performance improvement than Set-1.

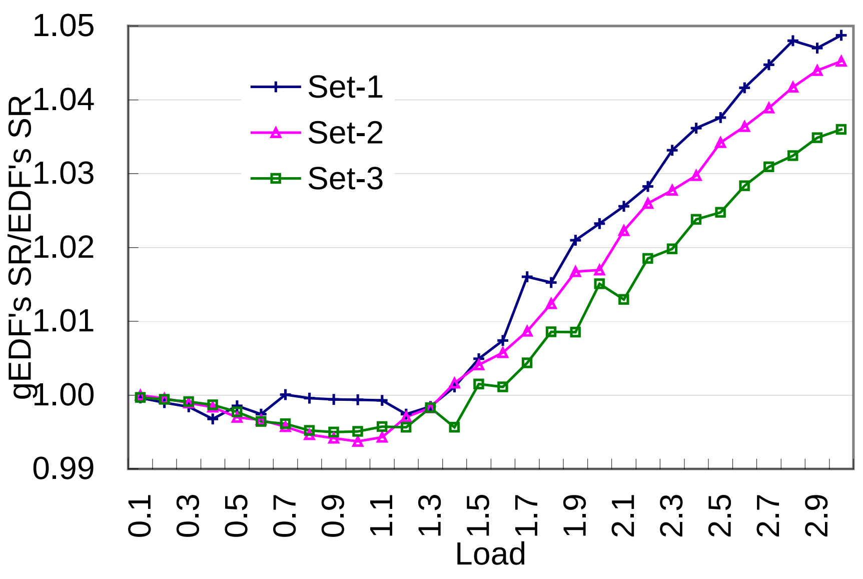

Figure 4.29: Success ratio of gEDF/success ratio of EDF when *Tr* = 0.

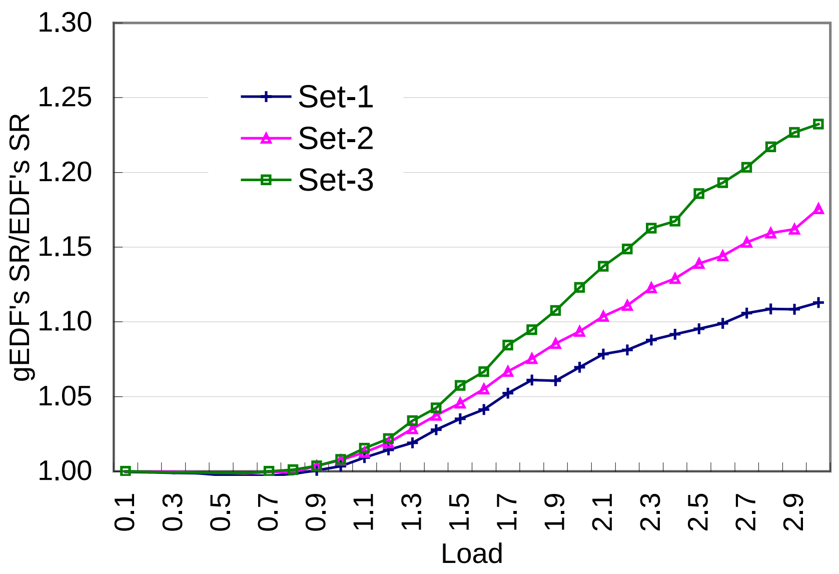

Figure 4.30: Success ratio of gEDF/success ratio of EDF when *Tr* = 0.2.

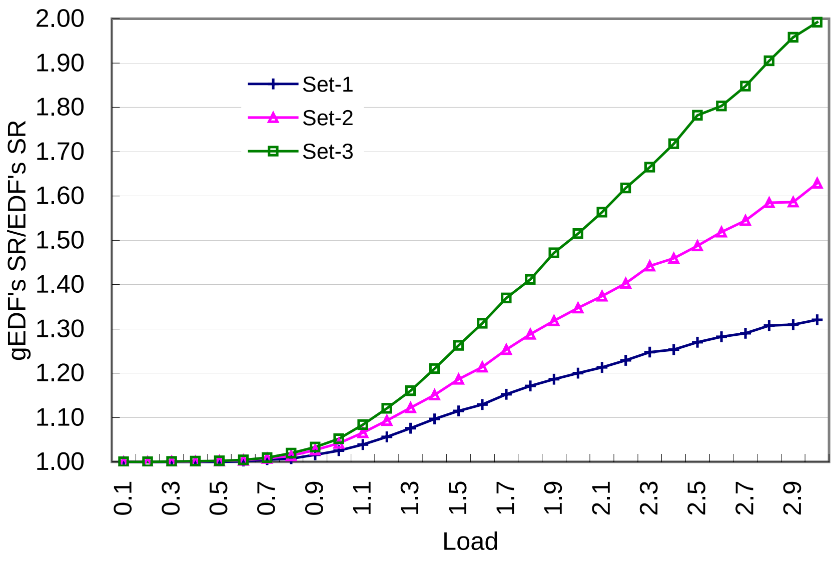

Figure 4.31: Success ratio of gEDF/success ratio of EDF when *Tr* = 0.5.

4.2.2 Experiment 9 – The Effect of Percentage of Small Jobs on γ

Previously, I analyzed the effect of data sets with different job classes on success rates and observed that a workload with more small jobs show higher success rates with gEDF over that of EDF. In this section, I will analyze the success ratios where I use two different job classes (with two different μ_es) but change the percentage of small jobs in the mix.

Distribution 1: 1/2 jobs with μ_e, 1/2 jobs with μ_e.
Distribution 2: 1/2 jobs with μ_e, 1/2 jobs with 1/2 μ_e.
Distribution 3: 2/5 jobs with μ_e, 3/5 jobs with 1/3 μ_e.
Distribution 4: 1/5 jobs with μ_e, 4/5 jobs with 1/8 μ_e.

I set *Tr* = 0.5. Table 1 presents the success ratio of gEDF/success ratio of EDF. Table 4.2 and Figure 4.32 show that the distribution with a larger percentage of small jobs obtains higher success ratio of gEDF when compared to the success ratio of EDF. Note that Distribution 4 has the most small jobs of any other distribution, and the data shows that gEDF benefits from this fact.

Table 4.2: The performance change (success ratio of gEDF/success ratio of EDF) for different percentages of small jobs

Load	Distribution 1	Distribution 2	Distribution 3	Distribution 4
0.1	1.000067	1.000274	0.999868	0.999877
0.2	1.000168	1.000265	1.000204	1.000363
0.3	1.000302	1.000253	1.000657	1.001042
0.4	1.000336	1.000864	1.001237	1.001779
0.5	1.000556	1.001620	1.003070	1.002737
0.6	1.002016	1.003082	1.005708	1.006286
0.7	1.003893	1.006661	1.010871	1.009740
0.8	1.007126	1.013990	1.019568	1.017424
0.9	1.014541	1.026174	1.035230	1.032902
1.0	1.025803	1.042502	1.055736	1.057957
1.1	1.040473	1.067228	1.084512	1.093741
1.2	1.058057	1.088816	1.120055	1.140386
1.3	1.071179	1.118099	1.167916	1.203977
1.4	1.093686	1.151395	1.208859	1.289895
1.5	1.114320	1.180786	1.252292	1.381070
1.6	1.132359	1.214086	1.311561	1.472425
1.7	1.153179	1.245111	1.353332	1.579499
1.8	1.164161	1.279391	1.407337	1.696627
1.9	1.185979	1.307189	1.459737	1.811491
2.0	1.202503	1.342744	1.512752	1.941354
2.1	1.213558	1.361546	1.551537	2.070311
2.2	1.230223	1.397064	1.592382	2.169966
2.3	1.251887	1.418970	1.636869	2.297082
2.4	1.260878	1.446233	1.681175	2.411978
2.5	1.272357	1.478179	1.722826	2.572307
2.6	1.282948	1.492337	1.774248	2.665443
2.7	1.289757	1.524958	1.820086	2.799853
2.8	1.309416	1.554312	1.867880	2.969026
2.9	1.330402	1.575254	1.900146	3.039547
3.0	1.323816	1.593459	1.942890	3.210040

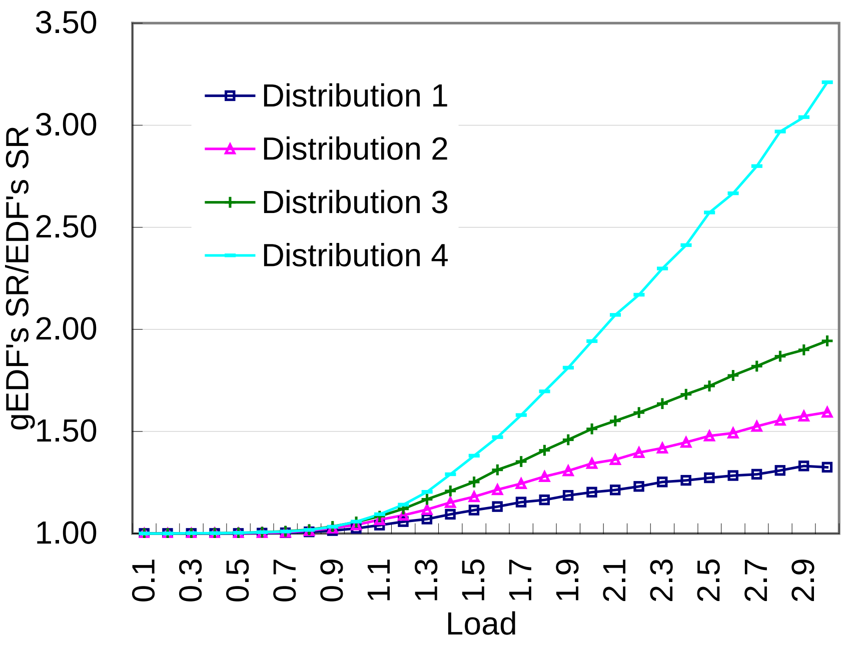

Figure 4.32: Success ratio of gEDF/success ratio of EDF *Tr* = 0.5.

For instance, Distribution 4, which has the most small jobs of any other distribution, shows that gEDF can achieve 200% higher success ratios than EDF. In comparison, Distribution 1, which has the fewest smaller jobs of any other distribution, shows only 120% higher success ratios than EDF.

4.3 Comparisons of gEDF, Best-Effort, and Guarantee Algorithms

4.3.1 Experiment 10 - Comparison of γ of gEDF and Best-Effort

Our gEDF method is not only an EDF-based overload strategy, it can also be used in underloaded conditions. I have shown that gEDF not only shows better performance than EDF when the system is overloaded, but performs as well as EDF when the system is underloaded. Thus, there is no need to switch between EDF and gEDF based on system load[5]. Researchers have explored adaptive algorithms to control the performance when the system is overloaded. One such algorithm is called the best-effort algorithm. In this dissertation, I will use the same best effort criteria (i.e., value-density: *V*/*C*) that Locke [9] used. To achieve a fair comparison of gEDF with best-effort, I set the same environments for gEDF and best-effort. For our experiments here the value-density *V*/*C* is set equal for all jobs. According to Locke's best-effort, depending on the utilization, if the system is not overloaded (utilization $\leq$ 1.0), best-effort becomes EDF; if the system is overloaded (utilization > 1.0), best-effort will schedule jobs with high *V/C* ratios in an attempt to maximize the overall value of the system.

The best-effort requires an estimation or prediction of utilization for switching between EDF algorithm and the best-effort. While it may be possible to predict the system load when the system processes only periodic jobs, it is very difficult to compute the system load if the system processes a mixture of periodic, aperiodic, and sporadic jobs. Recently, synthetic utilization bound has been

[5] It should be noted that gEDF does favor smaller jobs and thus cannot guarantee fairness.

proposed to measure real utilization. For the EDF-based schemes, however, synthetic utilization is very close to real utilization [37] and is an appropriate choice. Moreover, the estimated loads are imprecise because most real-time systems rely on worst-case execution times (WCET), while in most cases, the actual execution times are lower than these estimates. Switching to best-effort based on such imprecise load estimations can lead to inefficient utilization of the resources. In this dissertation, I use a clairvoyant scheme or profiling based on actual execution times of the real-time jobs. Thus, the comparisons shown are present most optimistic scenarios as far as the best-effort algorithm is concerned.

I set $\mu_r = \mu_e/\rho$, $\mu_e = 20$, $\mu_D = 5\mu_e$, $Gr = 0.4$. Figures 4.33 and 4.34 show that gEDF achieves higher success rates than best-effort when the deadline tolerance is varied to $Tr = 0.2$, 0.5, and 1.0.

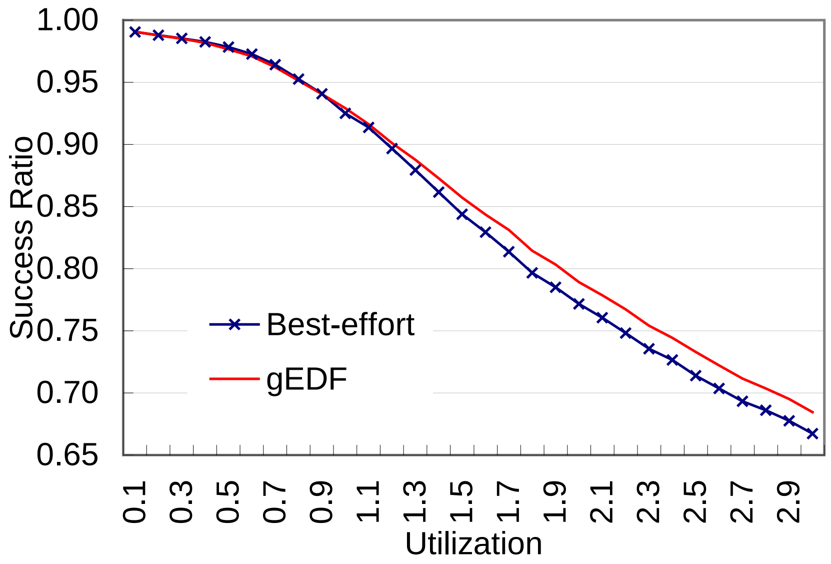

Figure 4.33: Success rates when deadline tolerance is 0.2.

Although the improvement of success ratios of gEDF are not significant, considering the difficulty of predicting the precise utilization required by best-effort, any improvements gained by gEDF should be viewed in a positive light.

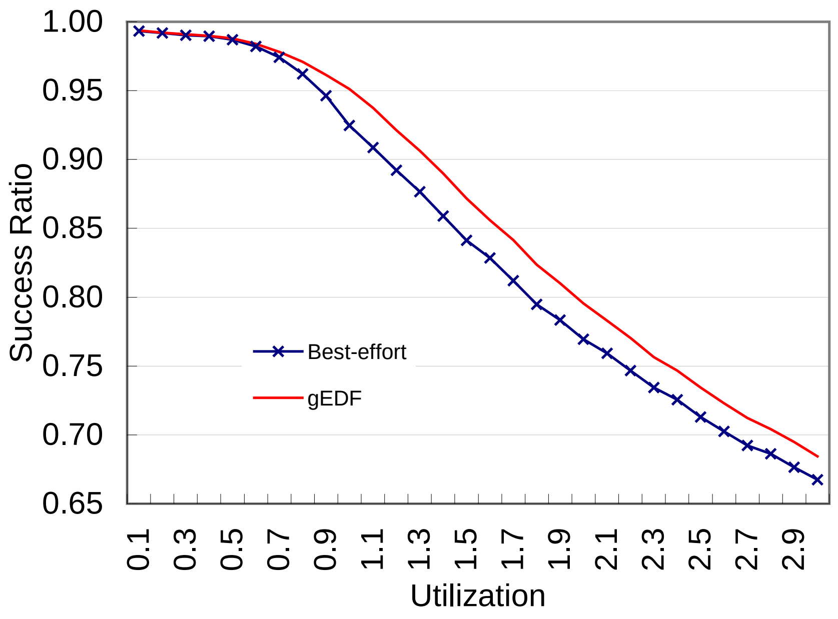

Figure 4.34: Success rates when deadline tolerance is 0.5.

The performance gains (i.e. success ratios) achieved by gEDF are even greater when the deadline tolerance is very lenient say 50%, as shown in Figure 4.34.

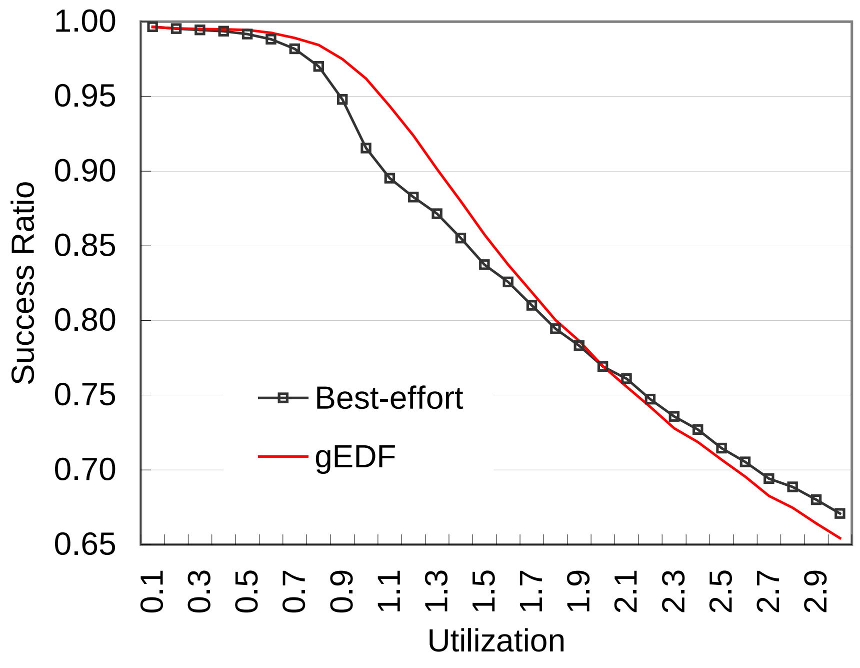

Figure 4.35: Success rates when deadline tolerance is 1.0.

When deadline tolerance is 100%, as shown in Figure 4.35, gEDF is better than best-effort for most loads except when the system is very heavily loaded but it should be noted that both gEDF and best-effort achieve very low success rates at such loads.

4.3.2 Experiment 11 – Comparison of $\mathfrak{R}$ of gEDF and Best-Effort

I have shown that gEDF results in higher success rates when compared with best-effort, particularly when the system is overloaded and the deadline tolerances are very lenient. Here I will compare the average response times achieved using gEDF with those achieved using best-effort. I set $\mu_r = \mu_e/\rho$, $\mu_e =$

20, μ_D = 5μ_e, *Gr* = 0.4. Figures 4.36, 4.37, and 4.38 show that gEDF can yield faster response times than best-effort (except when the loads are very high in Figure 4.38).

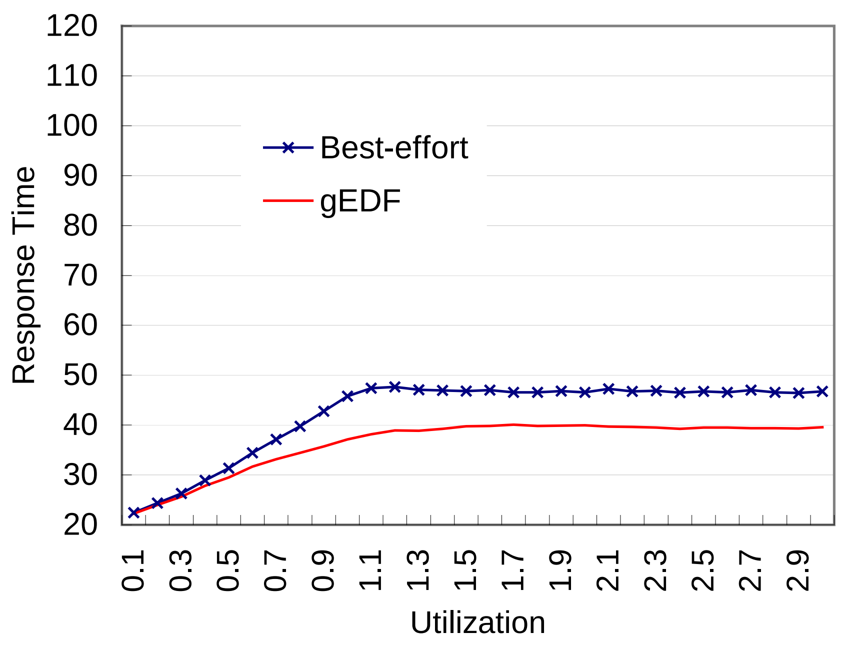

Figure 4.36: Response time when deadline tolerance is 0.

When *Tr* = 0 (i.e. hard real-time), gEDF can yield faster response times than best-effort in underloaded and overloaded conditions. For instance, gEDF can yield 30% faster response times than best-effort when ρ = 1.0; and gEDF can yield 20% faster response times than best-effort when ρ = 2.0.

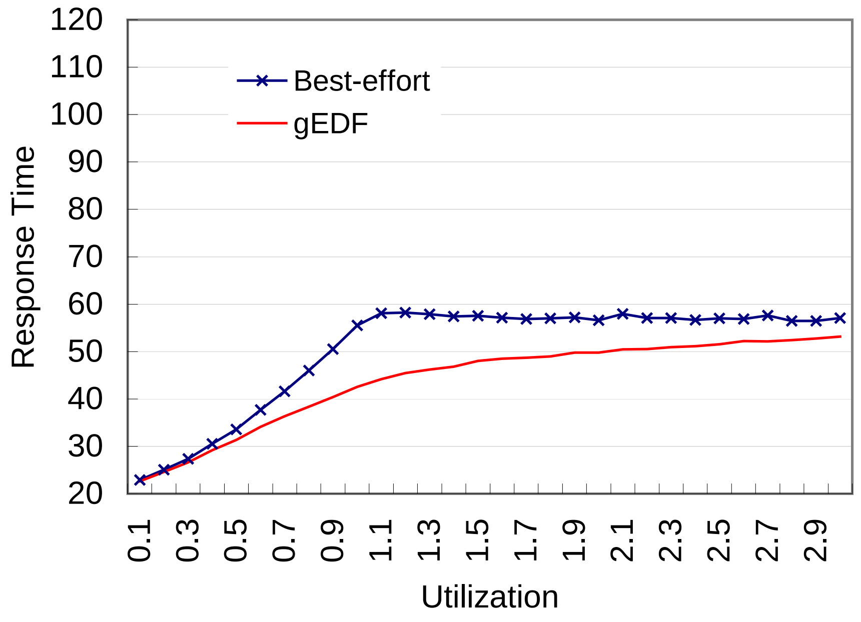

Figure 4.37: Response time when deadline tolerance is 0.2.

When *Tr* = 0.2, gEDF can result in even faster response times than best-effort in underloaded and overloaded situations.

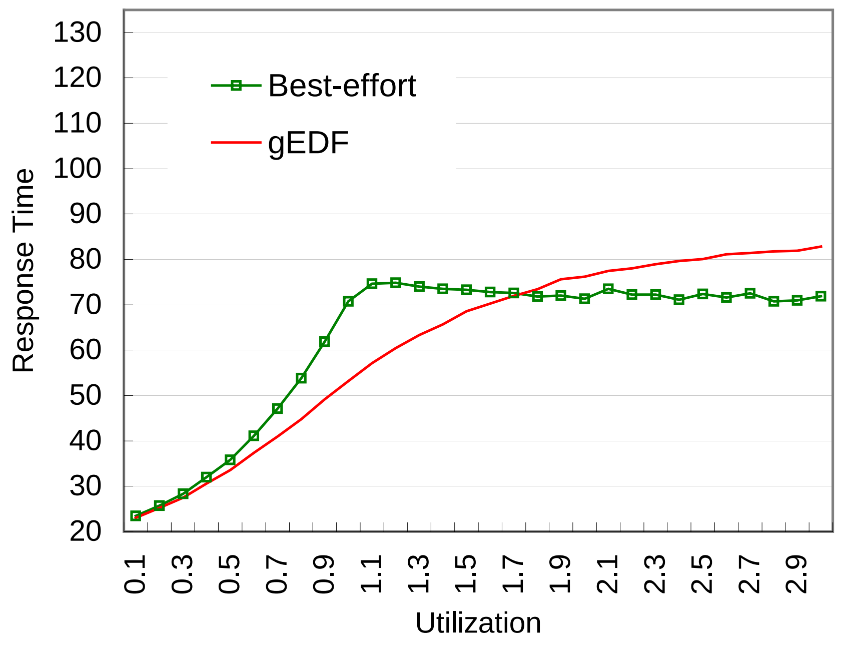

Figure 4.38: Response time when deadline tolerance is 0.5.

When *Tr* = 0.5, gEDF can still yield faster response times than best-effort until the system load reaches 1.7. It shows that choosing an appropriate deadline tolerance has a positive effect on average response time when using gEDF.

4.3.3 Experiment 12 – Comparison of γ of gEDF and Guarantee

The guarantee algorithm is inappropriate for soft real-time systems. However, I include the guarantee scheme algorithm (referred as guarantee) here only for the sake of completeness. When the system is underloaded, guarantee uses EDF for scheduling; when the system is overloaded, guarantee uses a specific policy to choose real-time jobs and guarantees execution of the jobs by their deadlines. In the simulations used here, incoming jobs are chosen for inclusion in the guarantee if they can be scheduled by their deadlines, without discarding any jobs already guaranteed.

I set $\mu_r = \mu_e/\rho$, $\mu_e = 20$, $\mu_D = 5\mu_e$, $Gr = 0.4$. Figures 4.39 and 4.40 show the success ratios of all the real-time scheduling algorithms discussed in this dissertation, including the guarantee algorithm, best-effort, EDF, and gEDF. Note that for guarantee algorithm, the success rate drops precipitously because tasks are rejected at a higher rate as the system load increases beyond 100%.

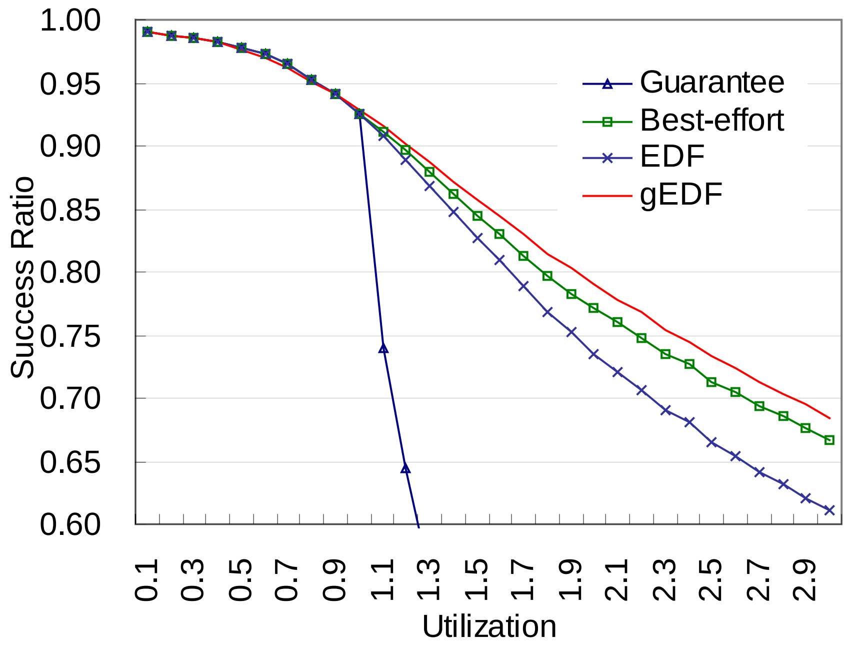

Figure 4.39: Success ratio when deadline tolerance is 0.2.

When *Tr* = 0.2, gEDF yields the highest success ratios than all the other methods when the system is overloaded.

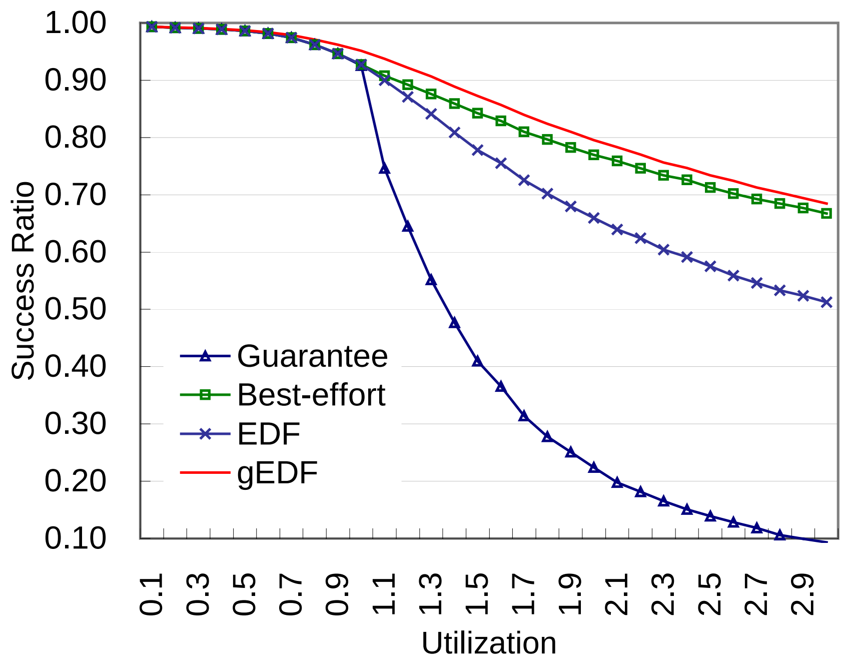

Figure 4.40: Success ratio when deadline tolerance is 0.5.

Our gEDF also outperforms all other methods when *Tr* = 0.5, and when the system is overloaded.

4.3.4 Experiment 13 – Comparison of $\Re$ of gEDF & Guarantee

In the final experiments, I summarize the response time performance of gEDF, EDF, best-effort and guarantee scheduling methods. I set $\mu_r = \mu_e/\rho$, μ_e = 20, $\mu_D = 5\mu_e$, *Gr* = 0.4. Figures 4.41 and 4.42 show the results.

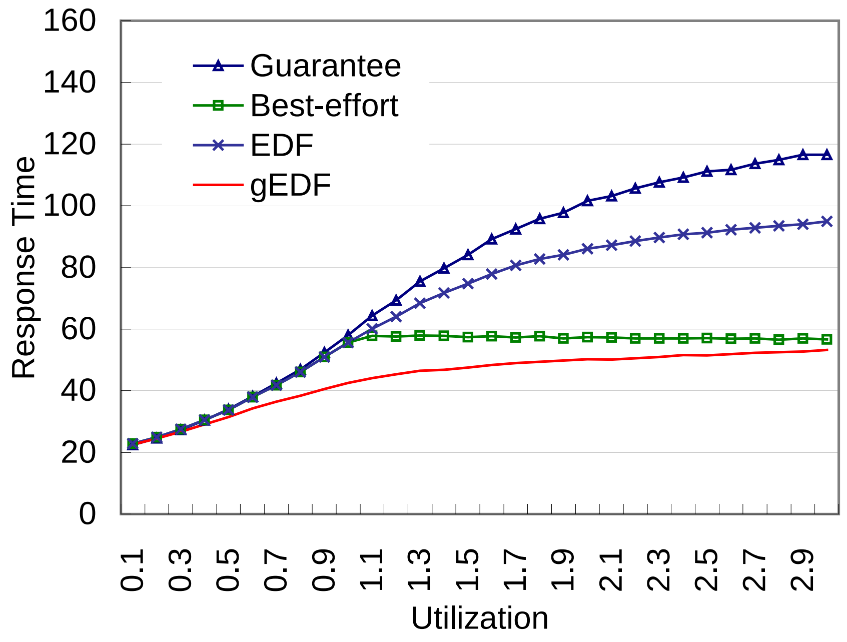

Figure 4.41: Response times when deadline tolerance is 0.2.

When *Tr* = 0.2, gEDF outperforms all other methods. For instance, when ρ = 1.0, gEDF yields about 35% faster response time than best-effort, EDF, and guarantee. Best-effort has the same performance as EDF and guarantee in underloaded but outperforms them in overloaded conditions.

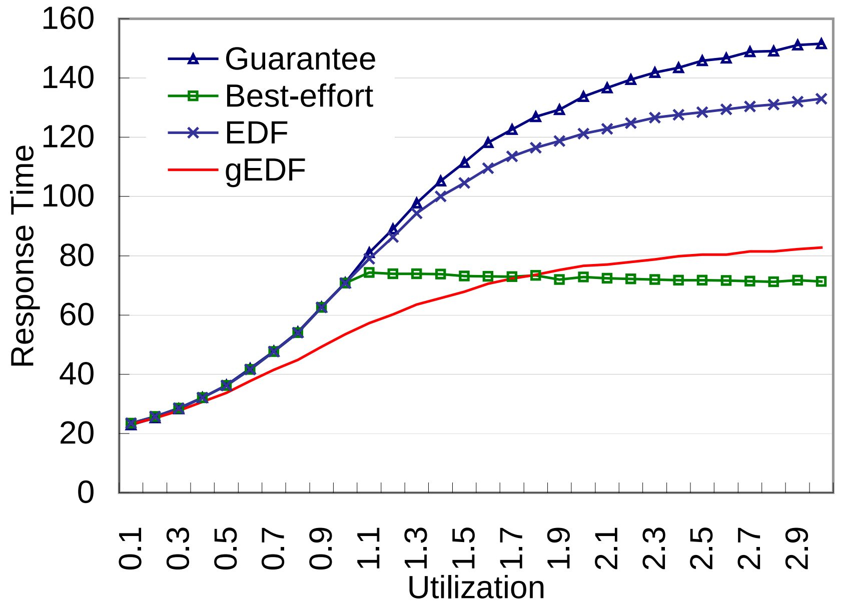

Figure 4.42: Response times when deadline tolerance is 0.5.

Our gEDF has faster response time than EDF and guarantee schemes when *Tr* = 0.5. Our gEDF outperforms the best-effort method when the system is not heavily loaded, but has slower response times than the best-effort technique when the system load exceeds 1.7.

CHAPTER 5

IMPLEMENTATION OF gEDF IN THE LINUX KERNEL

In this chapter, I will introduce practical and commercial real-time operating systems. I will show how our gEDF can be implemented in Linux systems and evaluate gEDF using a real workload.

5.1 Enhancing Linux with the gEDF Scheduling Scheme

The real-time operating system must deal with scheduling of jobs to meet timing constraints, and achieve desired response time, particularly in systems that are designed to process a mix of real-time and non real-time tasks.

Linux, as the most popular open source platform, fully supports POSIX 1003.1a and POSIX 1003.1c, but only partially supports the real-time extension of POSIX 1003.1b [31, 32]. For example, Linux doesn’t support real-time features such as system timers or message queues. It only supports a few simple scheduling policies, such as round robin (RR) and FIFO. In spite of that, most commercial and open source real-time operating systems (RTOS) are Linux-based.

In chapter 2, I introduced some examples of real-time operating systems based on the Linux kernel [37, 38]. Various scheduling policies are implemented in different RTOS, but EDF is not a common algorithm implemented in commercial systems. Since EDF is known for its efficiency in scheduling real-time tasks, some recent RTOS systems are providing for EDF based extensions to basic scheduling policies.

In our research, it is not our goal to provide a complete and full real-time OS, but only to compare EDF with gEDF in a realistic real-time environment, to complement our experiments described in the previous chapter. I also wanted to explore how difficult it would be to implement gEDF (and EDF) in a practical system. For this purpose, I implemented gEDF in the Linux kernel. Some typical real-time benchmarks are executed on the modified Linux kernel. The results show that gEDF can achieve (task completion) success ratios that are at least as good as or better than EDF success ratios. In addition, gEDF produces better response times to real-time tasks than EDF, both when the system is underloaded and overloaded. These results are in line with our observations from experiments described in Chapter 4. In this chapter I will focus on success rates of gEDF and EDF only. I will also provide a general framework for implementing EDF based algorithms that require additional parameters to describe task execution times and deadlines.

To support EDF and gEDF in the Linux kernel, two parameters for EDF and gEDF will be created. The related structures, particularly task structure, will be extended to support EDF and gEDF. Separate new runqueue for EDF and gEDF will be created without affecting the original runqueue for non real-time tasks of the Linux kernel. Because POSIX 1003.1b doesn’t support EDF or gEDF, I also need system calls to define the EDF or gEDF policies for real-time tasks. New scheduling functions to implement EDF and gEDF are also added to the Linux kernel. Inserting a new active task into the runqueue is the same for EDF and gEDF. However, selecting a task for scheduling from the runqueue is different for EDF and gEDF. The EDF algorithm is simple and straightforward; it selects the task at the head of the runqueue since the queue is already sorted by deadlines. The gEDF algorithm requires more computation to select the shortest task among a group of tasks. I will introduce these algorithms in more detail in the next section.

5.2 Modification of the Linux Kernel

5.2.1 Modification of Structure task_struct in the Linux Kernel

A normal process executes in its user space. When it executes a system call or triggers an exception, the kernel space is entered. If a higher priority process has become runnable in the interim, the Linux scheduler is invoked to select the higher priority task for running. Otherwise, it exits the kernel upon finishing the system call or exception ending.

Each task can be in one of six states.

TASK_RUNNING	0
TASK_INTERRUPTIBLE	1
TASK_UNINTERRUPTIBLE	2
TASK_STOPPED	4
TASK_ZOMBIE	8
TASK_DEAD	16

All processes in Linux are descendents of the init process, whose PID (process ID) is 1. The kernel starts init in the last step of the boot process. The existing task uses the fork() call to create a new process and enter TASK_RUNNING state. The Linux kernel scheduler dispatches tasks in runqueue to run. The running task may be preempted by a higher priority task with a call to context_switch(), executed by schedule(). Since we rely on non-preemptive scheduling, to implement non-preemptive scheduling, I modified the kernel to disable preemption between our real-time tasks. TASK_INTERRUPTIBLE (waiting for event) and TASK_UNINTERRUPTIBLE (waiting for event but does not wake up by a signal) states are for tasks in waiting status. If a task exists via do_exit(), it enters a TASK_ZOMBIE state. However, the process descriptor remains until the parent calls wait4().

Before I introduce structure task_struct, I need to discuss another structure, thread_info. Prior to Linux 2.6, task_struct was stored at the end of the kernel stack of each process. This allowed architectures with few registers, such as x86, to calculate the location of the process descriptor via the stack pointer without using an extra register to store the location. Since Linux 2.6, the process descriptor is dynamically created via the slab allocator; a new structure, thread_info, was created that lives at the bottom of the stack if it grows downwards or at the top of the stack if it grows upwards. The new structure makes it easier to calculate the offset of values needed for thread scheduling.

On x86 machines, the thread_info structure is defined in <asm/thread_info.h>.

```
struct thread_info {
        struct task_struct  *task;
        /* other definitions */
}
```

The task element of the structure is a pointer to the task's actual task_struct. If we want to find the current process descriptor, for instance in x86, we can mask out the least significant 13 bits of the stack pointer to obtain the thread_info structure. Assuming the stack size is 8KB, it is implemented by the following assembly code (a part of the current_thread_info() function):

```
movl $-8192, %eax
andl %esp, %eax
```

Thus, the current structure task_struct can be obtained by current_thread_info()->task; For the MIPS-based architectures, current (the pointer that points to the current structure task_struct) can be obtained with the value in register r2.

The following structure task_struct includes our added definitions for implementing an EDF/gEDF runqueue and real-time task parameters. The structure task_struct is defined in a header file that can be found at include/linux/sched.h.

```
struct task_struct {
   /* other definitions */
   struct thread_info *thread_info;
   int prio, static_prio;
   struct list_head run_list;
   prio_array_t *array;
   unsigned long policy;
   /* new pointer to EDF/gEDF runqueue */
   edf_queue_t *edf_queue;
   /* deadline of task, period of task, and execution time of task in ticks */
   unsigned long edf_deadline, edf_period, edf_length;
   struct list_head tasks;
   pid_t pid;
   unsigned long rt_priority;
   / other definitions */
}
```

5.2.2 Adding a New System Call

The file entry.S contains all system-calls and low-level fault handling routines. It also contains the timer-interrupt handler, as well as all interrupts and faults that can result in a task-switch. A new system call is added in arch/i386/kernel/entry.S as follows.

```
.long sys_sched_setscheduler_plus
```

5.2.3 Adding a New Structure and Several New Functions

Constants SCHED_EDF and SCHED_gEDF are defined to implement EDF and gEDF scheduling policies. A new structure edf_param is created for specifying real-time parameters, such as policy, period, length (i.e., execution time), and deadline. Structure runqueue is modified to create a new queue called edf_queue specifically for EDF/gEDF scheduling.

```
#define SCHED_NORMAL        0
#define SCHED_FIFO          1
#define SCHED_RR            2
```

Earliest deadline first real-time scheduling policies are defined as follows.

```
#define SCHED_EDF           3
#define SCHED_gEDF          4
```

The real-time task parameters are included in the new structure edf_param. For our purposes here, the period of a real-time task is assumed to be the same as its deadline.

```
struct edf_param {
   unsigned long policy;
   unsigned long period;
   unsigned long length;
}

struct sched_param {
   int sched_priority
}

struct runqueue {
   spinlock_t lock
   unsigned long nr_running, nr_switches, expired_timestamp, nr_uninterruptible
   task_t *curr, *idle
   struct mm_struct *prev_mm
   prio_array_t *active, *expired, arrays[2]
   /* edf_queue is defined for running EDF and gEDF tasks. */
   edf_queue_t edf_queue
   /* other definitions */
}
```

The new function dequeue_edf_task implements moving a task from EDF/gEDF task queue for scheduling.

```
static inline void dequeue_edf_task (struct task_struct *p, edf_queue_t *edf_queue)
begin
   edf_queue->nr_active--
   list_del(&p->run_list)
end
```

Function enqueue_edf_task() adds a new task to the gEDF/EDF runtime queue. Its main function is to insert a real-time task edf_queue based on deadlines.

```
static inline void enqueue_edf_task(struct task_struct *p, edf_queue_t *edf_queue)
begin
   struct list_head *rt_queue = &edf_queue->queue
   if (list_empty(rt_queue)) then
      list_add_tail(&p->run_list, &edf_queue->queue)
   else
      unsigned long deadline = p->edf_deadline
      struct task_struct *tsk = 0, *n = 0
      struct list_head *tmp, *nxt
      list_for_each_safe(tmp, nxt, rt_queue) do
         tsk = list_entry(tmp, struct task_struct, run_list)
         if (deadline < tsk->edf_deadline) then break
         n = tsk
      enddo
      if (n) then
         list_add(&p->run_list, &n->run_list)
      else
         list_add_tail(&p->run_list, &tsk->run_list)
      endif
   endif
   edf_queue->nr_active++
   p->edf_queue = edf_queue
end
```

In the above enqueue_edf_task() function, I use linked-list structure of the Linux kernel API (Application Programming Interface) for implementing queues. It is possible to improve the performance of the implementation using heaps (or a binary tree structure) so that tasks with earliest deadline and tasks with shortest jobs can appear at the top of the heap(s. Heap structures can be implemented with dynamic memory allocation with kmalloc() function of Linux kernels.

Compared with the original Linux or other RTOS scheduling functions, the modified schedule() is rewritten with the new gEDF scheduling feature. Usually, in schedule(), the task at the head of the queue has the minimum deadline and is selected for execution. In our gEDF approach, this function must perform additional computation to identify a task with the shortest execution time. While it is possible that the additional computation can increase the scheduling overhead, since we anticipate very small number of jobs in each group (or our gEDF groups), the actual search to find the shortest job should not consume excessive computing resources. Context switching in the Linux kernel is performed by the function schedule(). In addition, function schedule() needs to perform several other comparisons to determine if a new task needs to be scheduled. Thus, I feel that the added overhead of finding a shortest job by gEDF adds a very small amount of computing time.

Depending upon the length of jiffy (i.e. the duration of one tick of the system timer interrupt. Usually, it is 10ms or 1ms in Linux), the function schedule() may be called more than once during a task execution. Therefore, for the multiple callings, the schedule() function can be optimized to improve its performance. When a new real-time task arrives, the selection of the task with the minimum execution time within a group is invoked. Otherwise, the task selected previously is used. The key part of the function is shown as pseudo code.

```
asmlinkage void schedule(void)
  begin
     /* other definitions */
     array = rq->active
     edf_queue = &rq->edf_queue
     int pick_edf_one = 0
     if (unlikely(edf_queue->nr_active)) then
        q = &edf_queue->queue
        list_for_each_safe(tmp, nxt, q) do
           tsk = list_entry(tmp, task_t, run_list)
           if ((tsk->edf_deadline + Tr*tsk->edf_period) < (jiffies + tsk->edf_length)) then
              deactivate_task(tsk, rq)
           else
              pick_edf_one = 1
              if ( tsk->policy == SCHED_gEDF) then
                 if (first_element == 1 ) then
                    first_element = 0
                    d1 = tsk->edf_deadline
                    D1 = tsk->edf_deadline - jiffies
                    min_edf_length = tsk->edf_length
                    min_task = tsk
                 endif
                 if ((tsk->edf_deadline - d1) > (Gr*D1)) then break
                 if (tsk->edf_length < min_edf_length) then
                    min_edf_length = tsk->edf_length
                    min_task = tsk
                 endif
              endif
           endif
        enddo
        if (pick_edf_one == 1) then
           if ( prev->policy == SCHED_gEDF) then
              next = min_task
           else
              next = list_entry(edf_queue->queue.next, task_t, run_list)
           endif
           goto switch_tasks
        endif
     endif
     /* other part */
     switch_tasks:
     /* other part */
  end
```

As stated above, heap structures can be used instead of linked lists to improve implementation performance of our algorithms.

A new system call, sys_sched_setscheduler_plus, is available to run real-time applications. In this system call, structures task_struct, sched_param, and edf_param are used. In addition, a variable edf_queue of type edf_queue is defined. A variable array of real-time priority queue prio_array is defined for other Linux real-time tasks. A variable rq of runqueue is defined. In the following function, the parameters defined in the structure edf_param are copied to structure task_struct.

```
asmlinkage int sys_sched_setscheduler_plus(pid_t pid,
    struct edf_param __user *edf, struct sched_param __user *param)
begin
    struct task_struct *p
    struct sched_param lp
    struct edf_param ep
    edf_queue_t *edf_queue
    prio_array_t *array
    runqueue_t *rq
    /* other definitions */
    p = find_process_by_pid(pid)
    rq = task_rq_lock(p, &flags)
    /* other preparation */
    edf_queue = p->edf_queue
    array = p->array
    if (edf_queue || array) then
        deactivate_task(p, task_rq(p))
    endif
    if (array) p->array = NULL
    p->rt_priority = lp.sched_priority
    p->static_prio = p->prio = (int) (MAX_USER_RT_PRIO - 1 - p->rt_priority)
    p->policy = ep.policy
    p->time_slice = task_timeslice(p)
    p->edf_period = period_ticks
    p->edf_length = NS_TO_JIFFIES(ep.length)
    p->edf_deadline = jiffies + p->edf_period
    if (edf_queue || array) then
```

```
        __activate_task(p, task_rq(p))
    endif
    /* others */
end
```

5.3 The Complexity of gEDF in the Linux Kernel

Ingo Molnar [39] introduced an O(1) scheduler, as a patch for Linux 2.4. This is now accepted by most Linux 2.6 systems. It provides an O(1) scheduling algorithm and it can handle loads more smoothly. The approach is to use two split arrays, an active array, and an expired array. The active array contains all tasks that are affined to the CPU, and the expired array contains all tasks that have used up their time slices.

On the other hand, EDF and gEDF cannot guarantee O(1) complexity, since these algorithms require O(n) search to Enqueue newly arriving jobs. Although EDF and gEDF appear to have higher complexity, I feel that in most practical real-time systems, n is not large and thus the actual execution overheads associated with our new algorithms are not excessive. In addition, as I discussed in Section 5.3, if heap data structures, instead of list, are used for the EDF and gEDF queue operations, the algorithm complexity of EDF and gEDF will decrease to O(log(n)).

5.4 Real-Time Benchmark Testing

For the experiments I use Red Hat Inc. Fedora Core/Linux 2.6 with our EDF and gEDF enhancements running on a desktop CPU using AMD (Advanced Micro Devices) Athlon XP 1800+.

Soft real-time tolerance *Tr* is set to 10%. Group range of gEDF, that is, *Gr*, is set to 0.4. The applications for benchmarking are chosen from the embedded applications benchmark suite, called MiBench [40, 41], which is similar to the Embedded Microprocessor Benchmarking Consortium (EEMBC) [42].

Typical hard real-time applications, such as proportional, integral, and derivative (PID) control, are used in control systems. Soft real-time applications can be found in many areas. Multimedia and telecommunication are examples of soft real-time applications. Four well-known real-time applications are selected for as our first set of experiments. I designate these benchmarks as test suite 1. These programs are MPEG (Moving Picture Experts Group), GSM (Global Standards for Mobile) encode and GSM decode, and APCM (Adaptive Differential Pulse Code Modulation). MPEG Decode, Mad for short, is a high-quality MPEG audio decoder. It currently supports MPEG-1 and the MPEG-2 extensions at lower sampling frequencies, as well as the MPEG 2.5 format. All three audio layers (Layer I, Layer II, and Layer III or MP3) are fully implemented. Typically, thirty frames per second are needed for normal operation. Therefore, the deadline is set to 30 milliseconds in our experiments. GSM encode and decode are the Global Standards for Mobile (GSM) communications used in

Europe and other continents. It uses a combination of time- and frequency-division multiple access (TDMA/FDMA) to encode or decode data streams. According to the standard, GSM decode and encode have 20 or 40 milliseconds deadlines. Adaptive differential pulse code modulation (ADPCM) is a variation of the well-known standard pulse code modulation (PCM). A common implementation takes 16-bit linear PCM samples and converts them into 4-bit samples, yielding a compression rate of 4:1. ADPCM is the core part of G.726/VoIP technology. The deadline is set to 20 milliseconds.

The average execution times of the selected benchmarks are shown in Table 5.1. The executions times are derived by executing the programs on our target computing platform.

Table 5.1: Real-time benchmark suite 1

Benchmark Name	Deadline *Di* (ms)	Average Execution Time e_i (ms)
MPEG Decode	33	6
GSM Encode	20/40	12
GSM Decode	20/40	5
ADPCM Encode	20	8

Table 5.2: Real-time benchmark suite 1 – time constraints

Load	Benchmark	D_i (ms)	e_i (ms)
Case 1, ρ[6]= 1.00	MPEG Decode	33	6
	GSM Encode	40	12
	GSM Decode	40	5
	ADPCM Encode	20	8
Case 2, ρ = 1.13	MPEG Decode	33	6
	GSM Encode	40	12
	GSM Decode	20	5
	ADPCM Encode	20	8
Case 3, ρ = 1.31	MPEG Decode	33	6
	GSM Encode	20	12
	GSM Decode	40	5
	ADPCM Encode	20	8
Case 4, ρ = 1.43	MPEG Decode	33	6
	GSM Encode	20	12
	GSM Decode	20	5
	ADPCM Encode	20	8

[6] $\rho = \Sigma e_i \,/\, Di$, the utilization or load of the system (when $D_i = P_i$, where P_i is the period of a task).

Based on the deadlines of the applications in test suite 1, there are actually four cases representing four different system loads. I use (V_{ET}, V_D) to describe each application, the first value V_{ET} represents the average execution time of the real-time application and the second value V_D represents the deadline of the real-time application. The following lists the four cases. ρ is the load or utilization. As I claim, EDF performs poorly in overload ($\rho > 1$) situations. I validated this claim in experiments.

Case 1 ($\rho = 1.00$):
MPEG Decode: (6, 33)
GSM Encode: (12, 40)
GSM Decode: (5, 40)
ADPCM Encode: (8, 20)

Case 2 ($\rho = 1.13$):
MPEG Decode: (6, 33)
GSM Encode: (12, 40)
GSM Decode: (5, 20)
ADPCM Encode: (8, 20)

Case 3 ($\rho = 1.31$):
MPEG Decode: (6, 33)
GSM Encode: (12, 20)
GSM Decode: (5, 40)
ADPCM Encode: (8, 20)

Case 4 ($\rho = 1.43$):
MPEG Decode: (6, 33)
GSM Encode: (12, 20)
GSM Decode: (5, 20)
ADPCM Encode: (8, 20)

The results of our first experiment are shows in Table 5.3.

Table 5.3: Real-time benchmark suite 1 - performances of EDF/gEDF

Load	Success Ratio of EDF γ_{EDF}	Success Ratio of gEDF γ_{gEDF}	Success-Ratio Performance Factor $\eta_\gamma = \gamma_{gEDF} / \gamma_{EDF}$
Case 1, $\rho = 1.00$	0.981061	0.981061	100%
Case 2, $\rho = 1.13$	0.878981	0.885350	101%
Case 3, $\rho = 1.31$	0.770701	0.824841	107%
Case 4, $\rho = 1.43$	0.681319	0.793956	117%

When the load is 1, the success ratio of EDF and the success ratio of gEDF are about the same. When the system is slightly overloaded (1.13), gEDF outperforms EDF slightly. It should be noted that our test environment contains only 4 programs and thus the success rates are limited by this number. When the load increases to 1.31 and 1.43, gEDF shows even higher success rates than EDF and results in 107% and 117% improvements. I am confident that with more jobs in the suite, gEDF will consistently outperform EDF in overloaded conditions.

In the second experiment, I added more benchmark programs to our suite of benchmarks and I refer to this set as test suite 2. I added JPEG, CRC32 and Lame. JPEG is a standard, lossy compression algorithm. It is included in MiBench because it is a representative algorithm for image compression and decompression and is commonly used for viewing images embedded in documents. The deadline is set to 30 milliseconds. CRC32 benchmark performs a 32-bit cyclic redundancy check (CRC). CRC checks are often used to detect errors in data transmission. Assuming 15 frames per second are needed to transfer, the deadline is set to 67 milliseconds. Lame is a GNU general public licensed MP3 encoder that supports constant, average and variable bit-rate encoding. Assuming that 5 wave clips per second are needed to play, I set the deadline to 200 milliseconds.

Table 5.4: Real-time benchmark suite 2 – time constraints

Benchmark Name	D_i (ms)	e_i (ms)
MPEG Decode	33	6
GSM Encode	40	12
GSM Decode	40	5
ADPCM Encode	20	8
JPEG Decode	30	2
JPEG Encode	30	6
CRC32	67	11
MP3 Encode	200	38

Table 5.5: Real-time benchmark suite 2 - performances of EDF/gEDF

Load	Success Ratio of EDF γ_{EDF}	Success Ratio of gEDF γ_{gEDF}	Success-Ratio Performance Factor $\eta_{\gamma} = \gamma_{gEDF} / \gamma_{EDF}$
$\rho = 1.63$	0.558659	0.639020	114%

The test results from our second experiment are shown in Table 5.5. As can be seen, the success ratio of gEDF is 114% higher than that of EDF.

The following two figures, Figures 5.1 and 5.2, present the results graphically.

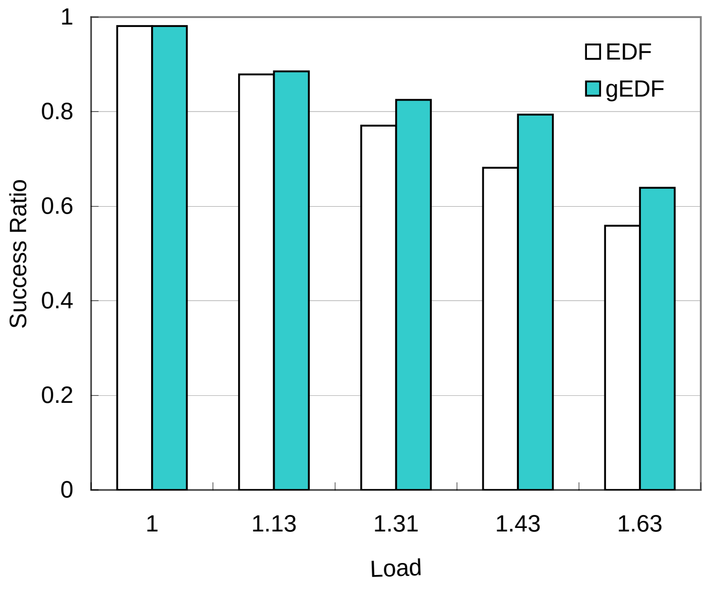

Figure 5.1: Performance comparison of EDF and gEDF in success ratio when ρ = 1.00, 1.13, 1.31, 1.41, and 1.63.

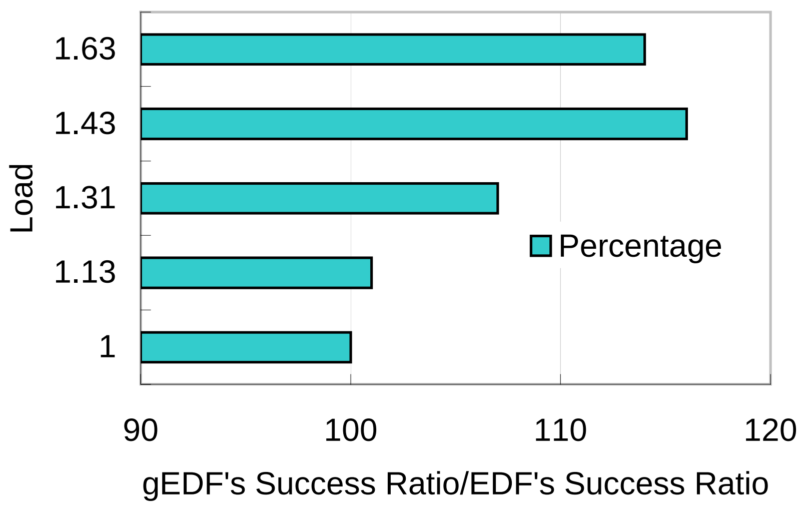

Figure 5.2: η_γ = γ_{gEDF} / γ_{EDF} = 100%, 101%, 107%, 117%, and 114% when ρ = 1.00, 1.13, 1.31, 1.41, and 1.63.

CHAPTER 6

CONCLUSIONS

In this dissertation, I presented a new real-time scheduling algorithm that combines shortest job first scheduling approach with the earliest deadline first scheduling algorithm. We grouped together tasks with deadlines that are very close to each other, and scheduled jobs within a group based on using SJF scheduling. Based on the experimental results included in this dissertation, I conclude that group EDF results in higher success rates (that is, the number of jobs that have completed successfully before their deadlines) as well as in faster response times.

It has been known that while EDF produces an optimum schedule (if one is available) for systems using preemptive scheduling, EDF is not as widely used for non-preemptive systems. I believe that for soft real-time systems that are implemented on multithreaded processors, non-preemptive scheduling is more efficient. Although EDF produces practically acceptable performance even for non-preemptive systems when the system is underloaded, EDF performs very poorly when the system is heavily loaded. Our gEDF algorithm performs as well as EDF in terms of success ratios when a system is underloaded. Even on systems that are underloaded, gEDF shows higher success rates than EDF when dealing with soft real-time tasks (using higher deadline tolerances). In addition, gEDF consistently outperforms EDF in overloaded systems.

In this dissertation, I also compared gEDF with schemes that using adaptive scheduling algorithms often used in conjunction with EDF when the system is overloaded. Among these I considered the best-effort and the guarantee algorithms. In general, gEDF, which can be used under all system loads, performs as well as or better than EDF and adaptive algorithms such as best-effort and guarantee schemes. It should be remembered that these adaptive algorithms require the ability to accurately measure system loads so that the overloaded conditions can be detected. In most practical workloads this is very difficult, particularly if the workload consists of periodic, aperiodic and sporadic jobs, or if the system consists of both real-time and non-real-time jobs. Moreover, estimating system load based on worst-case execution times leads to under-utilizations of the system resources. These problems are not encountered by gEDF, since there is no need to estimate system load or to switch between EDF and an adaptive method in overloaded conditions.

Last, I modified the Linux kernel scheduler to implement gEDF scheduling policy for real-time processes. I tested several real benchmarks and our test results show that gEDF can be used effectively in real world systems.

The group range (*Gr*) is an important factor in the gEDF algorithm. For instance, when *Gr* is very large to include all the tasks, gEDF degenerates to be SJF; and if *Gr* is set to 0, gEDF degenerates to be EDF. Figure 4.14 and Figure

4.15 show when deadline tolerance (*Tr*) is 0.1 and 0.5, and *Gr* is between 0.2 to 1.0, gEDF obtains the near optimal performance. However, the optimal value for *Gr* depends on several factors that depend on the application. These factors include the variations among task execution times, inter-arrival times and task deadlines. It may be possible to derive analytical models that show the dependence of *Gr* on these parameters (specified as mean values of underlying probability distributions).

I have shown that the gEDF algorithm can be applied in uniprocessor systems for soft real-time systems. In our future work, I will explore the applicability of gEDF algorithm for multi-processors systems as well as decoupled architectures such as the scheduled dataflow (SDF) architecture, which contains a SP that accesses memory and an EP that executes computations [4].

Because EDF is not optimal for multiprocessor real-time systems [43], I will explore if gEDF can be used to obtain acceptable (and near optimal) results for multiprocessor systems with soft real-time tasks. While the EDF scheme can be used to schedule dynamic groups on multiprocessors, an optimal or near optimal algorithm may be adopted to schedule jobs distributed on different processors within each dynamic group. I hope to show that gEDF results in higher, success ratios and response times in underloaded and overloaded situations.

In fact, exploring different scheduling scheme applied within each gEDF group is another promising research of applying the gEDF scenario. Scheme other than SJF may be used appropriately for the real-time systems depending an application domain. For example it may be necessary to reduce overall power consumption and one may need to explore a scheduling scheme that minimizes the power consumed by tasks in a group, accounting for any power consumed by tasks waiting in a queue.

APPENDIX A

BUILDING THE LINUX KERNEL

A1 The Linux Kernel Source and Configuration

The Linux kernels are written and maintained by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. The latest stable version of the Linux kernel can be downloaded from the Linux kernel archives website [44] and all the previous versions can be found on the mirror web sites listed under the directory [45]. I downloaded Linux kernel 2.6.0 from a mirror kernel web site [46]. The Linux distributions can be downloaded free from the web site [47]. I use the distribution, i.e., Fedora core, from Red Hat Inc. All the Fedora versions can also be found on the mirror web sites [48]. I downloaded Fedora core 1/kernel v2.4 and then upgraded it to kernel v2.6. The kernel tarball is distributed in both GNU zip (gzip) and bzip2 format. Bzip2 is the preferred format.

Assuming the old kernel with the Linux distribution is v2.4.22, and the downloaded kernel is located at /usr/src directory,

```
$ su root
# cd /usr/src
# tar xvjf linux-2.6.0.tar.bz2
# cd linux-2.6.0
# make mrproper
```

It's a smart idea to use an old configuration of the distribution as a head start because it could take dozens of minutes to go through all of the configurations diligently.

```
# cp /boot/config-2.4.22-1.2115.nptl  .config
# make menuconfig
```

Then, make sure to choose the correct processor type, and to check the Linux kernel hacking option. Finally, save and exit.

A2 Compiling the Linux Kernel

After successful configuration, open Makefile. The first four statements should be:

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 0
EXTRAVERSION = -x
```

In the fourth statement, *x* can be set to an appropriate name. For instance, it can be set as EXTRAVERSION = -edf or EXTRAVERSION = -gedf when a new version with EDF scheduling policy or gEDF scheduling policy of the Linux kernel is to be generated, while the older versions that generated before are also needed to be kept. If the new version of the Linux kernel has some critical problem or crashes, the older version can be rolled back to run instead of destroying the whole system.

```
# make bzImage
# make modules
# make modules_install
# /sbin/mkinitrd /boot/initrd-2.6.0-x.img  2.6.0-x
# make install
```

If the modification of the Linux kernel that does not affect the modules and the related structures, and if the same kernel version is to be generated, it may not be necessary to execute all the steps above. The most time-consuming “make modules” and “make modules_install” steps may not execute. That is a very efficient way to change and debug the kernel without waiting for a long compiling time.

After compiling the Linux kernel, /boot/grub/grub.conf, the GRUB (Grand Unified Bootloader) configuration file, will be the following,

```
# Note that you do not have to rerun grub after making changes to this file
# NOTICE:  You have a /boot partition.  This means that
#          all kernel and initrd paths are relative to /boot/, eg.
#          root (hd0,0)
#          kernel /vmlinuz-version ro root=/dev/hda2
#          initrd /initrd-version.img
#boot=/dev/hda
default=5
timeout=10
splashimage=(hd0,0)/grub/splash.xpm.gz
title Fedora Core (2.6.0-gedf)
        root (hd0,0)
        kernel /vmlinuz-2.6.0-gedf ro root=LABEL=/ hdc=ide-scsi rhgb
        initrd /initrd-2.6.0-gedf.img
title Fedora Core (2.6.0-edf)
        root (hd0,0)
        kernel /vmlinuz-2.6.0-edf ro root=LABEL=/ hdc=ide-scsi rhgb
        initrd /initrd-2.6.0-edf.img
title Fedora Core (2.6.0)
        root (hd0,0)
        kernel /vmlinuz-2.6.0 ro root=LABEL=/ hdc=ide-scsi rhgb
        initrd /initrd-2.6.0.img
title Fedora Core (2.4.22-1.2115.nptl)
        root (hd0,0)
        kernel /vmlinuz-2.4.22-1.2115.nptl ro root=LABEL=/ hdc=ide-scsi rhgb
        initrd /initrd-2.4.22-1.2115.nptl.img
```

In our configuration of the Linux kernel, to make it suitable for our goal, usually, the minimum configuration is set. However, it will be convenient to use Universal Serial Bus (USB) or CD-ROM (Compact Disc-Read-Only Memory). The following are the parameters in our system.

In the /etc/fstab file:

```
/dev/cdrom   /mnt/cdrom    udf,iso9600 noauto,owner, kudzu, ro 0 0
/dev/sda1   /mnt/usbstick  vfat  user,noauto,umask=0 0 0
```

For instance, for using USB device, use the command: mount /dev/sda1 /mnt/usbstick.

To debug the kernel bugs, it is straightforward and efficient to use printk(), the kernel's formatted print function, which is similar as printf() in C library. For displaying the messages that printed by printk() in the Linux kernel, command dmesg can be called. However, because the log buffer's default size is 16KB, we need to set the parameter CONFIG_LOG_BUF_SHIFT to 17 (i.e., 128KB) from the original value 14 (i.e., 16KB).

APPENDIX B

RUNNING REAL-TIME APPLICATIONS WITH gEDF POLICY

B1 Typical Real-Time Applications

The most common fixed-priority scheduling policies are:

FIFO **-** a task executes until it is finished, or a higher priority task wants to run, but is never preempted by a task with the same priority.

Round robin **-** as above, a task is preempted if a higher priority task wants to run. If there are several tasks of the same priority, each gets to run a predefined amount of time and is then put last in the queue in order to allow the next task to execute.

B2 Compiling Environment

I use GNU gcc. Its version is 3.3.2 20031022 (Red Hat Linux 3.3.2-1)

We also need to evacuate c header files in the directories /usr/include/asm and usr/include/linux. Then, set up the Linux kernel header files to the above two directories.

```
ln -s /usr/src/linux-2.6.0/include/asm-i386  /usr/include/asm
ln -s /usr/src/linux-2.6.0/include/linux  /usr/include/linux
```

B3 Real-Time Program Interface

The following system calls are common real-time program interfaces: the definitions for POSIX 1003.1b-1993 (aka POSIX.4) scheduling interface (1996-2003, part of the GNU C Library).

Set scheduling parameters for a process,

```
extern int sched_setparam (__pid_t __pid, __const struct sched_param *__param)
        __THROW;
```

Retrieve scheduling parameters for a particular process,

```
extern int sched_getparam (__pid_t __pid, struct sched_param *__param) __THROW;
```

Set scheduling algorithm and/or parameters for a process,

```
extern int sched_setscheduler (__pid_t __pid, int __policy,
                               __const struct sched_param *__param) __THROW;
```

Retrieve scheduling algorithm for a particular purpose,

```
extern int sched_getscheduler (__pid_t __pid) __THROW;
```

Yield the processor,

```
extern int sched_yield (void) __THROW;
```

Get maximum priority value for a scheduler.

```
extern int sched_get_priority_max (int __algorithm) __THROW;
```

Get minimum priority value for a scheduler.

```
extern int sched_get_priority_min (int __algorithm) __THROW;
```

Get the SCHED_RR interval for the named process.

```
extern int sched_rr_get_interval (__pid_t __pid, struct timespec *__t) __THROW;
```

B4 POSIX Extension of EDF and gEDF Scheduling Policy

Because the function sched_setscheduler_plus is not in the standard library of POSIX yet, it is defined in the header file edf.h, which must be included in the real-time applications with the gEDF policy.

```
#ifndef EDF_H
#define EDF_H
#include <sys/syscall.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
#include <sched.h>
```

```
struct edf_param {
    unsigned long policy;
    unsigned long period;
    unsigned long length;
}
inline int sched_setscheduler_plus (pid_t pid, struct edf_param *gedf,
                    struct sched_param *param)
{
   long ret;
   __asm__ volatile ("int $0x80" : "=a" (ret) : "a" (__NR_sched_setscheduler_plus),
                                        "b" (pid), "c" (gedf), "d" (param) );
   if ((unsigned long) ret >= (unsigned long) -125) {
      errno = -ret;
      ret = -1;
   }
   return (int) ret;
}
#endif
#endif
```

B5 Real-time Applications with gEDF Scheduling Policy

The following is a sample code in C style. Three processes of the real-time applications are created and scheduled by the gEDF scheduling policy.

```
#include <sched.h>
#include "edf.h"
#define SCHED_EDF 3
#define SCHED_gEDF 4
struct sched_param schp;
struct edf_param edfp;
int main (int argc, char *argv[])
{
    /* set parameters of real-time applications */
    memset(&schp, 0, sizeof(schp));
    schp.sched_priority = sched_get_priority_max(<POLICY>);
    memset(&edfp, 0, sizeof(edfp));
    edfp.policy = <POLICY>;
    /* skip some details of the definition here*/
    /* create some processes continuously */
    if (fork() == 0) {
        edfp.period = <PERIOD>;
        edfp.length = <ET>;
        sched_setscheduler_plus(0, &edfp, &schp);
        if (fork() == 0) {
            edfp.period = <PERIOD>;
            edfp.length = <ET>;
            sched_setscheduler_plus(0, &edfp, &schp);
            if (fork() == 0) {
                edfp.period = <PERIOD>;
                edfp.length = <ET>;
                sched_setscheduler_plus(0, &edfp, &schp);
                if (fork() == 0) {} else execl(<EXEP>);
            }
            else  execl(<EXEP>);
        }
        else execl(<EXEP>);
    }
    /* other work */
}
```

Note:	<POLICY>: SCHED_gEDF/SCHED_EDF
	<PERIOD>: Real-Time Period or Deadline
	<ET>: Average Execution Time
	<EXEP>: Executable Program

BIBLIOGRAPHY

[1] F. Balarin, L. Lavagno, P. Murthy, and A. S. Vincentelli, “Scheduling for Embedded Real-Time Systems”, IEEE Design & Test of Computer, January-March, 1998.

[2] J. H. Anderson, V. Bud, U. C. Devi, “An EDF-based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems”, 17th Euromicro Conference on Real-Time Systems, 2005.

[3] R. Jain, C. J. Hughes, and S. V. Adve, “Soft Real-Time Scheduling on Simultaneous Multithreaded Processors”, In Proceedings of the 23rd IEEE International Real-Time Systems Symposium, December 2002.

[4] K. M. Kavi, R. Giorgi, and J. Arul, “Scheduled Dataflow: Execution Paradigm, Architecture, and Performance Evaluation”, IEEE Transactions on Computers, Vol. 50, No. 8, August 2001.

[5] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”, Journal of the ACM, Vol. 20. No. 1, pp. 46-61.

[6] K. Jeffay and C. U. Martel, “On Non-Preemptive Scheduling of Periodic and Sporadic Tasks”, Proceedings of the 12th IEEE Real-Time Systems Symposium, San Antonio, Texas, December 1991, IEEE Computer Society Press, pp. 129-139.

[7] M. R. Garey, D. S. Johnson, “Computer and Intractability, a Guide to the Theory of NP-Completeness”, W. H. Freeman Company, San Francisco, 1979.

[8] L. Georges, P. Muehlethaler, N. Rivierre, “A Few Results on Non-Preemptive Real-time Scheduling”, INRIA Research Report nRR3926, 2000.

[9] C. D. Locke, “Best-Effort Decision Making for Real-Time Scheduling”, CMU-CS-86-134 (PhD Thesis), Computer Science Department, Carnegie-Mellon University, 1986.

[10] J. K. Dey, J. Kurose, and D. Towsley, "Online Processor Scheduling for a Class of IRIS (Increasing Reward with Increasing Service) Real-Time Tasks", Tech. Rep. 93-09, Department of Computer Science, University of Massachusetts, Amherst, Jan 1993.

[11] S. Zilberstein, "Using Anytime Algorithms in Intelligent Systems", AI Magazine, fall 1996, pp.71-83.

[12] T. Abdelzaher, V. Sharma, and C. Lu, "A Utilization Bound for Aperiodic Tasks and Priority Driven Scheduling", IEEE Trans. On Computers, March 2004.

[13] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The Influence of Processor Architecture on the Design and the Results of WCET Tools", Proceedings of IEEE July 2003, Special Issue on Real-time Systems.

[14] G. Bernat, A. Collin, and S. M. Petters, "WCET Analysis of Probabilistic Hard Real-Time Systems", IEEE Real-Time Systems Symposium 2002, 279-288.

[15] J. Nieh and M. S. Lam, "A SMART Scheduler for Multimedia Applications", ACM Transactions on Computer Systems, Vol. 21, No. 2, May 2003.

[16] G. Buttazzo, M. Spuri, and F. Sensini, Scuola Normale Superiore, Pisa, Italy, "Value vs. Deadline Scheduling in Overload Conditions", 16th IEEE Real-Time Systems Symposium (RTSS'95) December 05-07, 1995.

[17] S. K. Baruah and J. R. Haritsa, "Scheduling for Overload in Real-Time Systems", IEEE Transactions on Computers, Vol. 46, No. 9, September 1997.

[18] A. L. N. Reddy and J. Wyllie, "Disk Scheduling in Multimedia I/O system", In Proceedings of ACM multimedia'93, Anaheim, CA, 225-234, August 1993.

[19] B. D. Doytchinov, J. P. Lehoczky, and S. E. Shreve, "Real-Time Queues in Heavy Traffic with Earliest-Deadline-First Queue Discipline", Annals of Applied Probability, No. 11, 2001.

[20] J. P. Hansen, H. Zhu, J. P. Lehoczky, and R. Rajkumar, "Quantized EDF Scheduling in a Stochastic Environment", Proceedings of the International Parallel and Distributed Processing Symposium, 2002.

[21] W. T. Chan, T. W Lam, K. S. Liu, P. W. H. Wong, “Resource augmentation analysis of SRPT and SJF for minimizing total stretch in multiprocessor scheduling”, University of Liverpool, UK.

[22] P. Brucker, “Scheduling Algorithms”, Third Edition, Springer, 2001.

[23] J. P. Lehoczky, L. Sha, and Y. Ding, “The Rate Monotonic Scheduling Algorithms – Exact Characterization and Average-Case Behavior”, Proceedings IEEE Real-Time Systems Symposium, Santa Monica, California, 1989.

[24] N. C. Audsley, A. Burns, M. F. Richardson, A. J. Wellings “Hard Real-Time Scheduling: The Deadline-Monotonic Approach (1991)”, Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software

[25] L. Sha, R. Rajkumar, and S. S. Sathaye, “Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems”, Proceedings of the IEEE, Jan. 1994.

[26] J. R. Jackson, “Scheduling a Production Line to Minimize Maximum Tardiness”, Research Report 43, Management Science Research Project, University of California, Los Angels, 1955.

[27] J. K. Lenstra and A. H. G. Rinnooy Kan, “Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey”, Discrete Math, No. 5, 1977, pp. 287-326.

[28] G. Buttazzo, M. Spuri, F. Sensini, “Value vs. Deadline Scheduling in Overload Conditions”, Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS 1995), Pisa, Italy, pp. 90-99, December 5-7, 1995.

[29] J. A. Stankovic, M. Spuri, M. D. Natale, G. C. Buttazzo, “Implications of Classical Scheduling Results for Real-Time Systems”, Computer, Volume 28, Number 6, June 1995.

[30] S. Baskiyar, N. Meghanathan, “A Survey of Contemporary Real-Time Operating Systmes”, Informatica 29 (2005) 233-240.

[31] IEEE Information technology - Portable Operating System Interface (POSIX) - Part 1: Base Definitions; Part 2: System Interfaces; Part 3: Shell and Utilities; Part 4: Rationale.
http://standards.ieee.org/catalog/olis/posix.html

[32] IEEE Information Technology – Portable Operating System Interface (POSIX): IEEE/ANSI Std 1003.1, 1996 Edition.

[33] http://www.fsmlabs.com

[34] K. Lin, Y. C. Wang, "The Design and Implementation of Real-Time Schedulers in RED-Linux", Proceedings of the IEEE, Vol. 91, No. 7, July 2003.

[35] W. Dinkel, D. Niehaus, M. Frisbie, J. Woltersdorf, "KURT-Linux User Manual", Information and Telecommunication Technology Center, University of Kansas, 2002.

[36] MATLAB Software and Documents, http://www.mathworks.com.

[37] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner, "Linux Kernel Internals", Second Edition, Addison Wesley Longman 1998.

[38] R. Love, "Linux Kernel Development", 2nd Edition, Novell Press, 2005.

[39] I. Molnar, "Goals, Design and Implementation of the New Ultra-Scalable O(1) Scheduler", 2002.

[40] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, "MiBench: A free, Commercially Representative Embedded Benchmark Suite", IEEE 4th Annual Workshop on Workload Characterization, 2001

[41] http://www.eecs.umich.edu/mibench/.

[42] EDN Embedded Microprocessor Benchmark Consortium, http://www.eembc.org

[43] J. H. Anderson, V. Bud, U. C. Devi, "An EDF-based Scheduling Algorithm for Multiprocessor Soft Real-time Systems", Proceedings of 17th Euromicro Conference on Real-Time Systems, 2005 (ECRTS 2005).

[44] http://www.kernel.org

[45] The Previous Versions of the Linux Kernel: http://www.kernel.org/mirrors/

[46] One Mirror Web Site for the Linux Kernel: http://mirror.doit.wisc.edu/mirrors/linux/kernel/v2.6/

[47] The Linux Distributions: http://www.linux.org

[48] Red Hat Inc. Linux Distributions, Fedora Cores: http://fedora.redhat.com/download/mirrors.html