DYNAMIC SCHEDULING POLICIES
IN
A REAL-TIME MAIN MEMORY DATABASE SYSTEM

by

MARYAM MOGHADDAS
B.Sc., The University of Shahid Beheshti, Tehran, Iran, 1996

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE
in
THE FACULTY OF GRADUATE STUDIES
(Department of Electrical and Computer Engineering)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
January 2001
© Maryam Moghaddas, 2001
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Electrical and Computer Engineering

The University of British Columbia
Vancouver, Canada

Date Jan 31, 2001
Abstract

In a real-time system, the overhead associated with scheduling algorithms can be a significant factor in the resulting schedule, in both normal and overload situations. This overhead may be due to the frequency of scheduling-algorithm invocations and to the execution time of the algorithm at each invocation. We investigate the trade-off in the dynamic scheduling of real-time tasks between the frequency at which the scheduling algorithm is invoked and the quality of the resulting schedules in terms of deadline compliance. We also look into the size of the task-set to which the scheduling policy is applied at every invocation. Furthermore, we study two overload management policies to improve the quality of resulting schedules in heavy load systems. We perform experimental evaluation of the algorithms in a real-time main memory database system.

We propose a batching algorithm which, forms a batch of arrived tasks, then schedules and executes all tasks in a batch before considering other tasks that arrive in the mean time. The evaluation shows that our batching algorithms, also referred to as *Batching Earliest Deadline First (BEDF)* or *Batching Least Slack First (BLSF)* with respect to their used scheduling policy, EDF or LSF, outperform their non-batching counterparts under tighter time constraints.

In this thesis, we also study the behavioral change of batching algorithms in overload conditions in comparison to normal load. We propose the *Task Fusion (TF)* overload management policy in order to avoid context switch overhead in heavy load situations. The other policy applied to the algorithms is a *Not Tardy (NT)* overload management policy, through which we improve the performance of EDF-based algorithms. Experiments show effective performance results due to employment of these overload management policies.
Table of Contents

ABSTRACT ............................................. ii

TABLE OF CONTENTS ................................ iii

LIST OF FIGURES .................................... vi

LIST OF TABLES ..................................... vii

LIST OF ACRONYMS ................................ viii

ACKNOWLEDGEMENTS ............................... ix

DEDICATION .......................................... x

CHAPTER 1 INTRODUCTION ........................... 1

1-1 Real-Time Scheduling ................................ 1
1-2 Motivation ........................................ 3
1-3 Contributions ..................................... 4
1-4 Thesis Outline .................................... 6

CHAPTER 2 BACKGROUND ............................ 7

2-1 Real-Time Systems ............................... 7
  2-1-1 Real-Time Main Memory Database ............... 10
  2-1-2 Real-Time Operating System .................... 11
    2-1-2-1 Small, Fast, Proprietary Kernels ........... 12
2-2 Real-Time Scheduling ............................ 13
  2-2-1 Earliest Deadline First ......................... 15
  2-2-2 Least Slack First ............................. 15
2-3 Related Work .................................... 16
2-4 Summary ......................................... 17
<table>
<thead>
<tr>
<th>CHAPTER 3 EXPERIMENTAL PLATFORM</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1 Experimental Model</td>
<td>19</td>
</tr>
<tr>
<td>3-2 Query Environment</td>
<td>21</td>
</tr>
<tr>
<td>3-2-1 Query Format</td>
<td>22</td>
</tr>
<tr>
<td>3-2-2 Query Generation</td>
<td>24</td>
</tr>
<tr>
<td>3-2-3 Query Arrival</td>
<td>25</td>
</tr>
<tr>
<td>3-2-3-1 Modification on the Receiver</td>
<td>26</td>
</tr>
<tr>
<td>3-3 RTLinux</td>
<td>27</td>
</tr>
<tr>
<td>3-4 Query Processor</td>
<td>29</td>
</tr>
<tr>
<td>3-5 Summary</td>
<td>34</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 4 BATCHING ALGORITHMS</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-1 Batching Algorithms</td>
<td>36</td>
</tr>
<tr>
<td>4-1-1 Scheduling Steps</td>
<td>37</td>
</tr>
<tr>
<td>4-2 Experimental Evaluation</td>
<td>39</td>
</tr>
<tr>
<td>4-2-1 Comparison of EDF and BEDF Algorithms</td>
<td>40</td>
</tr>
<tr>
<td>4-2-2 Comparison of LSF and BLSF Algorithms</td>
<td>41</td>
</tr>
<tr>
<td>4-2-3 Comparison of BEDF and BLSF Algorithms</td>
<td>42</td>
</tr>
<tr>
<td>4-3 Batch Size Effect in Batching Algorithms</td>
<td>44</td>
</tr>
<tr>
<td>4-3-1 Experimental Evaluation</td>
<td>45</td>
</tr>
<tr>
<td>4-3-1-1 Effect of Batch size on BEDF</td>
<td>45</td>
</tr>
<tr>
<td>4-3-1-2 Effect of Batch size on BLSF</td>
<td>46</td>
</tr>
<tr>
<td>4-3-1-3 A Lower Boundary on Batch Size</td>
<td>47</td>
</tr>
<tr>
<td>4-4 Summary</td>
<td>49</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 5 BATCHING ALGORITHMS IN OVERLOAD CONDITIONS</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-1 Introduction</td>
<td>50</td>
</tr>
<tr>
<td>5-1-1 Experiment Outline</td>
<td>52</td>
</tr>
<tr>
<td>5-2 Batching Algorithms under Overload Conditions</td>
<td>53</td>
</tr>
<tr>
<td>5-3 Context Switching Overhead in Real-time Systems</td>
<td>55</td>
</tr>
<tr>
<td>5-3-1 Task Fusion Policy</td>
<td>57</td>
</tr>
<tr>
<td>5-3-1-1 Comparison of BEDF and BEDF/TF Algorithms</td>
<td>59</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2-1: Illustration of a Real-Time System .................................................. 9
Figure 3-1: Experimental Model .......................................................................... 20
Figure 3-2: Query Format ....................................................................................... 22
Figure 3-3: A Sample Query .................................................................................. 24
Figure 3-4: A view of Global Relation and Grid Structure of a sample database ...... 31
Figure 3-5: Pseudo Code for IDD Mechanism ..................................................... 33
Figure 4-1: Comparison of EDF and BEDF Algorithms ........................................ 41
Figure 4-2: Comparison of LSF and BLSF Algorithms .......................................... 42
Figure 4-3: Comparison of BEDF and BLSF Algorithms ...................................... 44
Figure 4-4: Effect of Batch Size on BEDF ............................................................ 46
Figure 4-5: Effect of Batch Size on BLSF ............................................................. 47
Figure 5-1: Behavior of Batching Algorithms in Overloaded Conditions ............. 55
Figure 5-2: Comparison of BEDF and BEDF/TF Algorithms ............................... 60
Figure 5-3: Comparison of BLSF and BLSF/TF Algorithms ................................. 61
Figure 5-4: Comparison of BEDF/TF and BLSF/TF Algorithms .......................... 62
Figure 5-5: Comparison of BLSF/TF and BEDF/TF/NT Algorithms ..................... 64
Figure 5-6: Summary ......................................................................................... 66
List of Tables

Table 4-1: Task Model Notation ................................................................. 37
Table 4-2: Percentage of Missed Deadlines for BEDF algorithms, NTASKS 1000 .... 48
Table 4-3: Percentage of Missed Deadlines for BLSF algorithms, NTASKS 1000 .... 49
Table 5-1: List of Scheduling Policies ........................................................... 53
List of Acronyms

BEDF  Batching Earliest Deadline First
BLSF  Batching Least Slack First
EDF   Earliest Deadline First
IDD   Index and Data Distribution
IRQP  In-memory Real-time Query Processor
LSF   Least Slack First
NT    Not Tardy
OS    Operating System
QE    Query Environment
QP    Query Processor
RTLinux Real-Time Linux
RTDB  Real-Time DataBase
RTMMDB Real-Time Main Memory DataBase
TF    Task Fusion
Acknowledgements

In one hand, I carry a page of transcripts and in another a copy of my thesis published in UBC. But my heart and my brain carry other assets. While my brain holds the useful lessons I learned for my future, my heart carries love and wonderful memories. It makes me sit back and think about

My supervisor, Dr. Hamidzadeh who helped me in many academic and non-academic ways in the last years,
The members of my committee, Dr. Ito and Dr. Alnuweiri, who dedicated time to read my thesis,
Dr. Marti and the dear secretaries who guide me in administration works,
Doug, Mark, Paria, Buke, Ramin, Geoff, Shree, colleagues in High Performance Computing lab and GSS who helped me in achieving my goals,
My sweet sister, Shirin, who gave me all her love and support at all time,
My parents, Hossein and Gitijoon, and my brother Milad, who went through lots of sacrifice at a long distance,
and all the beautiful moments that can’t be described in a page...

Thank you all!

Maryam Moghaddas

University of British Columbia

January 2001
to Shirin
Chapter 1

Introduction

This chapter introduces the thesis research topic. A quick review of two categories of real-time scheduling, static scheduling and dynamic scheduling, is provided. The motivation and contributions of the research are also discussed. Finally, an outline of the thesis is provided.

1-1 Real-Time Scheduling

Any operating system consists of a unit referred as the scheduler. The scheduler is the core unit, which determines the task execution sequence in a system. In order to meet the deadlines in a time-critical system such as a real-time system, a proper real-time scheduling algorithm needs to be applied. Scheduling of real-time tasks on a uni-processor system addresses the order in which tasks are to be executed so that the timing constraints of each task (e.g., deadlines) are met. Based on the time at which scheduling is performed, real-time scheduling algorithms have been divided into two categories, namely static and dynamic.

Static algorithms are invoked only once to determine a fixed schedule off-line prior to the start of problem solving. Such approaches require complete a priori knowledge of the tasks. Static scheduling may lead to poor utilization of CPU and/or deadline loss if a priori knowledge of tasks is not accurate.
One of the most difficult problems in the design of real-time systems is the scheduling of sporadic tasks [1]. Sporadic tasks are tasks that have random arrival times. Because of the unpredictable nature of their arrivals, it is extremely difficult to design a real-time system, which can guarantee that the deadlines of all sporadic tasks can be met. In practice, dynamic real-time systems aim to meet deadlines of the most important tasks first. The importance of the task is defined by its priority. Thus, we need a dynamic algorithm with which to obtain the best schedule.

Dynamic algorithms [1][2][3] produce schedules on line in the hope of using more comprehensive and up-to-date information about the tasks and the environment. In applications where tasks arrive over time, for example, these algorithms are invoked several times to account for new tasks. A main characteristic of these algorithms is that they consider only one group of tasks (rather than the entire task set from system start-up to system shutdown) at any invocation, to determine the schedule.

When one or more sporadic tasks arrive at time $t$, the execution times and deadlines of the tasks are made known to the system. The scheduler is called upon to pick a task from between the new arrivals and the unfinished tasks as of time $t$. The task is picked based on its level of importance or priority from the set of tasks. The priority of a task is set differently in each algorithm. In Earliest Deadline First (EDF) the task with a nearer deadline gets a higher priority. In another real-time scheduling algorithm, Least Slack First (LSF), slack determines the task priority. Slack time is an estimate of how long we can delay the execution of a task and still meet its deadline. We use EDF and LSF as the two base algorithms in this thesis and will explain them in more detail in the Chapter 2.
Some of the applications that can benefit from the proposed techniques are real-time simulation, virtual reality, multimedia, and computer games. For example, a computer game may be played on-line, in a virtual world, against a computer program. This may require a computer system to present multimedia data and information in real-time, and to react to the player’s action dynamically. Scheduling the moves that the computer will execute in reaction to the player’s previous moves and scheduling the retrieval and presentation of the multimedia data in real-time can benefit from considering the issues that are addressed and evaluated in this thesis.

1-2 Motivation

Real-time scheduling has been widely researched as one of the most important issues in real-time systems. Thus we would like to also study different scheduling algorithms, for the purpose of reducing the missed deadline ratio. This metric represents the percentage of tasks with missed deadlines within all the tasks existing in the real-time system.

In a uni-processor architecture, a dynamic scheduling algorithm competes for resources (e.g., CPU and memory), with the tasks it opts to schedule. Thus, the overhead associated with the scheduling algorithm can be an important performance factor that should not be ignored [4][5][6][7][8][9]. This overhead is partly due to the frequency of scheduling-algorithm invocations, and partly due to the time it takes to execute the scheduling algorithm at each invocation. An algorithm that is invoked frequently to decide the order of executing tasks can take into account more up-to-date information in the system (e.g., new task arrivals), at the cost of higher scheduling overheads. Using complex scheduling algorithms, on the other hand, can lead to nearer-to-optimal
schedules, also at the cost of higher scheduling overheads. These issues motivated us to propose algorithms with less complexity, namely *Batching EDF* and *Batching LSF*.

In order to achieve the lowest *missed deadline ratio*, the least amount of overhead in a real-time system is also required. Interrupt handling latency and context switch overheads are two of the major factors, causing overhead in a real-time system. In order to reduce these sources of overhead, we have chosen RTLinux [19][20][21][22] as our real-time operating system with a small and fast kernel. This type of operating system will be explained in more detail in Chapter 2.

Although we have chosen an operating system with fast context switches for the purpose of our thesis, we still suffer from context switch overhead in high load situations. We propose the *Task Fusion (TF)* policy that reduces the overhead caused by context switching, decreasing the *missed deadline ratio*.

In this thesis, we also address another issue to decrease the *missed deadline ratio* of our scheduling algorithms. The EDF-based algorithms used throughout this thesis, have a major weakness in overload conditions. These algorithms may assign the highest priority to a task that has already missed or is about to miss its deadline. By assigning a high priority and system resources to a task that will miss its deadline anyway, we deny resources to tasks that still have chance to meet their deadlines, causing them to be late as well. In order to overcome this weakness we apply the *Not Tardy (NT)* policy suggested in former studies [10], to our proposed EDF-based algorithms.

**1-3 Contributions**

In this thesis, we study the tradeoffs between the frequency of scheduling-algorithm invocations, the size of the task-set to which the scheduling policy is applied at every
invocation, and the quality of resulting schedules. We propose *Batching Earliest-Deadline-First (BEDF)* and *Batching Least Slack First (BLSF)* scheduling, to dynamically prioritize the execution of a set of aperiodic, non-preemptable, real-time tasks on a uni-processor architecture. A contribution of the thesis is to address the problem of deciding when to invoke the task scheduling strategy, and how many tasks to execute in a row, before considering the newly arrived tasks.

Another contribution is the evaluation of different scheduling strategies and policies through implementation and experimentation. We have evaluated the batching algorithms under different system loads.

We have evaluated the effect of context-switch overhead on batching algorithms in overload conditions and proposed the TF policy to reduce this overhead. This policy integrates all tasks in a batch into one single process. Integration reduces the overall number of context switches and therefore, the overhead caused by context switching. The results of experiments show the improved performance of batching algorithms under heavy loads using TF policy.

Finally, we have applied and evaluated the NT policy on the EDF-based algorithm to reduce the *missed deadline ratio* by eliminating tasks whose deadlines have passed. Experiments show that a batching EDF algorithm combined with TF and NT policies have the best performance in overload conditions in comparison to the rest of their counterparts.

As we constructed the experimental platform for the purpose of our thesis, we have modified the underlying operating system, RTLinux. We have added more
functionality to the operating system. We have also modified the network device driver to receive data in real-time with less overhead.

1-4 Thesis Outline

Following the introduction of the thesis topic in the first chapter, Chapter 2 provides background information on real-time systems and their attributes. It also explains different real-time system categories: soft, hard and firm. As will be shown in the second chapter of this thesis, real-time operating system is an important component of a real-time system. A brief overview of real-time operating systems will be provided. We introduce real-time scheduling in the second chapter, followed by the description of static and dynamic real-time scheduling. We conclude Chapter 2 with a description of the two base algorithms used in this thesis, namely Earliest Deadline First and Least Slack First.

In Chapter 3, an overview of the experimental platform is provided. The platform, an In-memory Real-time Query Processor (IRQP) is broken down and explained in this chapter.

In Chapter 4, we propose the BEDF and BLSF algorithms. We evaluate our proposed techniques through implementation and experimentation. The effect of different batch sizes on the performance of batching algorithms is evaluated in Chapter 4 as well.

Chapter 5 investigates the performance of batching algorithms in overload conditions. It introduces the impact of context switch overhead on BLSF and BEDF in heavy load. Thus a new strategy, TF policy for reducing this overhead is proposed. Finally, we evaluate another load management policy, NT, in the last section of Chapter 5.
Chapter 2

Background

In this chapter, we provide an overview of real-time systems and their specifications. We also introduce several variations of real-time systems and discuss how they differ from each other. A brief introduction of real-time operating systems and databases is also provided. This chapter focuses on achieving real-time scheduling with minimal overhead, the core contribution of this thesis. Existing scheduling schemes such as rate monotonic, dynamic scheduling and static scheduling in real-time systems will be discussed. Two basic dynamic scheduling algorithms, Earliest Deadline First and Least Slack First, will be explained. The chapter is concluded with related works in this topic.

2-1 Real-Time Systems

One of the fastest expanding areas in computer science involves applications whose prime function is not information processing alone, but processing information in a timely manner. A microprocessor controller system in an airplane is a good example of such a system. Here, the prime function is not only to process information but also to navigate the airplane, considering all safety matters within tight time constraints. This type of computer application is generally called a real-time system. Real-time systems are defined as those systems in which the correctness of the system depends not only on the
logical result of computation, but also on the time in which the results are produced [11][12][13][14][15].

The Predictably Dependable Computer Systems (PDCS) project [17] gives the following definition: "A real-time system is a system that is required to react to a stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment." Others such as Young [16] define a real-time system in similar ways: "any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified period." This finite and specified period is also referred to as a deadline. The deadline in these systems can be in milliseconds, minutes or even hours. The problem and the design principles remain the same, but the time scale changes from one real-time system to another.

Real-time systems are categorized into the following subgroups, depending on how critical is to meet the deadline:

**Soft real-time system:** The value of computing a result decreases gradually after the deadline expires. The decrease of the resulting value is parallel to an increase in the penalty. An example of a soft real-time application is a toaster control where the penalty for a late response is a darker toast.

**Hard real-time systems:** The deadlines are strict hence meeting them is crucial. The value of the computing result becomes zero after the deadline expires. Thus, the penalty of this miss can be catastrophic. An example of a hard real-time system is an airplane landing system, where failure to respond by the specific deadline may cause loss of life.
Firm real-time systems: Firm refers to a real-time system with hard deadlines but where a low probability of missing a deadline can be tolerated. In some systems optional firm components may be given probabilistic requirements.

A real-time system can be viewed as follows.

![Figure 2-1: Illustration of a Real-Time System](image)

As shown in Figure 2-1, a complete picture of a real-time system is an interactive system that reacts to the input received from the external or internal environment. A sensor senses the stimuli received from the environment in the real-time system. The sensor signals the processing unit through the hardware interface. The processing unit that includes the real-time operating system and the real-time application will then do the
real-time computing within defined timing constraints. The real-time operating system provides a schedule upon which the application executes. The art of real-time scheduling is to produce schedules with the least missed deadline ratio possible. The valid result of the computing process will activate the actuator.

In this thesis our platform, a real-time main memory database, is referred to as In-memory Real-time Query Processor (IRQP) system. This platform is a sub-class of Figure 2-1. Queries are the input generated in the environment, transferred for processing in task format to a real-time operating system. Within the operating system, the scheduler applies different scheduling schemes to find the eligible task for execution. The real-time operating system used in our thesis is RTLinux and the scheduling algorithms are implemented on the top of this operating system. Both the real-time operating system and the real-time scheduling algorithms will be explained later in more detail. After the task is chosen for execution, the real-time application, which in our case is a query processor, starts running.

2-1-1 Real-Time Main Memory Database

In recent years, Real-Time Databases (RTDB) have been widely used in many time-critical applications. RTDB can be defined as a database providing real-time responses to the queries of data-intensive applications, such as computer-integrated manufacturing, the stock market, banking and command and control systems. Similar to a conventional real-time system, the queries processed in the database have timing constraints, usually in the form of deadlines. What makes a RTDB different from a conventional real-time system is its requirement to preserve the consistency of data, as well as considering the real-time requirements of the queries [12][18].
With the decreasing cost of memory in recent years, it becomes feasible to store the databases in main memory for improving the overall performance of an RTDB. The main difference between a main memory database and a disk resident database is that no disk I/O is needed for a query to access the data in a main memory database [30]. In a Real-Time Main Memory Database (RTMMDB) data can be accessed directly in memory; therefore it can provide much better response times and query throughputs, as compared to conventional databases and RTDB. However, there exists a tradeoff between the performance improvement by using more memory to store databases and the memory cost required [28][29].

If a database permits the users to read and write data, concurrency control and data consistency should be considered. However since the platform in this thesis is a read-only RTMMDB, concurrency control problem is not addressed.

2-1-2 Real-Time Operating System

In real-time computing, the correctness of the system depends not only on the logical result, but also on the time by which the results are produced. Explicitly dealing with time makes building and analyzing real-time systems difficult. If one can show that a task or set of tasks can meet their timing constraints, it is said that they are predictable. One important component of an effective, predictable real-time system is the real-time operating system. The operating system itself must be predictable. This means that the execution times of operating system primitives in process management, memory management, IPC, etc. must be small and bounded.

In discussing real-time operating systems, it is possible to categorize them into three general groups [13]:

- Real-Time Extensions to Commercial Operating Systems
- Research Operating Systems
- Small, Fast, Proprietary Kernels

A real-time extension to conventional operating systems is generally the extension of commercial products; e.g., RT-UNIX is a real-time extension for UNIX. Although this type is slower and less predictable than the proprietary kernels, it has greater functionality and better software development environments. While many real-time applications are constructed for commercial real-time operating systems, significant problems still exist. In particular, commercial offerings emphasize speed rather than predictability, thereby perpetuating the myth that real-time operating computing is fast computing. Research operating systems on the other hand emphasize predictability. The operating system used in this thesis, RTLinux, is in the third category. In the following section, we explain the specifications of the third category in more detail.

2-1-2-1 Small, Fast, Proprietary Kernels

These kernels are often used for small, embedded systems when very fast and highly predictable execution time must be guaranteed. To achieve speed and predictability, these kernels are stripped down and are optimized from versions of time-sharing operating systems. To reduce run-time overheads incurred by the kernel and to make the system fast, the kernel must

- have a fast context switch (minimal overhead during context switch),
- have a small size (with its associated minimal functionality),
- respond to external interrupts quickly,
- minimize intervals during which interrupts are disabled,
- provide fixed or variable sized partitions for memory management (i.e., no virtual memory),
- provide special sequential files that can accumulate data at a fast rate.

In order to deal with timing requirements, the kernel maintains a real-time clock. Special alarms and timeouts are provided. A priority scheduling mechanism is also provided in the kernel. As the focus of this thesis is to evaluate scheduling techniques, background on scheduling mechanisms will be given in the following section.

2-2 Real-Time Scheduling

Scheduling involves the allocation of resources and time to tasks in such a way that certain performance requirements are met. Scheduling has been perhaps the most widely researched topic within real-time systems.

As it is stated in [1][5][6][13], real-time task scheduling is characterized by the sequencing a set of tasks and assigning system resources in order to comply with each task's timing constraints. Based on the time at which the scheduling is performed, the real-time scheduling algorithms have been divided into two categories: static and dynamic.

Static scheduling: In this strategy, the order of task execution and resource allocation is determined off-line prior to the start of problem solving. Such approaches require prior knowledge of the task set and its time constraints. The resulting schedule is fixed permanently. However, due to the commonly occurring worst case assumption of tasks' execution times, a fixed schedule may lead to poor utilization of CPU and/or incorrect decision making on the schedulability of tasks. Static scheduling has a complete knowledge of the task set and its
constraints, including deadlines, computation times, precedence constraints, and future release times. This set of assumptions is realistic for many real-time systems. For example, a simple laboratory experiment or a simple process control application might have a fixed set of sensors and actuators as well as a well-defined environment and processing requirements. The static scheduling algorithms operate on these task-sets and produce a single fixed schedule. If all future release times are known when the algorithm is developing the schedule, then it is still a static algorithm.

**Dynamic scheduling:** These algorithms perform on-line task sequencing and resource allocation for the purpose of including more comprehensive and up-to-date information about tasks and the environment. A dynamic scheduling algorithm has complete knowledge of currently active tasks. For example, teams of robots cleaning up a chemical spill, or military command and control applications, require dynamic scheduling. A major problem to be addressed in dynamic scheduling is the complexity of the scheduling algorithm. Due to the on-line nature of their problem solving, the complexity of those algorithms directly affects their performance. Using low-complexity algorithms results in negligible scheduling costs, however, they may diminish the quality of the resulting schedules in terms of guaranteeing deadline compliance. An optimal schedule of the currently active tasks, on the other hand may incur large scheduling costs resulting in shifting the deliverable schedule by a prohibitive time interval and introducing a hazardous delay for considering newly arrived tasks.
There are many ways to dynamically assign priorities to tasks and schedule them. We have chosen two of them as the basis of our scheduling algorithms in this thesis: *Earliest Deadline First* and *Least Slack First* [10].

**2-2-1 Earliest Deadline First**

From among the arriving tasks in the scheduler, one is chosen for execution. This mechanism gives a task with the earliest deadline the highest priority. The selection is done through either a *search* or a *sort* among tasks. The *search* mechanism applies a *Min* function to the tasks, searching for the task with minimum deadline. After the selected task is executed, the next task is chosen in the same way. The *sort* mechanism sorts the tasks once in increasing order. The tasks will then be chosen one after another from this sorted list for execution. In this thesis, we have employed the search-based version of EDF.

**2-2-2 Least Slack First**

For a task $T$ we define a slack time $S = d - (t + E - P)$, where

- $d$ is the deadline,
- $t$ is the current time,
- $E$ is the runtime estimate,
- and $P$ is the amount of service time received by $T$ so far.

Note that $P$ is zero if the scheduling is non-preemptive. The slack time is an estimate of how long we can delay the execution of $T$ and still meet its deadline. If $S \geq 0$ then we expect that if $T$ is executed without interruption then it will finish at or before its
deadline. When a task has already missed its deadline or when it is estimated that it cannot meet its deadline, a negative slack time results.

A natural question to consider is how often to evaluate a task’s slack. Two methods can be considered: static evaluation and continuous evaluation. In the former, the slack of a task is evaluated only once, when the task arrives; in the latter, the slack is recalculated whenever one wishes to know the task’s priority. In the static evaluation, the task priority is its slack value for as long as the task is in the system. Through out this thesis, static evaluation is applied to calculate slack value.

2-3 Related Work

EDF and LSF variations have been studied in many real-time systems. One of the main surveys done on the comparison of these algorithms is in [10]. In this survey, a study is presented on performance evaluation of EDF and LSF (static evaluation) algorithms in different system loads. The affect of different overload management policies such as NOT Tardy are also examined on some candidate algorithms. The survey approves our similar experiment results observed through this thesis. It can be seen that EDF algorithm outperforms the LSF algorithm in normal conditions, while it poorly performs under heavy loads. The suggested Not Tardy policy in the survey improves the EDF algorithm performance in heavy load, as it does in this thesis.

The overhead associated with the scheduling algorithm is an important factor in increasing the performance. In this thesis, we study the overhead due to the frequency of scheduling algorithm invocation and the size of the task-set to which the scheduling policy is applied. Others studies, by B. Hamidzadeh et al. [7], have different point of view on scheduling overhead. The proposed Time Controlled Dynamic Scheduling
algorithm [7], TCDS, addresses a fundamental trade-off in dynamic scheduling between the cost of scheduling and the quality of resulting schedules. While the same task model is used, it is shown that taking into account the scheduling time is crucial for honoring the deadlines of scheduled real-time tasks.

In this thesis the size of task-set to which the scheduling policy is applied is controlled using the batching approach. In studies by Haritsa et al. [24], a priority assignment algorithm called Adaptive Earliest Deadline (AED) is presented that stabilizes the EDF overload performance using a feedback control mechanism. This mechanism selects the group of tasks that are sustainable under an EDF schedule. Therefore, the EDF scheduling is applied only to a group of tasks in the system. A Random Priority assignment policy is applied to the remaining tasks. The batching approach presented in this thesis is similar to AED in that both apply their scheduling algorithm to a task-set with limited size.

2-4 Summary

In this chapter, a definition of a real-time system was provided. A real-time system can be hard, soft or firm depending on the importance of meeting its deadline on time. We also presented a structural overview of a real-time system. In the overview, we learned that a complete real-time system is composed of environment, sensors, actuators, hardware interface, real-time operating system and a real-time application. A brief review on real-time main memory databases was provided. As real-time operating system and scheduling are the focus of our thesis, a more detailed overview of the two subjects was provided. Under the scheduling section, the basis of all the algorithms studied in this
thesis, *Earliest Deadline First* and Least Slack First were discussed. Related work by other scientists in the area of dynamic scheduling concluded this chapter.
Chapter 3

Experimental Platform

This chapter gives an overview of the thesis' experimental platform, referred to as In-memory Real-time Query Processor (IRQP). IRQP is a real-time main memory database system. We continue with sections on query environment, query format and query generation. IRQP uses RTLinux as the real-time operating system. The properties of this operating system will be described in more detail later in this chapter. The application used in this real-time system is a query processor, which will also be described.

3-1 Experimental Model

We have built IRQP, a real-time main memory database system, as the platform for our experiments. As mentioned in section 2-1, a real-time system is composed of different units: a real-time application, a real-time operating system, an environment, etc.. Our customized real-time system in this thesis is composed of the following:

- Query Environment
- RTLinux
- Query Processor
As we also stated in Chapter 2, a real-time system must react to a stimuli from the environment within a certain timing constraint. In this platform, the environment creating stimuli for the system is the Query Environment (QE) and the stimuli are the queries sent to the operating system. QE generates and sends the queries in a Poisson arrival pattern. After RTLinux operating system receives the queries, it transforms them to a proper real-time task format, referred to as task, suitable for scheduling. The scheduler will select a task according to its scheduling policy. The selected task will be processed in Query Processor (QP).

We have chosen RTLinux operating system for its small scale, fast kernel, low interrupt latency and its simplicity. Its scheduler is a module easily modified by users for experimental purposes.

QP, the real-time application used in this system, is composed of a read-only memory-resident database, also referred to as global relation, and a grid structure. QP
processes the queries by applying an access method, *Index and Data Distribution mechanism*, to this database.

A client-server model is employed in QE. The client generates queries and transmits them in a special query format to the server, while the server receives and parses messages and passes them to the other layers for query processing.

**Sequence of Experiments:**

1. Series of queries are generated in QE;
2. The queries are transferred to RTLinux;
3. In the scheduler unit of the operating system, the suitable query is scheduled for execution based on a scheduling policy that aims to minimize the *missed deadline ratio*;
4. The selected query will be processed in QP;
5. The *missed deadline ratio* is calculated.

In the following sections, we briefly explain composing units of our real-time system platform: Query Environment, RTLinux and Query Processor.

### 3-2 Query Environment

We have employed a *client-server* model for query generation. The client generates the queries and sends them for scheduling to the server in special query format. Meanwhile, the server receives messages, in parallel with the scheduler, parses the queries out of the messages and passes them to the scheduler in an appropriate time.

This model gives us an opportunity to evaluate our candidate algorithms in a real world environment. A great learning experience, this model has introduced us to
technical issues regarding network device drivers, aspects of real-time networking and kernel programming. In the following sections details regarding, query format, query generation and required implementation issues are provided.

3-2-1 Query Format

A query consists of the following information:

- inquired key values,
- inquired key indices,
- query deadline.

The query format is database dependent; therefore, we have chosen a format to fulfill the requirement of our database, or in our case a global relation. As we will describe in more detail in section 3-4, each database record, or tuple, is composed of three fields of integer size. Therefore, the query format requires having three fields in integer size. One field indicates the key indices for the query and one field indicates the deadline. In order to speed up the query parsing in the real-time system, we chose a simple design for the query format. Thus the length of the message is fixed and is dependent on the database attributes.

<table>
<thead>
<tr>
<th>deadline</th>
<th>which-key</th>
<th>key1</th>
<th>key2</th>
<th>key3</th>
</tr>
</thead>
</table>

Integer

*Figure 3-2: Query Format*
**deadline:** Deadlines $d_i$ are uniformly distributed in intervals of *maximum query processing time* and *maximum deadline value*.

$$\text{deadline} = \text{uniform} (\text{MaxProcessTime}, D_{\text{max}}),$$

$D_{\text{max}}$ is the maximum value a deadline can have and is calculated as

$$D_{\text{max}} = \text{MaxProcessTime} + SF^{kd}.$$

$SF^{kd}$ is a parameter that controls the degree of *slack time* in query deadlines. *Slack time* is the maximum time span in which a query can be delayed without missing its deadline. $kd$ is fixed to five in our experiments. $SF$ ranges from 1 to 10. Larger $SF$ values represent larger slack time, whereas smaller $SF$ values represent smaller slack time.

*MaxProcessTime* is the worst case processing time of a task and it is determined by a series of experiments. Average processing time of the queries has been measured based on repetitive runs of QP with 10,000 queries, in hundreds of iterations. The worst case among all the iterations has been chosen for the worst case query processing time that is referred to as the *MaxProcessTime*.

**which_key:** This field is placed in order to avoid the overhead of expensive string operation on QP side and hence to provide better real-time performance. This field indicates the desired keys to be queried from the database. This method of indication uses binary coding. If a key is going to be queried, it will be assigned to digit one, otherwise to zero. Field *which_key* is assigned to the decimal equivalent of the digits. For example, if keys one and two are being queried, the indicator is 1-1-0 with a decimal value equal to six. In this case, *which_key* is assigned to six.
**key1, key2, and key3:** These fields are the desired key values to be inquired. If a query consists of only two keys, the field would be filled by zero.

Here is an example to clarify the query format. User X is inquiring for a tuple in the database. The query is to search for all the tuples in the database with $key1 = 9$ and $key3 = 11$. This query has to meet a deadline of 8912 units, or ticks (as will be explained in next section), from its receipt time. As only $key1$ and $key3$ are asked for in this query, the $which_key$ value is 101 or five. Thus, the query format for this example is 891259011 as shown in Figure 3-3:

![Figure 3-3: A Sample Query](image)

### 3-2-2 Query Generation

The client is responsible for generating a number of queries for application on the server side. A *Poisson process* is used to create a sequence of aperiodic tasks arrivals. The client sends the queries to the server based on this sequence. The process’ priority is set to real-time mode in order to have the least latency in sending queries. The time window, within which the arrivals are observed, is set to 1000 time units. The arrival rate $\lambda$, ranges from 1 to 10 in our experiments. Larger $\lambda$ values represent larger system load, whereas small $\lambda$ values represent smaller system load.
3-2-3 Query Arrival

On the server side the receiver unit or, the network device driver, is responsible for receiving messages and storing them in a buffer. The messages are stored in the buffer based on an increasing order of arrival time. In an appropriate time depending on the scheduling algorithm, the receiver passes the queries to the QP.

The process of receiving messages happens in parallel with query processing. In order not to lose any messages, two buffers are required: one on the receiver unit, and the other on the scheduling unit of the server. Dynamically allocating memory is time consuming and very non-deterministic. Thus, we have statically allocated both buffers on the initialization level of the system.

**Receiver Buffer:** The buffer that holds the queries in the receiver unit of the server upon query de-framing. This buffer is fed from the client.

**Scheduler Buffer:** The buffer that holds the queries in the real-time operating system, before query scheduling. This buffer is fed from the Receiver Buffer.

Upon pulling the frame off the data-link layer the data portion is extracted, or de-framed. The data or, query is stored into the Receiver Buffer until a scheduler indicates its readiness for accepting new queries. When the scheduler is ready to accept more queries, Receiver Buffer and Scheduler Buffer pointers are switched. This is done to avoid copying, which is expensive and time consuming, thus not suitable for a real-time system. The queries in the Scheduler Buffer are transformed to a real-time task format and scheduled for execution.
If the scheduler is not idle, but busy with scheduling or executing jobs, the scheduler readiness depends on its scheduling algorithm. If a non-batching algorithm is applied, the scheduler is ready to receive more queries after processing of a task is complete. If a batching algorithm is applied, the scheduler is ready to receive more queries after processing of a batch of tasks is complete.

In the following section, we explain the modification needed on the receiver to perform within real-time standards.

3-2-3-1 Modification on the Receiver

The messages on the client side are sent with short intervals to the server. To achieve the least latency on the client side, the sender process is set to real-time priority. In addition, the server networking should be modified to be capable of receiving messages in real-time.

On the server side the original protocol of the receiver, TCP/IP, is very complex and has lots of overhead, thus it is non-deterministic. A solution would be to implement a TCP/IP layer for the server operating system, RTLinux. However, TCP/IP protocol is fundamentally non real-time. Another simpler approach is adopted [23].

In this approach, the server network device communicates directly with the receiving process. It is important to note that this gives us an interface to RAW Ethernet frames on a data link layer, not IP/UDP/TCP networking layers. That is, the real-time receiver pulls out the raw frame before it goes to the higher layers of protocol. This is not an issue here because we have opened special sockets on the client side to send raw frames to the receiver.
3-3 RTLinux

Although Linux has all the features of a modern UNIX operating system, it has several problems preventing it from being a hard real-time operating system. The problems not allowing Linux to be real-time include its

- time-sharing scheduling,
- virtual memory system,
- timing unpredictability,
- and high interrupt latency.

These issues were the motivation for scientists at the New Mexico Institute of Mining and Technology, (Michael Barbanov and al.) [19][20][21] to design and implement RTLinux.

RTLinux is the hard real-time variant of Linux. Controlling robots, data acquisition systems, manufacturing plants and other time-sensitive instruments and machines is possible using RTLinux. As stated in Chapter 2, RTLinux has the properties of a fast, small, proprietary kernel real-time operating system.

**Interrupt Emulation:** The problem of unpredictable interrupt dispatch latency in Linux is solved in RTLinux by addition of an emulation software layer between the Linux kernel and the interrupt control hardware. Upon arrival of an interrupt, RTLinux fetches the interrupt and decides what to do with it. If that interrupt line is registered in RTLinux, immediately the special handler for that interrupt will be invoked. However, if the interrupt belongs to Linux, it will be marked as pending. If Linux has asked to enable the pending interrupt, its handler will be invoked. This will result in low interrupt latency in RTLinux. The worst case time between
the moment a hardware interrupt is detected by the processor and the moment an
interrupt handler starts to execute is under 15 microseconds on a generic x86
processor [20].

**Timing:** While operating systems like Linux use periodic clock interrupts as its
timer, RTLinux uses a programmable interval timer to interrupt the CPU only
when needed. An interrupt can be scheduled with approximately 1 microsecond
precision using this mode, while in Linux this precision is limited to 10
milliseconds. In this scheme, the overhead of interrupt processing is minimal
while the timer resolution is high. To keep track of the global time, all intervals
between interrupts are summed up together. The timer interface allows the
scheduler to obtain the current time. The return value of this interface is in *ticks*,
or the number of timer interrupts since the scheduler is loaded in the kernel. A
second is equal to 1,193,180 *ticks*.

**Scheduler:** In most real-time systems, the scheduler is a large, complex piece of
code that cannot be extended in anyway. The user can only modify the behavior
of the scheduler by adjusting parameters, which may not be enough. In contrast,
the RTLinux scheduler is a small piece of code and is easy to customize it for the
application programmer's needs. It can be implemented as *loadable kernel
modules*. Kernel modules are object files that can be dynamically loaded into the
kernel address space and linked with the kernel code. This makes it possible to
easily experiment with different scheduling policies and algorithms and to find
the ones that best suit the specific application.
Interrupt emulation results in low interrupt latency and fast context switch. Therefore, RTLinux is a suitable choice for our real-time system platform. The small and modifiable scheduler of RTLinux provides us an excellent experimental environment, where we can modify or substitute our desired scheduler according to our needs.

3-4 Query Processor

The application chosen for our thesis is Query Processor (QP). In QP, the main concern is to minimize the query processing time. Thus the query processing is designed to be simple and with minimum overhead. The simplicity and low processing time of our queries match the RTLinux design purpose. In RTLinux design, the intention is to give a small job to the real-time task application. Applying a special access method, an Index and Data Distribution mechanism, to our global relation in QP (using the Grid Structure) reduces the processing time per query.

Global Relation: This is a read-only memory resident database. Tuple (database record) attributes are arbitrary and the number of the attributes can be changed by the requirements of the environment. For this thesis, we have chosen tuples with three attributes, $a_1$, $a_2$, and $a_3$. The value of each attribute is an integer. The domain values of the attributes are, $a_1[1..10]$, $a_2[1..10]$, and $a_3[1..100]$. The generation of the global relation may be a time-consuming process, depending on its domain sizes and number of its attribute. Thus it is generated in the initialization stage at RTLinux module loading time and has no effect on the real-time task performance.
Query processing is done using a priori knowledge of the distribution and location of attributes of each tuple. This knowledge is obtained through a search in the global relation and generation of a grid structure at initialization time.

**Grid Structure:** A grid can be regarded as a regular decomposition of the space, for all possible tuples in the database into a collection of disjointed cells. Each attribute in the relation is assigned to a grid structure. A grid contains $n$ number of cells, where $n$ is the number of values in the attribute domain, or the cardinality of the attribute domain. Each grid cell is used to store the access information for its corresponding domain value. This information consists of the position of the first and the last tuple with attributes matched to the corresponding cell index, the number of the matched tuples, and the index list to all matched tuples.

The database in the Figure 3-4 is a small example, constructed of two attributes, $a1$ and $a2$. The attribute domain for $a1$ is [1,2,3] and the attribute domain for $a2$ is [a,b,c,d]. As this database has two attributes, $a1$ and $a2$, we generate two grid structures, $Gal$ and $Ga2$ respectively. In order to clarify the idea of grid structure we have a closer look at a grid cell. Cell 3 of $Ga2$ is representative of the third value of $a2$ domain. As there are three tuples in the global relation, with $a2 = 3$, the distribution for the value of this attribute is set to three. As shown in Figure 3-4, all tuples with the same attribute value are linked in a link list. The first and last nodes of the list are also provided in the grid structure. For example, the first tuple in the list with $a2 = 3$ is the third tuple in the global relation.
Figure 3-4: A view of Global Relation and Grid Structure of a sample database
Using the grid structures and the access method, Index and Data Distribution mechanism (IDD), the queries are processed quickly. As the result of the grid structure composition, a chain of pointers linking the tuples satisfying the queries is available at each grid cell. The distribution of the matching tuples is also available at the cell. The satisfying tuples can be accessed directly. Therefore, the searching can be done along the shortest chain. The shortest chain is determined by finding the minimum data distribution, among all inquired keys. No searching is required if the distribution is zero. This access method is referred to as the Index and Data Distribution mechanism. The pseudo code for this access method is presented below.
Index and Data Distribution mechanism (IDD)

1- Check for the validity of the range of key1, key2, key3
   • If all invalid STOP
   • Else CONTINUE

2- Determine the keys to be queried, using which_key
   • If valid STOP
   • Else CONTINUE

3- Determine the minimum data distribution among the requested keys in the grid cell
   • If zero STOP
   • Else CONTINUE

4- Search along the link list of the grid cell with the minimum data distribution, starting from the first tuple to the last tuple.

Figure 3-5: Pseudo Code for IDD Mechanism
3-5 Summary

In this chapter, the experimental platform of the thesis was presented. The platform is a real-time main memory database, also referred to as *In-memory Real-time Query Processor*. Its composing units are, Query Environment, RTLinux Real-time Operating System and Query Processor. The Query Environment is the unit in which the queries are generated. Queries are generated in a client based on a Poisson arrival pattern and later transmitted to server for process. RTLinux is employed in the platform for its ease of modification and its fast and small kernel. The chapter concludes with a definition of the Query Processor unit. This unit performs efficient query processing by employing a *Grid Structure* and an *Index and Data Distribution* mechanism.
Chapter 4

Batching Algorithms

In this chapter we investigate

- the trade-off in the dynamic scheduling of real-time tasks between the frequency at which the scheduling algorithm is invoked,

- the size of the task-set to which the scheduling policy is applied at every invocation,

- and the quality of the resulting schedules in terms of deadline compliance.

We identify two classes of algorithms, one of which forms a batch of arrived tasks, then schedules and executes all tasks in a batch before considering other tasks that arrive in the mean time. The other class schedules arrived tasks more frequently and applies the scheduling policy to all available tasks. We compare the performance of a batching and non-batching technique, both of which apply Earliest-Deadline-First (EDF) or Least Slack First (LSF) policies to prioritize tasks. The batching algorithms are also called *Batching Earliest Deadline First (BEDF)* or *Batching Least Slack First (BLSF)* with respect to their used scheduling policy, EDF or LSF.

Experimental evaluation of the proposed algorithms shows that our batching algorithms outperform their non-batching counterparts under tighter time constraints.
4-1 Batching Algorithms

A common characteristic of systems employing dynamic scheduling techniques is that scheduling and execution are interleaved as intervals of scheduling followed by execution of one or more scheduled tasks. Different patterns of interleaving the scheduling and execution can be conceived, each of which can suit different situations. The frequency at which a scheduling policy is applied and the number of tasks that are executed during the execution intervals can decide the interleaving pattern.

A common pattern [11] applies the scheduling policy to all available tasks after the arrival or execution of every task in the system. Note that using this type of strategy, the scheduling policy will be applied to a large number of tasks and invoked frequently, thus increasing the scheduling overhead significantly, even when simple and low-complexity algorithms are adopted. The execution interval, in many of these techniques, consists of executing one task (i.e., the highest priority one).

As in other dynamic schemes, a batching algorithm interleaves scheduling and execution. The distinct characteristic of this algorithm is that it forms batches of arrived tasks to which the scheduling policy is applied. We refer to the scheduling and execution of tasks in a batch as a phase. New tasks that arrive during a scheduling and its associated execution in a phase form a new batch for the following scheduling and execution intervals. The trade-off in this strategy is the reduction of the frequency and complexity of the scheduling algorithm at the cost of deferring consideration of new tasks.

We shall investigate the performance of such a strategy in later sections. In the remainder of this section, we first specify the basic scheduling and execution steps of a
phase in a batching algorithm. We then discuss the implementation of batching algorithm in the RTLinux operating system.

4-1-1 Scheduling Steps

In order to specify the basic scheduling steps of batching algorithms, we must explain the task model first.

Task Model and Problem: In this thesis, we address the problem of scheduling a set $T$ of $n$ aperiodic, non-preemptive real-time tasks with deadlines on an uni­processor architecture. Aperiodic tasks are tasks with arbitrary arrival times whose characteristics are not known a priori. Each task $T_i$ in $T$ is characterized by a processing time $p_i$, arrival time $a_i$, and a deadline $d_i$ (See Table 4-1 for definition of these and other notation).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Batch$</td>
<td>A set of tasks to schedule</td>
</tr>
<tr>
<td>$S_i$</td>
<td>A schedule: ordered(prioritized) set of tasks</td>
</tr>
<tr>
<td>$T_i$</td>
<td>A Task</td>
</tr>
<tr>
<td>$P_i$</td>
<td>Worst-case processing time of $T_i$</td>
</tr>
<tr>
<td>$A_i$</td>
<td>Arrival time of $T_i$</td>
</tr>
<tr>
<td>$D_i$</td>
<td>Deadline of $T_i$</td>
</tr>
<tr>
<td>Slack $i$</td>
<td>The maximum time by which the start of $T_i$'s execution can be delayed from time current time, without missing its deadline</td>
</tr>
</tbody>
</table>

Table 4-1: Task Model Notation

A task is defined as feasibly scheduled if that task meets its deadline when executed. The objective of a scheduler is to maximize the number of feasible tasks (or minimize missed deadlines).
The input to each new phase \( j \) is a set of tasks (i.e. \( \text{Batch}(j) \)). Initially, \( \text{Batch}(0) \) consists of a set of the arrived tasks. In BEDF, the EDF policy is applied to the tasks in each batch by applying a Min. function to the batch to search for the task with the earliest deadline, every time a task is to be selected for execution. In BLSF, the LSF scheduling policy is applied to a batch. Instead of searching for the earliest deadline, it selects the task with least slack.

Task execution in phase \( j \) of the search-based batching algorithm consists of executing one task. So, this technique applies the Min. function, which searches and selects for a minimum value, to \( \text{Batch}(j) \) of phase \( j \), executes the selected task, removes that task from the batch, and applies the Min. function to \( \text{Batch}(j) \) again to select the next task to execute. This process continues until all tasks in \( \text{Batch}(j) \) are executed. Phase \( (j+1) \) starts after the tasks in \( \text{Batch}(j) \) have been executed (i.e., after execution phase \( j \)). \( \text{Batch}(j+1) \) consists of the set of tasks that arrived during scheduling and execution in phase \( j \).

An important point to note is that the scheduling complexity of batching techniques (BEDF/BLSF) is both lower and controlled compared to a corresponding non-batching technique (e.g., basic EDF/LSF). This lower complexity is due to the fact that the scheduling policy is applied to a shrinking batch, in a batching scheme, whereas in a non-batching scheme the scheduling policy is applied on average to a growing set of tasks, particularly at higher arrival rates.

Tasks in a batching technique are executed non-preemptively in the sense that a new task out of the batch with a higher priority than the running task will not preempt the running task to take control of the CPU.
4-2 Experimental Evaluation

In this section, we compare batching algorithms with non-batching algorithms. We have selected EDF and LSF as our two non-batching algorithms that to be compared with BEDF and BLSF respectively.

In these series of experiments, BEDF and BLSF algorithms are implemented with a variable batch size. That is, all of the tasks in the batch will be scheduled in one phase. The effect of fixed batch size will be studied in the next sections of this chapter.

The EDF algorithm simply executes tasks in the ascending order of their deadlines, in a search-based fashion. The LSF algorithm follows the same sequence. Contrary to EDF, LSF executes tasks in the increasing order of their slack times. The drawback to either (non-batching) algorithm is that it applies prioritization process to the same tasks several times just to see where a new task stands in the entire set of arrived tasks. Note that the set of tasks can grow rapidly and increase the scheduling time as tasks arrive.

The graphs presented in this section represent a comparison of the percentage of missed deadlines over total number of tasks in the system. As stated earlier, this percentage is referred to as the missed deadline ratio. This ratio compares the candidate algorithms, on a slack factor range of 1 to 10. Note that arrival rate $\lambda$ [Section 3-2-2] is set to one in these experiments.

In this section, we evaluate batching algorithms through a number of performance-comparison experiments. The experiments are organized as follows:

- In one set of experiments, we compare the performance of the BEDF algorithm that searches the current batch each time for the next task to
execute, and an EDF algorithm that searches all arrived tasks for the next task to execute. Our goal is to reveal the situations under which a BEDF algorithm performs well.

- In a second set of experiments, we compare the performance of a BLSF algorithm that searches the current batch each time for the next task to execute, and an LSF algorithm that searches all arrived tasks for the next task to execute. Our goal is to reveal the situations under which a BLSF algorithm performs well.

- In a third set of experiments, we compare the performance of a BLSF algorithm and BEDF algorithm. Our goal is to reveal the situations under which a BEDF algorithm performs well.

4-2-1 Comparison of EDF and BEDF Algorithms

Figure 4-1 shows the performance of search-based EDF and BEDF algorithms. BEDF outperforms EDF over a wider range of slack factor values, up to a slack factor of seven. This is because during execution of a task many new tasks may have arrived, the scheduling of all of which will incur a large overhead for EDF. This longer scheduling period will cause many of the tasks in the task set to miss their deadlines. In the meantime, even more tasks arrive into the system and the snowball effect causes more deadline violations. BEDF, in such situations, applies its scheduling algorithm to a shrinking batch, so its complexity is maintained at a lower level than EDF. It is true that in slack factors higher than seven, by the time BEDF finishes execution of a batch, several tasks with higher priorities may have arrived and some tasks may have missed their deadlines, thus a priority inversion occurs. But the overall performance is improved
compared to non-batching EDF. This performance may be even better, if the number of tasks in a batch is reduced. In this set of experiments the batch size is variable. Therefore, as the number of tasks grows in the system, the number of tasks in the batch increases, thus the algorithm suffers from higher scheduling overhead. In section 4-3-1 the effect of limited batch size and its superior performance are presented.

Figure 4-1: Comparison of EDF and BEDF Algorithms

4-2-2 Comparison of LSF and BLSF Algorithms

Figure 4-2 shows the performance of search-based LSF and BLSF algorithms. BLSF outperforms LSF all over the range of slack factor values, from 1 to 10. The same reasoning as the previous section applies to this comparison. As explained for EDF in the previous section, during execution of a task many new tasks may have arrived scheduling
all of which will incur a large overhead for LSF. This longer scheduling period will cause many of the tasks in the task set to miss their deadlines. In the meantime, even more tasks arrive into the system and the snowball effect causes several deadline violations. In such situations, our batching algorithm, BLSF, applies the scheduling policy to a shrinking batch. As the BLSF scheduling policy is applied to a smaller number of tasks, complexity is maintained at a lower level than LSF.

![Missed Deadline Ratio for LSF and BLSF](image)

**Figure 4-2: Comparison of LSF and BLSF Algorithms**

**4-2-3 Comparison of BEDF and BLSF Algorithms**

Figure 4-3 shows the performance of search-based BEDF and BLSF. BEDF outperforms BLSF over a wide range of slack factor values, with slack factor of 3 to 10. We justify
BEDF outperformance by a reference to a theorem on EDF performance in normal load [25][26].

Liu's theorem concludes "the earliest deadline first scheduling will produce an optimal schedule in the sense that if a set of jobs can be scheduled to meet all the deadlines, the earliest deadline first scheduling algorithm will produce one such schedule. On the other hand, if according to the earliest deadline first scheduling algorithm, one or more jobs will miss their deadlines, then there is no schedule in which all jobs will meet their deadlines."

Studies [10][11][15][24] have also observed, however, that the performance of EDF sharply degrades in an overload system. Batching policy is equally applied to EDF and LSF producing BEDF and BLSF algorithms, respectively. Therefore, by justifying EDF outperformance on LSF, we can also explain the BEDF outperformance on BLSF algorithm. As this set of experiment is performed under a normal load, based on the above studies, it is reasonably expected that EDF will perform equal to or better than LSF; thus BEDF can not perform worse than BLSF.
Missed Deadline Ratio for BEDF and BLSF

![Comparison of BEDF and BLSF Algorithms](https://example.com/figure4-3.png)

**Figure 4-3: Comparison of BEDF and BLSF Algorithms**

However, Figure 4-3 implies a better performance for BEDF. We justify this argument with the overhead occurring in BLSF by computing the estimated execution time. Therefore, we can see that BEDF outperforms BLSF overall.

### 4-3 Batch Size Effect in Batching Algorithms

In a batching algorithm, a set of tasks is scheduled, while succeeding newly arrived tasks are held, forming a batch. Upon execution of all the tasks in the first set, the tasks in the batch will be sent for scheduling and execution. In the following section, we explore the effect of batch size, arguing that the number of tasks in a batch effects the performance of the batching algorithm; thus, limiting the number of tasks in the batch improves algorithm performance.
4-3-1 Experimental Evaluation

In the following set of experiments, we examine the algorithms with fixed batch size. Similar to the previous algorithm, it forms a waiting set of arrived tasks to which the scheduling policy is applied. This set is queued in order of arrival. Upon completion of Batch(j-1), the next batch is formed. Despite the previous method, only the first n arrived tasks in the set form Batch(j). While Batch(j) is scheduled and executed, the newly arriving tasks append to the end of the waiting set. Therefore, we always schedule a limited number of a task per batch.

We have chosen four different batch sizes: 100, 300, 500 and 700. We attach this number to the algorithm name in order to indicate its batch size. For example in BEDF100 a maximum of only 100 tasks will form a batch and will be scheduled at the same time. Note that throughout these experiments the arrival rate \( \lambda \) [Section 3-2-2] is set to one.

The experiments are organized as follows:

- In the first set of experiments, we compare the performance of BEDF algorithms with different batch sizes. The goal in this set of experiments is to see the effect of batch size on the candidate algorithms.

- In the last set of experiments, we compare the performance of BLSF algorithms with different batch sizes. The goal in this set of experiments is to see the effect of batch size on the candidate algorithms.

4-3-1-1 Effect of Batch size on BEDF

Figure 4-4 shows the performance of BEDF with different batch sizes. The candidate algorithms are BEDF700, BEDF500, BEDF300 and BEDF100. The algorithm with a
smaller batch size outperforms the algorithm with a higher batch size over the down range of slack factor values, up to slack value of four. As shown in Figure 4-4, the batch can hold a high percentage of total tasks in one phase; e.g. in BEDF700 holds a minimum of 70% of the tasks in one phase. In this situation, the overhead of scheduling the complete batch is almost the same as scheduling tasks using non-batching algorithms. Therefore a small batch size, or less than the number of tasks scheduled in one phase, will decrease the overhead in the phase. As the overhead lessens, the algorithm performance increases.

**Figure 4-4: Effect of Batch Size on BEDF**

**4-3-1-2 Effect of Batch size on BLSF**

Figure 4-5 shows the performance of BLSF with different batch sizes. The candidate algorithms are BLSF700, BLSF500, BLSF300 and BLSF100. The algorithm with a smaller batch size outperforms the algorithm with a higher batch size over a wider range
of slack factor values, up to slack value of seven. The same reasoning as used in section 4-3-1-1 applies to the improvement of performance in smaller batch sizes.

![Effect of batch size on BLSF](image)

**Figure 4-5: Effect of Batch Size on BLSF**

As presented in Figure 4-5, the batch can hold a high percentage of total tasks at one phase (minimum 70% of total tasks in a batch for BLSF700). In this situation, the scheduling overhead of the complete batch is almost the same as scheduling tasks using non-batching algorithms. Therefore, restricting the batch size, or the number of tasks scheduled at one phase will decrease the overhead in the phase. Consequently, the algorithm performance increases when the batch size decreases.

### 4-3-1-3 A Lower Boundary on Batch Size

As presented in the Figures 4-4 and 4-5, minimizing the batch size improves the *missed deadline ratio* in the algorithms. For both BEDF and BLSF algorithms, the smallest batch
size value was set to 100 in the experiments. In order to obtain a lower boundary on the batch size, we have continued the experiments with batch sizes less than 100. The candidate BEDF algorithms for the lower boundary evaluations are BEDF50, BEDF10, BEDF5 and BEDF1. The candidate BLSF algorithms are chosen with the same batch size values as their BEDF counterparts, BLSF50, BLSF10, BLSF5 and BLSF1. As it is presented in Table 4-2, all BEDF algorithms with batch sizes equal to or lower than 100 have very close missed deadline ratio for all ranges of slack factor.

<table>
<thead>
<tr>
<th>SF</th>
<th>BEDF1</th>
<th>BEDF5</th>
<th>BEDF10</th>
<th>BEDF50</th>
<th>BEDF100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>93.2</td>
<td>93.1</td>
<td>93.1</td>
<td>93.2</td>
<td>93.4</td>
</tr>
<tr>
<td>2</td>
<td>62.9</td>
<td>62.9</td>
<td>63.2</td>
<td>63.2</td>
<td>63.6</td>
</tr>
<tr>
<td>3</td>
<td>44.1</td>
<td>44.1</td>
<td>44.2</td>
<td>44.1</td>
<td>44.0</td>
</tr>
<tr>
<td>4</td>
<td>26.9</td>
<td>26.6</td>
<td>26.6</td>
<td>26.3</td>
<td>26.5</td>
</tr>
<tr>
<td>5</td>
<td>22.7</td>
<td>22.7</td>
<td>22.6</td>
<td>22.7</td>
<td>22.4</td>
</tr>
<tr>
<td>6</td>
<td>17.4</td>
<td>17.0</td>
<td>17.1</td>
<td>17.0</td>
<td>17.2</td>
</tr>
<tr>
<td>7</td>
<td>15.2</td>
<td>15.0</td>
<td>15.0</td>
<td>15.1</td>
<td>15.1</td>
</tr>
<tr>
<td>8</td>
<td>11.4</td>
<td>11.3</td>
<td>11.2</td>
<td>11.1</td>
<td>11.1</td>
</tr>
<tr>
<td>9</td>
<td>10.5</td>
<td>10.5</td>
<td>10.5</td>
<td>10.4</td>
<td>10.4</td>
</tr>
<tr>
<td>10</td>
<td>9.5</td>
<td>9.5</td>
<td>9.3</td>
<td>9.4</td>
<td>9.4</td>
</tr>
</tbody>
</table>

*Table 4-2: Percentage of Missed Deadlines for BEDF algorithms, NTASKS 1000*

The same behavior is observed in Table 4-3. The BLSF algorithms have close missed deadline ratio values for all slack factors, from 1 to 10. However, in both tables it can be observed that batch size of one is not necessarily the best, because of the unnecessary batching overhead. Other batches of reasonable sizes allow comparison and prioritization of tasks while keeping scheduling overhead small.
<table>
<thead>
<tr>
<th>SF</th>
<th>BLSF 1</th>
<th>BLSF 5</th>
<th>BLSF 10</th>
<th>BLSF50</th>
<th>BLSF100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>79.8</td>
<td>79.9</td>
<td>80.4</td>
<td>80.8</td>
<td>81.0</td>
</tr>
<tr>
<td>2</td>
<td>45.9</td>
<td>46.0</td>
<td>46.1</td>
<td>46.5</td>
<td>46.7</td>
</tr>
<tr>
<td>3</td>
<td>34.3</td>
<td>34.3</td>
<td>34.5</td>
<td>34.6</td>
<td>34.8</td>
</tr>
<tr>
<td>4</td>
<td>22.0</td>
<td>22.0</td>
<td>22.2</td>
<td>22.2</td>
<td>22.1</td>
</tr>
<tr>
<td>5</td>
<td>20.0</td>
<td>20.0</td>
<td>20.1</td>
<td>20.1</td>
<td>20.1</td>
</tr>
<tr>
<td>6</td>
<td>15.3</td>
<td>15.3</td>
<td>15.2</td>
<td>15.2</td>
<td>15.2</td>
</tr>
<tr>
<td>7</td>
<td>13.7</td>
<td>13.6</td>
<td>13.7</td>
<td>13.7</td>
<td>13.7</td>
</tr>
<tr>
<td>8</td>
<td>10.7</td>
<td>10.6</td>
<td>10.8</td>
<td>10.6</td>
<td>10.5</td>
</tr>
<tr>
<td>9</td>
<td>9.8</td>
<td>9.8</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
</tr>
<tr>
<td>10</td>
<td>8.6</td>
<td>8.6</td>
<td>8.6</td>
<td>8.6</td>
<td>8.6</td>
</tr>
</tbody>
</table>

*Table 4-3: Percentage of Missed Deadlines for BLSF algorithms, NTASKS 1000*

4-4 Summary

In this chapter, we have proposed two dynamic scheduling algorithms, *Batching Earliest Deadline First (BEDF)* and *Batching Least Slack First (BLSF)*, which schedule a set of aperiodic, non-preemptive real-time tasks. BEDF/BLSF schedule and execute tasks in batches, in order to maintain a low level of overhead associated with scheduling and task prioritization. The algorithms were designed to address trade-off, in dynamic scheduling between the frequency at which the scheduling process is invoked, the number of tasks it is applied to at each invocation, and the quality of the resulting schedules in terms of deadline compliance. Different batch size versions of the algorithms were described and their performance trade-off was investigated. The results of the experiments show an improved performance for the batching techniques under tighter deadlines..
Chapter 5

Batching Algorithms in Overload Conditions

In this chapter we investigate

- the behavioral change of batching algorithms in overload conditions in comparison to normal load,
- the effect of context switching overhead in overload conditions,
- two overload management policies and the quality of the resulting schedules in terms of deadline compliance.

We identify the behavioral changes of batching algorithms under heavy load systems. Therefore, we attempt to improve the performance of the two algorithms, BEDF and BLSF by applying overload management policies. We propose the Task Fusion (TF) policy in order to avoid context switch overhead. The second policy applied to the algorithms is a Not Tardy (NT) overload management policy, through which we improve the EDF-based algorithms.

5-1 Introduction

Real-time scheduling algorithms are used to schedule a wide range of applications. Embedded process-control systems, manufacturing production-control systems, aviation command and control systems and network management and stock market real-time database systems are examples of real-time systems. Some of the above systems, such as
a stock market real-time database, require performing under *heavy load* system conditions. One important aspect in obtaining the satisfactory performance for a real-time system is to employ the proper scheduling algorithms. Under different system loads, a scheduling algorithm behavior can change. Thus, it is important to study a scheduling algorithm not only under normal load, but also in *overload* conditions.

As stated in Chapter 2, a real-time system is a system to respond to frequent environmental *input* within a time interval (*deadline*) dictated by the environment. In a real-time system an overload condition can occur as a result of

- hardware error conditions like bus retries [15],
- software exception handling conditions [15],
- high frequency system input with relatively small deadlines,
- long execution times.

In the first section of this chapter, we examine the batching algorithms, BEDF and BLSF, under heavy load situations. BEDF algorithms outperform BLSF under normal load. The evaluation results in this chapter confirm a change of behavior in BEDF algorithms. The change of behavior is due to the core algorithm, EDF, employed in this strategy. EDF gives the highest priority to tasks with deadlines that are too tight to be met, thus wasting the CPU time for other eligible tasks. We shall provide a more detailed study on this aspect in Section 5-2.

As stated earlier, we have used RTLinux, for its fast, small, proprietary kernel with fast context switches. This property gives us a big performance advantage. However, this does not suffice for our system requirements under heavy load conditions. We attempt to reduce this overhead by presenting the TF policy in Section 5-3. In BEDF
and BLSF, a context switch is performed between each task in a batch. In the TF policy, the tasks in a batch are united as one task, therefore there is no context switching between them. There is only context switch per batch.

In the last section of this chapter, we examine a major weakness in EDF based algorithms. The EDF policy can assign the highest priority to a task that already has missed or is about to miss its deadline [10]. The NT policy is therefore suggested [10] to overcome this weakness. We shall perform an experimental evaluation on this policy.

5-1-1 Experiment Outline

The experiments in this chapter represent the missed deadline ratio over the growth of the load factor. Load Factor represents the number of queries in the system in the scale of 0.001 and is determined by the arrival rate $\lambda$ in Query Environment as described in Section 3-2-2. For example, in a system with $\lambda = 2$, the load factor is two and the number of queries in the system is 2000. The load factor ranges from 1 to 10. Larger values of $\lambda$ represent a larger load factor, therefore a larger number of tasks in the system. Note that the batch size is set to 100 for all candidate algorithms in this chapter.

These experiments represent comparisons of different combinations of batching algorithms when used with overload management policies. Table 5-1 summarizes the scheduling methods used in this chapter. We use format BEDF/NT to denote the algorithm formed by combining the BEDF and NT policies. Other algorithms are denoted similarly.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEDF</td>
<td>Batching Earliest Deadline First</td>
</tr>
<tr>
<td>BLSF</td>
<td>Batching Least Slack First</td>
</tr>
<tr>
<td>TF</td>
<td>Task Fusion policy</td>
</tr>
<tr>
<td>NT</td>
<td>Not Tardy policy</td>
</tr>
</tbody>
</table>

*Table 5-1: List of Scheduling Policies*

The experiments are organized as follows:

- In one set of experiments, in Section 5-2, we compare the performance of the BEDF and BLSF algorithm. Our goal is to investigate on BEDF behavioral change in an overloaded system.
- In three set of experiments, in Section 5-3, we compare the performance of BEDF and BLSF algorithm with BEDF/TF and BLSF/TF. Our goal is to investigate the effect of TF policy.
- In the last set of experiments, in Section 5-4, we investigate the effect of NT on EDF-based algorithms, BEDF, and BEDF/TF.

### 5-2 Batching Algorithms under Overload Conditions

In this set of experiments, we compare the performance of the BLSF algorithm with the BEDF algorithm in overload conditions. Our goal is to reveal the situations under which the algorithm performs well.

Figure 5-1 shows the performance of search-based BEDF and BLSF algorithms under overload conditions. In these conditions, BLSF outperforms BEDF as the number of tasks grows in each run of the experiment.
Batching policy is equally applied to EDF and LSF producing BEDF and BLSF algorithms, respectively. Therefore, by justifying LSF superior performance on EDF in overload conditions, we can also explain the BLSF outperformance on BEDF algorithm.

Several previous studies [10][11][15][24] have addressed the issue of scheduling with the objective of minimizing the number of late tasks. A common observation of these studies has been that assigning priorities to tasks according to an EDF Scheduling [25] minimizes the number of late tasks when under low or moderate levels of resource and data contention. These studies have also observed, however, that the performance of EDF sharply degrades in an overloaded system. This is because under heavy loading tasks gain high priority only when they near deadlines. Gaining high priority at this late stage may not leave sufficient time for tasks to complete before their deadlines. Under heavy loads then, a fundamental weakness of EDF is that it assigns the highest priority to tasks that are too close to missing their deadlines, thus delaying other tasks that might still be able to meet their deadlines. As EDF is the core scheduling strategy used in BEDF, we can easily justify why BEDF performs poorly in comparison to BLSF.
Behavioral Change of Batching Algorithms in Overload Conditions

![Graph showing behavioral change of batching algorithms in overload conditions]

Figure 5-1: Behavior of Batching Algorithms in Overloaded Conditions

5-3 Context Switching Overhead in Real-time Systems

An important issue in a real-time system is determinism. The goal of real-time systems engineering is to assess whether or not a system will meet its deadlines. The more deterministically the system behaves, the more easily analyzed it is [15]. In a real-time system, long interrupt handling or frequent context switches between tasks can make a system non-deterministic and unpredictable, thus hard to schedule. Therefore, context switching and interrupt servicing are necessary components to consider in task management and execution [9][27]. In order to reduce these overheads we have chosen

- a fast, small, proprietary kernel real-time-operating system [13]
- non-preemptive scheduling.

The real-time operating system chosen for the purpose of our thesis is RTLinux.
The RTLinux operating system has the properties of a fast, small, proprietary kernel operating system. Fast context-switching, small kernel size, quick response to external interrupts and bounded execution time for most system primitives are a list of these properties. Through the interrupt emulation technique used in the design of RTLinux, the real-time interrupt handling time becomes optimal [19][20]. The designers of RTLinux have overcome the overhead of context switching by using one address space for all real-time tasks. By using the kernel address space, the overhead of protection level changes through context switching is eliminated.

The second aspect that we consider in this thesis in order to reduce the context switch overhead is the choice of non-preemptive scheduling. Non-preemptive scheduling on a uni-processor is important for a variety of reasons [8]:

- Non-preemptive scheduling algorithms are easier to implement than preemptive algorithms and can exhibit dramatically lower overhead at runtime.

- Due to the higher number of context switches and task invocations, the overhead of preemptive algorithms is more difficult to characterize and predict than that of non-preemptive algorithms.

- Non-preemptive scheduling on a uni-processor naturally guarantees exclusive access to shared resources and data, thus eliminating both the need for synchronization and its associated overhead.

Although the efforts have given desirable results in normal load conditions, they will not suffice for scheduling under overload conditions. Therefore, we propose the Task
Fusion policy to reduce the frequency of context switching and task invocations. Following that, we will also evaluate our proposal with an experimental approach.

5-3-1 Task Fusion Policy

Under this policy, we use an integrated task execution mechanism in which a group of tasks is executed as a single process. By employing this method, fewer context switches are performed, thus less overhead occurs in the system. This strategy is not an add-on policy. That is, in order to apply this strategy to a scheduling algorithm, one should redesign and re-implement the algorithm with the TF consideration. Therefore, the BEDF/TF and BLSF/TF versions of the batching algorithms have been re-implemented using TF guidelines.

In order to explain this method clearly, we first give a review of the steps of the batching algorithms, BEDF and BLSF, to which this policy will be applied. The sequence of actions taken by a batching algorithm in the IRQP platform is reviewed below:

**Batching algorithm with out a TF policy :**

- **Query Reception:** Receive a query from the query generation unit;
- **Buffering:** Insert the query in a buffer, wait for the scheduler;
- **Query Transfer:** If the scheduler is done with the previous batch,  
  *Tasks structure association:* Associate a real-time task structure to each query in the buffer. This generates a batch of real-time tasks;
- In the scheduler:
  - **Scheduling:** Select a task from the batch by applying a scheduling policy (EDF or LSF) to the batch;
• **Execution**: *Context switch* to the selected task and execute it to completion;

• If the batch is empty, go back to the *Query Transfer*. Otherwise, go back to *Scheduling*.

As presented in the above scheme, one context switch is performed after every task execution, in the *execution* step of the algorithm. In the TF policy, there is only one context switch per batch. The following are the steps of the TF policy in IRQP system:

**Batching algorithm with TF policy :**

- **Query Reception**: Receive a query from the query generation unit;

- **Buffering**: Insert the query in a *buffer*, wait for the scheduler;

- **Query transfer**: If the scheduler is done with the previous batch;

  *Tasks structure association*: **Associate a single real-time task** to all the queries in the *buffer*. This generates a **single task** with a *local task buffer* full of query data as local input;

- In the scheduler:

  - **Scheduling**: Select the **single task** for execution;

  - **Execution**: *Context switch* to the selected single task and execute it to completion;

  - In execution of the single task:

    - **Task Selection**: Select a query from the *local task buffer* by applying a scheduling policy (EDF or LSF) to the buffer;
If the buffer is empty, go back to *Query Reception*. Otherwise, repeat the *Task Selection*.

In this scheme, we fuse the queries into one task, instead of associating each query with an individual task. The queries in the batch form and run as a single task; therefore, the number of context switches reduces to one per batch upon scheduler invocation.

This strategy cannot be applied to preemptive algorithms. This is clear, since any context switch among tasks in a batch is omitted. As our applied algorithms, BEDF and BLSF, are non-preemptive, we do not encounter this problem. Thus BEDF and BLSF are easily redesigned in using TF policy.

In the following sections, we evaluate the TF policy through a number of performance-comparison experiments.

### 5-3-1-1 Comparison of BEDF and BEDF/TF Algorithms

Figure 5-2 shows the performance of BEDF and BEDF/TF algorithms in overload conditions. As expected, both algorithms miss a greater number of deadlines as the load increases. BEDF/TF outperforms BEDF overall as the task load grows. Under the load factor of 1, or 1000 tasks, BEDF/TF shows no performance difference to BEDF. In the load factor of 10, or 10000 tasks, BEDF/TF shows the most difference in performance.
Effect of TF policy on BEDF

Figure 5-2: Comparison of BEDF and BEDF/TF Algorithms

compared to BEDF. We can easily observe the effect that TF policy has on the BEDF scheduling as the load grows. As we explained earlier in the Section 5-3-1, the TF policy reduces the number of context switches, thus decreasing the overhead. Therefore, the TF policy effectively reduces the missed deadline ratio in an overloaded system.

5-3-1-2 Comparison of BLSF and BLSF/TF Algorithms

Figure 5-3 shows the performance of BLSF and BLSF/TF algorithms in overload conditions. As expected, both algorithms miss a greater number of deadlines as the load increases. Overall, the same behavior is observed in this set of experiments as that exhibited by BEDF algorithms. BLSF/TF outperforms BLSF on a wide range of load growth. While in the load factor of 1, or 1000 tasks, BLSF/TF shows a slightly lower performance in comparison to BLSF. In load factor of 10, or 10000 tasks, BLSF/TF shows the most difference in performance to BLSF. The same reasoning used in the
previous section applies to this comparison. We can easily observe the effect that TF policy has on the BLSF scheduling as the load grows. As we explained earlier in the Section 5-3-1, the TF policy reduces the number of context switches, thus decreases the overhead. Therefore applying TF policy to BLSF effectively reduces the missed deadline ratio on an overloaded system.

![Effect of TF policy on BLSF](image)

**Figure 5-3: Comparison of BLSF and BLSF/TF Algorithms**

5-3-1-3 Comparison of BEDF/TF and BLSF/TF Algorithms

Figure 5-4 shows the performance of BEDF/TF and BLSF/TF algorithms in overload conditions. As expected, both algorithms miss a greater number of deadlines as the load increases. We can compare the behavior shown in this graph with that observed in the set of experiments on Figure 5-1, which compares pure BEDF to BLSF. In Figure 5-4, both algorithms perform better compared to their pure counterparts with no TF. We can also observe that BLSF still outperforms BEDF as the load increases, although with a less
difference. In Figure 5-1, the performance difference between BEDF and BLSF in overload conditions at a load factor of 10 is 14%. This difference is reduced to 5% under the same load factor as in Figure 5-4. As stated earlier, in heavier loads, BEDF wastes CPU time by scheduling already overdue tasks. TF policy, however, reduces the number of context switches, thus decreasing the overhead. This compensates for some of the time wasted by BEDF. Therefore, by applying TF policy to BEDF we can reduce its performance difference against BLSF in overload conditions.

![Comparison of TF effect on BEDF and BLSF](image)

*Figure 5-4: Comparison of BEDF/TF and BLSF/TF Algorithms*

### 5-4 Not Tardy Overload Management Policy

In the previous section, we observed the effect of TF policy on the performance of batching algorithms under overload conditions. As the system load increased both
BEDF/TF and BLSF/TF showed improvement. As shown in Section 5-3-1-3, however, BEDF/TF still performs poorly at higher loads in comparison to BLSF/TF. Obviously, the TF strategy has not yet completely overcome the major weakness of EDF-based algorithms such as BEDF in overload conditions. As previously stated, EDF assigns the highest priority to a task that already has missed or is about to miss its deadline. By assigning a high priority and system resources to a task that will miss its deadline anyway, we deny resources to tasks that still have a chance to meet their deadlines and cause them to be late as well.

*Not Tardy overload management* (NT) policy is suggested in [10]. In this policy, tasks that have missed their deadlines are aborted. Tasks that currently are not tardy remain eligible for execution.

By applying the NT policy to the EDF-based scheduling algorithms, BEDF and BEDF/TF, we can solve the major weakness pointed out earlier. This policy screens out the tasks that have missed or are about to miss their deadlines. We can screen out the tasks that have missed their deadlines, by comparing theirs deadline with the *current system time* before scheduling. If the deadline has already passed, the task will be aborted. In order to monitor the tasks with imminent deadlines, the *estimated execution time* of the tasks is required. This can be computed either before system start-up or at runtime. We have determined this value based on the average run-time of several task executions before system start-up.

Despite TF, the NT policy can be applied to a preemptive algorithm as well. Another advantage of this policy is its ease of implementation. NT policy can simply be
applied to the scheduling algorithm as an add-on policy at each scheduling invocation. Therefore, no major change of design and implementation is needed.

In the following section, the effects of applying NT policy to batching algorithms and their TF versions are examined using an experimental approach.

5-4-1 Experimental Evaluation on NT Policy

In this experiment, we compare the performance of a BLSF/TF algorithm and the BEDF/TF/NT algorithm, which applies the NT policy to the BEDF/TF scheduling algorithm. Our goal is to reveal the situations under which a BEDF/TF/NT algorithm performs well.

Figure 5-5 shows the performance of BEDF/TF/NT and BLSF/TF algorithms in overload conditions. As shown in Figure 5-4, BLSF/TF outperforms BEDF/TF in overload conditions. This is due to BEDF/TF assigning priority to already late deadlines,
as do other EDF-based algorithms. Therefore, we apply NT to BEDF/TF to overcome this weakness. Note that this policy is already integrated in LSF based algorithms as tasks with negative slack are aborted. Therefore, the comparison of BEDF/TF/NT with BLSF/TF is on an even basis. As shown in Figure 5-5, BEDF/TF/NT outperforms BLSF/TF throughout the load growth range in the experiments. It is obvious that an NT policy can overcome the weakness of EDF-based algorithms in overloaded systems.

5-5 Summary

In this chapter, we have studied the performance of batching algorithms in overload conditions. Figure 5-6 summarizes the evaluation we have done throughout this chapter.

Despite its performance under normal load, experiments show that BEDF performs poorly in comparison to BLSF under overload conditions. This is due to the major EDF-based algorithm weakness, which gives the highest priority to already late tasks. We have proposed Task Fusion policy in order to increase the performance of batching algorithms under heavy load. This policy reduces the context switch overhead by amalgamating all tasks in the batch into one single task. Both BEDF/TF and BLSF/TF show a better performance in comparison to their non-TF counter parts. Figure 5-6 also shows that BEDF/TF still suffers from the weakness mentioned above. Therefore, the Not Tardy overload management policy is applied to BEDF/TF and BEDF algorithms. Using this policy, the missed or tight deadlines can be screened out. Because of this policy BEDF/TF/NT overcomes its weakness and outperforms all the algorithms, including BLSF/TF, under heavy load conditions.
Summary of performance in overload conditions

Figure 5-6: Summary
Chapter 6

Final Words

To conclude this thesis, a summary of the research and results, as well as some suggestions for possible future work in this area are outlined.

6-1 Summary

In this thesis, we have studied the trade-off between the frequency of scheduling-algorithm invocations and the quality of resulting schedules. We proposed *Batching* scheduling, *Batching Earliest Deadline First* and *Batching Least Slack First*, to dynamically prioritize the execution of a set of aperiodic, non-preemptable, real-time tasks on a uni-processor architecture. The effect of *batch-size* on the performance of batching scheduling was studied in this thesis. We also studied the behavior of batching algorithms in *overload* conditions. As the load increases, the performance decreases, therefore we suggested *Task Fusion* to improve performance. Finally, we observed the positive effect of applying *Not Tardy* overload policy to BEDF algorithm for additional performance improvement. Throughout this thesis, we have evaluated the different scheduling strategies and policies through implementation and experimentation.

We first proposed the batching scheduling algorithms, BEDF and BLSF, in which the EDF or LSF policies are applied to a *batch* of tasks, respectively. While the scheduling policy selects and executes task from the current batch, the newly arrived
tasks form the batch for the next phase of scheduling. As the scheduling policy is applied to a shrinking batch in most batching algorithms, the scheduling overhead is minimized. Therefore, the BEDF and BLSF showed a better performance compared to their non-batching counterparts.

The number of tasks scheduled in a batch affects the scheduling performance. We therefore suggested scheduling a limited number of tasks in each scheduling phase. The newly arrived tasks will append to the remainder of tasks in the current batch. Through experiments, we have shown that a batching algorithm with smaller batch-sizes outperforms batching algorithms with bigger batch-sizes. The experiments also showed that there is an optimal lower bound on the batch-size.

We have evaluated the batching algorithms under different system loads, normal and heavy. In a series of experiments, we evaluated the batching algorithms over an increasing task load. Experiments show that both BEDF and BLSF miss more deadlines as the load increases. However, it was observed that BLSF out-performs BEDF under heavy load.

We have studied the effect of context-switch overhead on batching algorithms in overload conditions and proposed the TF policy to reduce this overhead. The TF policy recommends uniting all the tasks in a batch into one. This mechanism reduces the number of context switches to only one per batch. Results show a performance improvement by batching algorithms under heavy loads using TF policy.

Examination revealed that the batching EDF-based algorithm, BEDF, performs poorly in comparison to BLSF under heavy load conditions. Our studies explained that this is due to the EDF major weakness, where it assigns high priority to already late tasks.
We have applied and evaluated NT overload management policy to the EDF-based algorithm to overcome this weakness. The NT policy screens out the late tasks before they are executed. Results demonstrate the positive effect of NT management on EDF based algorithms under heavy load conditions. It was also revealed that a batching EDF algorithm combined with TF and NT policies has the best performance in overload conditions in comparison to the rest of its LSF-based counterparts.

The construction of the experimental platform for this thesis was an invaluable technical experiment. The experimental platform, a real-time main memory database system, also referred to as In-memory Real-time Query Processor, was composed of a real-time operating system, real-time application and a query generator. A client-server model was employed for query generation. In this model queries are generated in a client. We have modified the network device driver to receive queries in real-time with less overhead. We have also modified the underlying operating system, RTLinux to add more functionality. Building the real-time application, a query processor, was also a part of this invaluable experiment.

6-2 Future Work

Our studies can be further extended in the following areas:

- Batch size affects the number of deadlines missed in a real-time system using a batching algorithm. A mechanism for batch-size control is needed to determine the batch-size at which the batching algorithm performs the best.

- In our thesis, the batching approach is applied to non-preemptive EDF and LSF scheduling algorithms. Preemption complicates the scheduling problem by increasing
the unpredictability factor. An interesting study is the performance evaluation of batching approach on preemptive scheduling algorithms

- EDF and LSF algorithms are both simple algorithms, with minimal complexity. The effect of batching approach on a complex scheduling algorithm can be studied.

- The experimental platform can be expanded to a multi-client platform. In the existing platform, a single client inquired the server; thus, the overhead of service in a multi-user system is minimized.

- The IRQP employed in this thesis is a read-only database. Neither disk read/write nor any other source contention is required in this application. As the application is simple, the overhead is minimal. A more complex application with resource contention challenges us with other issues such as priority inversion.
References


