Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

LEAST SPACE-TIME FIRST SCHEDULING ALGORITHM: SCHEDULING COMPLEX TASKS WITH HARD DEADLINE ON PARALLEL MACHINES

by
Bo-Chao Cheng

Both time constraints and logical correctness are essential to real-time systems and failure to specify and observe a time constraint may result in disaster. Two orthogonal issues arise in the design and analysis of real-time systems: one is the specification of the system, and the semantic model describing the properties of real-time programs; the other is the scheduling and allocation of resources that may be shared by real-time program modules.

The problem of scheduling tasks with precedence and timing constraints onto a set of processors in a way that minimizes maximum tardiness is here considered. A new scheduling heuristic, Least Space Time First (LSTF), is proposed for this NP-Complete problem. Basic properties of LSTF are explored; for example, it is shown that (1) LSTF dominates Earliest-Deadline-First (EDF) for scheduling a set of tasks on a single processor (i.e., if a set of tasks are schedulable under EDF, they are also schedulable under LSTF); and (2) LSTF is more effective than EDF for scheduling a set of independent simple tasks on multiple processors.

Within an idealized framework, theoretical bounds on maximum tardiness for scheduling algorithms in general, and tighter bounds for LSTF in particular, are proven for worst case behavior. Furthermore, simulation benchmarks are developed, comparing the performance of LSTF with other scheduling disciplines for average case behavior.

Several techniques are introduced to integrate overhead (for example, scheduler and context switch) and more realistic assumptions (such as inter-processor commu-

nication cost) in various execution models. A workload generator and symbolic simulator have been implemented for comparing the performance of LSTF (and a variant — LSTF$^+$) with that of several standard scheduling algorithms.

LSTF's execution model, basic theories, and overhead considerations have been defined and developed. Based upon the evidence, it is proposed that LSTF is a good and practical scheduling algorithm for building predictable, analyzable, and reliable complex real-time systems.

There remain some open issues to be explored, such as relaxing some current restrictions, discovering more properties and theorems of LSTF under different models, etc. We strongly believe that LSTF can be a practical scheduling algorithm in the near future.

LEAST SPACE-TIME FIRST SCHEDULING ALGORITHM: SCHEDULING COMPLEX TASKS WITH HARD DEADLINE ON PARALLEL MACHINES

by
Bo-Chao Cheng

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Computer and Information Science

January 1997

APPROVAL PAGE
(1 of 2)

LEAST SPACE-TIME FIRST SCHEDULING ALGORITHM: SCHEDULING COMPLEX TASKS WITH HARD DEADLINE ON PARALLEL MACHINES

Bo-Chao Cheng

Dr. Alexander D. Stoyenko, Dissertation Advisor — Date
Director of Real-Time Computing Laboratory
Associate Professor of Computer and Information Science,
New Jersey Institute of Technology, Newark, New Jersey

Dr. Sanjoy Baruah , Committee Member — Date
Assistant Professor of Computer Science,
University of Vermont, Burlington, Vermont

Dr. Tadao Ichikawa, Committee Member — Date
Chair of Graduate School of Information Engineering,
Hiroshima University, Hiroshima, Japan

Dr. Phillip A. Laplante, Committee Member — Date
Dean of Engineering and Technology,
Burlington County College-NJIT, Mount Laurel, New Jersey

Dr. C. L. Liu, Committee Member — Date
Professor of Computer Science,
University of Illinois at Urbana-Champaign, Illinois

APPROVAL PAGE
(2 of 2)

LEAST SPACE-TIME FIRST SCHEDULING ALGORITHM: SCHEDULING COMPLEX TASKS WITH HARD DEADLINE ON PARALLEL MACHINES

Bo-Chao Cheng

Dr. Thomas J. Marlowe, Committee Member — Date
Professor of Mathematics and Computer Science,
Seton Hall University, South Orange, New Jersey

Dr. James A. McHugh, Committee Member — Date
Associate Chairperson and Professor of Computer and Information Science,
New Jersey Institute of Technology, Newark, New Jersey

Dr. Peter A. Ng, Committee Member — Date
Chairperson and Professor of Computer and Information Science,
New Jersey Institute of Technology, Newark, New Jersey

Dr. Lui Sha, Committee Member — Date
Senior Member of Technical Staff,
Software Engineering Institute,
Carnegie Mellon University, Pittsburgh, Pennsylvania

BIOGRAPHICAL SKETCH

Author: Bo-Chao Cheng

Degree: Doctor of Philosophy

Date: January 1997

Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1997
- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1989
- Bachelor of Science in Chemical Engineering,
  National Central University, Chung-Li, Taiwan, 1984

Major: Computer and Information Science

Publications:

B.-C. Cheng, A. D. Stoyenko, T.J. Marlowe, and S. Baruah, "LSTF: A New Scheduling Policy for Complex Real-Time Tasks in Multiple Processor Systems," *Automatica*, vol. 33, no. 5, pp. 921–926, 1997.

T.-H. Wu, I. Korpeoglu, B.-C. Cheng, "Distributed Interactive Video System Design and Analysis," *IEEE Communications Magazine*, vol. 35, no. 3, pp. 100–108, 1997.

B.-C. Cheng, A.D. Stoyenko, T.J. Marlowe, and S. Baruah, "An Experiment With Real-Time Channel Establishment for Video-on-Demand Service," *The Fifth International Conference on Computer Communications and Networks*, Rockville, Maryland, pp. 358–363, October, 1996.

B.-C. Cheng, A.D. Stoyenko, T.J. Marlowe, and S. Baruah, "The Allocation and Scheduling of Precedence- and Timing-Constrained Tasks with Communication Delays," *Proceedings Second IEEE International Conference on Engineering of Complex Computer Systems*, Quebec, Canada, pp. 91–94, October, 1996.

A.D. Stoyenko, L.R. Welch, and B.-C. Cheng, "Response Time Prediction in Object-Based, Parallel Embedded Systems," *Euromicro*, vol. 40, pp. 135–150, 1994.

This work is dedicated to
my family

ACKNOWLEDGMENT

I would like to express my appreciation to my advisor, Professor Alexander D. Stoyenko, for his outstanding assistance and support throughout this whole year. Also, I would like to thank Professors Thomas J. Marlowe and Sanjoy Baruah for their guidance.

Special thanks are given to Dr. Tadao Ichikawa, Dr. Phillip A. Laplante, Dr. C. L. Liu, Dr. James A. McHugh, Dr. Peter A. Ng and Dr. Lui Sha for actively participating in my committee. The members of Real-Time Computing Laboratory (RTCL) are deserving for their help and assistance.

Finally, I want to thank my family members. Only they continuously gave me support during my most difficult days. Without their support and encouragement, my accomplishment would not be possible.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

Figure Page

CHAPTER 1

INTRODUCTION

1.1 Motivation

A number of real-time scheduling algorithms have been developed; these algorithms can be categorized into *static/dynamic* and *on-line/off-line* scheduling. When the task priority does not change during its execution, we term this *static* scheduling; otherwise, we have *dynamic* scheduling. For example, the rate monotonic scheduling algorithm (RMS) [50] is a well-known static scheduling policy, and earliest deadline first (EDF) [23] is the best-known dynamic one. Scheduling of tasks at compile time is termed *off-line scheduling*; whereas *on-line policy* makes a decision after receiving requests.

We label tasks with only a single resource computation requirement, and without any synchronization or parallelism (other than trivial inter-task parallelism), '*simple*', and all other tasks '*complex*' or '*non-simple*'. We likewise characterize computer systems as 'single processor' or 'multiple processor'. Many of the applications and research efforts found in existing scheduling disciplines are restricted to "*Simple Tasks on Single Machine*" [54, 63, 64], "*Simple Tasks on Multiple Machines*" [26] or "*Complex Tasks on Single Machine*" [51, 60, 70]. Although some work can be found on "*Complex Tasks on Multiple Machines*", the network configurations are limited to particular topologies such as two machines [59], three machines [5] or hypercubes [56].

We propose a new scheduling algorithm, **least space-time first (LSTF)**, to deal with "Complex Tasks on Multiple Machines" for an arbitrary topology. LSTF makes scheduling decisions based on a combination of precedence and real-time constraints.

1.2 Context

In the past forty years, many scheduling papers have been published. Most focus on "Simple Tasks on Single Machine", "Simple Tasks on Multiple Machines", or "Complex Tasks on Single Machine". In the "Complex Tasks on Multiple Machines" category, there are no provably optimal scheduling disciplines, in the sense of meeting hard real-time deadlines.

Task scheduling also naturally divides into on-line and off-line scheduling. Off-line task scheduling occurs at compile time and it needs to have a good planning agent to evaluate interactions of tasks and resources and to make a scheduling list, based on which the dispatcher will allocate the resources to tasks [3, 14, 17, 28, 30, 34]. For example, requirement-driven scheduling (REDS) [35], an artificial-intelligence-based architecture, is composed of planning agents and has a number of advantages over the conventional hierarchical, distributed, and subsumption architectures for real-time distributed systems. Similarly, the Branch-and-Bound method is based on a depth-first solution strategy; in the solution tree, each nodes represents a resource and precedence feasible partial schedule. Branches emanating from a parent node correspond to exhaustive and minimal combination scheduling [22, 71]. The module allocation algorithm (MAA) [42] uses branch-and-bound based on objective functions embedding timing constraints. The objective functions drive MAA in assigning task modules to processing nodes and using a module scheduling algorithm to schedule all modules for meeting all task deadlines.

On-line scheduling orders the execution based on some a *priori* knowledge of the tasks allocated by assignment algorithms. Rate-monotonic-next-fit-scheduling (RMNFS) [25], an extended basic rate-monotonic-scheduling algorithm, assigns the task to the processor where it can be feasibly scheduled based on RMS; however, RMNFS does not consider communication. Agne [1] suggests that the scheduler compute local scheduling plans, one for each node that reserves time slots for

execution of software modules with hard execution deadlines, and the other for the transmission medium that reserves time slots for the transmission with hard deadline. It produces low utilization when a reserve technique is applied. A heuristic algorithm is proposed by Wang et al. [72], combining list scheduling (finding a good bound) with H scheduling (finding a feasible schedule). The algorithm performs well in both respects, but is not scalable in the number of tasks. The algorithm, scheduling real-time periodic tasks on a fault-tolerant multiprocessor system, incorporates both redundancy and masking techniques, and an imprecise computation model. A trade-off between the quality of the result and the processing time used to produce the result is provided by their imprecise computation model [74].

An on-line algorithm makes a decision immediately after receiving the requests, while an off-line algorithm makes a decision based on its planning. An off-line algorithm knows the entire sequence of requests in advance and chooses its actions optimally. The competitive ratio is defined as $\frac{Cost(on-line)}{Cost(off-line)}$, where *Cost(on-line)* is the worst-case cost of an on-line scheduling algorithm and *Cost(off-line)* is the worst-case cost for optimal off-line scheduling. Hsu et. al. [43] use competitive ratios to compare the performance of on/off line scheduling in lists, off-line static search trees, dynamic search trees, and the k-server problem.

Generally, there are advantages and disadvantages in both on and off-line scheduling [47]:

- flexibility:

  On-line scheduling is good at handling unexpected events.

- extension:

  In on-line scheduling, it is easy to add new tasks to the systems or increase the load of an original task (by increasing code, data input size, etc.).

- dynamic execution effect:

  On-line scheduling is effective in the presence of synchronization, contention for shared resources, communication between processors.

- information:

  On-line scheduling needs less information than off-line because off-line should know the entire sequence of requests, and the parameters of all tasks and resources.

- overhead:

  In off-line scheduling, there exists a dispatcher allocating the resources to tasks based on the scheduling list determined at compiling time. The dispatcher need not compute a priori information and so can spend less execution time than the scheduler.

- schedulability analyzer:

  For both scheduling algorithms, it is hard to implement a schedulability analyzer which guarantees whether tasks meet deadline.

- provability and complexity:

  Except for EDF and RMS, both on and off-line scheduling algorithms should be investigated further.

LSTF is designed as an on-line scheduling approach, because we are interested in flexible and extensible systems (such as multimedia systems). We can use a simulator to assist LSTF building a schedulability analyzer.

1.3 Problem and Approach

Evaluation of performance of real-time systems is based on satisfaction of deadline constraints. In the "Simple Tasks on Single Machine" case, RMS is an optimal

static scheduling algorithm in the term of *schedulability* or *feasibility*, and EDF is an optimal dynamic one. When a set of tasks is scheduled by an algorithm, and all tasks meet their deadline, we call this set *schedulable*. If a set of tasks is schedulable under an algorithm, it is also schedulable under an optimal scheduling algorithm. We have two observations. First, both RMS and EDF deal with a simple model which is the classic one for real-time scheduling community (Table 1.1, page 7). Second, it is difficult to use "schedulability" to compare two algorithms when both of them satisfy or both miss deadlines in the "Simple Tasks on Multiple Machines", "Complex Tasks on Single Machine", or "Complex Tasks on Multiple Machines" models. In some applications, we need information on how late the tasks will be and how much penalty each should pay if it misses its deadline. French [29] defined the *lateness* of process i, $L(i)$, as the difference between its completion time and its deadline. This implicitly suggests that if process i completes before its deadline, then $L(i)$ is negative. A process is *tardy* when it completes after its deadline, otherwise, it is *early*; the non-negative value variable $T(i)$, the *tardiness* of process i, is then defined as $T(i) = \max(L(i), 0)$. That is, when a job completes before its deadline, $T(i) = 0$, but it will pay a penalty if it completes after its deadline. Let τ_{max} define as the maximum value of $T(1), T(2), ..., T(n)$. When a set of tasks is schedulable, $\tau_{max} = 0$. A scheduling algorithm which minimizes tardiness will necessary satisfy all deadlines for any schedulable set of tasks.

Under certain conditions, EDF minimizes the lateness and the tardiness of a set of tasks with deadlines executing on a single processor [52, 64]. Previous work surveyed by Cheng [15] treats several lateness and tardiness performance criteria.

Different criteria can be used to classify different time requirement applications, to specify precise requirements on applications, and to evaluate the performance relative to requirements. For example, musical instrument digital interface (MIDI) applications need to consider the lateness of early and tardy tasks (both early and

tardy tasks should be penalized); in contrast, most real-time systems should be evaluated by a tardiness criterion only, because the system should pay small (soft-real-time-system) or huge (hard-real-time-system) penalty for tardy tasks, but will not be either penalized or rewarded for tasks which complete before deadline.

Determining minimum weighted tardiness, for preemptible and for non-preemptible jobs on both single and parallel machines, is NP-Complete [32] in each case. Some heuristic algorithms seek to minimize the total tardiness [16, 37, 40] or the mean tardiness [9]. If all jobs' weights are equal, a weighted tardiness problem reduces to the *tardiness problem*. The tardiness problem is NP-Complete for the two machine case, and is open for more than two machines [48]. Root's algorithm [58] is the only published approach for minimizing the tardiness problem [15].

Our new scheduling algorithm, Least Space Time First Algorithm (LSTF), seeks to minimize *tardiness* (τ_{max}) in the "Complex Tasks on Multiple Machines" model. We assume that N *non-simple* and preemptible tasks with deadlines are ready at the same scheduling time 0 and should complete by its deadline. Tasks require fixed processing time, must satisfy arbitrary precedence constraints, may share resources, and may execute in parallel with other tasks during their execution. M identical parallel machines are available to process these complex tasks, where each machine can execute any tasks at any time. The classical model, which has provable properties, is much simpler than the extended model handled by LSTF (Table 1.1). In fact, LSTF also handles relaxed guarantees of machine characteristics (for example, a machine may break down during a scheduling period) and task properties (such as an aperiodic arrival pattern). We start with the outlined model to study LSTF properties; we will extend the model further in future work.

Without a *priori* knowledge (such as deadline, processing time, start time, period, etc.), it is impossible for a scheduler to get an optimal result in the sense of schedulability even if there is no restriction on preemption, owing to precedence

Table 1.1 Comparison of classical and LSTF models

	Classical	LSTF
processing time	various and fixed	various and fixed
arrival pattern	periodic	deterministic
deadline	hard	hard
ready time	same	same
preemptive	preemptive	preemptive
precedence constraint	independence	arbitrary dependence
resource sharing	none	software and hardware
actions	single	multiple
call parallelism	without	with
number of jobs	fixed	fixed
machine	single	multiple
configuration	n/a	parallel and identical
execution	singly-threaded	singly-threaded

and/or mutual exclusion constraints [24]. For example, EDF is driven by deadline; RMS is driven by period. The *Precedence Graph (PG)* and *Unit Precedence Graph (UPG)* represent program structures and offer a priori information for the LSTF scheduler. To construct the PG, program translation, code unrolling and inlining, transformation and conditional linking techniques are applied [65, 66, 67, 68] [1]. A transformation to a *mutually commensurable* unit [53] is applied to transform the PG into the UPG. The scheduler computes the *Space-Time* for each executable point over all processes (initial vertices in UPG), and allocates an available processor to an initial vertex with the least Space-Time. LSTF makes scheduling decisions based on precedence and real-time constraints.

[1] See page 15.

1.4 Contributions of this Thesis

We propose a new scheduling algorithm, LSTF, for scheduling complex tasks on parallel machines. Obviously, the operation research and real-time communities are interested in this work. Our major contributions are as follows:

- Theoretical
  - LSTF has been proved to be more effective than EDF in four different scheduling models: 'simple-tasks-single-processor', 'simple-tasks-multiple-processor', and 'complex-tasks-single-processor' (Chapter 4) and 'complex-tasks-multiple-processor' (Chapter 7).
  - Relatively sharp lower and upper bounds of work-conserving scheduling algorithms in general and LSTF in particular, have been obtained (Chapter 5).
- Practical/Pragmatic
  - Context switch overhead and communication cost have been incorporated into LSTF (Chapter 6).
  - We have built a task generator that generates precedence-constrained real-time tasks, and a symbolic simulator that evaluates the performance of LSTF and other scheduling disciplines under different platforms (Chapter 7).

1.5 Why LSTF?

When a system's performance is based on a tardiness criterion, LSTF is a good scheduling algorithm to minimize the maximum tardiness under certain simplifying assumptions (such as no context switch overhead). LSTF is also a practical

scheduling algorithm to integrate and work well in realistic systems. Some features of LSTF are:

- Simplicity

  LSTF is an algorithm combining precedence and real-time constraints. It is simple to understand, implement and maintain. As input to the algorithm, we need the N task graphs, including the graph structure, node costs, and sink-node deadlines, plus the number M of processors. For the bound evaluator, we need only N and M, plus, for each task, the length of the longest path, the total CPU requirement, and the deadline. To incorporate context-switch or communication, we need respectively the context-switch cost and the communication costs for each edge.

- Performance

  We have been able to show, under a number of restrictions on tasks and operating system properties, that LSTF minimizes maximum tardiness, and so produces a deadline-satisfying schedule whenever one exists (Chapter 4).

- Schedulability

  Relatively sharp lower and upper bounds can be provided for LSTF; these tight bounds enable us to design predictable real-time systems.

- Integration into realistic systems

  Unlike some other algorithms, LSTF is not difficult to accumulate, minimize and integrate the cost of various overheads (for example, context switch) into LSTF algorithm (Chapter 6).

1.6 Outline of Dissertation

The rest of this dissertation is organized as follows.

- Chapter 2: Reviews the scheduling model and presents a new scheduling algorithm, Least Space-Time First (LSTF), to minimize the tardiness of all processes.
- Chapter 3: Gives an overview of the related work such as PERT and DDM.
- Chapter 4: Uses an example to illustrate LSTF and explore some properties of LSTF.
- Chapter 5: Investigates the lower and upper bound of tardiness on general algorithms and LSTF.
- Chapter 6: Discusses a variety of cost measures in a realistic system.
- Chapter 7: Presents the simulation results on three different platforms: ideal, context switch and communication, and discusses a substantial treatment of modelling algorithmic issues of real-time systems.
- Chapter 8: Makes the conclusions and outlines future work.

CHAPTER 2

THE SCHEDULING MODEL AND DEFINITIONS

In this Chapter, we characterize our scheduling model and introduce a new algorithm, Least Space-Time First (LSTF), to deal with the general 'complex-task-multiple-processor' model.

2.1 Execution Model

Two types of parallelism are popular in practical use: one is single-instruction stream, multiple-data stream (SIMD), and the other is multiple-instruction stream, multiple-data stream (MIMD) [27]. SIMD is the easiest to understand and implement. On entering a parallel region, a serial program can be transformed from one process into multiple processes (threads). In SIMD, all threads execute the same (possibly guarded) code on different array elements in a *for* loop processing members of an array. MIMD provides *independent-block* parallelism. A block of code, with operations on global and local (private) data different from other blocks, is executed in its own thread independently. Variables within code executed in parallel are either shared between threads or are local to a specific thread. Shared variables are stored in shared memory pointed to by each execution thread, while for local variables, each thread has its own private copy.

Our execution model uses both *blocking* and *non-blocking* calls on MIMD parallel machines, where each node (PE) has a single CPU and contains the code of some tasks and objects [69]. When a task makes an non-blocking call to other objects, the application program (parent) *forks* a thread for the non-blocking call (child). These two threads execute in parallel until they reach a synchronization point, where the parent thread needs to reference IN/OUT or OUT parameters of

the child thread (or one or the other terminates). At the synchronization point, parent and child threads *join* together.

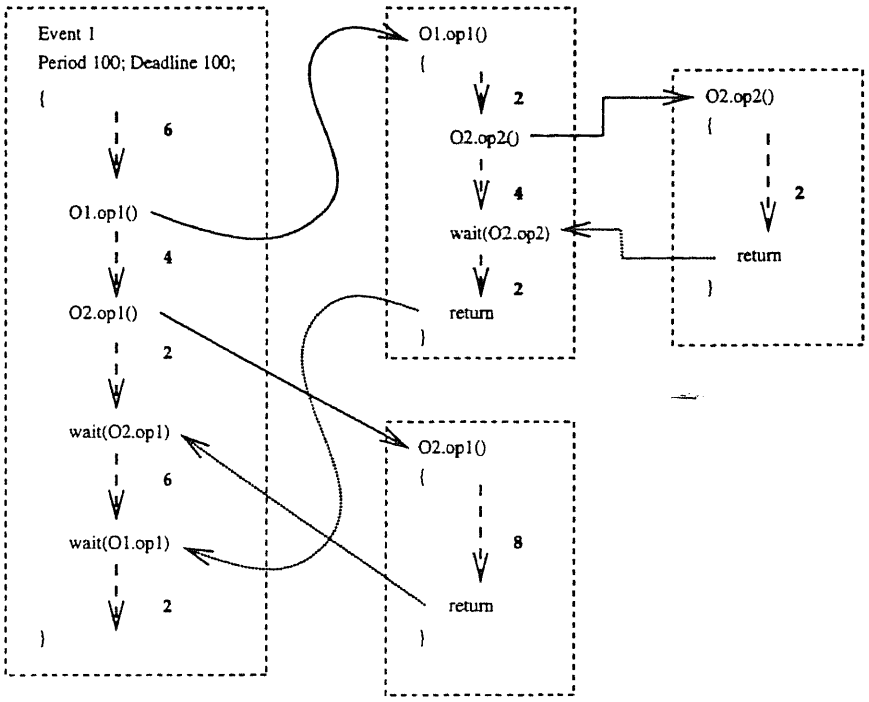

Figure 2.1 Application program for event 1

In Figure 2.1, Event 1 makes a non-blocking call to *object 1* after it has executed 6 instruction units so that Event 1 forks one child thread to execute *object 1.* After executing four more instruction units, Event 1 also generates another thread for calling *object 2. Object 1* must synchronize (join) with Event 1 at the the synchronization point, *wait(O1.op1)*, added to application program by the compiler.

2.2 Scheduling Framework

The scheduling framework is based upon a given precedence graph (PG), G = (V, E), representation of a real-time application program. V is a set of vertices, each associated with a distinct execution weight, and E is a set of directed edges, each associated with a data volume. Standard program translation, code unrolling and

inlining [46, 65], transformation [67] and conditional linking techniques [68] are applied to facilitate PG construction. We will henceforward assume that all PGs have been transformed by such techniques, and that in particular, there is no conditional execution which affects the timing behavior of the code [1]. Transforming from programming source code into PG, a synchronization point has two in-edges, one from task itself and the other one from the non-blocking region (Figure 2.2 is a PG for event 1 in Figure 2.1).

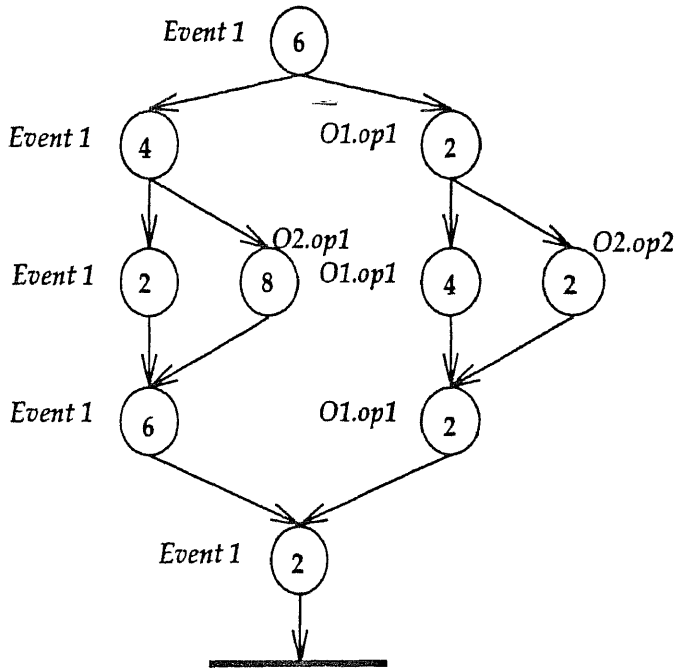

Figure 2.2 Precedence graph (PG) for event 1

Now, we specify the task and hardware model formally. Suppose that we have a number of M identical processors and a set $T = \{T_1, T_2, ..., T_n\}$ of N independent tasks with each task, T_i, having a distinct deadline D_i and fixed processing time X_i. All tasks are ready at the same scheduling time 0. T_i^j is a subtask of task i

[1]This is of course an unreasonable simplification, but any scheduler without perfect future knowledge can perform arbitrarily badly in the presence of conditionally-executed code which affects timing.

and has the execution requirement X_i^j. Precedence constraints, denoted by "$\rightarrow$", between subtasks; edges are denoted by $T_i^j \stackrel{\lambda}{\rightarrow} T_i^k$, which represent (1) a precedence relationship between predecessor T_i^j and successor T_i^k, and/or (2) a data volume λ sent from subtask T_i^j to subtask T_i^k. The data volume on a pure precedence edge is $\lambda = 0$. Task T_i^j is defined to be *ready* when all its predecessors have completed execution and it has received all messages from all its predecessors. A ready vertex corresponds to a point where the thread is ready to execute, and competes for resources with other ready vertices. If we have 10 ready vertices in the PG graph, the scheduler has 10 threads competing for processors. A PG-based scheduler now operates more-or-less as a topological sort. After an ready vertex has been scheduled, the vertex with its out-edges can be deleted from PG, and its direct successors then become new ready vertices. With a single sink subtask, each task T_i can be represented as a directed acyclic graph (DAG). M identical parallel machines are available to process these complex tasks, where each machine can execute any tasks at any time.

A set of nodes is said to be *mutually commensurable* [53] if there exists a U such that each node's weight is an integer multiple of U — in practical programs, U is of course also a multiple of the real-time unit. Based upon this concept of commensurable nodes, we will sometimes conceptually look upon the PG as a *unit* precedence graph.

It is convenient to treat processors as non-preemptible during a real-time unit U. Using UPG, the scheduler treats each node equally and schedules ready vertices step-by-step.

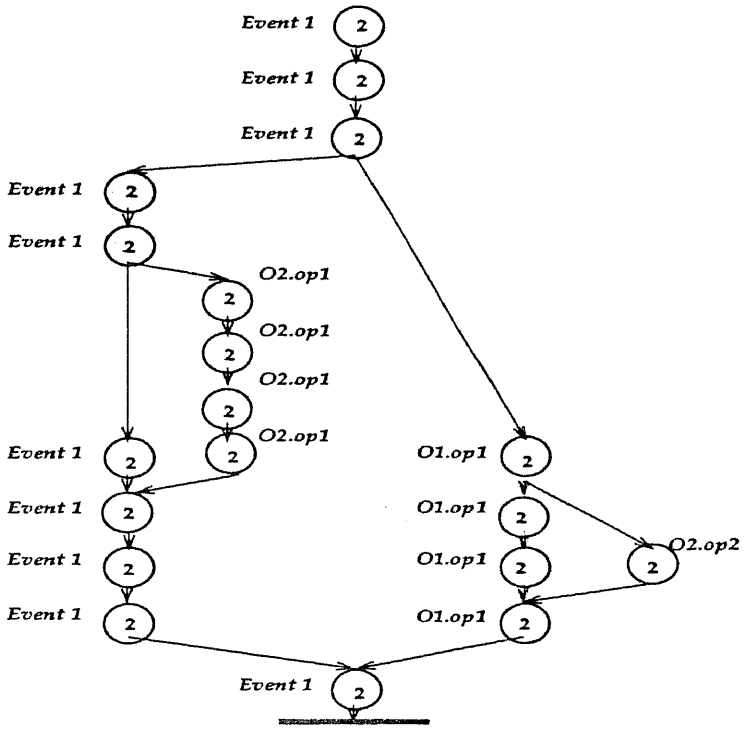

Figure 2.3 Unit precedence graph (UPG) for event 1

2.3 Definitions

To demonstrate LSTF and its properties clearly, we define a number of terms.

- $TASK_v$: a task with ready vertex v. By extension, a (partially executed) task on which v is one of the ready nodes.
- $D(v)$: deadline of $TASK_v$.
- $C(v)$: completion time of $TASK_v$.
- $\tau(v)$: the tardiness of $TASK_v$; $\tau(v) = \max(C(v) - D(v), 0)$.
- τ_{max}: the maximum τ among all tasks.
- $R(v)$: remaining time for $TASK_v$; $R(v) = D(v) - Present\_Time$.

- $L(v)$: the length (level) of a longest path from the vertex v to a sink (a vertex with no descendants) of the PG.
- $P(v)$: the longest path in $TASK_v$.
- LP_v: the longest path length in $TASK_v$.
- TX_v: the total execution cost of $TASK_v$.
- $BX_{(v,i,j)}$: the execution cost of all $TASK_v$'s branches except P_v between level i and j; $BX_{(v,0,LP_v)} = TX_v - LP_v$.
- S_v: *space time* of a vertex v at a particular scheduling point: $S(v) = R(v) - L(v)$.

2.4 Least Space Time First (LSTF) Scheduling Algorithm

Without *a priori* knowledge (such as deadline, processing time, start time, period, etc.), it is impossible for a scheduler to get an optimal result in the sense of schedulability, even if there is no restriction on preemption, owing to precedence and/or mutual exclusion constraints [24]. Most common schedulers assume the availability of such information; for example, EDF is driven by deadline, RMS by period. LSTF uses the *Unit Precedence Graph (UPG)*, which encodes *a priori* information representing program structures and deadline [2].

At a particular time, the remaining time of a task is its deadline less the present time. The *level* of a vertex is the length (measured in time units as per UPG construction) of the longest path from the vertex to a leaf of the UPG. The Space-Time of a vertex is then defined to be the remaining time of the task less the level

[2]The UPG is a *conceptual* model. It is a straightforward to implement LSTF using just the PG and unit preemption, with concomitant savings in space, but slightly higher time cost.

of the vertex [3]. The LSTF algorithm instructs the scheduler to allocate an available processor to a ready subtask (whose predecessors have all already executed) which has the least space time. When a tie occurs between subtasks, a subtask with early deadline is granted a processor. If there is still a tie, it is handled arbitrarily [13].

[3]Space time (*Time_before_deadline* - *Longest_execution_path_remaining*) is different from the laxity (*Time_before_deadline* - *Remaining_exec_time*) used by Least Laxity First (LLF) algorithm.

CHAPTER 3

RELATED WORK

To satisfy different real-time applications, various scheduling algorithms were investigated before. Usually, feasibility is used to evaluate the hard real-time applications (i.e., τ_{max} should be equal to 0; every task should meet its own deadline.). For soft real-time applications, several criteria are also discussed, such as total/mean lateness, total/mean tardiness, total/mean number of tardy jobs and imprecise results [15, 18, 32, 39, 62]. In multimedia applications, the engineers look other quality of service, for example, (m,k)-firm scheduling [38]. A stream is said to have (m,k)-firm deadlines if at least m out of any k consecutive customers must meet their deadlines. The basic idea is to assign higher priorities to customers from streams that are close to a dynamic failure (fewer than m out any k consecutive customers meet their deadlines).

For World-Wide Web (WWW) applications, the server should respond to all requests prior to specified deadlines. If the server fails to do so, it should "fairly" schedule all tasks, and in particular tasks behind in their schedules. LSTF assures that all tasks are finished as close possible to their deadlines, while (m,k)-firm scheduling may starve some tasks. For such applications, it seems sensible to choose maximum tardiness as the performance criterion, and thus (as we shall see) LSTF as the scheduling discipline.

The following example demonstrates the difference between LSTF and (m,k)-firm scheduling. Suppose that there are six tasks on one processor, each with a pair of parameters (processing time, deadline), are listed as follows:

$$\{A(1,1),\ B(1,2),\ C(1,2),\ D(1,3),\ E(1,5),\ F(1,6)\}$$

The sequence of executions for (2,3) scheduling algorithm is (A, B, D, C, E, F). Therefore, (2,3) scheduling algorithm has the maximum tardiness 2. On the other hand, LSTF would schedule (A, B, C, D, E, F) and The maximum tardiness is 1.

3.1 Transformation between LSTF and DDM

Program Evaluation and Review Technique (PERT) and Critical Path Method (CPM) are two well-known and closely related network analysis methods in the field of planning, scheduling and construction projects [19]. For example, Navy used PERT to complete the Polaris missile project two years earlier than scheduled and Du Pont Cooperation used CPM to reduce maintenance time of plant by one-third. PERT/CPM can find the answer for questions such as: when does the whole project finish as early as possible? when does a specific activity (sub-project) start and finish? which sub-projects are "critical"? how long can "non-critical" sub-projects delay? The difference between PERT and CPM is that CPM has an option of cost-time trade-offs, i.e., CPM can reduce sub-project time dynamically by adding/subtracting resources.

In management science, PERT/CPM is a good method for minimizing the make-span where the application has an effectively infinite number of processors in the presence of precedence constraints between tasks. Without taking some considerations and constraints into account, PERT/CPM is an inadequate model for planning and control paradigm [33]. LSTF can be viewed as a variant of a least-laxity scheduler accounting for parallelism, or as a critical path method with a limited resources scheduler. However, LSTF is quite different in execution from techniques for computer systems process scheduling usually identified as CPM-schedulers [2, 4, 55].

The least-space-time-first (LSTF) algorithm is a preemptible CPM variant closely related to the deterministic deadline modification (DDM) algorithm [7]. DDM is an optimal algorithm for scheduling in-tree unit-processing-time tasks, in the sense of tardiness minimization on an overhead-free platform.

The DDM algorithm consists of two phases. The first phase modifies the deadline, propagating through Equation 3.1, and produces an (increasing) ordered

list based on the new deadline. Then, the second phase schedules the ready task with the smallest modified deadline.

$$D_{i,j} = \min(D_{i,j}, D_{i,k} - 1) \tag{3.1}$$

where $T_{i,k}$ is $T_{i,j}$'s successors and $D_{i,j}$ is the deadline of subtask $T_{i,j}$ ($D_{i,j}$ is updated for each predecessor $T_{i,k}$ of $T_{i,j}$) [1].

We use Figure 3.1 to illustrate the DDM algorithm. There are three list tasks, each with its own time-dependent subtasks. Each node represents a subtask and is labeled by the ordered pair (old deadline, new deadline), where old deadline is the original deadline, and the new deadline is the new modified deadline based on Equation 3.1. With two processors, the DDM scheduler will choose the two smallest modified-deadline ready nodes at each step; the execution sequence is as follows:

$$\{(T_{1,4}, T_{2,4}), (T_{1,3}, T_{2,3}), (T_{2,2}, T_{3,4}), (T_{1,2}, T_{3,3}), (T_{2,1}, T_{3,2}), (T_{1,1}, T_{3,1})\}$$

It is easy to transform the models between LSTF and DDM.

- DDM $\rightarrow$ LSTF:

  1. Group nodes with the same deadline into a cluster node such that there is no communication and synchronization within the cluster node.
  2. Find the GCD (Greatest Common Divisor) of node execution costs and constructing the UPG.
  3. Assign the same appropriate deadline to each node.
  4. Add a sink node (κ) with zero cpu requirement and infinite deadline, and construct arcs with zero data volume from original sink nodes to κ.

[1] For the PG, the equation becomes $D_{i,j} = \min(D_{i,j}, D_{i,k} - W_{i,k})$, where $W_{i,k}$ is the execution cost of $T_{i,k}$.

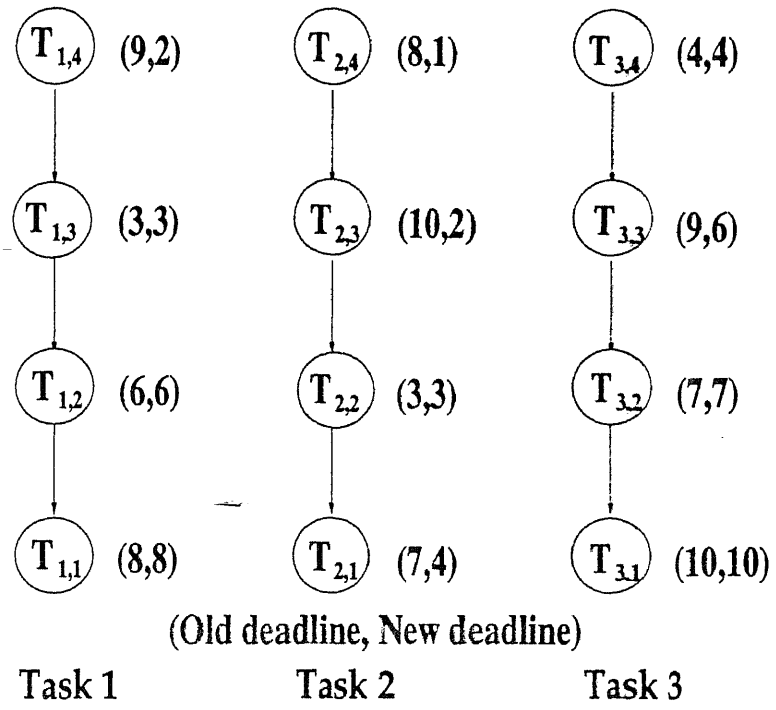

Figure 3.1 An example for DDM

- LSTF $\rightarrow$ DDM:

  1. Divide each node into a list of unit processing nodes, i.e., each node has unit cpu requirement in the list. Every arc has no communication cost within the list.
  2. Calculate space time as follows: $S(v) = \min\{R(v) - L(v), S(w)\}$, where w is v's successors and predecessors.

Although those two models can be mutually transformed, we investigate more properties based on the LSTF model (such as feasibility and bounds) and integrate some realistic issues (such as context switch and communication).

CHAPTER 4

EXAMPLES AND PROPERTIES OF LSTF

Through an example, we demonstrate that LSTF can outperform the rate-monotonic-scheduling (RMS), highest-level-first [44], earliest-deadline-first and least-laxity-first scheduling disciplines [41] in the sense of minimizing maximum tardiness of a set of tasks. Some properties of LSTF as applied upon idealized platforms are presented.

4.1 Example

As an example, assume there are four tasks to be scheduled on two identical processors, and that tasks can migrate freely without migration overhead. We assume that independent units of a task can execute in parallel. Each task has its own unit precedence graph and deadline, as illustrated in Figure 4.1. Using Figure 4.1, Table 4.1 and Figure 4.2, we demonstrate how LSTF outperforms Rate-Monotonic-Scheduling, Highest-Level-First, Earliest-Deadline-First, and a naive Least-Laxity-First scheduling discipline. Intuitively, we have cases which show LSTF outperforming other algorithms, formal results in Section 4.2, and simulations in Chapter 7.

The Rate Monotonic Scheduling (RMS) algorithm assigns higher priorities to the higher request rate. Under certain assumptions, RMS is optimal, in the sense that no other fixed priority assignment rule can schedule a set of tasks which can not be scheduled by RMS [50]. RMS performs well in single processor systems but not in multiple processor systems [25]. Assigning priorities to the four tasks in Figure 4.1 based on their request rates, twenty-four possibilities may occur. In Table 4.1, H stands for highest priority, HM for High-medium, M for medium and L for lowest priority and $C(TASK_i)$ is the completion time of $TASK_i$. The *tardiness* of all tasks

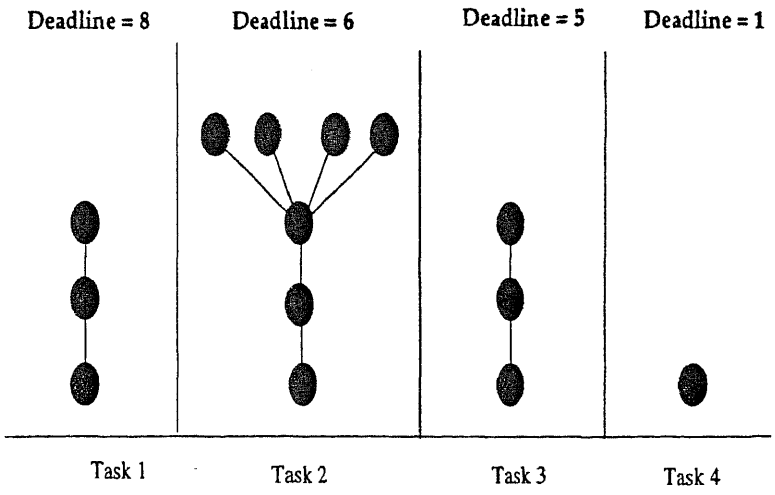

Figure 4.1 Example of four tasks to be scheduled on two processors

is denoted by τ_{max}. In most of cases, τ_{max} is greater than 0 (i.e., some tasks miss their deadlines) and the schedulable ratio is low (2 out 24).

The Highest Level First (HLF) policy gives priority to higher-level tasks. It produces the optimal completion time schedule for inforest [44] and outforest [8] precedence graphs. HLF schedules task 2 with the highest level at scheduling point 0 and 1, and it misses task 4 at scheduling point 1. HLF executes (in the format of $T_{taski}.level.vertex$): { $(T_2.4.1, T_2.4.2)$, $(T_2.4.3, T_2.4.4)$, $(T_2.3.1, T_3.3.1)$, $(T_1.3.1, T_3.2.1)$, $(T_1.2.1, T_2.2.1)$, $(T_4.1.1, T_3.1.1)$, $(T_1.1.1, T_2.1.1)$ }. τ_{max} is 5 because of task 2 finishing with tardiness 1, task 2 finishing with tardiness 2 and task 4 finishing with tardiness 5 (Figure 4.2).

Earliest Deadline First (EDF) policy takes deadline as priority. EDF minimizes the lateness and the tardiness of a set of tasks with deadlines on a single processor [52] Even relaxing some assumptions significantly — allowing, for example, an arbitrary (possibly infinite) number of tasks with arbitrary arrival and service times and deadlines EDF remains an optimal algorithm [64]. EDF executes: { $(T_4.1.1, T_3.3.1)$, $(T_3.2.1, T_2.4.1)$, $(T_3.1.1, T_2.4.2)$, $(T_2.4.3, T_2.4.4)$, $(T_2.3.1, T_1.3.1)$, $(T_2.2.1, T_1.2.1)$,

Table 4.1 Layout of rate monotonic scheduling algorithm

Priority				Task				τ_{max}
H	HM	M	L	$C(TASK_1)$	$C(TASK_2)$	$C(TASK_3)$	$C(TASK_4)$	
1	2	3	4	3	7	6	7	6
1	2	4	3	3	7	7	4	3
1	3	2	4	3	9	3	4	3
1	3	4	2	3	9	3	4	3
1	4	2	3	3	7	7	1	2
1	4	3	2	3	9	4	1	3
2	1	3	4	5	5	7	6	5
2	1	4	3	5	5	7	6	5
2	3	1	4	5	7	5	6	5
2	3	4	1	5	7	5	6	5
2	4	1	3	5	6	8	1	3
2	4	3	1	8	6	5	1	0
3	1	2	4	3	8	3	6	5
3	1	4	2	3	9	3	4	3
3	2	1	4	6	7	3	7	6
3	2	4	1	7	7	3	4	3
3	4	1	2	4	9	3	1	3
3	4	2	1	7	7	3	1	1
4	1	2	3	3	7	7	1	2
4	1	3	2	3	9	4	1	3
4	2	1	3	5	6	8	1	3
4	2	3	1	8	6	5	1	0
4	3	1	2	4	9	3	1	3
4	3	2	1	7	7	3	1	1

$(T_2.1.1, T_1.1.1)$ }. EDF can meet task 4's deadline but misses task 2 with tardiness 1, so that τ_{max} is equal to 1 (Figure 4.2).

Defining *laxity* to be the difference between remaining time and remaining requested computation time (initially the difference between deadline and total computation), the Least-Laxity-First (LLF) policy assigns the highest priority to the least laxity task. Under certain assumptions, EDF and LLF are optimal algorithms in single processor systems but not in multiple processor systems [47]. LLF misses task 4 because task 2 with least laxity (6 - 7 = -1) will be scheduled

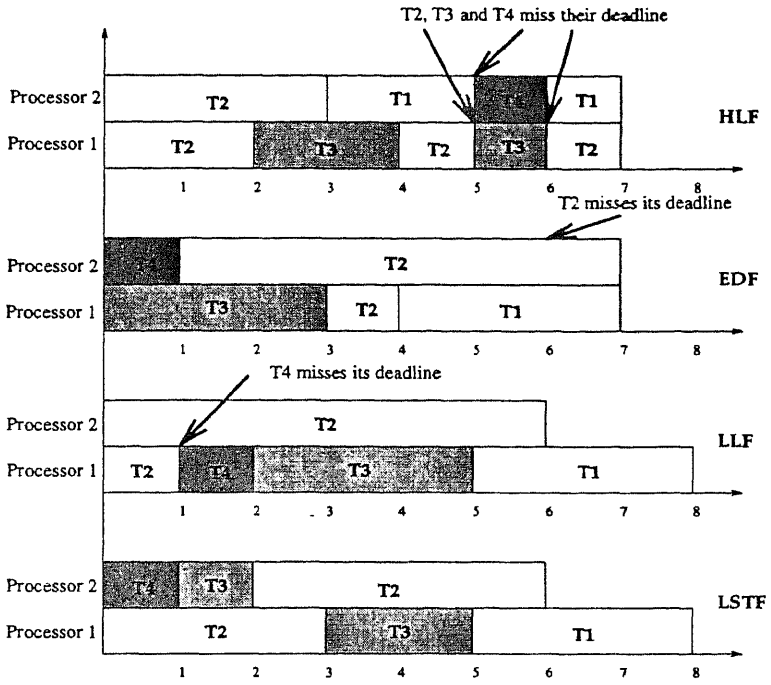

Figure 4.2 Different results of scheduling disciplines

ahead of task 4. LLF executes: { $(T_2.4.1, T_2.4.2)$, $(T_2.4.3, T_4.1.1)$, $(T_2.4.4, T_3.3.1)$, $(T_2.3.1, T_3.2.1)$, $(T_2.2.1, T_3.1.1)$, $(T_2.1.1, T_1.3.1)$, $(T_1.2.1)$, $(T_1.1.1)$ }. Task 4 finishes with tardiness 1 and τ_{max} is equal to 1 (Figure 4.2). Although LLF considers execution request and deadline together, LLF does not account for parallelism. When independent units of a task can not execute in parallel, LSTF reduces to LLF.

Tasks are free to run on any processor so that the LSTF scheduler will assign time slots of processors to executable nodes with the lowest space-time. The procedures for the LSTF scheduler are as follows:

LSTF Scheduler

Step 1: Calculate the Space-Time, S(v), for each initial vertex.

Step 2: Find $\nu(T)$, a set of initial vertices which have low S(v), so that the cardinality of $\nu(T)$ is equal to the number of

processors available at scheduling point T.

Step 3: Dispatch to each processor a member of $\nu(T)$.

Step 4: Advance the scheduling point by 1, and update UPG.

Step 5: While there are units left to schedule, go to Step 1.

At scheduling point 0, LSTF calculates the Space-Time for each initial vertices, $S(T_1.3.1) = 5$, $S(T_2.4.1) = S(T_2.4.2) = S(T_2.4.3) = S(T_2.4.4) = S(T_3.3.1) = 2$, $S(T_1.1.1) = 0$ (Table 4.2). The level of initial vertices in $TASK_2$ are higher than $TASK_3$ and LSTF assigns the processors to task 2 and 4. LSTF executes: { $(T_2.4.1, T_4.1.1)$, $(T_2.4.2, T_3.3.1)$, $(T_2.4.3, T_2.3.4)$, $(T_2.3.1, T_3.2.1)$, $(T_2.2.1, T_3.1.1)$, $(T_2.1.1, T_1.3.1)$, $(T_1.2.1)$, $(T_1.1.1)$ }. LSTF meets all tasks' deadlines and τ_{max} is equal to 0 (Figure 4.2). We summarize the scheduling results for various algorithms in Table 4.3.

Table 4.2 LSTF scheduling example

Scheduling Point	$S(v)$	$\nu(T)$
0	$S(T_1.3.1) = 5$; $S(T_2.4.1) = 2$; $S(T_2.4.2) = 2$; $S(T_2.4.3) = 2$; $S(T_2.4.4) = 2$; $S(T_3.3.1) = 2$; $S(T_4.1.1) = 0$.	$\{T_4.1.1, T_2.4.1\}$
1	$S(T_1.3.1) = 4$; $S(T_2.4.2) = 1$; $S(T_2.4.3) = 1$; $S(T_2.4.4) = 1$; $S(T_3.3.1) = 1$.	$\{T_3.3.1, T_2.4.2\}$
2	$S(T_1.3.1) = 3$; $S(T_2.4.3) = 0$; $S(T_2.4.4) = 0$; $S(T_3.2.1) = 1$.	$\{T_2.4.3, T_2.4.4\}$
3	$S(T_1.3.1) = 2$; $S(T_2.3.1) = 0$; $S(T_3.2.1) = 0$.	$\{T_2.3.1, T_3.2.1\}$
4	$S(T_1.3.1) = 1$; $S(T_2.2.1) = 0$; $S(T_3.1.1) = 0$.	$\{T_2.2.1, T_3.1.1\}$
5	$S(T_1.3.1) = 0$; $S(T_2.1.1) = 0$.	$\{T_1.3.1, T_2.1.1\}$
6	$S(T_1.2.1) = 0$.	$\{T_1.2.1\}$
7	$S(T_1.1.1) = 0$.	$\{T_1.1.1\}$

Table 4.3 The summary table

Discipline	τ_{max}	Remarks
RMS	3.2	the mean value of 24 cases
HLF	5	
EDF	1	
LLF	1	
LSTF	0	

4.2 Properties of LSTF

We have discovered some properties of LSTF. In the following, we assume that every task is ready at scheduling point 0. We also assume all conditionals are either balanced (same computation requirements on both branches) or evaluable before task execution. Any scheduler which relies on the computational structure of a task will in fact fail to construct optimal schedules (in any reasonable sense) in some cases if the above condition fails.

The following two lemmas characterize the runtime behavior of LSTF:

Lemma 1 *Suppose that the ready node a is in the longest path of* $TASK_a$. *The space time of vertex a does not change if the system schedules vertex a, otherwise it decreases by one.*

Lemma 2 *If, at some scheduling point, ready nodes a and b of* P_a *and* P_b *have identical space-time, then the nodes in* P_a *and* P_b *are executed by LSTF one level at a time until one finishes.*

Theorem 1 compares the performance of LSTF and EDF in a uniprocessor environment.

Theorem 1 *LSTF dominates EDF for scheduling a set of tasks on a single processor; i.e., if a set of tasks are schedulable under EDF, they are also schedulable under LSTF.*

Proof:

We assume that there are two tasks ($TASK_a$ and $TASK_b$) and the deadline of $TASK_a$ is less than $TASK_b$. They are schedulable under EDF. At some scheduling point, we have two possibilities:

1. $S_a \leq S_b$: LSTF has the same action as EDF.
2. $S_a > S_b$: We have C_a and C_b under EDF:

$$D_a \geq C_a = TX_a = LP_a + BX_{(a,0,LP_a)}$$

$$D_b \geq C_b = TX_a + TX_b = LP_a + BX_{(a,0,LP_a)} + LP_b + BX_{(b,0,LP_b)} \tag{4.1}$$

Suppose that $k = D_b - D_a$ and $i = S_a - S_b$. From Lemma 2, the processor is devoted to executing $TASK_b$ before finishing $TASK_a$ in two ways: (1) the longest path, P_b: $i + LP_a$ time units, and (2) the branches: $BX_{(b,k+i,LP_b)} + BX_{(b,k,k+i)}$. So, we have C'_a and C'_b under LSTF as follows:

$$C'_a = LP_a + BX_{(a,0,LP_a)} + i + LP_a + BX_{(b,k+i,LP_b)} + BX_{(b,k,k+i)}$$

$$C'_b = C_b$$

Now, we should prove $D_a \geq C'_a$ if they are also schedulable under LSTF.

$$S_a > S_b \Rightarrow D_a - LP_a - i = D_b - LP_b \tag{4.2}$$

We subtract $BX_{(b,0,LP_b)} + LP_a + BX_{(a,0,LP_a)}$ on both side in Equation (4.2).

$$\begin{aligned} & D_a - LP_a - i - (BX_{(b,0,LP_b)} + LP_a + BX_{(a,0,LP_a)}) \\ & = D_b - LP_b - (BX_{(b,0,LP_b)} + LP_a + BX_{(a,0,LP_a)}) \end{aligned} \tag{4.3}$$

From Equation (4.1) and (4.3), we derive $D_a \geq LP_a + i + BX_{(b,0,LP_b)} + LP_a + BX_{(a,0,LP_a)}$. Since $BX_{(b,0,LP_b)} \geq BX_{(b,k+i,LP_b)} + BX_{(b,k,k+i)}$, we can obtain

$$\begin{aligned} D_a \geq\ & LP_a + BX_{(a,0,LP_a)} + i + LP_a + \\ & BX_{(b,k+i,LP_b)} + BX_{(b,k,k+i)} = C'_a \ \square \end{aligned}$$

Corollary 1 *LSTF is optimal in the simple-tasks-single-processor environment.*

Proof:

EDF is optimal [50]. □

Theorem 2 *LSTF is more effective than EDF for scheduling a set of tasks in the simple-tasks-multiple-processor environment.*

Proof:

A simple task does not have any branches; hence, the total requirement is equal to the length of the longest and only path ($TX_v = LP_v$). The space time is the same as laxity in this special case. However LLF is more effective than EDF for scheduling a set of tasks on $m > 1$ processors [49]. □

CHAPTER 5

BOUNDS ON TARDINESS

There are two ways to evaluate the performance of heuristic algorithms: (a) worst case behavior, (b) average case behavior. In this chapter, we prove theoretical bounds on maximum tardiness for scheduling algorithms in general and for LSTF in particular, within an idealized framework that ignores several implementations details such as context switch time, etc.

5.1 General Bounds on Tardiness

Since the target problem is known to be NP-Complete, many approaches have been focused on the development of heuristic which do not provide optimal solutions but whose algorithms run in polynomial time [20, 61, 73]. Unfortunately, they provide neither relative error bounds nor absolute lower and upper bounds. In this section, we discuss lower/upper bounds for general algorithms based on R_i, D_i and M. Of course, bounds can be sharpen if more task information is acquired. For purposes of this analysis, we assume that there are n tasks $\{T_1, ... T_n\}$ to be scheduled on M processors, and task each task T_i is characterized by

- A deadline D_i.
- A computational requirement at time t, $R_i(t)$. For simplicity, let $R_i = R_i(0)$.
- A maximum height at time t, $L_i(t)$. Again, $L_i = L_i(0)$.

We assume that tasks are ready at time $t = 0$.

All scheduling algorithms are assumed to be *work-conserving*, that is, they schedule as many nodes as possible at each scheduling instant t [1]. The performance of a scheduling algorithm x on task set T is characterized by [2]:

[1] That is, the server cannot be idle if there is a job in the waiting queue.

[2] Where context is clear or irrelevant, we will suppress the superscript x.

- The time it completes task T_i, γ_i^x.
- The makespan, or maximum completion time, $\Gamma^x = \max_{i=1}^{n} \gamma_i^x$.
- The tardiness of each task, τ_i^x, and the maximum tardiness τ^x.

Finally, we use [n] for $\{1, 2, 3, \ldots, n\}$. We identify several useful permutations of task indices [n]: α is the permutation of indices by increasing D_i, β by decreasing R_i, and δ by decreasing $\frac{R_i}{M} - D_i$, which is an approximation of slack if task i is executed in isolation. To minimize nested subscripts, we will use function notation, e.g., $\alpha(i)$, rather than the more usual α_i.

Bounds for general scheduling algorithms follow largely from the precedence structure and the work-conserving property.

Lower Bound. Our principal lower bound schema follows from looking at a subset of tasks, and assuming the algorithm can schedule the subset exclusively, without wasted cycles. We also give a lower bound based on precedence structure.

Lemma 3 *Let x be any scheduling algorithm, and $\mathcal{S} \subseteq$ [n]. Then*

$$\max\left\{0, \left\lceil \sum_{i \in \mathcal{S}} \frac{R_i}{M} \right\rceil - \max_{i \in \mathcal{S}}\{D_i\}\right\} \leq \tau^x$$

Proof:

$\mathcal{S}$ involves $\sum_{i \in \mathcal{S}} R_i$ work, which requires time at least $\lceil \sum_{i \in \mathcal{S}} \frac{R_i}{M} \rceil$ to schedule, and $\max_{i \in \mathcal{S}}\{D_i\}$ is the time by which the last task in $\mathcal{S}$ must finish. □

Clearly, using Lemma 3 for all subsets would require exponential time. However, we can obtain good and efficient bounds by by stressing one or both of the terms in the bound. Let

- $\ell_1 = \max_{k=1}^{n} \left\{ \left\lceil \sum_{i=1}^{k} \frac{R_{\alpha(i)}}{M} \right\rceil - D_{\alpha(k)} \right\}$
- $\ell_2 = \max_{k=1}^{n} \left\{ \left\lceil \sum_{i=1}^{k} \frac{R_{\beta(i)}}{M} \right\rceil - \max_{i=1}^{k}\{D_{\beta(i)}\} \right\}$

- $\ell_3 = \max_{k=1}^{n} \left\{ \left\lceil \sum_{i=1}^{k} \frac{R_{\delta(i)}}{M} \right\rceil - \max_{i=1}^{k} \{D_{\delta(i)}\} \right\}$
- $\ell_4 = \max_{k=1}^{n} \{L_i - D_i\}$

Theorem 3 *Let x be any scheduling algorithm. Then*

$$\max \{ 0, \ell_1, \ell_2, \ell_3, \ell_4 \} \leq \tau^x.$$

Further, this bound can be computed in $O(n)$ (plus the $O(n \log n)$ cost of forming the permutations α, β, δ).

Proof:

ℓ_1, ℓ_2, ℓ_3 are special cases of Lemma 3; $D_{\alpha(k)}$ is the last and necessarily the largest D_i by definition of α.

ℓ_4 is a lower bound since only one unit of computation on the longest path can be scheduled at any time t.

Finally, note that each of the first three ℓ_i can be computed by maintaining a running sum and max, at cost n, and that ℓ_4 trivially has cost n. □

Upper Bound. Upper bounds on tardiness follow from deadlines and bounds on makespan. Moreover, there is a well-known relative upper bound for makespan Γ^x due to Graham [31]:

$$\Gamma^x \leq (2 - \frac{1}{M})\Gamma^{opt} \tag{5.1}$$

where Γ^{opt} is the makespan when scheduled by an optimum algorithm, *opt*.

When all task deadlines are 0, the problem of minimizing tardiness reduces to the problem of minimizing the makespan. However, given this or any other upper bound for makespan, an upper bound on tardiness can easily be obtained:

$$\tau^x \leq \Gamma^x - D_{\alpha(1)}.$$

Further, this bound can be attained, if some node from $T_{\alpha\ (1)}$ is scheduled at time $t = \Gamma^x$.

The problem thus reduces to finding upper bounds for makespan. We can obtain a slightly sharper absolute version of the Graham result by first considering some properties of work-conserving schedulers; Lemma 4 gives these key observations.

Lemma 4 *Suppose x is a work-conserving algorithm, and T a task set. Let T_f be a task scheduled by x at scheduling instant Γ^x. Then:*

1. *At any time t, $0 \leq t < \Gamma^x$, either x schedules m nodes, or x schedules at least one node of T_f.*

2. *At any time t at which a node of T_f is scheduled, either x schedules m nodes, or $L_f\ (t+1) = L_f\ (t) - 1$.*

Proof:

For every task T_i, and at any time t between its ready time and its completion time, every node $v \in T_i$ with height $h\ (v) = L_i\ (t)$ is ready. If there are r ready nodes at time t, a work-conserving scheduler will schedule min (r, m) nodes. Finally, all tasks, and in particular T_f, are ready at time 0. □

Theorem 4 *Suppose x is a work-conserving algorithm, and T a task set. Let L_{max} be* max $\{L_i\}$. *Then*

$$\Gamma^x \leq \left\lfloor \frac{\sum_{i=1}^{n} R_i - L_{max}}{M} \right\rfloor + L_{max}.$$

Further, this bound can be computed in time $O\ (n)$.

Proof:

Every scheduling instant either schedules m nodes, or reduces $L_f\ (t)$. Clearly, makespan is maximized when $L_f\ (t)$ is never reduced when m nodes are scheduled, and when T_f is T_{max}. □

Corollary 2 *Let U be the upper bound of Theorem 4. Then*

$$\tau^x \leq U - D_{\alpha(1)} = U'.$$

Corollary 3 *Graham's result follows from Theorem 4.*

Proof:

Let $C = \left\lceil \frac{\sum_{i=1}^{n} R_i}{M} \right\rceil$. C is a lower bound for Γ^x, and, since $L_{max} \leq C$, we have

$$C \leq \Gamma^{opt} \leq \Gamma^x \leq (2 - \frac{1}{M})\, C,$$

and Graham's bound follows. □

5.2 Sharper Upper Bounds on Tardiness for Specific Algorithms

Some heuristics offer *a priori* information that determines the execution orders of tasks, but not their order of completion (i.e., priority can order which task should run first but it cannot guarantee which task will finish first). Consider, for example, the three tasks which are to be scheduled on two processors in Figure 5.1. Although TASK1 has the highest priority, TASK1 finishes at the last. This phenomenon — a higher priority task finishes after a lower-priority one — is called *contrapositive termination.* Due to contrapositive termination, any task can complete at the end of the makespan, and hence bounds sharper that are in Theorem 4 cannot be obtained for most algorithms. However, for some algorithms, it is possible to deduce further information about the order in which tasks complete when executed by these algorithms and sharper bounds may be possible. We present below each improved bounds for the EDF and LSTF scheduling algorithms.

Upper Bounds for EDF. Under EDF, tasks are known to execute in deadline order, and the tardiness of each task can be computed by considering only those tasks with earlier and equal deadlines.

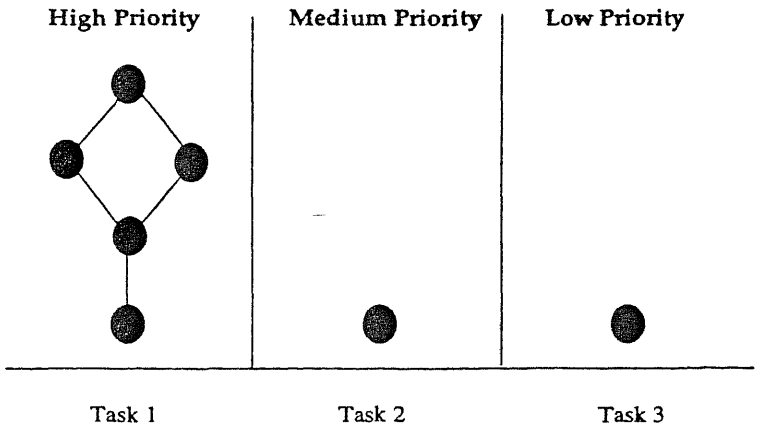

Figure 5.1 Example of three tasks on two processors

Theorem 5 *Let T be a task system, and suppose all deadlines are distinct. Then the maximum tardiness under EDF is bounded by*

$$\tau^x \leq \max \left\{ 0, \max_{k=1}^{n} \left\{ \left\lfloor \frac{\sum_{i=1}^{k} R_i - L_k}{M} \right\rfloor + L_k - D_k \right\} \right\}$$

Even allowing for equal deadlines, the conclusion of the theorem holds if EDF always favors the task earlier in the α-list, or if the α-list breaks ties in favor of the shortest L_k (so that the last task with a given D_k will necessarily result in the worst-case tardiness for tasks with that deadline). Otherwise, it is sufficient to replace L_k by $\max \{D_k - D_j, L_j\}$, with the same effect.

Upper Bounds for LSTF. The following lemma characterizes the finish order under LSTF:

Lemma 5 *Suppose that we have two task, T_i and T_j with space time S_i and S_j respectively. If $S_i \leq S_j$, $L_i \leq L_j$ and $D_i < D_j$ then T_i completes at least $(L_j - L_i)$ time units before T_j; i.e., $\gamma_i \leq \gamma_j - (L_j - L_i)$.*

Proof:

When $S_i = S_j$, the subtasks in L_i and L_j are executed by LSTF one level at a time until one finishes. Since of $L_i \leq L_j$, the completion time of T_j is therefore equal to or greater than $(\gamma_i + (L_j - L_i))$. When $S_i < S_j$, T_j may execute in parallel with T_i or be inactive, the completion time of T_i is thus less than or equal to $(\gamma_j - (L_j - L_i))$. □

Let $E_{j,i}$ be defined to equal $(L_j$ - $L_i)$. For each task T_i, we build a list Δ_i of task T_j such that $D_j > D_i$ and $L_j \geq Li$, ordered by decreasing $E_{j,i}$. Let $\Delta_i(j)$ denote the jth item on list Δ_i, and $\mid \Delta_i \mid$ denote the cardinality of Δ_i. When LSTF completes the execution of T_i, each task in Δ_i, $\Delta_i(j)$, has $E_{j,i}$ levels remaining to be executed. All tasks in Δ_i may execute in parallel. Clearly [3], we can obtain a sharper bound for the γ_i based on Theorem 4 and Lemma 5. Let

- $u_1 = \left\lfloor \frac{\sum_{k=1}^{n} R_k - L_i}{M} \right\rfloor + L_i$

  u_1 estimates the completion time of T_i without any finish order constraints.

- $u_2 = \min_{j \in \Delta_i} \left\{ \left\lfloor \frac{\sum_{k=1}^{n} R_k - L_j}{M} \right\rfloor + L_j - E_{j,i} \right\}$

  Task T_i has to finish at least $E_{j,i}$ time units (which is remaining required level of T_j) before the expected completion time of task T_j for any T_j in list Δ_i. Thus, T_i must complete its execution before the smallest of the differencies between expected completion time and the corresponding $E_{j,i}$.

Theorem 6 *Let T be a task system, and suppose all deadlines are distinct. Then the maximum tardiness under LSTF is bounded by*

$$\tau^{LSTF} \leq \max \left\{ 0, \max_{k=1}^{n} \{\hat{\gamma}_i - D_i\} \right\},$$

where $\hat{\gamma}_i = \min\{u_1, u_2\}$.

Proof:

[3] If $\Delta_i = \emptyset$, u_2 does not exist.

Follows from the definition of the maximum tardiness. □

Example:

Suppose that a task set $T = \{T_1, T_2, T_3, T_4\}$ with each task T_i being characterized by an ordered 3-tuple (D_i, R_i, L_i), where D_i is the deadline, and R_i is cpu execution requirement, and L_i is the longest path length of task T_i respectively (Table 5.1). There are 2 processors are available to schedule the task set T.

Table 5.1 Bounds of trace table

T_i	(D_i, R_i, L_i)	u_1	Δ_i	u_2	$\hat{\gamma}_i$
T_1	(38, 30, 10)	$\frac{138-10}{2} + 10 = 74$	$\{T_3, T_4, T_2\}$	$\min\{(84 - 20), (83 - 18), (79 - 10)\} = 64$	64
T_2	(49, 38, 20)	$\frac{138-20}{2} + 20 = 79$	$\{T_3, T_4\}$	$\min\{(84-10), (83-8)\} = 74$	74
T_3	(68, 40, 30)	$\frac{138-30}{2} + 30 = 84$	$\emptyset$	—	84
T_4	(70, 30, 28)	$\frac{138-28}{2} + 28 = 83$	$\emptyset$	—	83
$\sum_{i=1}^{4} R_i = 138$					

It is easy to obtain u_1 for T_1 as follows:

$$u_1 = \left\lfloor \frac{\sum_{k=1}^{n} R_k - L_1}{M} \right\rfloor + L_1 = \frac{138 - 10}{2} + 10 = 74.$$

From Lemma 5, we construct $\Delta_1 = \{T_3, T_4, T_2\}$ ordered list, where $E_{3,1} = 20$, $E_{4,1} = 18$, and $E_{2,1} = 10$. Then, we calculate the smallest of the differencies between expected completion time and the corresponding $E_{j,1}$ as follows:

$$u_2 = \min\{(84 - 20), (83 - 18), (79 - 10)\} = 64.$$

Finally, $\hat{\gamma}_1 = \min\{74, 64\} = 64$. With the similar way, $\hat{\gamma}_2 = 74$, $\hat{\gamma}_3 = 84$, and $\hat{\gamma}_4 = 83$ (See Table 5.1). Now, we follow the definition of the maximum tardiness,

$$\tau^{LSTF} \leq \max\ \{0, (64 - 38), (74 - 49), (84 - 68), (83 - 70)\} = 26.$$

CHAPTER 6

INTEGRATION OF LSTF INTO REALISTIC REAL-TIME SYSTEMS

When we build a realistic system, we need to consider practical issues such as interrupts, timers, context switch, the scheduler, and communication because they consume system resources. We discuss how to integrate these issues with LSTF in Chapter 6.

6.1 Interrupts and Timers

There are two categories of real-time processes: event-driven and time-driven [36]. Event-driven processes generally are based on external hardware devices which generate interrupts to 'wake up' corresponding processes. An interrupt invokes an interrupt handler to recognize which task to execute. Time-driven processes use a periodic timer to generate interrupts. We can thus treat time-driven processes as a subcase of the event-driven. If a hardware interrupt priority is mapped to software task priorities then we name it an *integrated interrupt event*, otherwise a *non-integrated interrupt event* [45]. A non-integrated interrupt event consists of two sections; the interrupt and software sections, the interrupt section is responsible for handling the interrupt (e.g., acquisition of data) and sending the information to the software section to process. These two sections each have their own priorities.

Under the same assumptions as in Cesar [10], there is an interrupt for each task. Each interrupt is an integrated interrupt and consistent with the corresponding task period. In this case, we can simply add a weighted interrupt node to the root of PG for every event before translating PGs into UPGs. When an event becomes active, the processor starts to executes the interrupt part.

6.2 Scheduler Overhead and Context Switch

Let Λ_i denote the task multiset executed by the system at time i. When Λ_i is different from Λ_{i+1}, the system incurs a context-switch cost, W. When a processor performs a context switch, it needs to save and restore the entire processor state, no matter whether the restored states will be used or not before the next context switch. This procedure takes a lot of resources and time, and may cause LSTF to miss deadlines that have been met in a schedule with fewer context switches.

To study how to minimize the context switch overhead, we had better understand when the context switch occurs.

- Memory miss: Instructions reference the data which causes block misses in the cache or page faults in the main memory. In a distributed system, the system may across the network to get the data from a remote side.
- Preemption: The processor is preempted by the arriving interrupts or higher priority process.
- Remote procedure call: The program make a procedure call which is located at a remote side. If the program needs to wait for the result, the program should be be blocked.
- Synchronization: To assure consistency of variables and data structures during updates, locks are used.

When the system meet this situation, the processor has either to be idle (spinning) or to make a context switch to load the head process of the ready queue (blocking). The choice between spinning and blocking involves balancing the processor time lost for spinning against the processor time to the context switch. In [6, 11], ways of estimating an optimal limit on spinning time before blocking are explored.

In general, LSTF schedules may involve a large number of context switches. For example, if there are two identical list tasks (each with L associated subtasks) running on one processor, LSTF will flip-flop between these two tasks (performing at least L context switches). If the scheduler does too many unnecessary context switches, system performance (such as CPU utilization and feasibility of meeting deadline) will drop tremendously.

We propose two techniques to minimize the context switch overhead: soft-precedence edges and a modified heuristic, LSTF$^+$ [12]. The evaluation of these two techniques will be addressed in Chapter 7.

6.3 Soft-Precedence Edge

We introduce *soft-precedence edges* to coalesce two non-waiting segments, sequences of consecutive nodes without synchronization points, of the same UPG or different UPGs, into one Super UPG (SUPG). The SUPG may resemble, but is not necessary equivalent to, the original PG plus soft precedence edges. This helps the scheduler to decide which thread to run first, as well as to prevent flip-flop between threads with same space-time.

6.4 Further Definitions

- *Object*(V_i): the process or object containing graph node V_i
- *Non-waiting execution code from vertex* V_i, $\mu(V_i)$: the maximal single-entry subgraph of the nodes associated with *Object*(V_i) and containing V_i. i.e. if $V_i \in \mu(V_i)$ then
  1. V_i is reachable from V_i inside $\mu(V_i)$
  2. V_j is a node of *Object*(V_i)

3. if V_k reaches V_j on path P, then either $V_k \in \mu(V_i)$ or V_i is an internal node of P.

6.4.1 Example

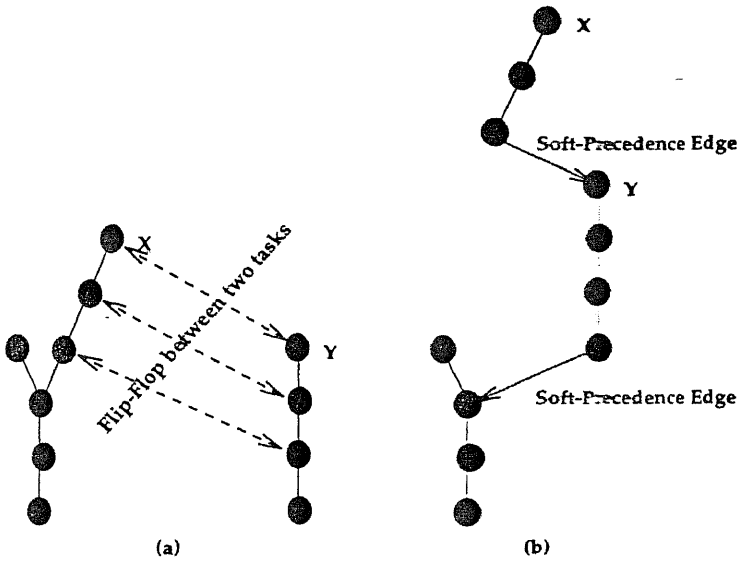

Figure 6.1 Soft-precedence

In Fig 6.1-a, there are two threads with initial vertex x (thread x) and y (thread y) are ready to execute. Based on definitions, $L(x) = 3$ and $L(y) = 4$. If thread x and y have to execute on the same processor, then obviously only one can be executed . *Soft-precedence* uses $L(a)$, $L(x)$, $S(a)$ and $S(x)$ to decide which one should execute first. A soft-precedence edge is constructed from lower S to higher one. If two threads have the same S, the thread with higher L runs first; if L's are the same, the choice is arbitrary. For example, if $S(x)$ is smaller then $S(y)$, a directed soft-precedence edge is built from $\mu(x)$ to $\mu(y)$ (Figure 6.1-b).

The use of soft-precedence edges reduces the number, and thus the cost, of context switches and simplifies scheduler migration and execution decisions. In Figure 6.1-a, 'flip-flop' execution will occur between the tasks with initial vertices X and Y, since they have the same space-time. This situation results in numerous

context switches. Using soft-precedence edges to construct a SUPG (Figure 6.1-b), execution will entail only two context switches instead of six.

When one non-waiting contiguous segment finishes execution, the scheduler switches the CPU from this finished segment to a new group of non-blocking segments.

6.4.2 LSTF$^+$ Scheduler

We propose a modified heuristic, LSTF$^+$ for scheduling real-time tasks under context switch overheads. First, LSTF$^+$ computes space time in a manner identical to LSTF. LSTF$^+$ maintains a priority queue Q of all ready subtasks prioritized by space time . Let m denote the number of processor available. At any time t, let η denote the set of (at most m) subtasks scheduled for execution. We define two vertex sets: Y is the set of all "fresh" ready subtasks at the start of the next-time slot (i.e., those ready subtasks which have at least one predecessor in η), and Z is the set of (at most m) smallest modified deadline ready subtasks. Let W denote the overhead of context switch (measured as $\frac{the\_time\_for\_context\_switch}{the\_time\_for\_quantum}$), and S_y denotes the highest space time in Y, and S_z denotes the smallest space time in Z. The pseudo-code for LSTF$^+$ scheduler is as follows (each iteration of the which loop coresponds to the scheduling of another time quantum):

LSTF$^+$ Scheduler

construct Q;
while ($Q \neq \emptyset$) {
  Determine new Y, Z, S_y and S_z;
  if ($W > X$) { /* context switch cost is above threshold */
    if ($S_y \leq S_z + K$) {
      $C = Y$;
    else

```
            C = Z;
        }
    }
    else { /* context switch cost is below threshold */
        C = Z;
    }
    schedule all the members of C;
    delete them from Q;
    insert all ready children of C into Q
}
```

LSTF$^+$ handles context switch through these two control parameters (X and K). X is the threshold for context-switch cost that determines whether LSTF$^+$ makes the same decision as LSTF or not. If $W \leq X$, LSTF$^+$ behaves the same way as LSTF does. When W exceeds X, the parameter K quantifies the degree to which LSTF$^+$ favors continuing to execute the current tasks rather than paying for a context switch. Adjusting these two values (X and K), we can improve the performance of LSTF$^+$ and satisfy the feasibility requirement.

Theorem 7 *Let LSTF$^+[X, K]$ denote LSTF$^+$ with threshold X and degree K. LSTF$^+[0.5, 1]$ dominates LSTF for scheduling list tasks in the sense of feasibility.*

Proof:

We consider a set of tasks that is successfully scheduled by LSTF, and prove that LSTF$^+$ with $X = 0.5$ and $K = 1$ will successfully schedule this system as well. At time $(i+1)$, we assume that $(m-1)$ ready nodes gain the processors and node k and p compete for the last processor. When the condition $(S_k < S_p)$, $(S_k > S_p + 1)$, and $(S_k = S_p + 1$ and $W \leq 0.5)$, LSTF$^+$ has the same execution sequence as LSTF. Now, we discuss the rest of cases.

- $S_k = S_p$ (Figure 6.2-Case 1): LSTF picks up either node k or p randomly. If LSTF chooses k, LSTF is the same as LSTF$^+$ Suppose that LSTF schedules p first and, finishes p at $(i+2+W)$; node k is completed at $(i+3+2W)$. Let $C_{(a,b)}$ denote the completion time of subtask b under algorithm a.

$$\begin{aligned} S_p = S_k &\geq \lceil C_{(LSTF,k)} \rceil = \lceil i+3+2W \rceil \\ &\geq C_{(LSTF^+,p)} \geq C_{(LSTF^+,k)} \end{aligned} \tag{6.1}$$

From Equation 6.1, LSTF$^+$ will finish them on time.

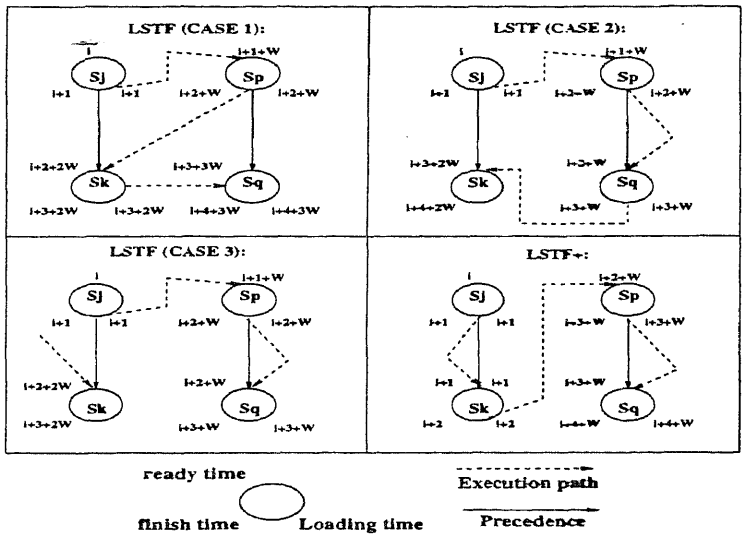

Figure 6.2 Execution path of LSTF and LSTF$^+$

- $S_k = S_p + 1$ and $W > 0.5$ (Figure 6.2): LSTF may have three possible execution sequences.

  Case 1: LSTF executes as $p \rightarrow k \rightarrow q$

  This situation is the same as the previous category.

  Case 2: LSTF executes as $p \rightarrow q \rightarrow k$

  From the execution path, we know $S_k = S_q$ and $S_p = S_k - 1$.

$$S_k = S_q \geq \lceil C_{(LSTF,k)} \rceil = \lceil i+4+2W \rceil$$

$$\geq C_{(LSTF^+,q)} \geq C_{(LSTF^+,k)}$$

$$S_p = S_k - 1 \geq \lceil i + 3 + 2W \rceil$$

$$\geq i + 3 + W = C_{(LSTF^+,p)}$$

Case 3: LSTF executes q and k in parallel as $p \rightarrow (q \mid\mid k)$ (more processors for execution and $S_k \leq S_q$)

$$S_q \geq S_k \geq \lceil C_{(LSTF,k)} \rceil = \lceil i + 3 + 2W \rceil$$

$$\geq i + 4 + W = C_{(LSTF^+,q)} \geq C_{(LSTF^+,k)}$$

$$S_p = S_k - 1 \geq \lceil i + 2 + 2W \rceil$$

$$\geq i + 3 + W = C_{(LSTF^+,p)} \ \square$$

The following example illustrates how LSTF$^+$[0.5, 1] improves upon LSTF. Suppose that there are M processors and N ($N = K * M$) identical list tasks each with L levels. Let W be greater than 0.5. LSTF would have $(K-1) * L$ context switches, but LSTF$^+$ would have only $(K-1) * \lceil \frac{L}{2} \rceil$. With this example, the LSTF$^+$[0.5, 1] scheduler achieves a 50% reduction in context-switch cost.

6.5 Communication

To handle communication, we add a new type of node representing communication, where the vertex's weight is set to the cost of call request signal plus transmission time requirement plus acknowledgement signal. The unit of communication nodes should be compatible with processor nodes. This may result in a smaller unit and larger graph. Alternatively, the larger grain size can be used, at the cost of larger communication or processor nodes. For example, suppose two procedure calls are made by an application program, A→B and A→C (Figure 6.3-a). After the assignment algorithm is applied, we can distinguish whether the procedure call is local or remote. If the

caller and callee are located on different processors, the caller should make a Remote-Procedure-Call (RPC) to the callee (for example, A→B). The weights of communication nodes are dependent on a number of parameters (such as the network capacity, the size of message, ..., etc.). One sending node (in front of procedure body) and one returning node (at back of procedure body) are added to the PG (Figure 6.3-b). These nodes are responsible for sending and returning arguments of the procedure respectively.

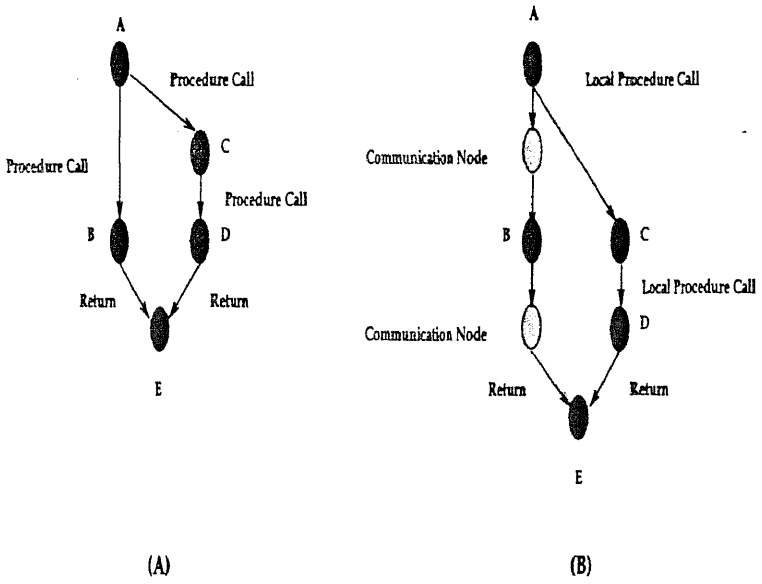

Figure 6.3 Adding new communication nodes corresponds to a RPC call

We can schedule network nodes based on our network assumption. A network link is treated as a virtual processor. In the fully connected network case, the unit network node can be scheduled without any delay. In the bus case, only one process at a time can use the bus, so soft-precedence edges will be needed. The major differences between a CPU and a network node is the restriction on in-degree; the degree of a network node always is equal one, but degree of CPU node can be greater.

CHAPTER 7

SIMULATION RESULTS

We have implemented a prototype symbolic simulator for a preliminary evaluation of LSTF performance. With different parameters of the workload generator, a set of benchmarks are used to compare LSTF with other established dynamic scheduling algorithms in our simulation study. Three benchmarks are used:

- Ideal case: There is no cost except execution requirement.
- Context overhead: The cost of context switch overhead is added to costs of the ideal case.
- Communication: There is a communication cost between subtasks, paid if segments execute on different processors.

7.1 Workload Generator

We have implemented a workload generator to produce in-tree-like structured tasks (each node representing as a subtask). The set of workload parameters is defined as (N, M, L, B): N is the total number of tasks (each task with a single sink node); M is the the number of processors; L is the maximum PG depth of a task; and B is the largest possible number of ancestors for a node. The pseudo-code for the in-tree generator is showed as following:

The Single Sink In-tree Generator

```
for (i = 1; i ≤ N; i++) {
    put sink in queue Q;
    while(Q ≠ ∅) {
        q = first (Q);
```

```
                Preds = level(q)/L;
                if (Preds ≥ rand(0,1)) {
                        Num_Preds = rand (1, B);
                        insert all Num_Preds children into Q;
                        delete q from Q;
                }
        }
}
```

To obtain precedence graph (PG) structures, we add a parameter, E (the ratio of cross edges to nodes), to the original parameter set. We randomly pick one pair of source and destination nodes as two end-points of one cross edge, where $the\_level\_of\_source > the\_level\_of\_destination$ and there does not exist a directed edges between them. Each subtask (node) will be assigned with an execution weight generated by a truncated non-negative binomial distribution as defined in Equation 7.1. We have used $r = 4$, $p = 0.4$ and $X = 28$ for our simulation study.

$$P(x = k) = P_k = \begin{cases} C_{(r-1)}^{(k-1)} P^{(r-1)} Q^{(k-r)} & \text{if } r \leq k < X \\ 1 - \sum_{i=r}^{X-1} P_i & \text{if } k = X \\ 0 & \text{otherwise} \end{cases} \quad (7.1)$$

After obtaining the graph-like tasks, each task will be assigned with a random deadline as defined in Equation 7.2.

$$D_i = \min\left(\left(L_i + \frac{w_i - L_i}{M^{0.4}} + \frac{\omega - w_i}{M^{1.2}}\right), \left(w_i \left(\frac{N}{M}\right)^{0.5} + F(N^{0.25})\left(L_i + \frac{w_i}{M}\right)\right)\right) \quad (7.2)$$

where w_i is the total weight, and L_i is the weight of the longest path for T_i respectively, ω is equal to $\sum w_i$, and F is the normal distribution (0,1).

Finally, a parameter for communication cost (U) is introduced. Let w_i is the total weight of cpu requirement and c_i is the total weight of communication requirement for T_i respectively. Four communication load conditions are discussed.

1. Light amount of communication (computation bound): $\frac{\sum c_i}{\sum w_i} < 0.25$
2. Moderate amount of communication: $0.25 \leq \frac{\sum c_i}{\sum w_i} < 0.75$
3. Heavy amount of communication : $0.75 \leq \frac{\sum c_i}{\sum w_i} < 1.25$
4. Super-heavy amount of communication (IO bound): $1.25 \leq \frac{\sum c_i}{\sum w_i} < 15$

The complete parameter set is (N, M, L, B, E, U); we will test each benchmark through varying each parameter, alone and in certain combinations.

7.2 Benchmark I

We simulate four established (HLF, LLF, EDF and MPF), one linear integrated heuristic algorithm (H3) and two other heuristic algorithms (H1 and H2) in this benchmark. Let $P(v)$ denote the priority of vertex v, where the vertex v with lowest $P(v)$ has highest priority. The definitions of various heuristics are listed as follows:

- Earliest-Deadline-First (EDF): $P(v) = D_v$.
- Highest-Level-First (HLF): $P(v) = -L_v$.
- Minimum Processing-Time First (MPF): $P(v) = A(v)$, where $A(v)$ is the requested computation time of $TASK_v$.
- H1 (combination of EDF, HLF and LLF): each vertex has a priority function of processor number, m and fan-out degree, f. Intuitively, the completion time of $TASK_v$ should be greater than or equal to $\frac{RX_v}{\min(m,f)}$ and L_v, where RX_v is the remaining computation time of $TASK_v$. The heuristic calculates the $P(v)$ as follows: $P(v) = D_v - \max(L_v, \frac{RX_v}{\min(m,f)})$.
- H2 (combination of EDF, HLF and LLF): similar to H1, the heuristic calculates the $P(v)$ as follows: $P(v) = D_v - (L_v + \frac{RX_v - L_v}{\min(m,f)})$
- H3 (combination of EDF and MPF): $P(v) = D(v) + A(v)$
- Least-Laxity-First (LLF): $P(v) = D_v - RX_v$
- CPM: the non-preemptive mode of LSTF, with the same $P(v_i^t) = S(v_i^t) = D(v) - L(v) - t$.

We have implemented a prototype symbolic simulator (without considering overhead) for a preliminary evaluation of LSTF performance. Tasks are free to execute on any processor; e.g., if *task i* is executing on processor k at time t, then it can continue execution on any processor(s) at time *(t+1)* without extra cost.

Different applications have different performance criteria, so we compared LSTF with the above heuristics with respect to the number of tasks missing their deadline (ψ), tardiness (τ), and make-span (η). We run 200 different seeds of one particular parameter set (10, 10, 7, 3, 2, 0) on 10 processors. The average results of 200 simulations with 95% confidence intervals (CI) are tabulated in Table 7.1. From Table 7.1, we observe that EDF is good at minimizing missing tasks; HLF is good at minimizing make-span; LSTF is good at minimizing tardiness. In fact, LSTF and HLF appear optimal with respect to the appropriate metrics among these algorithms with 95 % confidence, and EDF is better than all but H3 with 95 % confidence.

Table 7.1 The performance of various heuristics

Algorithm	ψ		τ		η	
	Mean	95% CI	Mean	95% CI	Mean	95% CI
LSTF	6.66	[6.32, 7.00]	25.11	[23.58, 26.65]	150.07	[146.08, 154.07]
HLF	8.93	[8.83, 9.03]	125.48	[121.75, 129.21]	132.10	[128.01, 136.19]
LLF	9.32	[9.22, 9.42]	70.69	[68.56, 72.82]	147.54	[143.35, 151.74]
EDF	5.70	[5.47, 5.94]	29.12	[27.46, 30.79]	155.06	[150.93, 159.19]
MPF	6.34	[6.06, 6.62]	36.18	[33.97, 38.39]	155.07	[150.97, 159.17]
H1	8.47	[8.20, 8.73]	30.05	[28.44, 31.65]	151.48	[147.41, 155.55]
H2	9.14	[8.97, 9.31]	42.90	[41.12, 44.67]	150.46	[146.39, 154.52]
H3	5.89	[5.64, 6.13]	33.35	[31.45, 35.24]	155.07	[150.97, 159.18]
CPM	7.05	[6.71, 7.29]	34.70	[34.59, 34.81]	163.15	[157.38, 168.87]

We have compared the performance of these algorithms by varying each element of the parameter set, without including any other costs, such as communication and context switch overhead.

Each of the points in the following simulation results represent the average data of 100 random graphs (with the generator initialized at time-dependent seeds). The performance penalty is measured as $\frac{A(\tau^x)-A(\tau^{LSTF})}{A(\tau^{LSTF})}$, where $A(\tau^x)$ is the mean τ^x of 100 runs ($\frac{\sum \tau^x}{100}$).

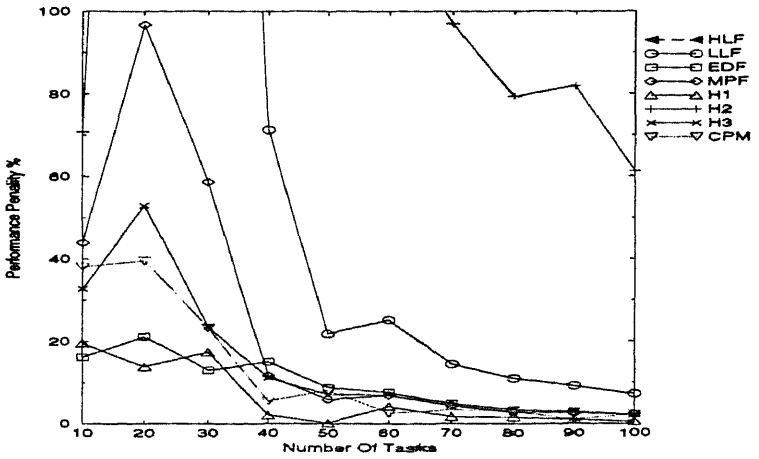

Figure 7.1 Effect of number of tasks (N) in benchmark I

In the following simulations, we randomly choose 10 processors for our simulation study. To observe a considerable difference between LSTF and other algorithms (such as H1, H3 and CPM), the number of tasks should be less than 40 (Figure 7.1). Except for HLF and LLF, there have similar results for $N = 10$, $N = 20$ and $N = 30$. Simply, we take ($N = 10$, $M = 10$) as our test-seed. Also, we can see the clear comparisons when L is greater than or equal to 6 from Figure 7.3. Some target algorithms (such as MPF, H1, H3 and EDF) have a "constant" difference between LSTF for the range (from $L = 7$ to $L = 10$). The value of factor L is collected as our test-seed. Under certain range of B (not too small or large; $B = 3$ or $B = 4$), LSTF has remarkable difference between LSTF and target algorithms (Figure 7.4). Therefore, the parameter set (10, 10, 7, 3, E, 0) is chosen for observing the effect of E.

7.2.1 Single Factors

The Number of Tasks, N: Figure 7.1 shows the relation between the number of tasks and the performance penalty $(\frac{\tau_x - \tau_{LSTF}}{\tau_{LSTF}} \times 100)$, where τ_x is the maximum

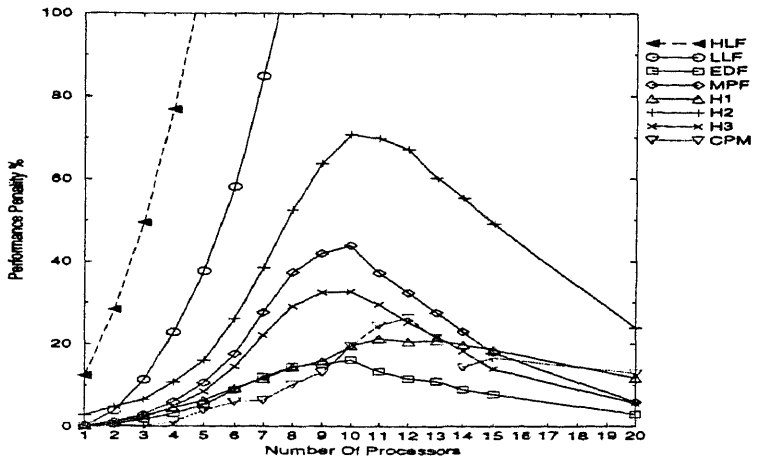

Figure 7.2 Effect of number of processors (M) in benchmark I

tardiness when scheduled by algorithm x. We use the parameter set (N, 10, 7, 3, 2, 0) for workload generator. When $\frac{N}{M} > 4$, the performance difference between LSTF and target algorithm is not great. Except for HLF, H2 and LLF, the heuristics do not pay a high penalty relative to LSTF under heavy load conditions. EDF and H1 are only lightly affected by factor N; they have a steady relative performance with LSTF.

The Number of Processors, M: Figure 7.2 shows the relation between the number of processors and the performance penalty $(\frac{\tau_x - \tau_{LSTF}}{\tau_{LSTF}} \times 100)$. We use the parameter set (10, 10, 7, 3, 2, 0) for workload generator, and run on different numbers of processors. When the system has a single processor, LSTF has the same performance as LLF, EDF and H1. With increase in the number of processors, the gain versus LLF increases rapidly, because LLF does not consider parallelism at all. Also, the gain versus other algorithms increases with increasing the number of processors, but declines as the number of processors becomes very large relative to

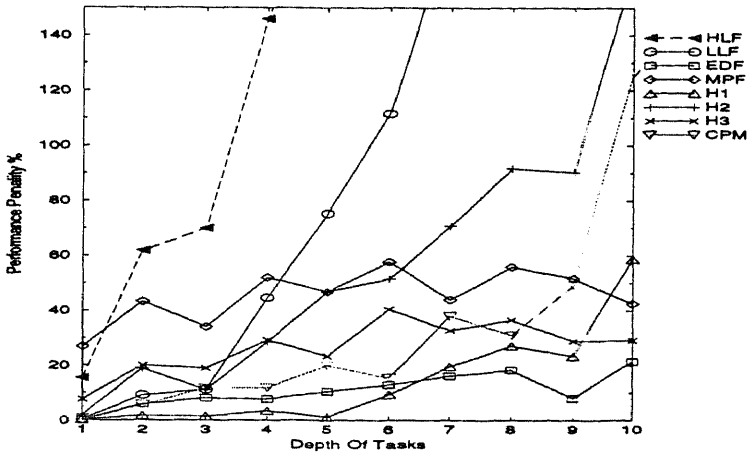

Figure 7.3 Effect of depth of task (L) in benchmark I

the number of tasks. If the number of processors is infinite, the schedule for LSTF is the same as the other algorithms which schedule the task level by level.

Maximum Task Depth, L: Figure 7.3 shows the relation between the depth of task and the performance penalty $(\frac{\tau_x - \tau_{LSTF}}{\tau_{LSTF}} \times 100)$. We use the parameter set (10, 10, L, 3, 2, 0) for workload generator. EDF and MPF have steady relative performance to LSTF; the other heuristics have increased penalty as L increases. When L = 1 (each task has one or two levels), LLF, EDF, H1 and CPM have the same performance as LSTF.

Maximum Branching Factor, B: Figure 7.4 shows the relation between the branches of tasks and the performance penalty $(\frac{\tau_x - \tau_{LSTF}}{\tau_{LSTF}} \times 100)$. For low density of cross edges, vertices with high space time can not obtain processors in general. For high density of cross edges, high space time vertices may be executed because

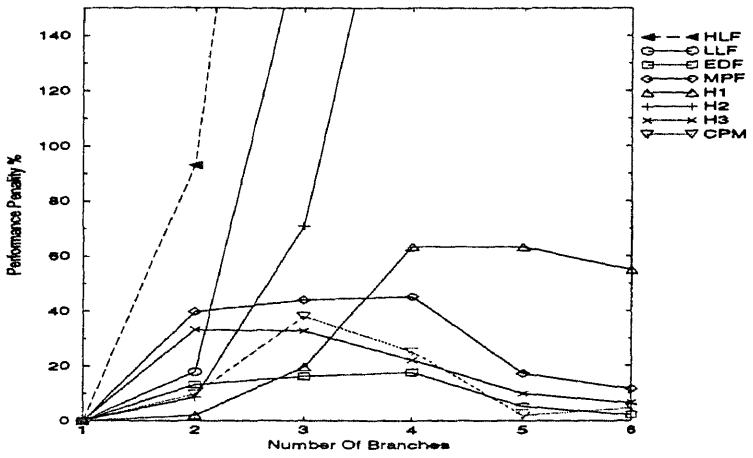

Figure 7.4 Effect of number of branches (B) in benchmark I

least space time tasks needed to synchronize. Thus, CPM (non-preemptive mode) has troubles in scheduling high density cross edges graphs.

Why does the performance penalty for CPM vary so irregularly with high E ($E > 4$)? Actually, the penalty is in a certain range (110 - 250). The density of cross edges is defined as the ratio of the number of cross edges to the number of in-tree edges. For example, considering a list (N nodes and N-1 in-tree edges) and $N = 8$, the maximum density of different cross edges will be (N-1)/2 = 3.5. Because we allow parallel cross edges in task graphs, the density of distinct cross edges may be higher in a task graph with lower total density of cross edges; we surmise that this is the major cause of this phenomenon.

We took the longest path = 7 as an input parameter of the workload generator so that the density of different cross edges will be less 3.5 (we do not build the cross edges between two nodes with the same level). Therefore, there are three major range for the penalty of CPM in the Figure 7.4:

1. $0 \leq E \leq 1$, CPM almost has the same results as LSTF.

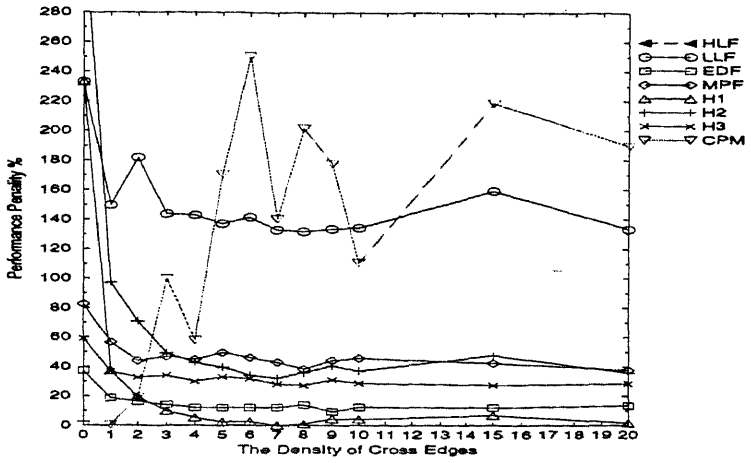

Figure 7.5 Effect of cross edge (E) in benchmark I

2. $2 \leq E \leq 4$, CPM has penalty between 20 and 100.

3. $5 \leq E$, CPM has penalty between 110 and 250.

The penalty of HLF, LLF and H2 increases sharply when $B \geq 3$. The rest of the algorithms keep a certain range of relative performance. The number of branches indicate the number of parent processes needed to be synchronized. With a low number of branches, H1 performs similarly to LSTF. When the number of branches increases, parallelism drops. Due to considering fan-out degree only (no fan-in consideration), the penalty for H1 is large when B is high. On the other hand, the penalty for CPM decreases when B is high.

Cross-Edge Density, E: Figure 7.5 shows the relation between the density of cross edges and the performance penalty $(\frac{\tau_x - \tau_{LSTF}}{\tau_{LSTF}} \times 100)$. We use the parameter set (10, 10, 7, 3, E, 0) for workload generator to run on 10 processors, and increase E step by step. When $E = 0$, H1 is the same as LLF, because $P(v) = D_v - \max(L_v, \frac{RX_v}{\min(m,f)}) = D_v - RX_v$. The parameter E does not affect the performance a

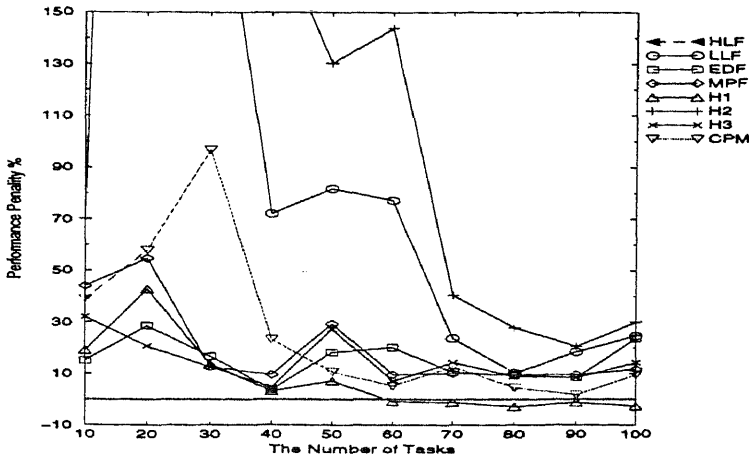

Figure 7.6 Effect of N and M (N = 3M) in benchmark I

lot, i.e., LSTF has steadily better performance as we go from trees to the complex-structure graphs. We also can see the clear performance difference between non-preemptive and preemptive mode when E is greater than 5.

7.2.2 Combination of Factors

Scaling and Changing both N and M: Figure 7.6 shows the relation between the combination of the numbers of tasks and processors and the performance penalty ($\frac{\tau_x - \tau_{LSTF}}{\tau_{LSTF}} \times 100$). For $N = 3M$, we use the parameter set (N, M, 7, 3, 2, 0) for workload generator to run on M processors, and increase N step by step There is no particular pattern in this simulation, but LSTF has 10% better performance than most of the algorithms, although H1 is competitive. We have a similar result when $N = 5M$.

Figure 7.7 shows the relation between the combination of the numbers of tasks and processors and the performance penalty ($\frac{\tau_x - \tau_{LSTF}}{\tau_{LSTF}} \times 100$). Under $N = M - 2$, we use the parameter set (N, M, 7, 3, 2, 0) for the workload generator to run on M processors, and increase N step by step. Every algorithm has steady relative

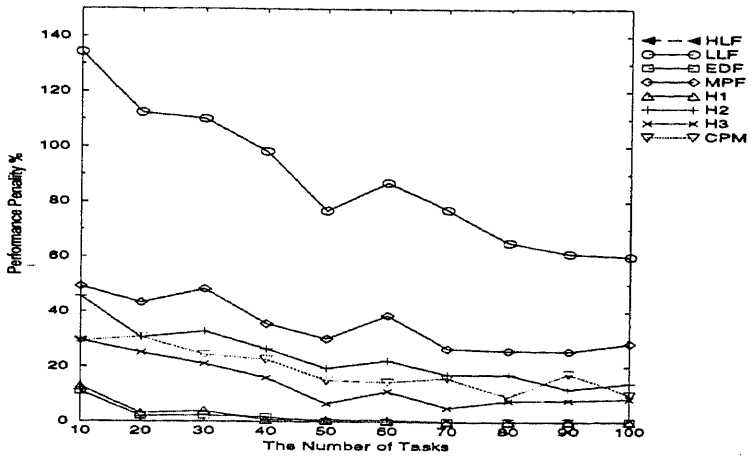

Figure 7.7 Effect of N and M ($N = M - 2$) in benchmark I

performance to LSTF. H1 and EDF perform very close to LSTF under a lightly loaded condition.

Tree vs. Cross Edge, B and E: Figure 7.8 shows the relation between the relative density of tree cross edges and performance penalty ($\frac{\tau_x - \tau_{LSTF}}{\tau_{LSTF}} \times 100$). With $B + E = 10$, we use the parameter set (10, 10, 7, B, E, 0) for workload generator to run on 10 processors, and increases B step by step. We have two observations: (1) HLF, LLF, H2 do very badly as B increases. (2) EDF,MPF, H3, CPM do well at either low or high B, but perform badly in the middle ($B = 3$ or $B = 4$).

Figure 7.9 shows the relation between the combination of density of cross edges and number of branches and performance penalty ($\frac{\tau_x - \tau_{LSTF}}{\tau_{LSTF}} \times 100$). For $E = 3B$, we use the parameter set (10, 10, 7, B, E, 0) for workload generator to run on 10 processors, and increase B step by step. The results are similar to the case of $B + E = 10$, but CPM has different relative performance from Figure 7.8 (the

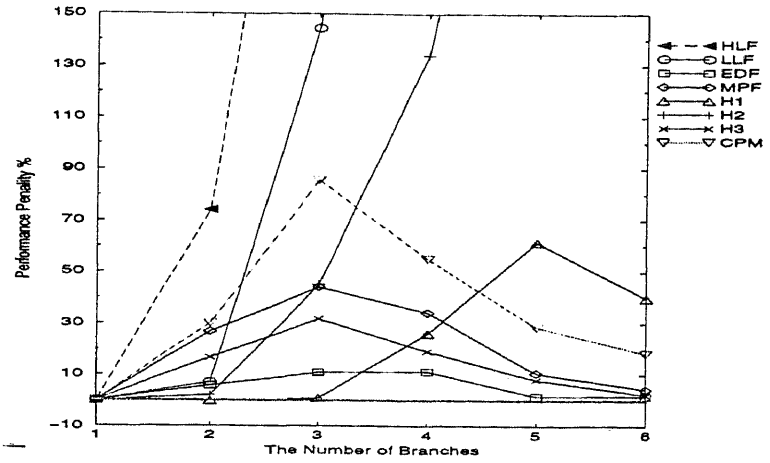

Figure 7.8 Effect of B and E (B + E = 10) in benchmark I

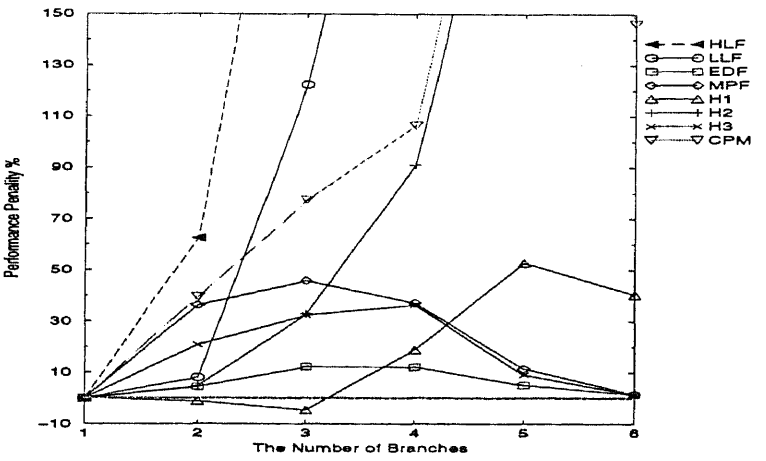

Figure 7.9 Effect of B and E (E = 3B) in benchmark I

relative performance of CPM is not bell-shaped; it does not droop as B increases). We conclude that E has more influence on CPM than B.

7.2.3 Insight from the Simulation

Under light load ($N = M - 2$), all algorithms have steady relative performance to LSTF. Although there is no particular pattern in a heavy load condition, LSTF has better performance (10%) than other algorithms (except H1). The factors B and E have a lot of influence on CPM and H1. H1 performs well with a low value of B (Figure 7.4) and CPM does well with low E (Figure 7.5). We also observe that E has more influence on CPM than B (Figure 7.8 and 7.9). EDF and MPF are not sensitive to factor L. Except for H1, LSTF performs better than the target algorithms under our simulations (and H1 only performs well with low B).

7.3 Benchmark II

From the previous simulation study, LSTF works well without overhead, but does not do well with context switch overhead. For example, if two tasks have same space time, both tasks will execute at the same rate under LSTF until one finishes. Thus, there will be large numbers of context switches between tasks with same space time. In this benchmark, we compare target algorithms with original LSTF and L-S (integrated LSTF and soft-precedence edge) and additional modifications of LSTF ($LSTF^+$) in presence of context switch overhead.

To achieve preemption in practice, a hardware interrupt clock is typically set for a certain quantum period. Using this interrupt clock, the CPU regains control and decides whether the first process of the ready queue gains control based on scheduling policy. A good quantum size is essential to the performance of operating systems [21]. If the quantum size is infinity, the tasks become non-preemptable. For small quanta, the context-switch overhead becomes unacceptable. For a practical system, the time for context switch is around 10,000 *ns*, and the quantum time is around 100,000,000 *ns* [57]. Thus, the overhead per quantum is around 0.1%. Arguably, real-time systems will require finer context switch granularity in order to meet deadlines, so we have explored an overhead range of 0 to 10 %.

Table 7.2 shows the relation between the context overhead and the performance penalty ($\frac{\tau_x - \tau_{LSTF}}{\tau_{LSTF}} \times 100$). Each attribute represents the average of penalty of 100 different seeds of simulations with the parameter set (10, 3, 7, 3, 2) running on 3 processors. As the context switch overhead grows, the penalty for EDF, H1, H2, and H3 decreases, and the gain for L-S increases. We observe that the soft-precedence edge technique helps LSTF when context-switch overhead is large.

In the following results, we omit the simulations of HLF, LLF, and H2 because of poor performance; on the other hand, various versions of $LSTF^+$ have been added.

Table 7.2 The performance of various algorithms with context switch overhead

Algorithm	Overhead to Quantum						
	0%	1%	2%	3%	4%	5%	10%
HLF	1669.3	1658.0	1646.9	1636.6	1625.8	1615.3	1564.5
LLF	157.4	165.0	172.4	179.7	186.8	193.7	227.8
EDF	13.2	11.9	10.6	9.3	8.1	6.9	1.2
MPF	47.4	45.5	43.7	42.0	40.3	38.6	30.7
H1	22.7	23.0	23.3	23.6	23.8	24.1	25.0
H2	136.1	129.1	122.3	115.7	109.3	103.0	72.9
H3	17.5	16.1	14.7	13.4	12.1	10.8	4.7
CPM	8.5	7.3	6.4	5.5	4.8	4.0	-1.7
L-S	1.7	0.8	0	-0.8	-1.6	-2.4	-6.1

The Context Switch Overhead, W: Figure 7.10 shows the relation between the context switch overhead and the performance penalty $(\frac{\tau_x - \tau_{LSTF}}{\tau_{LSTF}} \times 100)$. Let W denote the overhead of context switch (measured as $\frac{the\_time\_for\_context\_switch}{the\_time\_for\_quantum}$). We use the parameter set (10, 10, 7, 3, 2) for the workload generator, and simulate different overhead platforms. We assign X to the value of W for the LSTF$^+[X, K]$. Not surprisingly, LSTF has the best performance for an overhead-free environment. With even a small context switch overhead, LSTF's performance is not acceptable (all other heuristics are better than LSTF). We observed that LSTF$^+[X, 6]$ has the best general performance when context-switch overheads are considered.

The Maximum Branching Factor, B: Figure 7.11 shows the relation between the number of branches and the performance penalty $(\frac{\tau_x - \tau_{LSTF}}{\tau_{LSTF}} \times 100)$. We use the parameter set (10, 10, 7, B, 2) for the workload. With small B, all tasks have very simple task structure, so there is no difference among heuristics. When B is greater than 2, the performance of various LSTF$^+$ varies widely for varying (X, K) values, and seems to follow no definite trend.

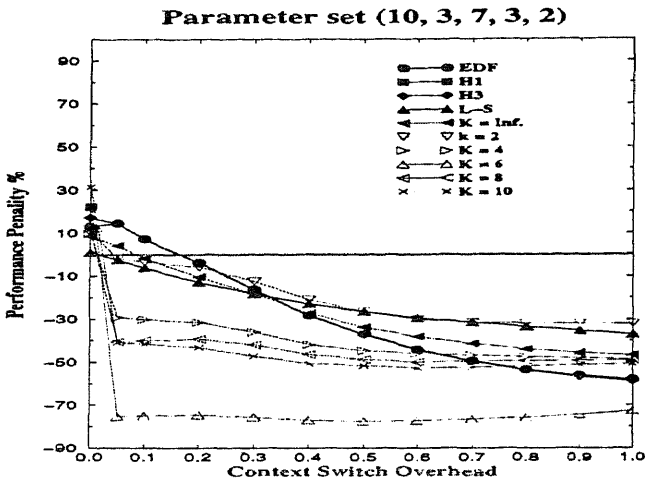

Figure 7.10 Effect of context switch overhead

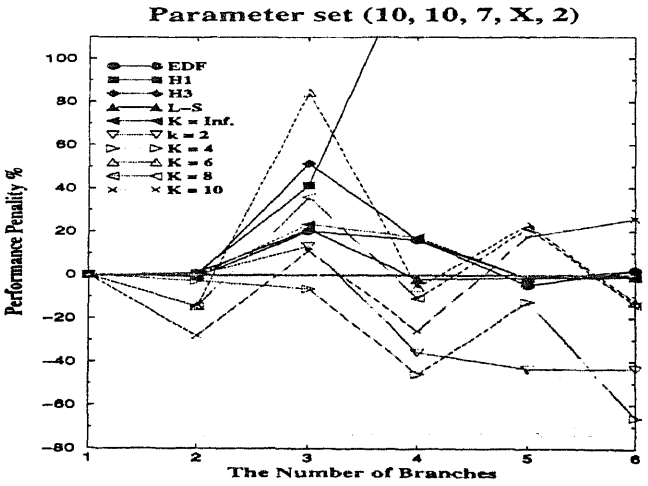

Figure 7.11 Effect of factor B

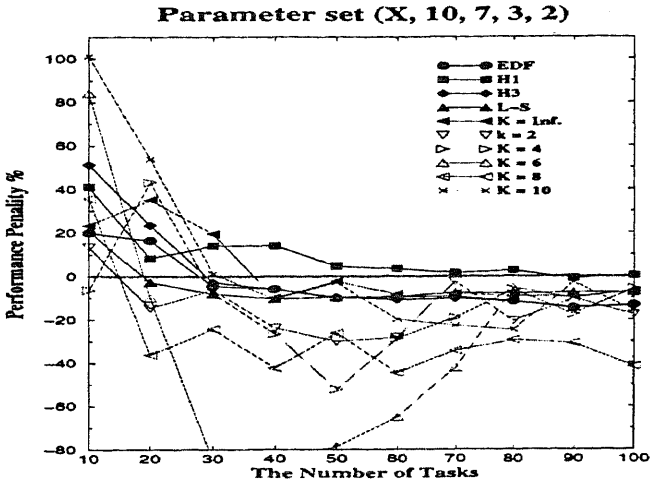

Figure 7.12 Effect of factor N

From Figures 7.10 and 7.11, we conclude that the selection of the appropriate K is sensitive to statistical properties of the task distribution, i.e., indicate a need for profiling (to estimate these properties) and simulation (to select K).

The Number of Tasks, N: Figure 7.12 shows the relation between the number of tasks and the performance penalty $(\frac{\tau_E - \tau_{LSTF}}{\tau_{LSTF}} \times 100)$. We use the parameter set (N, 10, 7, 3, 2) for the workload generator, and run on 10 processors with 0.1 overhead. When $N \approx M$, the performance of various LSTF$^+$ varies widely for all chosen (X, K) values. This implies that there exist enough processors for critical subtasks, and LSTF$^+$ cannot get a lot of benefit from this workload. The profit for LSTF$^+$ increases as N increases until the number of tasks reaches a certain value. After that, the profit for LSTF$^+$ decreases because the denominator of the performance penalty computation is big.

The Number of Processors, M: Figure 7.13 shows the relation between number of processors and performance penalty $(\frac{\tau_E - \tau_{LSTF}}{\tau_{LSTF}} \times 100)$. We use the parameter set

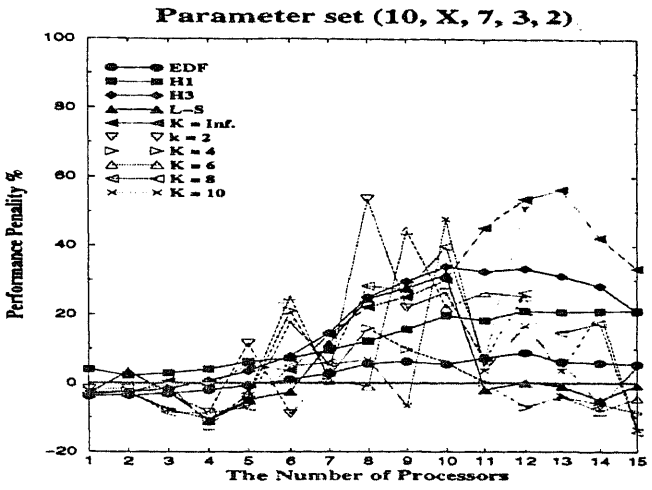

Figure 7.13 Effect of factor M

(10, 10, 7, 3, 2) for the workload generator, and run on different numbers of processors with 0.1 overhead. From Figure 7.13 and corresponding graphs for other parameter sets, we observe that graphs can be divided into three regions: $M \ll N$, $M \approx N$, and $M \gg N$. In the first and third regions, choosing an appropriate K results in uniformly better performance; in the second region, the relation between algorithms is highly unpredictable. Intuitively, when $M \ll N$, intertask contention results in frequent context-switch; when $M \gg N$, an algorithm which accounts for context-switch can better take advantage of intratask parallelism. For the current task set, with a low edge density ($B + E = 5$), $M \approx N$ more-or-less assigns each task to a single processor. We have observed that for higher $B + E$, LSTF$^+$ algorithms again does better when $M \approx N$; we suspect this reflects the frequent need to schedule multiple predecessors of critical nodes.

Scaling & Changing both N and M: Figure 7.14 shows the relation between the combination of the numbers of tasks and processors and the performance penalty ($\frac{\tau_x - \tau_{LSTF}}{\tau_{LSTF}} \times 100$). We use the parameter set (N, M, 7, 3, 2) for the workload generator,

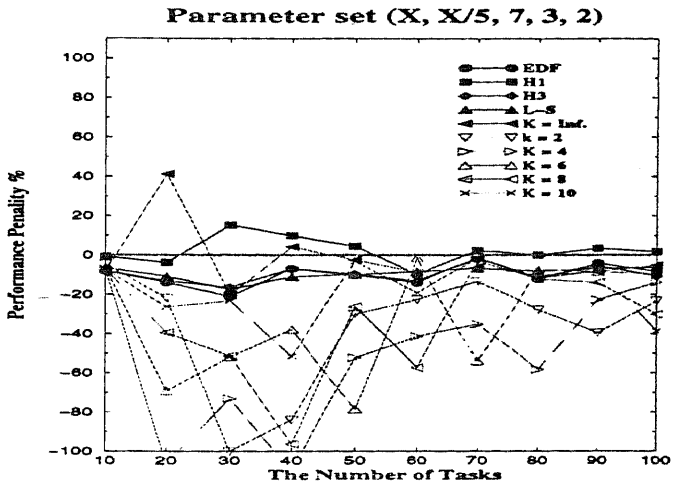

Figure 7.14 Effect of factor N and M ($N = 5M$)

where $N = 5M$, and run on M processors with 0.1 overhead. We observe that LSTF$^+$ has better performance most of the time. Again, the profitability of LSTF$^+$ drops when N is large because the performance penalty denominator is big.

7.3.1 Insight from the Simulation

For idealized platforms where context-switch operations incur no cost, LSTF is known to perform extremely well. An enhanced version, LSTF$^+$, is proposed for environments in which context switch incurs an overhead cost. To obtain a feasible schedule, two parameters (X and K) are adjusted for different task and machine properties. From the simulation results, we conclude that the selection of the appropriate parameters for LSTF$^+$ is sensitive to statistical properties of the task distribution.

7.4 Benchmark III

This benchmark investigates the model where communication cost is considered. The tasks should be fixed on host processors and there exist inter-process communications between subtasks. The arcs of PG represent inter-process communication relationship and a weight associated with each arc represents the data volume. For example, $\alpha \xrightarrow{\lambda} \beta$ denotes that subtask α sends data volume λ to subtask β. If subtasks α and β reside on different hosts, the inter-process communication incurs a cost.

7.4.1 The Factor of Workload Generator Parameters

We analyze the workload parameter under the "heavy amount of communication" condition ($0.75 \leq \frac{\sum c_i}{\sum w_i} < 1.25$). There are two heuristics added to our simulation study.

- LSTFC: After transforming a arc into a communication node with a communication cost, PG consists of two kind of nodes, cpu and communication, as described in Section 6.5. LSTFC calculates the space time based on the new transformed PG (i.e., LSTFC takes account of the communication weight for the critical path).

- Static LSTF (LSTFS): The heuristic assigns the static space time at the entrance of the node and keeps the same space time while the node is being executed.

After obtaining the data from the workload generator, we assign every subtask (vertex) randomly. Then, we run the symbolic simulator with different algorithms (such as LSTFC, LSTFS, LLF, EDF, and various LSTF$^+$). In this simulation benchmark, the processors are fully connected to each other and communication is contention-free. Similar to the previous benchmarks, each of the points in the following simulation results represents the average data of 100 runs.

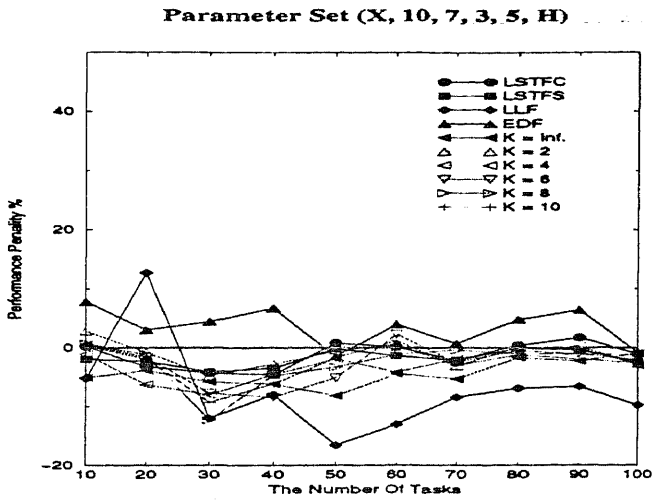

Figure 7.15 Effect of factor N

The Number of Tasks, N: Figure 7.15 shows the relation between the number of the tasks and the performance penalty $(\frac{\tau_x - \tau_{LSTF}}{\tau_{LSTF}} \times 100)$. We use the parameter set (N, 10, 7, 3, 5, Heavy) for the workload generator, and run on 10 processors. LSTFC and LSTFS have a slightly better performance over LSTF along the number of tasks axis. EDF and LLF do not have any particular trend in this experiment. The performance penalty for LSTF$^+$ starts around 0%. It monotonically drops to -15% at N=30 then monotonically increases to 0% as the number of tasks (N) is greater than 50.

The Number of Processors, M: Figure 7.16 shows the relation between the number of the processors and the performance penalty. We use the parameter set (10, 10, 7, 3, 5, Heavy) for the workload generator, and run on different processors. The performance penalty to LSTF$^+$ jumps up and down along the number of processors axis. With a trend, the performance penalty of LSTF$^+$ increases to 10% as the number of processors increases to 15. Except for LSTFC and LSTFS, which have

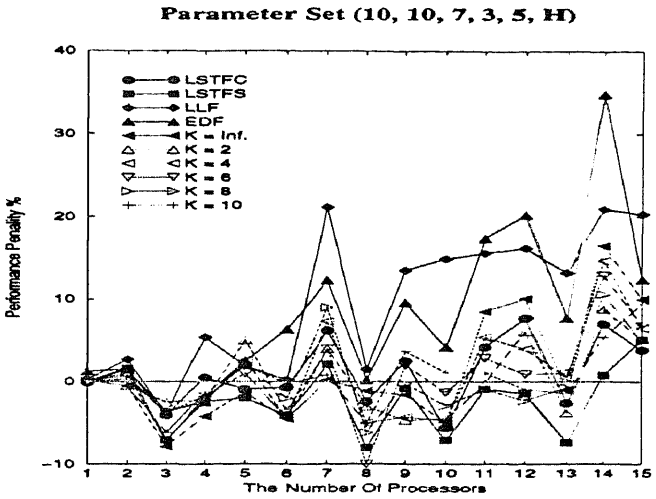

Figure 7.16 Effect of factor M

performance close to LSTF, other heuristics show lower gains as the number of processors increases.

Maximum Branching Factor, B: Figure 7.17 shows the relation between the number of the branches and the performance penalty. We use the parameter set (40, 10, 7, B, 5, Heavy) for the workload generator, and run on 10 processors. Again, LSTFC and LSTFS have a performance close to LSTF. The performance penalty for LSTF$^+$ starts at a lower value of -10% and monotonically increases to 0% as the number of branches (B) increases to 6.

Cross-Edge Density, E: Figure 7.18 shows the relation between the density of the cross edges and the performance penalty. We use the parameter set (10, 10, 7, 3, E, Heavy) for the workload generator, and run on 10 processors. The performance penalty for LSTF$^+$ starts at a higher mark of 10% and nearly monotonically drops to -10% as the number of cross edges (E) increases to 10. Except for LSTFC and LSTFS, other heuristics increase profit as the number of cross edges increase.

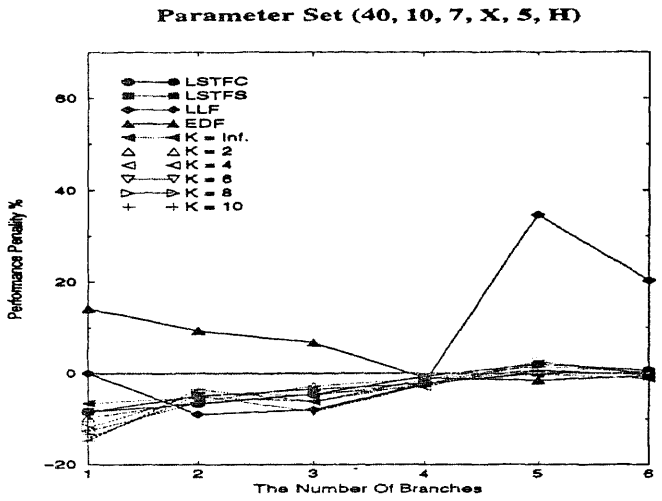

Figure 7.17 Effect of factor B

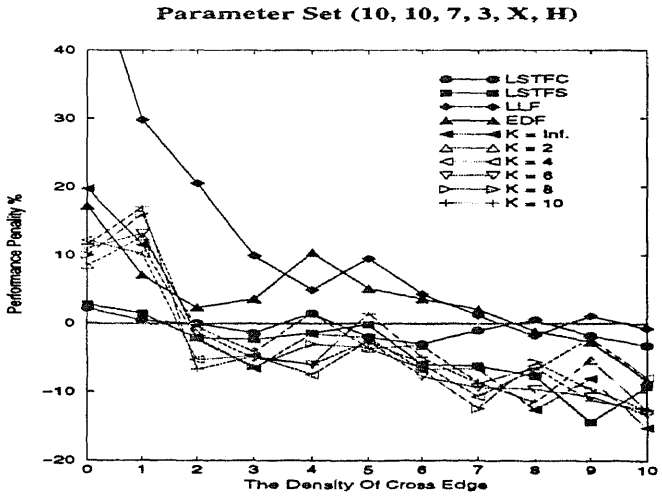

Figure 7.18 Effect of factor E

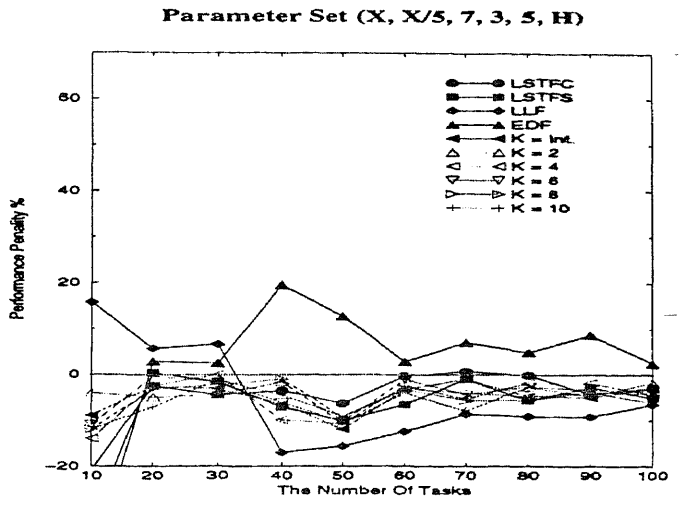

Figure 7.19 Effect of factor N and M ($N = 5M$)

Scaling and Changing both N and M: Figure 7.19 shows the relation between the combination factor of the number of tasks and processors and the performance penalty. We use the parameter set (N, M, 7, 3, 5, Heavy) for the workload generator, where $N = 5M$, and run on M processors. As the value of $\frac{N}{M}$ is kept as a constant 5, the performance penalty for LSTF$^+$ is pretty stable (around -5%).

Tree vs. Cross Edge, B and E: Figure 7.20 and 7.21 show the relation between the combination factor of the density of cross edges and the number of branches and the performance penalty. We use the parameter set (40, 10, 7, B, E, Heavy) for the workload generator, and run on 10 processors. From Figure 7.17 and 7.18, factor B and E work oppositely (LSTF$^+$s work well with low B but with high E). To show which the dominate factor is, we run two sets of experiments, $E = 3B$ and $B + E = 10$. Based on Figure 7.20 and 7.21, we conclude that the performance penalty is dominated by the number of branches (B).

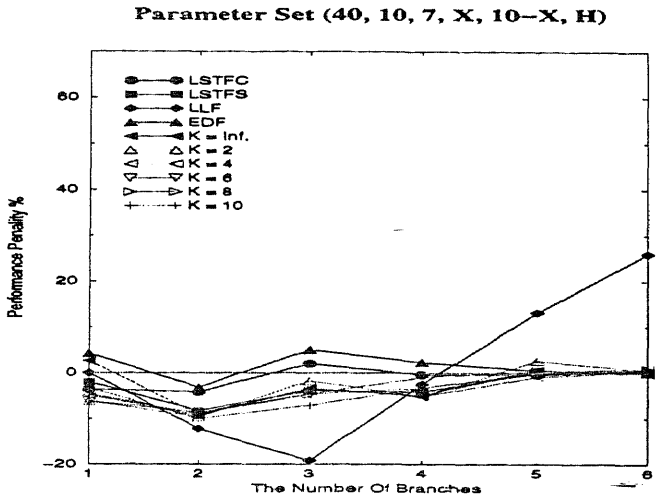

Figure 7.20 Effect of factor B and E ($B + E = 10$)

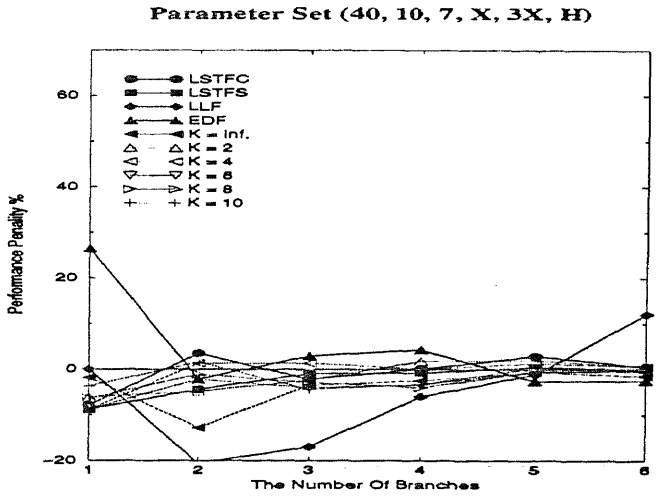

Figure 7.21 Effect of factor B and E ($E = 3B$)

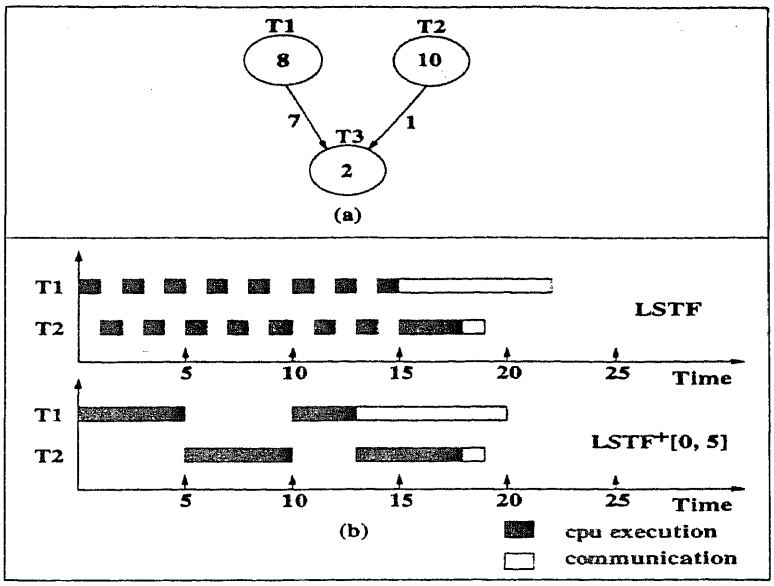

Figure 7.22 An example of concurrence in cpu execution and communication

7.4.2 Insight from the Simulation

To explain the results of experiments, we use one example to demonstrate. Suppose that there are T_1 with cpu requirement 8 and T_2 with cpu requirement 10 on the same processor in Figure 7.22-a. With the same space time, they share the cpu in a round robin manner. As both tasks complete, they will send the data volume, 7 and 1 (compatible with cpu requirement), respectively to the critical task T_3 located at another processor. The communication link delivers data without any delay because of absence of contention. To start T_3 as early as possible, all predecessors of T_3 are scheduled in such a manner that total completion time (which consist of computation and communication times) is minimized. Obviously, computation and communication should be performed concurrently.

Figure 7.22-b demonstrates the case of LSTF and LSTF$^+$[0, 5] scheduling T_1 at time 0. We observe that T_3 can start execution on time 20 under LSTF$^+$[0, 5] which is earlier than under LSTF (time 22). On the other hand, we list various LSTF$^+$ results, where case one denotes T_1 is running on time 0 and case two denotes T_2 is

running on time 0. The mean of case one and case two is also tabulated in column "Average".

Table 7.3 The starting time of T_3 of various LSTF$^+$

	Case One	Case Two	Average	Remarks
LSTF	22	23	22.5	see case one in Figure 7.22-b
LSTF$^+$[0, 4]	19	23	21	
LSTF$^+$[0, 5]	20	25	22.5	see case one in Figure 7.22-b
LSTF$^+$[0, 8]	15	23	19	
LSTF$^+$[0, 10]	15	25	20	

From the Table 7.3, LSTF$^+$ has a better performance than LSTF in case one, but not always in case two. This phenomenon can explain how the performance of LSTF$^+$ can jump up and down over LSTF.

In the example of Figure 7.22-a, LSTF$^+$ can get a lot of profit if T_1 has more out-edge (i.e., high cross edges density) because T_1 completes earlier under LSTF$^+$. Thus, communication and computation can perform concurrently. The higher density, the more profit of LSTF$^+$ has (Figure 7.18).

With a large number of branches, the processes are needed to be synchronized with a large number of predecessors. Suppose that T_3 has another predecessor, T_4. Although T_1 complete early under LSTF$^+$, T_3 need to wait the completion of T_4. Intuitively, LSTF$^+$ can not get a lot of profit with a large number of branches, B, (Figure 7.17).

From Figure 7.20 and 7.21, we conclude that the factor B has more influence on the performance penalty of LSTF$^+$ than factor E.

7.4.3 Impact of Assignments on the Performance of Algorithms

Because of the tasks represented as PGs, we view the tasks in three granularity: vertex, path, and task. Besides various workload generator parameters, we also

investigated the impact of assignments on the performance of algorithms based on granularity, When a whole task is assigned to a processor, communication cost will be zero and the model can be reduced into a model: each task runs on its own host processors independently. We have provided the theoretical result in Chapter 4 (such as Theorem 1) so that we only investigate the granularity of vertex and path.

- Random (RD): Let V denote the set containing all vertices of the PG. We label the processor from 1 to M, and construct the array *Placement* keeping the location information for each vertex. The pseudo-code for random assignment is presented as follows (where the function *rand* returns the integer number between 1 and M randomly):

```
for i = 1 to | V | {
    Placement[i] = rand(1, M);
}
```

- Round Robin (RR): The pseudo-code is similar to the random assignment. The function *mod* returns the reminder of i over M.

```
for i = 1 to | V | {
    Placement[i] = mod (i, M) + 1;
}
```

- Path-W (PW): Clustered by the path, a group of vertices are assigned together to the least loaded processor, where the heavier weight path is assigned earlier. Let P denote the set containing all paths.

```
group_vertices (&P);
sort_by_execution_weight (&P);
```

```
for i = 1 to | P | {
    m = least_loaded();
    for j ∈ P_i {
        Placement[j] = m;
    }
}
```

- Path-D (PD): The pseudo-code is similar to Path-W assignment. The path will be sorted by deadline and the earlier deadline path will be assigned earlier.

```
group_vertices (&P);
sort_by_execution_deadline (&P);
for i = 1 to | P | {
    m = least_loaded();
    for j ∈ P_i {
        Placement[j] = m;
    }
}
```

After we receive the data from the workload generator, we apply different assignment algorithms (RD, RR, PW and PD) to allocate the tasks to processors. We use the parameter set (10, 3, 7, 3, 5, U) for workload generator and run on 3 processors with four different traffic load (light, moderate, heavy and super-heavy). The symbol C in the following tables stands for the testbed with communication cost and CC stands for the testbed with communication cost and context switch overhead (0.1). For example, With PD assignment, LSTFC has performance profit over LSTF -0.7% under light communication condition and -0.3% under light communication and context switch overhead (Table 7.4).

We summarize some observations from our preliminary simulation results (Table 7.4, 7.5, 7.6 and 7.7):

- Trends in column C:
  - We observe that the technique of transformation of arcs into communication nodes helps LSTF when the communication takes place (LSTFC has better performance than LSTF under four different types of traffic load). The heavier communication, the more profitable LSTFC is.
  - Similar to LSTFC, LSTFS has better performance than LSTF under four different traffic loads. Usually, LSTFS outperforms LSTFC.
  - Except for LSTFC and LSTFS, LSTF gets more profit under light communication when assignment is done on the path basis rather than vertex basis.
  - With PD assignment, LSTF is usually better than with RD assignment.
  - The heavier load is, the smaller gains in performance LSTF gives over LLF.
  - $LSTF^+$ becomes more profitable under heavy load conditions (especially in the super-heavy condition).
- Trends in column CC: LSTF is not a good algorithm in these cases and $LSTF^+$ has better performance than other algorithms.

Table 7.4 The light communication simulation

	RD		RR		PW		PD	
	C	CC	C	CC	C	CC	C	CC
LSTFC	-0.7	-0.1	-0.6	-0.4	-0.3	-0.2	-0.7	-0.3
LSTFS	-2.8	-1.0	-2.2	-0.5	-0.5	-0.4	-1.4	-0.4
LLF	7.0	-7.8	36.8	-3.4	49.4	-2.6	50.0	-3.0
EDF	0.4	-0.9	0.4	-0.9	1.4	-0.9	1.3	-1.6
K=1	1.3	-63.3	6.5	-61.3	7.1	-62.5	7.3	-63.0
K=2	1.5	-64.5	6.9	-62.7	10.9	-63.9	10.9	-63.8
K=3	0.7	-65.4	7.3	-63.5	10.2	-64.8	10.2	-64.5
K=4	1.1	-65.6	8.0	-63.7	9.9	-65.1	9.2	-65.4
K = Inf.	3.6	-65.2	13.5	-63.9	21.0	-64.3	19.2	-64.6

Table 7.5 The moderate communication simulation

	RD		RR		PW		PD	
	C	CC	C	CC	C	CC	C	CC
LSTFC	-1.6	-0.4	-1.5	-0.0	-2.7	-0.0	-0.8	0.1
LSTFS	-3.7	-1.0	-5.3	-0.4	-5.6	-0.9	-4.4	-1.1
LLF	9.4	-6.4	29.0	-2.8	36.4	-2.5	46.0	-2.8
EDF	1.7	-1.1	2.6	-0.2	0.5	-1.0	1.9	-1.4
K=1	1.4	-62.1	3.3	-59.9	5.3	-61.1	8.1	-61.6
K=2	2.9	-63.2	3.3	-61.4	5.3	-62.6	9.3	-62.9
K=3	3.4	-63.6	5.2	-61.6	5.8	-63.6	8.8	-63.6
K=4	2.4	-64.3	6.1	-62.0	4.9	-63.9	9.4	-63.9
K = Inf.	5.7	-64.8	11.2	-63.2	16.5	-63.8	23.2	-64.3

Table 7.6 The heavy communication simulation

	RD		RR		PW		PD	
	C	CC	C	CC	C	CC	C	CC
LSTFC	-1.0	-0.6	-6.0	-1.5	-4.6	-1.1	-5.8	-1.5
LSTFS	-3.8	-1.6	-10.1	-2.1	-6.5	-1.9	-9.1	-1.9
LLF	8.6	-6.1	25.1	-3.6	30.9	-3.5	32.9	-3.6
EDF	5.3	-1.2	9.4	-0.5	8.6	-1.3	3.6	-1.5
K=1	-0.4	-61.7	1.1	-59.9	4.9	-61.1	7.1	-61.1
K=2	-1.7	-62.7	-0.3	-61.1	6.0	-62.1	6.5	-62.3
K=3	1.1	-62.8	2.7	-61.3	6.9	-62.4	7.9	-62.6
K=4	-0.7	-63.3	1.6	-61.6	4.5	-62.9	6.7	-62.9
K = Inf.	1.7	-64.5	10.4	-62.5	11.5	-64.1	16.4	-63.9

Table 7.7 The super-heavy communication simulation

	RD		RR		PW		PD	
	C	CC	C	CC	C	CC	C	CC
LSTFC	-5.5	-2.6	-6.3	-1.9	-7.2	-3.5	-9.9	-4.4
LSTFS	-9.5	-4.4	-8.7	-4.0	-11.2	-5.2	-12.3	-5.7
LLF	5.2	-4.4	4.8	-3.2	1.3	-0.5	6.5	-4.7
EDF	1.6	1.2	-0.6	0.4	-0.1	-0.1	1.9	0.4
K=1	-8.3	-50.3	-5.8	-47.2	-5.2	-49.6	-7.3	-49.9
K=2	-7.0	-50.3	-8.0	-48.4	-4.9	-50.0	-7.5	-50.3
K=3	-8.1	-50.7	-8.1	-48.6	-5.3	-50.4	-8.9	-51.2
K=4	-7.7	-50.7	-7.0	-48.3	-4.6	-50.2	-10.7	-51.8
K = Inf.	-13.9	-54.5	-11.7	-52.0	-8.7	-53.6	-10.9	-54.0

CHAPTER 8

SUMMARY AND FUTURE WORK

Least-Space-Time-First (LSTF) is a new scheduling policy aimed at the 'complex-tasks-multiple-processor' category of problems. We have proven that LSTF is a more effective scheduling algorithm than EDF in three different scheduling models ('simple-tasks-single-processor', 'simple-tasks-multiple-processor', and 'complex-tasks-single-processor'). We have also been able to show, under a number of restrictions on the tasks and the operating system, that LSTF minimizes maximum tardiness when compared to other scheduling disciplines in the 'complex-tasks-multiple-processor' model.

We present both lower and upper bounds on tardiness of schedules for general work-conserving scheduling algorithms, and refinements of the upper bounds for EDF scheduling and our LSTF algorithm. This information helps system engineers to know how badly scheduling algorithms perform so that the upper bound can be used a schedulability test in the design of hard real-time systems.

Also, we have explored other refinements of LSTF, LSTF$^+$, for use with context switch overhead and communication cost. We are able to show the outperformance of LSTF$^+$ through theoretic and experimental results. From the simulation results, we conclude that the selection of the appropriate parameters for LSTF$^+$ is sensitive to statistical properties of the task distribution.

We give simulation results on three different platforms and show that (1) LSTF outperforms EDF and other scheduling algorithms on the ideal platform. (2) LSTF$^+$ is a good algorithm in the presence of context switch and communication.

8.1 Extent to Dynamic Task Sets and Environment

In this dissertation, LSTF schedules the task set statically. We provide a brief discussion of some of the ways in which LSTF can be extended to handle dynamics in the task set of environment, as a partial skeleton of future work.

Ready time: The original assumption of ready time is released; some tasks may not be ready at time 0. Given a task j, with a deadline and task graph information (such as the total requirement R_j, the longest path L_j and the deadline D_j), arrives dynamically (the ready time of task j is not equal to 0). There are two possible ways for LSTF to handle the new incoming task j:

1. If the system does not care about quality of service (i.e., no threshold for tardiness), the control systems simply adds the ready nodes of this new task into the ready queue. LSTF treats the new task in the way as old ones.

2. The control system invokes the bound-evaluator to estimate the tardiness for each task. Intuitively, the evaluator considers (R_i', L_i', D_i) for each old task and (R_j, L_j, D_j) for the new task j, where R' is the remaining requirement and L_i' is the reaming execution longest path of task j respectively. If LSTF would degrade the service of existing old tasks (i.e., exceed the threshold for any one of old tasks) based on the evaluator, the control system regretfully rejects the request of task j. The task j may j may request to be scheduled again in the future.

Reclaiming unused time: Our scheduling frame work is based on static task graphs and weights which are known a priori from the worst case. If the scheduler does not take the worst branch during the execution, there may remain processing time units left in a execution node. Suppose that node X finishes F time units earlier

than the worst-case time; the processor may then either execute other ready nodes, or remain idle. LSTF handles the situation as follows:

- Context switch is free:

  LSTF continues to execute any ready node with the least space time for the F time units. When the real-time quantum is expired, the scheduler calculates the space time of each node in the same way as before (ignoring the fragment). For example, node i with L_i and D_i has been executed for F fragment time units. The space time of node i is equal to $(D_i - present\_time - L_i)$.

- Context switch is not free (W cost):

  - X's successors are ready

    * successors have least space time:

      LSTF continues to execute its successors for the F fragment time units.

    * other ready nodes have least space time:

      LSTF pays the W cost and jumps to execute the least space time task.

  - X's successors are not ready :

    * successors have least space time:

      LSTF executes the fragment $(F - 2W)$ time units for other ready nodes, if $(F - 2W)$ is greater than 0; otherwise, the processor keeps idle.

    * other ready nodes have least space time:

      LSTF pays the W cost and jumps to execute the least space time task.

8.2 Future Work

From the history of RMS, we know a lot of work needs to be done. We will continue to extend other work in several directions:

- Discover more properties and theorems of LSTF under different models.
- Formulate objective functions for making migration decisions and simulate the results.
- Find and apply LSTF to different case studies. For example, multimedia systems.
- Apply the developed methodology to different system models.
- Relax some current restrictions, such as on ready time, periodic tasks, etc.
- Cooperate with other heuristics to assignment and scheduling.

The RMS community spent almost 20 years investigating these latter issues and they keep on going. Optimistically, we may have significant results in a couple of years.

REFERENCES

1. R. Agne, "A distributed off-line scheduler for distributed hard real-time systems," *Proceedings of the 10th IFAC Workshop*, pp. 35–40, 1991.

2. B. Arinze, "Knowledge-based decision support system for project management," *Computer & Operations Research*, vol. 19, pp. 321–334, 1992.

3. T. P. Baker, "A stack-based resource allocation policy for realtime processes," *In Proceedings of the Real-Time Systems Symposium*, pp. 191–200, 1990.

4. M. Bandelloni, "Optimal resource leveling using non-serial dynamic programming," *European Journal of Operational Research*, vol. 78, pp. 162–177, 1994.

5. J. Blazewicz, P. Dell'Olmo, M. Drozdowski, and M. Speranza, "Scheduling multiprocessor tasks on three dedicated processors," *Information Processing Letters (IPL)*, vol. 41, pp. 275–280, 1992.

6. L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein, "Optimal strategies for spinning and blocking," *Journal of Parallel and Distributed Computing*, vol. 21, pp. 246–54, 1994.

7. P. Brucker, M. R. Garey, and D. S. Johnson, "Scheduling equal length tasks under treelike precedence constraints to minimize maximum lateness," *Mathematics of Operations Research*, vol. 2, pp. 275–284, 1977.

8. J. Bruno, "Deterministic and stochastic scheduling problems with treelike precedence constraints," *NATO Conference*, 1981.

9. B. Cadambi, "One machine scheduling to minimize expected mean tardiness i," *Computer & Operations Research*, vol. 18, pp. 787–796, 1991.

10. C. L. Cesar and P. G. Jessel, "Real-time task scheduling with overheads considered," *Naval Research Logistics*, vol. 39, pp. 247–64, 1992.

11. G. Chaudhry and X. Li, "A case for the multithreaded processor architecture," *Computer Architecture News*, vol. 22, pp. 55–59, 1994.

12. B.-C. Cheng, A. Stoyenko, T. Marlowe, and S. Baruah, "Minimizing maximum tardiness for dsp programs with context switch overheads considered," *The International Conference on Signal Processing Applications and Technology (ICSPAT'96)*, 1996.

13. B.-C. Cheng, A. D. Stoyenko, and T. J. Marlowe, "Least-space-time-first scheduling algorithm: a policy for non-simple real-time tasks in multiple processor systems," *accepted by 19th IFAC/IFIP Workshop on Real Time Programming (WRTP'94)*, 1994.

14. S. Cheng and J. A. Stankovic, "Scheduling algorithms for hard real-time systems - a brief survey," *IEEE Tutorial on Hard Real-Time Systems*, pp. 150–173, 1988.

15. T. C. E. Cheng and C. C. S. Sin, "A state-of-the-art review of parallel-machine scheduling research," *European Journal of Operational Research*, vol. 47, pp. 271–292, 1990.

16. C. Chu, "A branch-and-bound algorithm to minimize total tardiness with different release dates," *Naval Research Logistics*, vol. 39, pp. 265–283, 1992.

17. W. W. Chu, C.-M. Sit, and K. K. Leung, "Task response time for real-time distributed systems with resource contentions," *IEEE Transaction on Software Engineering*, vol. 17, pp. 1076–1092, 1991.

18. J.-Y. Chung, J. W. S. Liu, and K.-J. Lin, "Scheduling periodic jobs that allow imprecise results," *IEEE Transaction on Computers*, vol. 39, pp. 1156–1174, 1990.

19. T. M. Cook and R. A. Russell, *Introduction to management science*, Prentice-Hall, Englewood Cliffs, N.J., 1981.

20. P. De and T. E. Morton, "Scheduling to minimize maximum latenss on unequal parallel processors," *Computer Oper*, vol. 9, pp. 221–232, 1982.

21. H. M. Deitel and H. Lorin, "An introduction to operating systems," *Addison-Wesley Systems Programming Series*, 1981.

22. E. Demeulemeester and W. Herroelen, "A branch-and-bound procedure for the multiple resource-constrainted project scheduling problem," *Management Science (MCI)*, vol. 38, pp. 1803–1818, 1992.

23. M. Dertouzos, "Control robotics: the procedural control of physical process," *In Proc. of the IFIP Congress*, pp. 807–813, 1974.

24. M. L. Dertouzos and A. K.-L. Mok, "Multiprocessor on-line scheduling of hard-real-time tasks," *IEEE Transactions on Software Engineering*, vol. 15, pp. 1497–1506, 1989.

25. S. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, pp. 127–140, 1978.

26. H. Emmons and M. Pinedo, "Scheduling stochastic jobs with due dates on parallel machines," *European Journal of Operational Research (EJO)*, vol. 47, pp. 49–55, 1990.

27. M. J. Flynn, "Very high-speed computers," *Proceedings of the IEEE, 54*, pp. 1901–1909, 1966.

28. A. Fredette and R. Cleaveland, "Rtsl: a language for real-time schedulability analysis," *In Proceedings of the Real-Time Systems Symposium*, pp. 274–283, 1993.

29. S. French, "Sequencing and scheduling: an introduction to the mathematics of the job-shop," *Ellis Horwood Ltd*, 1982.

30. R. Gerber and S. Hong, "Semantics-based compiler transformations for enhanced schedulability," *In Proceedings of the Real-Time Systems Symposium*, pp. 232–242, 1993.

31. R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM J. Appl. Math.*, vol. 17, pp. 416–429, 1969.

32. R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan, "Optimization and approximation in deterministic sequencing and scheduling: A survey," *Annals of Discrete Mathematics*, vol. 5, pp. 287–326, 1979.

33. F. Grobler, K. P. Reddy, and C. Subick, "Beyond cpm: the deck-of-cards paradigm," in *Proceedings of the 1st Congress on Computing in Civil Engineering*, Washington, DC, pp. 2046–2053, June 20-22 1994.

34. R. Gupta and M. L. Soffa, "Region scheduling: an approach for detecting and redistributing parallelism," *IEEE Transactions of Software Engineering*, vol. 16, pp. 421–431, 1990.

35. K. Hadavi, W. L. Hsu, T. Chen, and C.-N. Lee, "An architecture for real-time distributed scheduling," *AI Magazine*, vol. 13, pp. 46–56, 1992.

36. W. A. Halang and A. D. Stoyenko, "Constructing predictable real time system," *Kluwer Academic Publishers*, 1991.

37. N. G. Hall and M. E. Posner, "Earliness-tardiness scheduling problems. i. weighted deviation of completion times about a common due date," *Operations Research*, vol. 39, pp. 836–946, 1991.

38. M. Hamdaoui and P. Ramanathan, "A dynamic priority assignment technique for streams with (m,k)-firm deadlines," *IEEE Transactions on Computers*, vol. 44, pp. 1443–1451, 1995.

39. C. M. Harmonosky and S. F. Robohn, "Real-time scheduling in computer intergrated manufacturing: a review of recent research," *International Journal of Computer Integrated Manufacturing*, vol. 4, pp. 331–340, 1991.

40. J. E. Holsenback and R. M. Russell, "A heuristic algorithm for sequencing on one machine to minimize total tardiness," *Journal of the Operational Research Society*, vol. 43, pp. 53–62, 1992.

41. K. Hong and J. Y. T. Leung, "On-line scheduling of real-time tasks," *In Proceedings of the real-time Systems Symposium*, pp. 244–250, 1988.

42. C.-K. Hou and K. G. Shin, "Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems," *In Proceedings of the Real-Time Systems Symposium*, pp. 146–155, 1992.

43. W. L. Hsu and R. C. T. Lee, "Competitiveness and response time in on-line algorithms," *ISA '91 Algorithms. 2nd International Symposium on Algorithms*, pp. 284–293, 1991.

44. T. C. Hu, "Parallel sequencing and assembly line program," *Operations Res.*, vol. Sep., pp. 841–848, 1961.

45. D. I. Katcher, H. Arakawa, and J. K. Strosnider, "Engineering and analysis of fixed priority schedulers," *CMU/ECE Technical Report*, 1991.

46. E. Kligerman and A. D. Stoyenko, "Real-time euclid: a language for reliable real-time systems," *IEEE Transactions on Software Engineering*, vol. 12, pp. 940–949, 1986.

47. H. Kopetz, "Scheduling," *Distributed Systems, Second Edition, Edited by Sape Mullender*, 1993.

48. J. K. Lenstra, A. H. G. R. Kan, and P. Brucker. "Complexity of machine scheduling problems," *Annals of Discrete Mathematics*, vol. 1, pp. 343–362, 1977.

49. J. Y.-T. Leung, "A new algorithm for scheduling periodic, real-time tasks," *Algorithmica*, vol. 4, pp. 209–219, 1989.

50. C. L. Liu and J. Layland, "Scheduling algorithm for multiprogramming in a hard real-time environment," *J. ACM*, vol. 20, 1973.

51. J. Mittenthal and M. Raghavachari, "Stochastic single machine scheduling with quadratic early-tardy penalties," *Operations Research (OPR)*, vol. 41, pp. 786–796, 1993.

52. J. J. Moder and S. E. Elmaghraby, *Handbook of operations research*, Van Nostrand Reinhold, Andover, England, 1978.

53. R. R. Muntz and E. G. Coffman, "Preemptive scheduling of real-time tasks on multiprocessor systems," *J. of ACM*, vol. 17, pp. 324–338, 1970.

54. R. Nord and B.-C. Cheng, "Using rma for evaluating design decisions," *accepted to Second IEEE Workshop on Real-Time Applications*, 1994.

55. F. Y. Partovi, "Timing of monitoring and control of cpm projects," *IEEE Transactions on Engineering Management*, vol. 40, pp. 68–75, 1993.

56. J. Plehn, "Preemptive scheduling of independent jobs with release times and deadlines on a hypercube," *Information Processing Letters (IPL)*, vol. 34, pp. 161–166, 1990.

57. K. A. Robbins and S. Robbins, *Practical UNIX programming: a guide to concurrency, Communication, and Multithreading*, Prentice Hall, Englewood Cliffs, N.J., 1995.

58. J. G. Root, "Scheduling with deadlines and loss functions on k parallel machines," *Management Science*, vol. 11, pp. 460–475, 1965.

59. T. Sen, P. Dileepan, and J. N. D. Gupta, "The two-machine flowshop scheduling problem with total tardiness," *Computers & Operations Research (CRO)*, vol. 16, pp. 333–340, 1989.

60. L. Sha and J. B. Goodenough, "Real-time scheduling theory and ada," Tech. Rep. SEI-89-TR-014, Software Engineering Institute, 1989.

61. J. A. Stankovic and K. Ramamritham, "The spring kernel: a new paradigm for real-time systems," *IEEE Software*, vol. 8, pp. 62–72, 1991.

62. J. A. Stankovic, M. Spuri, M. D. Natale, and G. Buttazzo, "Implications of classical scheduling results for real-time systems," *IEEE Computer*, vol. 28, pp. 16–25, 1995.

63. D. B. Stewart and P. K. Khosla, "Real-time scheduling of sensor-based control systems," *Real Time Programming. Proceedings of the IFAC/IFIP Workshop*, pp. 139–44, 1990.

64. A. D. Stoyenko and L. Georgiadis, "On optimal lateness and tardiness scheduling in real-time systems," *Computing*, vol. 47, pp. 215–34, 1992.

65. A. D. Stoyenko, V. C. Hamacher, and R. C. Holt, "Analyzing hard-real-time programs for guaranteed schedulability," *IEEE Transactions on Software Enginering*, vol. 17, pp. 737–750, 1991.

66. A. D. Stoyenko and T. J. Marlowe, "Schedulability, program transformations and real-time programming," *IEEE/IFAC Real-Time Operating Systems Workshop*, 1991.

67. A. D. Stoyenko and T. J. Marlowe, "Polynomial-time transformations and schedulability analysis of parallel, real-time programs with restricted resource contention," *Journal of Real-Time Systems*, vol. 4, 1992.

68. A. D. Stoyenko, T. J. Marlowe, W. A. Halang, and M. Younis, "Enabling efficient schedulability analysis through conditional linking and program transformations," *Control Enginering Practice*, vol. 1, 1993.

69. A. D. Stoyenko, L. R. Welch, and B.-C. Cheng, "Response time prediction in object-based, parallel embedded systems," *Microprocessing and Microprogramming*, vol. 40, pp. 135–150, 1994.

70. W. Szwarc and J. J. Liu, "Weighted tardiness single machine scheduling with proportional weights," *Management Science (MCI)*, vol. 39, pp. 626–632, 1993.

71. J. P. C. Verhoosel, E. J. Luit, D. K. Hammer, and E. Jansen, "A static scheduling algorithm for distributed hard real-time systems," *Real-Time Systems*, vol. 3, pp. 227–246, 1991.

72. F. Wong, K. Ramamritham, and J. A. Stankovic, "Bound on the the performance of heuristic algorithm for multiprocessor scheduling of hard real-time tasks," *In Proceedings of the Real-Time Systems Symposium*, pp. 136–145, 1992.

73. J. Xu, "Multiprocessor scheduling of process with release time, deadline, precedence, and exclusion relations," *IEEE transaction on Software Engineering*, vol. 19, pp. 139–154, Feb. 1993.

74. A. C. Yu and K. J. Lin, "A scheduling algorithm for replicated real-time tasks," *11th Annual International Phoenix Conference on Computers and Communications*, pp. 395–402, 1992.