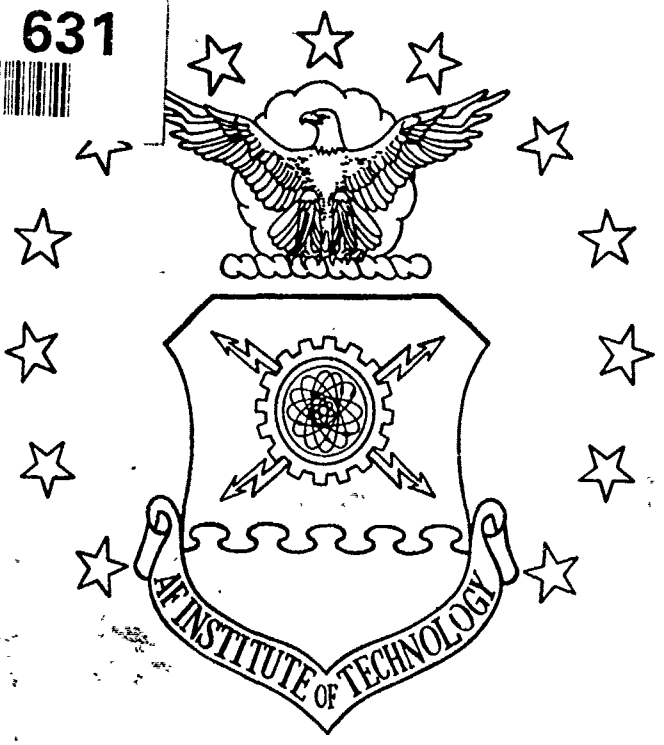


DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/91M-03

Optimal Iterative Task Scheduling for Parallel Simulations

THESIS

JoAnn M. Sartor
Captain, USAF

AFIT/GCS/ENG/91M-03

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1991	*Master's Thesis*

4. TITLE AND SUBTITLE
Optimal Iterative Task Scheduling for Parallel Simulations

5. FUNDING NUMBERS

6. AUTHOR(S)
JoAnn M. Sartor, Capt USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/91M-03

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
LTC John C. Toole
DARPA/ISTO
1400 Wilson Blvd
Arlington, VA 22209-2308

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The ultimate purpose of this research is to reduce the time needed for execution of parallel computer simulations. In particular, the impact of task assignment strategies is determined for parallel VHDL circuit simulations. The classical scheduling problem, which assigns n precedence-constrained tasks to m processors is NP-complete in all but the simplest cases. The problem of assigning simulation tasks is further complicated by the iterative nature of computer simulations: each task is required to execute multiple times as the simulation executes. This investigation develops a polynomial-time algorithm (the *level strategy*) which provides optimal assignment for iterative systems with specific constraints. A mathematical foundation for iterative task systems is proved. In particular, it is shown that restricted cases of iterative systems achieve minimal latency, (time between successive iterations of a given task), when the level strategy is used for task assignment.

To verify the theoretical results, various task scheduling strategies are compared using VHDL logic-circuit simulations on the iPSC/2 Hypercube computer. Tests are run with mappings based on the level strategy, the classical optimal assignment, a greedy technique for assignment, and an unbalanced assignment. The best results of these experiments, in terms of speedup, occur consistently in cases where the level strategy is used.

14. SUBJECT TERMS
simulation task scheduling, iterative scheduling, latency, optimal schedules, NP-complete problems, precedence-constrained scheduling, polynomial-time algorithms, repetitive schedules

15. NUMBER OF PAGES
128

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to **stay within the lines to meet optical scanning requirements.**

Block 1. Agency Use Only (Leave Blank)

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Names(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency. Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of ..., To be published in .... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availablity Statement. Denote public availability or limitation. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR)

DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE - See authorities
NASA - See Handbook NHB 2200.2.
NTIS - Leave blank.

Block 12b. Distribution Code.

DOD - DOD - Leave blank
DOE - DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports
NASA - NASA - Leave blank
NTIS - NTIS - Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

AFIT/GCS/ENG/91M-03

Optimal Iterative Task Scheduling for Parallel Simulations

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

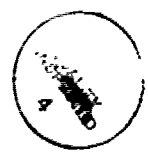
Requirements for the Degree of

Master of Science in Computer Science

JoAnn M. Sartor, B.S.

Captain, USAF

14 March 1991



Acknowledgments

There are many, many people who deserve mention here! I'd like to take this opportunity to thank some of them:

First of all, my family. Whenever I'd bring home report cards from school, my mother's response was, "I don't care what grade you get, as long as you do your best." (This, of course, doesn't let a student off-the-hook even if she brings home A's!) My dad, with his stubborn outlook on life, taught me to persevere through the tough times. My three siblings, Kathy, Joe, and Mike, (along with their spouses) performed a chameleon act: a cheering section when I did well – and a support group when I needed a shoulder to cry on. And my brother's children, Josh and Jessie, added sparkle to my life when research looked dreary.

The library staff (in particular, Kathy, Barbara, and Jeff) searched databases; located sources; and answered innumerable questions. Rick, the Keeper of the Hypercubes, simplified the VHDL simulation input, answered computer questions – and helped me scrounge an office. Dr. Roth guided me through the maze of LaTeX. My classmates (Ann Lee, James Jaques, and especially Betty Topp) listened to my griping and helped me keep my sense of proportion. And AFIT faculty members (Dr Khatri, Dr Howatt, Dr Bailor, Dr Brown, ...) all answered questions and helped me to decipher technical material. Dr Potoczny worked with me to derive the Combinatoric Example in Appendix A, and Jeff Simmers offered a method of simplifying induction proofs.

My committee members, MAJ Robert Hammell, Dr Gary Lamont, and Maj William Hobart all contributed valuable insights to this effort. Dr Hartrum, though not a member of my committee, took an interest in this thesis and provided several of the ideas that led to breakthroughs. In fact, Dr Hartrum was the first one to mention the iterative property of VHDL simulations!

MAJ Hammell gave me constant encouragement; his confidence-building speeches helped me through several low points: "Writing a thesis is like building a house..." "In graduate school, you'll find that no one has 'THE' answer!" ... MAJ Hammell, I am greatly in your debt for all your work...If ever you need a blood donor, call on me!

Dr Lamont! You have been my role model, as well as my advisor: Sometimes, I've come back from a discussion with you feeling the way King Arthur's knights might have felt after leaving the Round Table – as if I have the strength to triumph over any obstacle. You have been a profound influence in my life, as well as the driving force behind this research. Thank you!

And most of all, I'd like to acknowledge the patience and support given by my husband Ray. (Often, Ray was more excited about this thesis than I was: "They're going to let you work on the **Hypercube!!!**") Although putting in "80 hours a week" on his PhD research, Ray found the time to cook huge batches of jambalaya; to go for walks around the neighborhood; and to roar laughing at the escapades of Bertie Wooster![1] Ray, you've been a great blessing throughout this struggle; I wouldn't trade you for

- All the tea in China,
- All the rum in Jamaica,
- Or even the jackpot in last week's Lotto!

JoAnn M. Sartor

[1] For those of you who haven't had a good laugh lately, check out any "Bertie and Jeeves" book by P.G. Wodehouse!

Table of Contents

		Page
List of Figures		viii
List of Theorems		xi
Abstract		xii
I.	Introduction	1-1
	1.1 Overview	1-1
	1.2 Simulation Tasking Background	1-2
	1.3 Task Scheduling Problem	1-7
	1.4 Assumptions	1-8
	1.5 Scope (Context)	1-8
	1.6 Approach	1-9
	1.7 Structure of Thesis	1-10
II.	Background	2-1
	2.1 General Task Scheduling Problem Description	2-1
	2.1.1 Algorithm Considerations	2-5
	2.1.2 Notation and Diagrammatic Representation	2-6
	2.2 Classes of Repeating Schedules	2-7
	2.3 Latency	2-9
	2.4 General approaches to the scheduling problem	2-9
	2.5 Iterative nature of the problem	2-12
	2.6 Assumptions and Environment	2-14
	2.7 Additional Considerations	2-16
	2.8 Summary	2-16

Page

III. Scheduling Algorithm Design 3-1

3.1 Basis for Level Strategy (One-pass Systems) 3-1

3.1.1 One-Pass Level Algorithm 3-1

3.2 Level Algorithm for Iterative Tasks 3-3

3.3 Search Process 3-3

3.3.1 Exhaustive Search 3-4

3.3.2 Informed Search 3-4

3.4 Reducing the Search Space 3-8

3.4.1 Lower Bound Metrics 3-8

3.5 Heuristics for Schedule-Building 3-14

3.6 Summary 3-16

IV. Low-level Analysis 4-1

4.1 Theoretical Design of Schedule 4-1

4.1.1 Justification for Latency Measure 4-3

4.2 Restrictions 4-5

4.3 Level Strategy 4-6

4.4 Scheduling Within a Processor 4-9

4.4.1 Decision Strategy 4-9

4.5 Lower Bound on Latency 4-10

4.6 Upper Bound on Latency 4-12

4.6.1 Processor Delay Time 4-13

4.6.2 Chain of Tasks 4-14

4.6.3 Arbitrary Precedence 4-18

4.7 Equal Execution Time Task Systems 4-24

4.8 Variable Execution Times 4-25

4.8.1 Problem Description 4-25

4.8.2 Reasons for Level-Strategy Failure 4-26

Page

4.8.3 Bounds on Variable-Execution-Time Latency 4-28

4.8.4 Minimizing Number of Processors 4-32

4.9 NP-Complete Aspects 4-35

4.9.1 Background 4-35

4.9.2 Proving NP-Completeness 4-36

4.9.3 Variable Execution Time Systems 4-37

4.10 Summary 4-42

V. Application/Experimental Results 5-1

5.1 VHDL Application 5-1

5.1.1 VHDL Parameters and Results 5-1

5.2 Gaming Simulation Results 5-4

5.3 Summary 5-7

VI. Conclusions and Recommendations 6-1

6.1 Conclusions and Contributions 6-1

6.2 Recommendations for Further Research 6-4

Appendix A. Combinatoric Complexity Example A-1

A.1 Detailed Example A-1

Appendix B. Level Strategy Example B-1

B.1 Level Strategy B-3

Appendix C. A* Search C-1

C.1 A* overview C-1

C.2 Sample A* Search C-1

C.3 Evaluation Function C-3

Appendix D. Iteration-Number Decision Strategy D-1

Page

Vita VITA-1

Index IND-1

Bibliography BIB-1

List of Figures

Figure		Page
1.1.	Unbalanced Processor Loads	1-3
1.2.	Relationships Between Tasks (1) and (2)	1-4
1.3.	Branch and Bound Search Tree	1-6
2.1.	Relationship Between Multiprocessor Scheduling Problems	2-3
2.2.	Taxonomy of Scheduling Problems	2-5
2.3.	Task system	2-7
2.4.	Arbitrary Precedence Graph with Optimal Schedule	2-10
2.5.	Tree-Structured Precedence Graph with Optimal Schedule	2-11
2.6.	Graph(a); Chordal Complement(b)	2-12
2.7.	CORBAN Tasks	2-13
2.8.	Optimal (1-pass) vs Pipelined	2-14
3.1.	Assignment by level strategy	3-2
3.2.	Exhaustive Search	3-5
3.3.	Partial Search Tree	3-6
3.4.	Branch and Bound Search Tree	3-7
3.5.	Chain of 9 Tasks	3-9
3.6.	Longest Path = 5	3-10
3.7.	Search Graph and Optimal Schedule for Independent Task Example	3-13
3.8.	Scheduling Tasks with Most Successors	3-15
3.9.	Search Tree for Tasks with Most Successors	3-17
4.1.	Precedence Graph: 8-bit Adder	4-2
4.2.	Latency	4-3
4.3.	Search Tree (Level-Strategy Assignment)	4-4

Figure Page

4.4. Mapping for 10 tasks and 10 processors; (latency = 1) 4-4

4.5. Task System with Non-optimal Mapping 4-7

4.6. Assigning Levels to Tasks 4-8

4.7. Effects of Decision Strategy 4-9

4.8. 5 independent tasks; 2 iterations 4-11

4.9. Maximum Latency for Task System 4-13

4.10. Comparison: Assignment Strategies with Schedule 1 4-14

4.11. Level Strategy Assignment: chain of m tasks on m processors 4-15

4.12. Level Strategy Assignment: chain of $m+1$ tasks on m processors . . . 4-15

4.13. Level Strategy Assignment: Serial Precedence (chain) 4-16

4.14. Last task execution on all processors 4-17

4.15. Level Strategy Assignment: arbitrary precedence *vs.* serial precedence . 4-19

4.16. Idle time due to level change 4-20

4.17. Variable execution time task system 4-27

4.18. Task System with Variable Execution Time 4-28

4.19. Variable execution time task system 4-29

4.20. Relationship between bounds 4-33

4.21. Lower Bound on Processors 4-33

4.22. Transformation to Show NP-Completeness 4-37

4.23. Nondeterministic Turing Machine Assignment 4-39

4.24. Mapping NP-Complete Problem into Open Problem 4-40

4.25. Transforming the Iterative Problem to the Classical Problem 4-41

4.26. Transforming the Classical Problem to the Iterative Problem 4-43

5.1. Precedence Graph: VHDL tasks 5-2

5.2. Varied Mapping Strategies 5-5

5.3. Speedup 5-6

5.4. Gaming Simulation Precedence Graph 5-7

Figure Page

5.5. Varied Mapping Strategies (Spin Loops 10,000) 5-8

5.6. Varied Mapping Strategies (Spin Loops 100,000) 5-8

6.1. Precedence Graph with Feedback 6-5

B.1. Precedence Graph B-1

B.2. Level Assignment B-2

B.3. More ready tasks than processors B-4

C.1. Graph for A* Example C-2

D.1. Delay Time Due to Arbitrary Decision Strategy D-1

D.2. Optimal Schedule D-2

List of Theorems

Theorem	Page
UET Lower Bound	4-11
Delay Time: chain	4-15
Delay Time: arbitrary precedence	4-18
Maximum Delay Time	4-21
UET Upper Bound	4-22
Firm Bound on UET Latency	4-23
Equal Execution Time	4-24
Variable-execution-time Lower Bound	4-29
Variable-execution-time Upper Bound	4-31
Number of Processors: Variable-execution-time	4-34
NP-completeness	4-37

Abstract

The ultimate purpose of this research is to reduce the time needed for execution of parallel computer simulations. In particular, the impact of task assignment strategies is determined for parallel VHDL circuit simulations.

The classical scheduling problem, which assigns n precedence-constrained tasks to m processors is NP-complete in all but the simplest cases. The problem of assigning simulation tasks is further complicated by the iterative nature of computer simulations: each task is required to execute multiple times as the simulation executes.

This investigation develops a polynomial-time algorithm (the *level strategy*) which provides optimal assignment for iterative systems with specific constraints. A mathematical foundation for iterative task systems is proved. In particular, it is shown that restricted cases of iterative systems achieve minimal latency, (time between successive iterations of a given task), when the level strategy is used for task assignment. In addition, the iterative scheduling problem is proved NP-complete when constraints are relaxed.

To validate the theoretical results, various task scheduling strategies are compared using VHDL logic-circuit simulations on the iPSC/2 Hypercube computer. Tests are run with mappings based on the level strategy, the classical optimal assignment, a greedy technique for assignment, and an unbalanced assignment. The best results of these experiments, in terms of speedup, occur consistently in cases where the level strategy is used.

Optimal Iterative Task Scheduling for Parallel Simulations

I. Introduction

1.1 Overview

Although computer technology has evolved dramatically in recent years, some applications are limited by intense computational requirements. Examples of these include aircraft design; weather prediction; and computational fluid dynamics (5, 13).

Some computer applications, such as matrix operations and search problems, may execute faster when *parallel processing* is used. If a designer can decompose the problem into tasks, each processor in a parallel architecture can work on a subset of the original problem. This may be done through *data decomposition*, a technique which partitions a data structure into pieces with different processors working on separate parts of the problem, or by *control/algorithm decompositon*, which allows different processors to perform different functions (31). Dividing the problem in this manner may enable the overall solution to develop faster than on a sequential (one-processor) machine.

After a large problem has been decomposed into tasks, each task must be assigned to a processor so that the computation can be performed. Assigning tasks to processors in an optimal schedule to minimize execution time is, in general, a *NP-complete* problem (10). All NP-complete problems have certain characteristics:

- There is no known polynomial-order solution for any NP-complete problem.
- NP-complete problems have combinatoric or exponential search spaces.
- Every NP-complete problem can be mapped (in polynomial time) to every other NP-complete problem. This implies that *if a polynomial-time (p-time) solution is found for* any *NP-complete problem, then a p-time solution can be found for* all *NP-complete problems.*

NP-complete problems can be approached in several ways:

- If an approximate solution is acceptable, heuristics can be developed to produce "good" solutions for specific cases, rather than *optimal* solutions.
- Informed search strategies can be used to reduce the amount of time needed to find an optimal solution in most instances. There are, however, cases where informed search strategies produce no better results than an exhaustive search.
- The problem can be restricted so that it conforms to a problem which has a known polynomial-time solution.

This thesis investigation concentrates on methods for finding an optimal solution to the task scheduling problem, as defined by iterative task systems. In iterative systems, such as those reflected in electronic circuit simulations, each task executes several times during the course of the simulation. This problem is shown to have a polynomial-time solution, for highly-restricted cases, which means that a deterministic Turing machine (DTM) can solve the problem with an algorithm of complexity $O(n^c)$, where c is constant (?). Most task scheduling problems, however, require NP-time (10), which means that a nondeterministic Turing machine (NDTM), can solve the problem with an algorithm of complexity $O(n^c)$, but that a DTM cannot (?).

1.2 Simulation Tasking Background

The development of electronic circuits, such as Very Large Scale Integration (VLSI) circuits, involves numerous steps, from circuit design to physical implementation (30). One attempt to streamline the process of circuit design uses the *Very High Speed Integrated Circuit (VHSIC) Hardware Design Language*, or VHDL, a design methodology which allows VLSI circuit behavior to be modelled on a computer (2). If VHDL simulations efficiently model circuit behavior, the turnaround time from design to final implementation could be decreased; instead of building intermediate designs and physically measuring the outputs, computer simulations could be used to iteratively refine the circuit before implementation.

Unfortunately, VHDL simulations for current applications take a disproportionate amount of computer time compared to the size of the circuit which is modeled. For

example, one projection estimates that simulating a VHDL design for a circuit with 100,000 transistors would require 700 hours to execute on a VAX 11/780 computer (30).

Previous research (30), attempted to decrease the time needed for VHDL processes by running VHDL test cases on a parallel computer. Although successful VHDL circuit simulations were implemented in parallel, the parallel simulation required approximately the same amount of time as the sequential version. This may have been due to any of several factors which prevent parallel applications from achieving the theoretical speedup:

- *Communications Overhead:*

  The simulation must be amenable to parallelization. For example, if each simulation task has a small amount of computation compared to the required amount of communication, then the simulation can spend more time communicating between processors than performing actual work.

- *Inefficient Loading:*

  The task scheduling process, which assigns sub-problems to different processors, must provide an assignment so that each processor is kept busy. If tasks are assigned to processors so that processors are not fully utilized, execution time is dominated by the processor with the heaviest workload. For example, tasks are assigned to 3 processors in Figure 1.1. Although the average processor finishing time is 17, the finishing time for the entire job is 21.

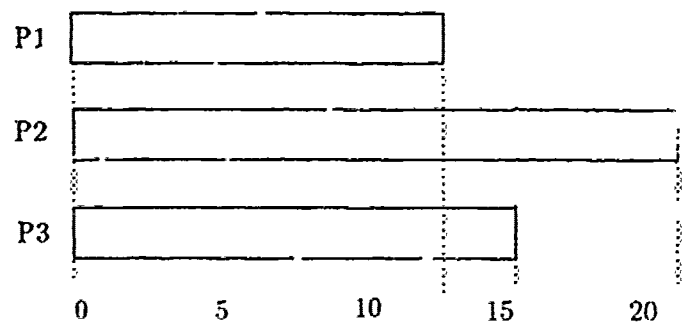


Figure 1.1. Unbalanced Processor Loads

Since dependencies exist between inputs and outputs in VHDL tasks, the process of assigning tasks to processors involves *precedence-constrained* scheduling. In a precedence-constrained system, some tasks must delay until previous tasks are complete. For example, the following task system, shown in Figure 1.2, has a constraint between tasks:

Task 1: Calculate $a = (b + c)/(364 \times 972)$

Task 2: Calculate $g = a/2094$

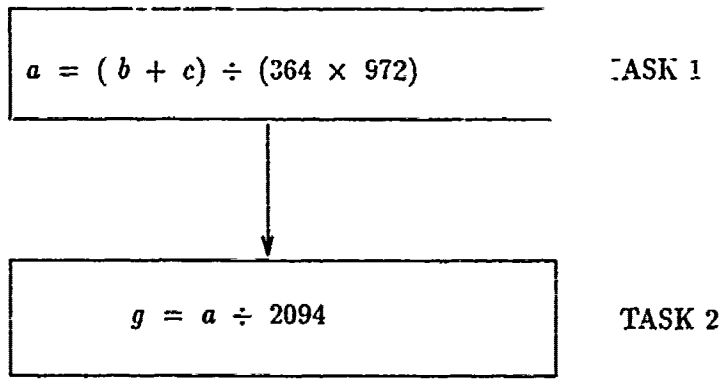


Figure 1.2. Relationships Between Tasks (1) and (2)

In this task system, Task 2 cannot begin until a value is given for a. Since the value of a is the final result of Task 1, Task 2 cannot begin until Task 1 has completed. Therefore, Task 2 is *constrained* by Task 1.

The problem of mapping precedence-constrained tasks to processors to obtain an optimal schedule falls into the class of NP-complete problems (10, 15). One characteristic of NP-complete problems is that a large search space exists implicitly and is partially generated explicitly when deriving optimal solutions; another characteristic of these problems is that an optimal solution may require an exceedingly long time to generate using a graph search algorithm (4).

For example, suppose an optimal schedule is desired for 60 independent tasks assigned to 2 processors. If the problem is simplified by assuming that only 30 time slots can be filled on each processor, it is possible to derive the number of possible combinations:

- There are 60 choices for the placement of the first task.

- After the first task is chosen, there are 59 choices for the placement of the second task.
- After the second task is chosen, there are 58 choices for the placement of the third task.

  ⋮

- For the final task, there is one choice for the placement.
- Thus, there are $60 \times 59 \times 58 \times \ldots \times 1 = 60!$ unique ways to schedule 60 tasks on 2 processors so that there are 30 tasks on each processor.

In order to determine an optimal schedule for this problem by exhaustive search, 60! combinations must be examined. Assuming that schedules are generated by a computer at the rate of 1,000,000 schedules per second, the time required to generate all schedules would exceed hundreds of billions of centuries!

$$\frac{60!\ schedules}{\frac{1{,}000{,}000\ schedules}{second} \times \frac{60\ seconds}{minute} \times \frac{60\ minutes}{hour} \times \frac{24\ hours}{day} \times \frac{365\ days}{year} \times \frac{100\ years}{century}}$$

$$= \frac{8.321 \times 10^{81}}{3.15 \times 10^{15}} = 2.63 \times 10^{66}\ centuries$$

Thus, optimal schedules must be derived using methods other than exhaustive search. (Appendix A contains more complete calculations for this example).

Intelligent choice of search techniques can generate the optimal schedule without exploring all possible combinations. One method of informed search is called *branch and bound.* Branch and bound techniques place a bound, or limit, on the branches of the search space which are traversed (8). As partial solutions exceed the cost of the current solution, the search abandons the high-cost branch and backtracks to a previous state. In this way, a limited search, rather than an exhaustive search is used to achieve an optimal solution.

For example, suppose 4 independent tasks are to be scheduled on 2 processors. These tasks may execute in any order, and there are no restrictions which limit the number of tasks assigned to any processor.

Tasks 1, 3, 4 take 2 time units to execute.

Task 2 takes 1 time unit to execute.

Figure 1.3 shows a partial search tree for this problem.

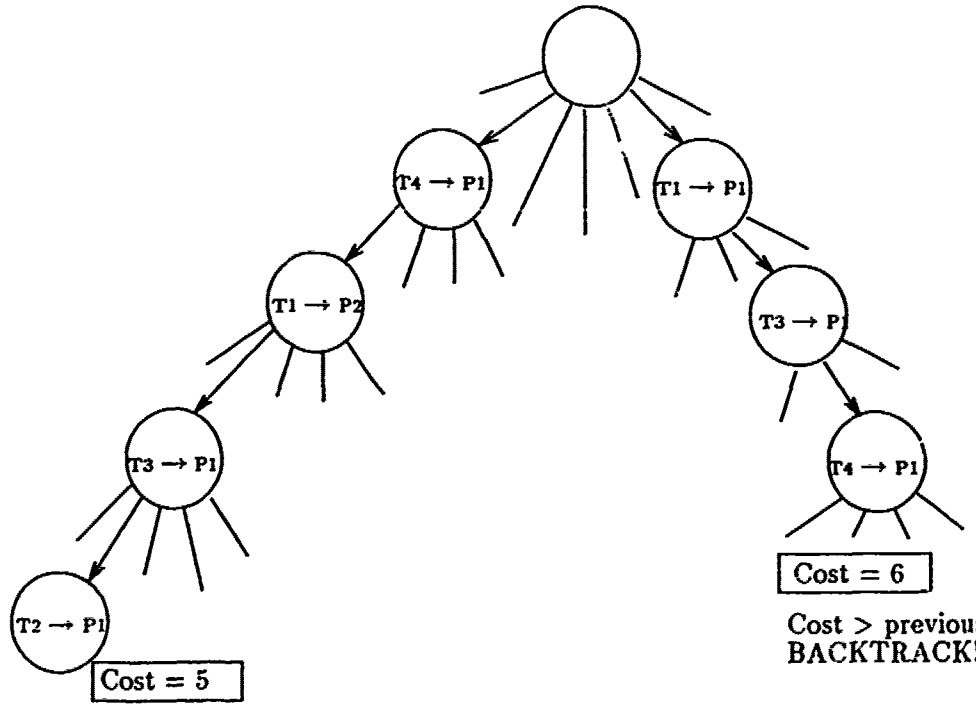


Figure 1.3. Branch and Bound Search Tree

At some point in the search, the following schedule is generated (shown on the left branch of Figure 1.3):

P1	T4	T4	T3	T3	T2
P2	T1	T1			

Since this schedule can be completed in 5 time units, all partial schedules of more than 5 time units can be abandoned. For example, the partial schedule shown on the right branch

of the search tree exceeds the current minimum complete schedule; thus, this schedule and its variations can be eliminated from consideration:

P1	T3	T3	T1	T1	T4	T4
P2						

Using techniques which allow the problem to be bounded by considering only those options which fall beneath the current lowest-cost schedule, it is possible to generate an optimal schedule to an NP-complete problem without an exhaustive search of all alternatives in most cases.

1.3 Task Scheduling Problem

Timing analysis has shown that large computer applications can often execute faster on a parallel architecture than on a sequential machine (22). When VHDL circuit simulations were ported to the parallel iPSC/2 Hypercube, however, the execution time remained the same as for serial implementations (30). This may have been due to communications overhead or to unbalanced processor workload. In order to distribute the workload among processors, the mapping process, which assigns tasks to processors, must be examined.

Previous AFIT research (30) used a *greedy,* or polynomial-time, technique to generate a quick solution when mapping tasks to processors. In this method, the assignment strategy is based on a candidate selection, but optimal results are not guaranteed. It is possible that the overall execution time of the simulation can be reduced if more informed techniques are used to assign VHDL tasks to processors.

In general, the task scheduling problem attempts to assign tasks to processors in such a way that some criterion is optimized. (A formal definition is given in Chapter 2). The VHDL problem attempts to assign n tasks to m processors (where $n \gg m$) in such a way that overall execution time is minimized.

1.4 Assumptions

Before the scheduling process is begun, the task system must be defined. At a minimum, the following must be known (10):

- number of tasks
- execution time for each task
- task precedences
- number of processors

It is also assumed that the parallel simulations generated by previous AFIT research (23, 30) work effectively.

1.5 Scope (Context)

The primary focus of this study is the problem of assigning n precedence-constrained tasks to a multiprocessor system consisting of m identical processors such that $n \gg m$ and such that a minimal schedule in terms of overall execution time is generated. Since scheduling theory, even with these constraints, encompasses a vast range of problems (9), this investigation is further restricted to simulations which meet the following criteria:

- The task graph contains no feedback loops.
- Messages between processors are held in a buffer until required.
- Every task in the task graph is executed multiple times.
- Once a task is assigned to a processor, all subsequent iterations of that task remain on that processor.

Using informed search techniques, an algorithm for assigning tasks to processors in an optimal manner is developed. This algorithm, which derives a minimal schedule based on the iterative nature of simulation tasks, is proved to be optimal in terms of *latency*, the time between successive iterations of a task.

Finally, parallel VHDL implementations generated by previous AFIT research (30) are executed with the following scheduling algorithms on the 8-node iPSC/2 Hypercube and results are compared:

- greedy algorithm
- optimal one-iteration algorithm
- level strategy
- unbalanced mapping

1.6 Approach

This research begins with an analysis of scheduling algorithms which generate optimal mappings of tasks to processors. These algorithms are examined to determine an optimal method for scheduling tasks which conform to circuit simulation constraints.

Existing algorithms fail to capture one of the primary aspects of simulation task systems: the simulation *iterates* through each task in the system numerous times during execution. This iterative behavior dominates all other mapping considerations in some simulations. A mapping strategy based on iterative tasks is developed. This strategy is proved to result in optimal execution time for highly-restricted systems of simulation tasks. As constraints are loosened, the problem of generating optimal solutions to iterative task systems is shown to be NP-complete.

The next aspect of this effort requires an implementation of mapping algorithms to generate optimal schedules for mapping n tasks to m processors, where $n \gg m$. The implementation produced as a result of this thesis investigation directly supports mapping strategies for iterative tasks in a precedence-constrained environment.

Finally, execution time is analyzed with respect to varied mapping strategies, using VHDL and gaming simulations which were implemented in parallel in previous research (23, 30).

1.7 Structure of Thesis

Chapter 2 of this investigation contains a detailed background on the general task scheduling problem. The general problem is covered in detail, as a predecessor to the iterative problem. In order to allow the iterative problem to be scheduled in polynomial time, restrictions are defined, and the environment is limited to identical processors.

In Chapter 3, factors which impact the design of the level strategy algorithm are defined. The search process is discussed, and methods for reducing the search space are presented.

A detailed mathematical basis for the iterative scheduling problem is given in Chapter 4. When the problem is restricted to Unit Execution Time (UET) tasks with specific precedence, the level strategy is formally proved to produce optimal mappings in terms of latency. A proof is given to show that the iterative problem with variable-execution-time tasks is NP-complete. A variation on the basic problem determines the optimal number of processors for variable-execution-time tasks.

Chapter 5 presents experimental results for simulation runs tested with various mapping strategies. The level strategy, a greedy algorithm, and an unbalanced assignment are compared for identical simulation runs.

Conclusions are summarized in Chapter 6, and recommendations for future study are given.

II. Background

Chapter 1 gives a brief introduction to the precedence-constrained scheduling problem, and to the ultimate goal of task scheduling for VHDL simulations. In this chapter, more detailed aspects of the problem are considered. The general scheduling problem, which encompasses many different variants, is restricted to conform to the VHDL mapping problem. Decisions which impact the mapping algorithms are made, based on *a priori* knowledge of the simulation. Further constraints allow the iterative scheduling problem to be mapped to an optimal solution in polynomial-time for some cases.

2.1 General Task Scheduling Problem Description

The problem of assigning tasks to processors in an optimal manner is referenced by several names: the Assignment Problem (6); the Mapping Problem (which takes the characteristics of the target machine into consideration) (6); and the Scheduling Problem, described below (10). The general scheduling problem may be defined in terms of the available resources, task systems, sequencing constraints, and performance measures (10):

- System Resources

  System resources consist of a set of m processors $\{P_1, ..., P_m\}$

  Additional resource types $\{R_1, ..., R_s\}$ (for example, I/O devices), may also be considered.

- Task Systems

  Defined by $(\mathcal{T}, \prec, [\tau_{ij}], \{\mathcal{R}_j\}, \{w_j\})$, where

  Tasks: $\mathcal{T} = \{T_1, ..., T_n\}$ is the set of tasks to be executed

  Precedence: $\prec$ is an (irreflexive) partial order defined on T which specifies precedence constraints. $T_i \prec T_j$ signifies that T_i must be completed before T_j can begin.

  Execution Times:$[\tau_{ij}]$ is an $m \times n$ matrix of execution times, where τ_{ij} is the time required to execute T_j on processor P_i.

  Resources: If additional resources, such as I/O devices, must be considered in the scheduling problem, these resources are designated by a vector: $\{\mathcal{R}_1(\mathcal{T}_j\}), ..., \mathcal{R}_s(\mathcal{T}_j\})$.

The ith component of this vector specifies the amount of resource type $\mathcal{R}_i$ required through the execution of task T_j, $(1 \leq j \leq n)$. This constrains the problem so that a set of tasks requiring more of a given resource than is allowed in the system cannot execute at the same time.

For example, suppose that available resources consist of 3 I/O devices and 4 Processors. If 4 tasks, each requiring 1 I/O device are to be scheduled, they cannot all execute in the same time slot. Although several different types of resources may be allowed, this investigation concentrates on scheduling only one resource type (processors).

Weighting: The weights $\{w_j\}$ are interpreted as cost rates, which are taken as constants. For example, each task may be weighted with a profit factor or a tardiness penalty.

- **sequencing constraints**

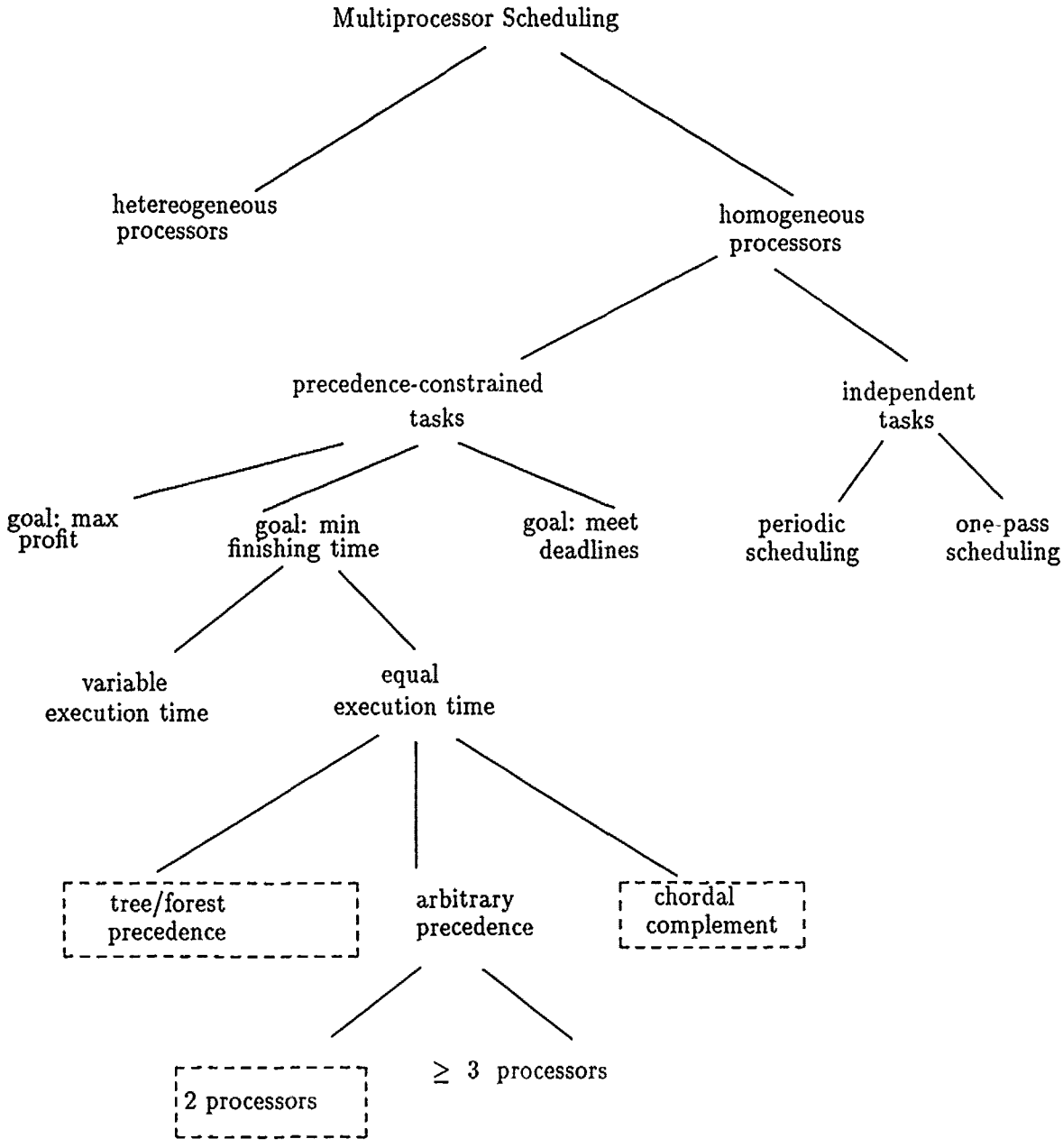
  non-preemptive: tasks run uninterrupted until completion.

  preemptive: tasks may be interrupted during execution.

- **performance measures:** Performance measures define the criteria that must be optimized as the schedule is built. For example, the overall goal of the scheduling process could be to minimize the schedule length (or execution time). Another goal relates to the weighting criteria which may be associated with each task; objectives such as *maximize profit and minimize tardiness* fall into this category.

There are many different variations of the basic scheduling problem. Figure 2.1 shows the relationships between some of these variants, based on such parameters as processor type (identical or heterogeneous) and precedence structure (tree, forest, arbitrary, ...) (10).

The scheduling problem considers both single-processor and multiprocessor scheduling. The single-processor case attempts to assign many tasks to one processor in order to minimize or maximize some criterion. For example if each task is associated with a deadline and a penalty for each missed deadline, the goal of the schedule might be to assign tasks so that the penalty is minimized. The multiprocessor scheduling problem can involve



Note: Boxes denote polynomial-time solutions

Figure 2.1. Relationship Between Multiprocessor Scheduling Problems

identical (homogeneous) processors or heterogeneous processors, which operate at different rates. Within the homogeneous processor branch of the tree, precedence-constrained task systems involve tasks which are related in some manner. Several goals for scheduling precedence-constrained systems are considered:

- Minimize execution time.
- Minimize weighted factors (tardiness, deadlines)

Within the precedence-constrained, homogeneous processor condition, task systems with unit execution time (UET) differ from those with variable execution time. Although both cases contain NP-complete problems, there are no known polynomial-time algorithms for mapping variable-execution-time tasks. Figure 2.1 indicates that polynomial algorithms which find optimal solutions to the scheduling problem are rare: For example, a task system with arbitrary precedence can be mapped to 2 processors in polynomial time. If there are 3 or more available processors, there is no known polynomial-time algorithm which provides an optimal solution (10). Table 2.1 illustrates the limited range of p-time solutions to the scheduling problem (10).

Number of Processors	*Task Lengths*	*Precedence*	*Problem complexity*
arbitrary	equal	forest	$O(n)$
2	equal	arbitrary	$O(n^2)$
fixed, $\geq$ 3	equal	arbitrary	Open
arbitrary	equal	arbitrary	NP-complete
fixed, $\geq$ 2	1 *or* 2	arbitrary	NP-complete

Table 2.1. Results for Minimizing Execution Time (non-preemptive; no resource constraints)

One version of the classical scheduling problem attempts to **minimize the overall execution time** when n related tasks are to be mapped to m identical processors (10). Since the goal of parallelizing VHDL processes is to reduce overall execution time, this version of the scheduling problem is most pertinent to this research.

2.1.1 Algorithm Considerations The scheduling problems examined in this investigation involve multiprocessor scheduling with precedence-constrained task systems on identical processors. At this stage, some decisions are required to determine the types of scheduling algorithms to be considered. A taxonomy of scheduling problems (Figure 2.2) highlights these decisions (9):

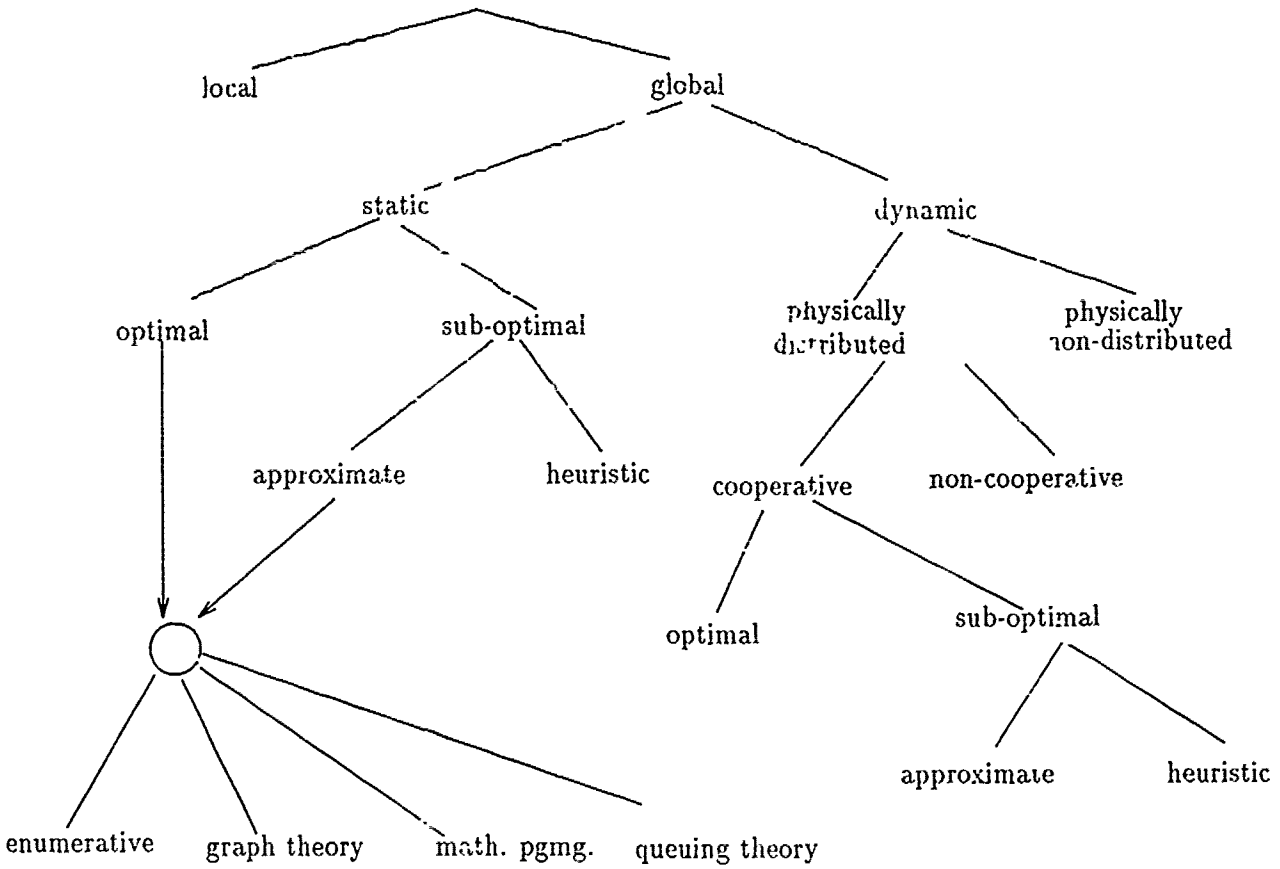


Figure 2.2. Taxonomy of Scheduling Problems

- *Approximate vs. Optimal*

  Because of the inherent intractibility of NP-complete problems, numerous researchers have developed *approximation algorithms* which find near-optimal solutions for the scheduling problem (3, 12, 21). In most cases, however, approximation algorithms are not guaranteed to be within a specified ϵ of the optimal solution; if an optimal solution

is required, the problem is still NP-complete. Therefore, this research considers the generation of *optimal* solutions for the scheduling problem (10, 32).

- *Static vs. Dynamic*

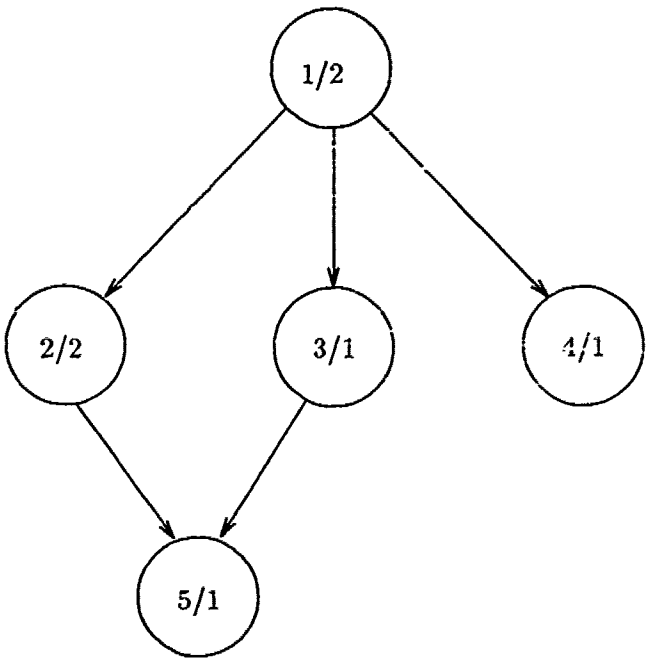
  Processors may be assigned their workload in one of two ways: static assignment or dynamic load-balancing. Static assignment divides the problem into 'chunks.' Each processor is given one chunk of the problem to solve. If a processor finishes its portion of the problem, it remains idle until all other processors have completed their workload.

  Dynamic load balancing algorithms begin with an assignment of work for each processor. As each processor completes its assigned tasks, the workload is redistributed so that idle processors take on tasks which were originally assigned to other processors. In order to make this method practical, the overhead associated with redistributing work must be offset by the gain in processor efficiency. Since the VHDL simulations have been constructed so that all work is assigned to processors only at the start of the simulation (30), static assignment techniques are considered for this research.

2.1.2 Notation and Diagrammatic Representation. Precedence-constrained task systems can be defined in terms of the relationship between tasks. In a task graph representation, each task is represented by a circle on the chart, with task numbers and task execution times in the circle. Directed arcs between tasks indicate precedence. Figure 2.3 shows an example of a task graph, consisting of 5 tasks to be scheduled onto 3 processors. Task 1 requires 2 time units to execute. The arrow between task 1 and task 2 indicates a precedence relationship, $(t1 \prec t2)$, which means that task 1 must complete execution before task 2 can begin. The complete set of precedence relationships can be expressed as follows:

$$\{(t1 \prec t2 \prec t5),(t1 \prec t3 \prec t5),(t1 \prec t4)\}$$

Since the longest chain of tasks requires 5 time units, an optimal schedule for this task system also requires 5 time units, as shown in the Gantt chart of Figure 2.3.



P1	T1	T1	T3		
P2			T2	T2	T5
P3			T4		

Figure 2.3. Task system

2.2 Classes of Repeating Schedules

The classical scheduling problem is concerned with a single pass through the task system; however, VHDL simulations iterate through the system numerous times. Several categories of nonpreemptive scheduling problems (*periodic scheduling, fixed-cycle scheduling, and iterative scheduling*) require tasks to repeat multiple times. These categories are summarized in Table 2.2.

In *periodic scheduling*, each task is associated with a repetition frequency (or period). The objective of periodic scheduling is to assign tasks to processors such that all tasks execute within their given period (32). Variations include minimizing the number of processors and scheduling tasks so that no task executes before a given release time.

Task systems which use *fixed-cycle scheduling* are given a set of tasks and a cycle-length (16). The objective of fixed-cycle scheduling is to determine the time points where a new task set must start in order to minimize the delay in task sets which have already begun execution. This problem has polynomial-time solutions in two cases: no precedence constraints; unit-execution-time tasks *and* serial precedence constraints; unit-execution-time tasks (16). One variation attempts to minimize the number of processors when tasks are not precedence-constrained, an NP-complete problem (27).

This investigation introduces the concept of *iterative scheduling,* which attempts, under specific constraints, to maximize the repetition rate at which each task is executed.

Type	*Parameters*	*Goal*	*Comments*
periodic	n tasks execution time E_i task period T_i $\prec$	min # processors	*NP complete* if empty precedence (32) *NP complete* if precedence-constraints (32)
fixed-cycle	cycle-length n m processors unit execution time l tasks $\prec$	minimize delay by choosing times to insert tasks	p-time for empty precedence (16) p-time for serial precedence (17)
fixed-cycle	cycle-length n m processors execution time l_i l tasks	min # processors	*NP complete* (27)
iterative	n tasks m processors execution time l_i $\prec$	minimize latency	$O(n^3)$ if equal execution time *NP complete* if variable execution time
iterative	n tasks m processors execution time l_i $\prec$	min # processors	*NP complete* (27)

Table 2.2. Categories of Non-Preemptive Repeating Schedules for Multiprocessors

2.3 *Latency*

The iterative scheduling problem attempts to minimize overall execution when each task in the system is required to execute more than once. When multiple-iteration systems are scheduled, the concept of *latency,* time between successive iterations of a given task, is used to measure the quality of the mapping.

Since both the one-pass and the iterative problems have many similarities, a study of the classical scheduling problem yields insight about the iterative scheduling problem, especially for cases where efficient solutions for the classical problem are known to exist. In particular, the *level strategy,* which assigns tasks based on the longest chain of unscheduled tasks, forms a core for the iterative scheduling algorithm.

Finding an optimal schedule for a precedence-constrained task system is usually an NP-complete problem (15)(Table 2.1). This means that there is no known efficient (polynomial-time) algorithm to solve the optimization problem.

2.4 *General approaches to the scheduling problem*

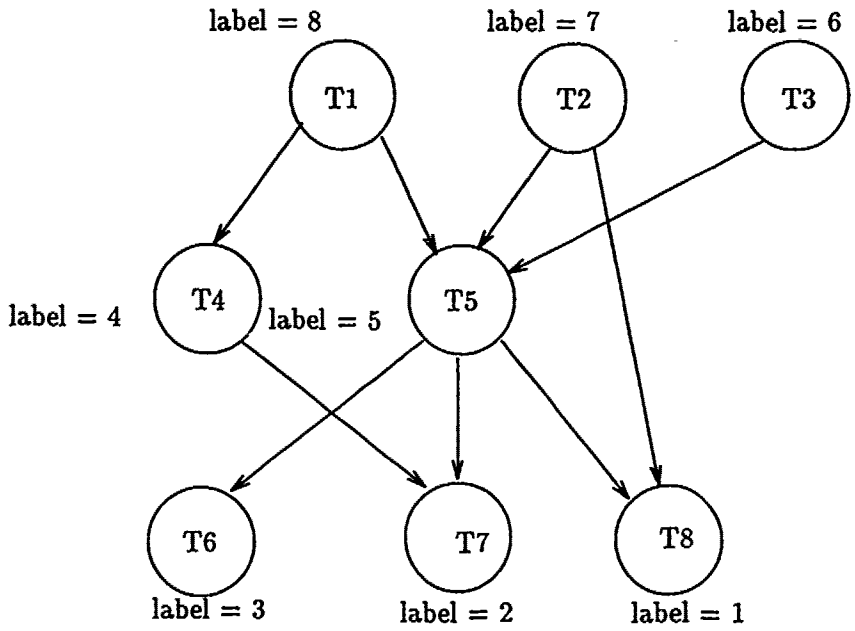
As shown in Table 2.1, the problem of assigning tasks to processors in an optimal manner has been proved NP-complete (33) for most task systems. There are, however, efficient algorithms for several specialized instances of this problem (15):

- Two identical processors; arbitrary precedence (10)

  If the multiprocessing system consists of two identical processors, then a *list strategy,* which scans an ordered list of prioritized tasks each time a processor is free, is used to build the schedule. Tasks are prioritized based on the number of immediate successors, beginning with the terminal nodes; ready tasks with the highest priority are assigned first. Since this algorithm assigns higher-level tasks first, simulation mappings which fit this schema can be expected to perform well. Figure 2.4 shows an arbitrary precedence graph and an optimal 2-processor schedule generated by this gorithm.

- Arbitrary number of processors; precedence graph is a forest structure (10)

  If the task system precedence graph is a *forest* of trees (each node has only one source



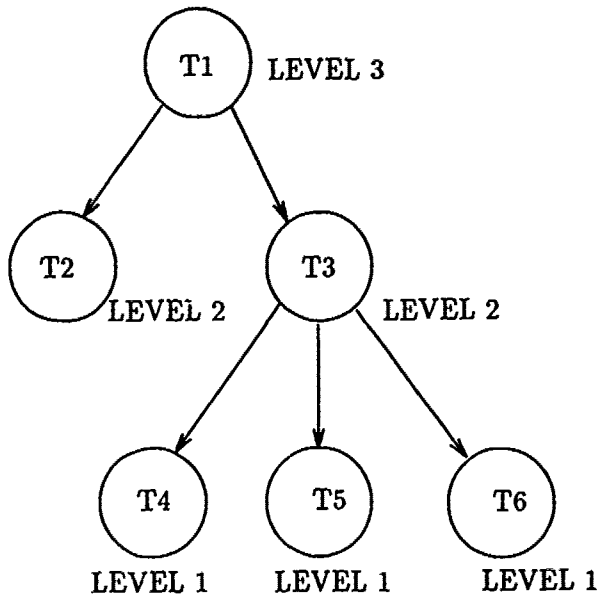
P1	$T1$	$T3$	$T5$	$T7$	
P2	$T2$	$T4$		$T6$	$T8$

Figure 2.4. Arbitrary Precedence Graph with Optimal Schedule

[sink]), then a *level strategy,* which computes the level for each task, [$level(x) = max$ $\sum$(execution time associated with the nodes in a path from x to a terminal vertex),] for assigning tasks to processors so that overall execution time in minimized. As in the above case, this algorithm can be used to derive an optimal iterative algorithm for simulation tasks; since the assignment is made level-by-level, tasks are assigned in much the same manner as the iterative level strategy. Figure 2.5 shows an example of this type of system.

- chordal complement

  Often, *a priori* knowledge about a system can provide insight about improved ways to solve a specific problem. If it is known that the graph of the task system has a chordal complement, [i.e. The complemented graph has chords connecting vertices in



P1	T1	T2	T4	T6
P2		T3	T5	

Figure 2.5. Tree-Structured Precedence Graph with Optimal Schedule

each subgraph of 4 or more nodes.], then there is a p-time algorithm which generates an optimal schedule (28). In general, however, the question "Is graph G a chordal graph?" is itself an NP-complete problem (15). In large simulations, the question of chordality may overwhelm any benefit which could be derived from the algorithm. Figure 2.6(a) shows a graph containing 4 nodes. In Figure 2.6(b), the complement (obtained by placing arcs between all vertices which are unconnected in the original graph, and removing the arcs of the original graph) of the graph in (a) is shown .

- heterogeneous processors; cyclic forest

  In the case of heterogeneous processors, even the cases described above are NP-complete (18). However, if the tasks fall into a precedence relationship such that the tasks form a 'cyclic forest,' (all tasks at any given level *must* execute on the same processor), then there is a p-time algorithm that generates an optimal schedule (18).

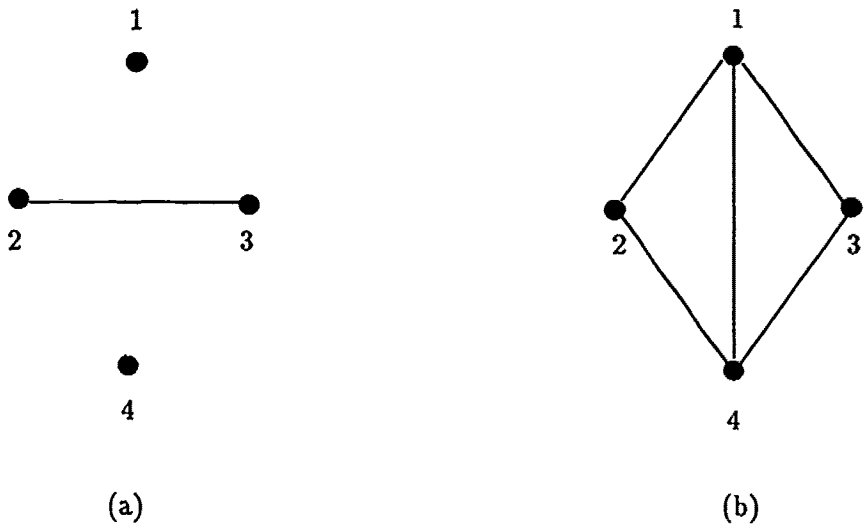


Figure 2.6. Graph(a); Chordal Complement(b)

Since this effort is concerned with **homogeneous** processors, and since the tasks in this simulation are not pre-assigned to a specific processor, the cyclic forest algorithm does not apply.

2.5 Iterative nature of the problem

Although the problem of mapping tasks to processors in a simulation environment has many similarities to the general scheduling problem, the task system may be required to execute multiple times during the course of the simulation. For example, the Corps Battle Analyzer (CORBAN) simulation executes tasks LOOK, SHOOT, DECIDE, MOVE, COMMUNICATE, RECOVER, PROVIDE, and SUPPORT at each time step in the simulation (11). A precedence graph for this simulation is shown in Figure 2.7.

If these are the only tasks in the simulation, and if these tasks were to be mapped to a 3-processor system, an optimal mapping, (in the classical sense) could assign all tasks to one processor. This ensures that communication is minimized, which, in a one-iteration mapping, would be a reasonable decision. **But** if multiple iterations of the task system are needed for a complete simulation run, then a mapping which assigns each task to a separate processor can complete the simulation in less time, assuming that certain conditions are

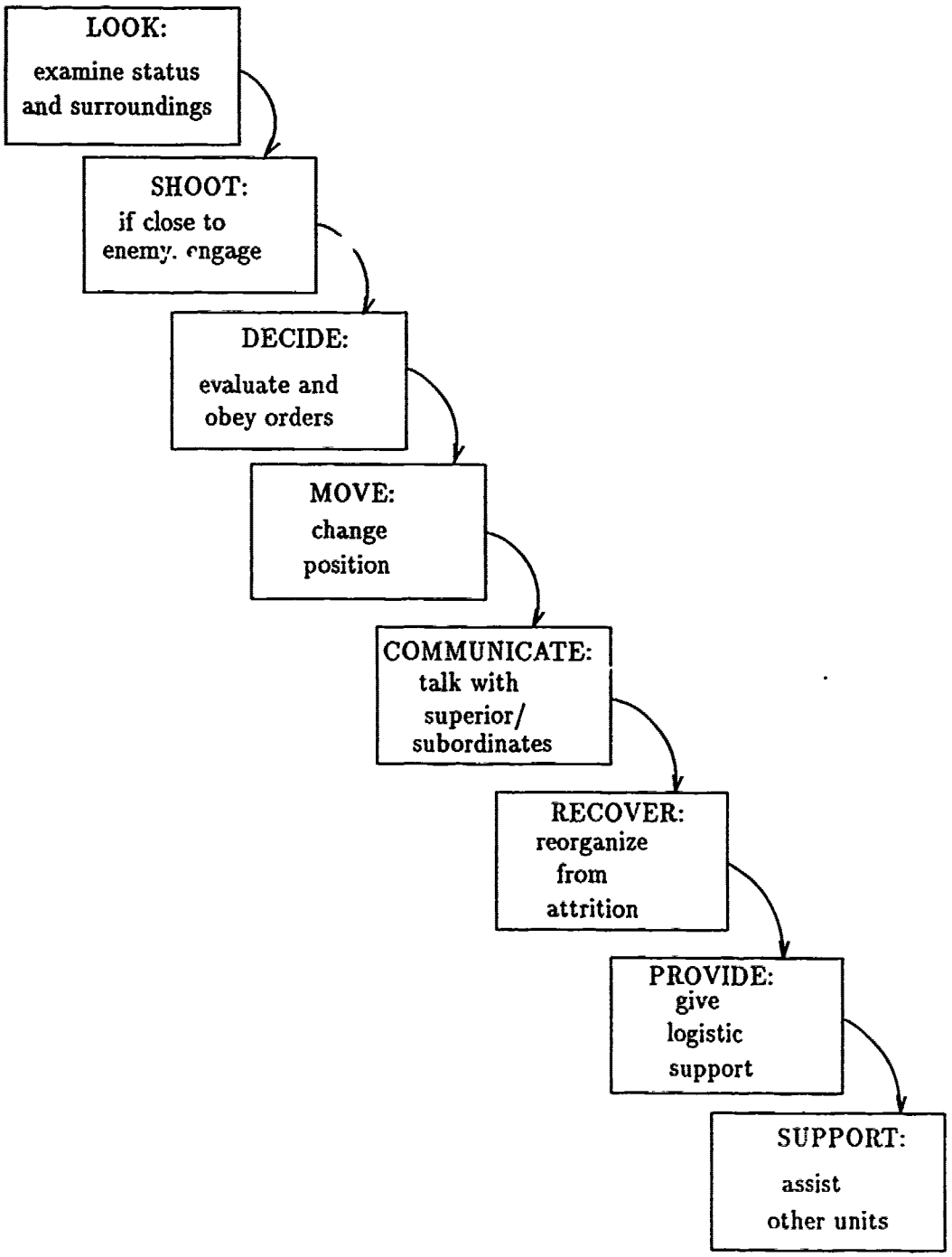
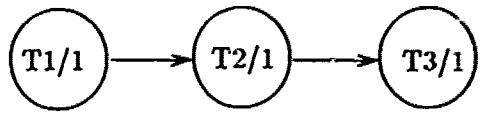


Figure 2.7. CORBAN Tasks

met. For example, Figure 2.8 shows a task system and a comparison of an optimal one-iteration mapping which is executed 3 times and an optimal 'pipelined' mapping, which also requires each task to execute 3 times.



P1	T1	T2	T3	*T1′*	*T2′*	*T3′*	*T1″*	*T2″*	*T3″*
P2									
P3									

3 iterations of 1–pass mapping: 9 time units

P1	T1	*T1′*	*T1″*		
P2		T2	*T2′*	*T2″*	
P3			T3	*T3′*	*T3″*

3 iterations of pipelined mapping: 5 time units

Figure 2.8. Optimal (1–pass) vs Pipelined

2.6 Assumptions and Environment

The general scheduling problem comes in many variations; so does the iterative scheduling problem. This section discusses the restrictions which the current VHDL simulations, implemented on the iPSC/2, impose on the iterative scheduling problem:

- homogeneous processors

  Since an 8-node Intel iPSC/2 hypercube was used for previous research, this thesis concentrates on solutions for identical processors. Athough 8 nodes are used for all experiments, the mapping strategy developed as a result of this thesis can be extended to any number of nodes.

- no preemption

  The algorithms for preemptive task assignment differ significantly from non-preemp-

tive algorithms. Since preemptive algorithms require specific knowledge about timing information (for example, time that a process is suspended for memory reads), no preemption is assumed throughout.

- unit-execution-time tasks

  All tasks in the system are assumed to have equal execution time. In the VHDL simulation, each VHDL assignment statement is a separate task. The very small amount of computation required by each of these tasks (5 or 6 floating-point operations) produced unstable timing results when the simulation was executed. For example, one execution took 18 seconds. The next execution, with identical parameters, ran in 42 seconds. In order to stabilize the test runs, spin loops, (large computational segments of code which force the processor into a small $\frac{communication}{calculation}$ ratio), have been added to dominate the activity of each task. This allows task-to-processor assignment to be based solely on the computation time of each task. In the simulation, spin loops of 100,000 operations were added to each task to ensure that calculation is the overriding factor and that all tasks have equal execution time.

- queued message passing

  The VHDL simulations implemented on the iPSC/2 use a queued-message paradigm (30). Messages from one task to another are buffered until the recieving task is ready to receive the message. This allows predecessor tasks to send messages to successor tasks and then to execute subsequent iterations without waiting for successor tasks to begin. This protocol permits simulation execution time to be minimized when an pipelined approach is used for task assignment.

- task exists on only one processor

  In the current VHDL simulations, all iterations of a task execute on the same processor as the initial iteration of that task. Tasks are loaded onto computing nodes at the start of the simulation and remain on those nodes until simulation is complete. (No dynamic load-balancing.)

- no feedback loops

  The level algorithm developed as a product of this research is based on the assumption that all task systems can be represented by directed acyclic graphs (dags). This

limits the simulations to 'forward flow' applications which do not have feedback loops between tasks. Such applications include various **wargame simulations, logic circuit simulations,** and **assembly-line simulations.**

2.7 Additional Considerations

If communication time must be incorporated into the scheduling problem, then there is no known case where a p-time algorithm exists (12).

If the precedence graph of the system to be modeled contains cycles and if buffer sizes between communicating nodes must be considered, then a methodology exists to transform these graphs into directed acyclic graphs (dags). These dags may be scheduled, using the one-pass level algorithm, to obtain near-optimal solutions in terms of overall execution time (24).

2.8 Summary

There are many variations of the classical problem for mapping tasks to processors. This research explores optimal solutions for the multiprocessor, precedence-constrained problem, with identical processors. Since simulation tasks iterate through many passes of the task system, iterative scheduling techniques are investigated. Primary emphasis is given to tasks with equal computation requirements, which reflects the system used for existing VHDL simulations.

The problem of deriving an optimal assignment for a precedence-constrained task system is essentially a search problem. In highly-restricted cases, a polynomial-time algorithm will produce an optimal schedule for iterative task systems; however, most instances of the iterative scheduling problem require sophisticated search techniques to generate an optimal solution.

III. Scheduling Algorithm Design

Although simulation tasks differ from the classical scheduling problem in that they iterate numerous times through the task system, insight into the iterative problem is gained from a study of the classical problem. This chapter discusses the design of the Level Strategy for iterative task assignment. The Level Strategy is a polynomial-time algorithm which produces optimal latency schedules for highly-restricted forms of the iterative problem.

Since the Level Strategy only provides optimal schedules for restricted forms of the iterative scheduling problem, the rationale is presented for limiting the search space in more complex forms of this problem. Tests for optimality are discussed. Heuristics are chosen to guide the A* search algorithm (29) which uses additive costs to generate optimal schedules for an NP-complete variation of the iterative task scheduling problem.

3.1 Basis for Level Strategy (One-pass Systems)

Polynomial-time algorithms which provide optimal solutions to the classical scheduling problem can be found for limited parameter constraints. Of these algorithms, there are two theorems which provide insight into the iterative scheduling problem (10):

1. Unit Execution Time (UET) tasks; $\prec$ a forest; arbitrary number of processors
   The algorithm which gives an efficient solution for this case uses a level strategy of assignment: The precedence graph is divided into levels, and tasks at the highest level are assigned first.

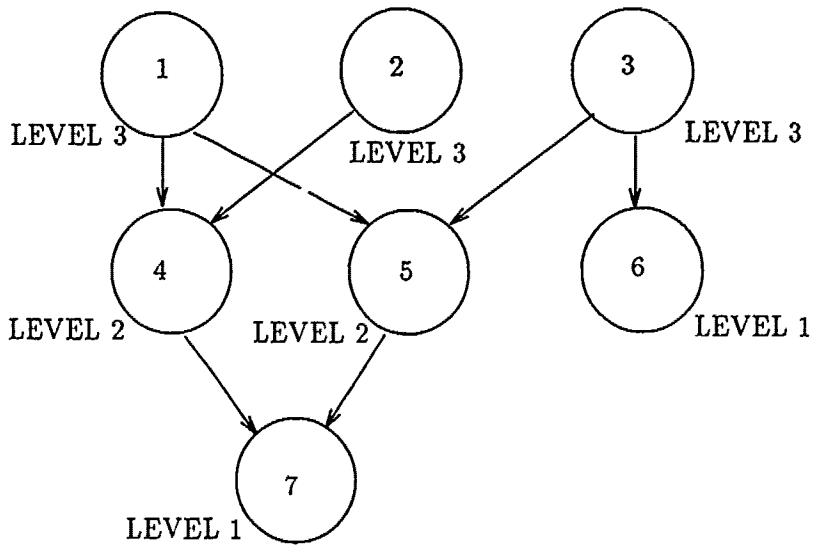
2. UET tasks; $\prec$ arbitrary; 2 processors
   The p-time algorithm is based on number of successor tasks: Ready tasks are prioritized according to the successors of each task. The ready task with the highest priority is then assigned to the first available processor.

3.1.1 One-Pass Level Algorithm The level algorithm for single-execution task systems is as follows (10):

Let the *level* of a node x in a dag [directed acyclic graph] be the maximum number of nodes (including x) on any path from x to a terminal task. In a forest, there is exactly one such path. A terminal task is at level 1. Let a task be *ready* when all its predecessors have been executed.

Level strategy: whenever a processor becomes available, assign it an unexecuted ready task at the highest level (farthest from a terminal).

The level strategy for iterative tasks uses the same criteria as the one-pass level strategy: Tasks are assigned to a level, based on their distance from the terminal task. Ready tasks at the highest level are assigned first. Since no transitive arcs and no level-traversing arcs are allowed in the task system, successor tasks become ready as soon as their predecessor level is finished. This allows all tasks at the successor level to be assigned without delay. Figure 3.1 shows an example of assignment made by the level strategy.



P1	$T1$	$T5$	
P2	$T2$	$T6$	
P3	$T3$		$T7$
P4		$T4$	

Figure 3.1. Assignment by level strategy

3.2 Level Algorithm for Iterative Tasks

The level strategy for iterative task assignment can be formalized as follows:

Let n be defined as the number of tasks and m the number of processors. Let the *level* of a node x in a task graph be the maximum number of nodes (including x) on any path from x to a terminal task. A terminal task is at level 1. Let a task be *ready* when all its predecessors have been executed.

Assign tasks to processors in the following manner:

last-assigned-processor = m
while levels remain
    calculate number-ready
    for i in 1..number-ready
        assign task i to Processor $((last - assigned - processor + 1) \ mod \ m)$
        last-assigned-processor = $((last - assigned - processor + 1) \ mod \ m)$
    end for
end while

Since the last-assigned-processor is used to determine the next-processor for assignment, the amount of tasks assigned to each processor is roughly equivalent. If the queued-message paradigm is used, this allows predecessor tasks to iterate in 'vacant' time slots.

If the system meets the restrictions outlined in Chapter 2, and if all tasks are of the same length, then the level strategy of task assignment is used to generate an optimal schedule (in terms of latency) in polynomial time. If, however, variable execution times are allowed in the task system, a search process is necessary to produce an optimal schedule. (Proofs of these assertions are given in Chapter 4 (Theorems 1, 3, and 6.))

3.3 Search Process

The problem of mapping tasks to processors is essentially a search problem: How can n tasks be mapped to m processors, where $n \gg m$, in such a way that an optimal

schedule is found?

3.3.1 Exhaustive Search For very small problems, an exhaustive search may be appropriate. Using exhaustive search, all possible schedules are generated, and the smallest schedule is chosen.

For example, Figure 3.2 illustrates that a task system with 3 tasks to be assigned to 2 processors. An exhaustive search of this system would generate 12 unique schedules, some of which are shown in the figure.

Figure 3.3 shows a partial search space generated by an exhaustive search of this task system. For example, if Task 1 is assigned to Processor 1 at time 0, then either

- Task 2 may be assigned to Processor 1 at time 3, or
- Task 2 may be assigned to Processor 2 at time 0, or
- Task 3 may be assigned to Processor 1 at time 3, or
- Task 3 may be assigned to Processor 2 at time 3.

If Task 2 is assigned to Processor 2 at time 0, then there are two choices for the assignment for Task 3, (Processor 1 at time 3, or Processor 2 at time 3). In an exhaustive search, even redundant schedules are generated. (For example, the schedule shown below is generated in the (1,1,0) case and the (2,2,0) case.)

P1	T1	T1	T1		
P2	T2			T3	T3

3.3.2 Informed Search If an exhaustive search is used, the scheduling problem has complexity on the order of $n!, [O(n!)]$ (10, 15). Table 3.1 shows the amount of time it would take to generate an optimal schedule if

- Exhaustive search is used; and

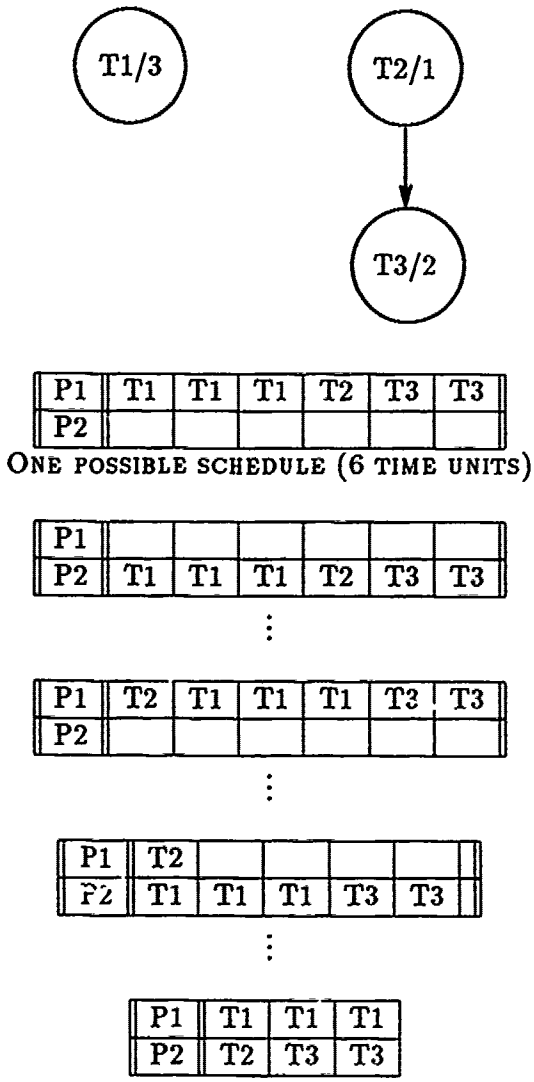
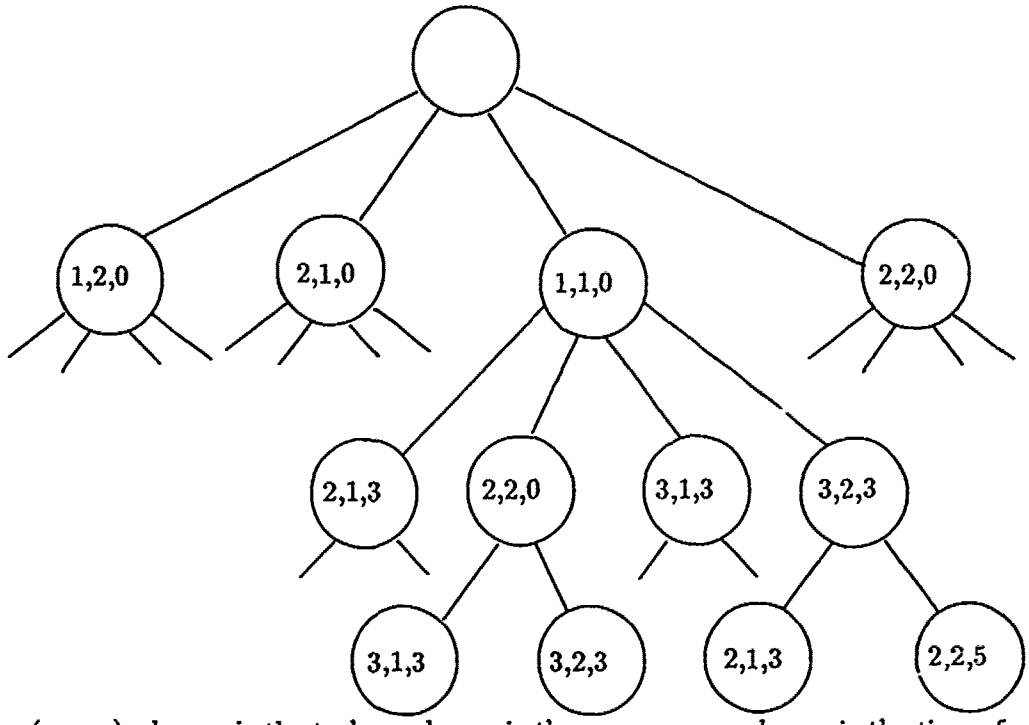


Figure 3.2. Exhaustive Search



(x, y, z) where x is the task number, y is the processor number, z is the time of assignment.

Figure 3.3. Partial Search Tree

- 1,000,000 schedules are generated every second.

n	Time to Schedule all $n!$ Combinations
5	.00012 seconds
10	3.6288 seconds
15	1307674 seconds
20	77,146 *years*

Table 3.1. Time for exhaustive search on n tasks

Informed searches use *a priori* information about a problem to generate a solution without exploring every possibility. These branch and bound techniques place a bound, or limit, on the branches of the search tree which are traversed for a solution. Often this

is done by assigning costs to each node in the search tree. During the search process, the additive costs are computed, and lowest-cost paths are traversed before high-cost paths, in an effort to minimize the solution cost. For example, suppose 4 independent tasks are to be scheduled on 2 processors. These tasks may execute in any order, and there are no restrictions which limit the number of tasks assigned to any processor.

Tasks 1, 3, 4 take 2 time units to execute.

Task 2 takes 1 time unit to execute.

Figure 3.4 shows a partial search tree for this problem. At some point in the search, the following schedule is generated (shown on the left branch of Figure 3.4):

P1	T4	T4	T3	T3	T2
P2	T1	T1			

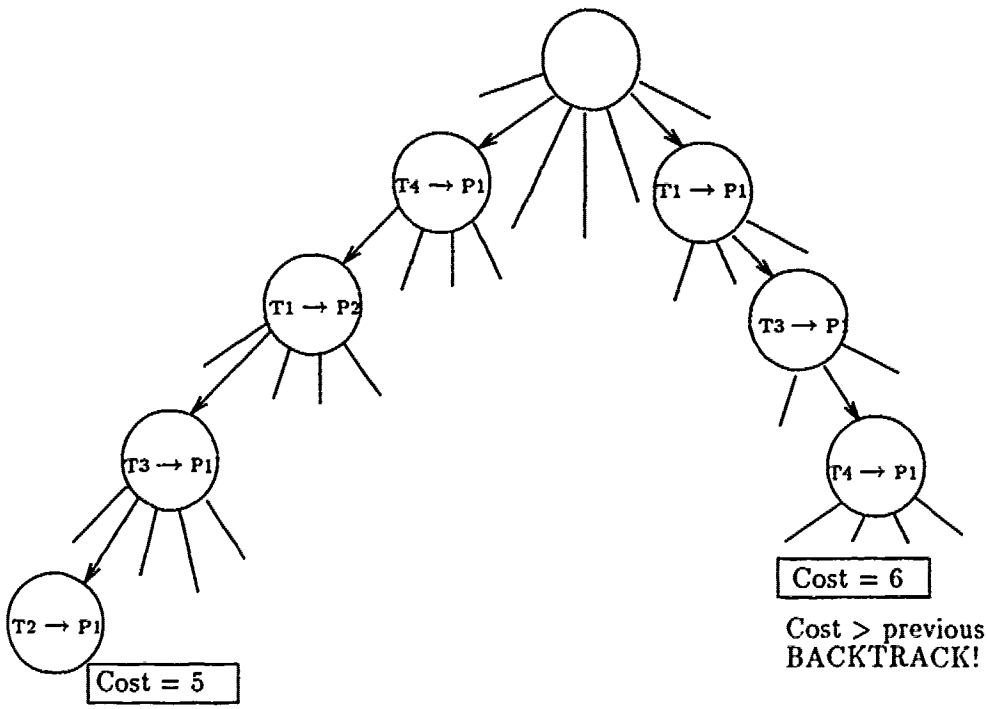


Figure 3.4. Branch and Bound Search Tree

Since this schedule can be completed in 5 time units, all partial schedules of more than 5 time units can be abandoned. For example, the partial schedule shown on the right branch of the search tree exceeds the current minimum complete schedule; thus, this schedule and its variations can be eliminated from consideration:

P1	T3	T3	T1	T1	T4	T4
P2						

Using techniques which allow the problem to be bounded by considering only those options which fall beneath the current lowest-cost schedule, it is possible to generate an optimal schedule without an exhaustive search of all alternatives.

3.4 Reducing the Search Space

In order to generate an optimal schedule without an exhaustive search of all possible schedules, several tactics can be employed to drive toward an optimal schedule more quickly. Lower-bound calculations allow the search process to terminate if a lower-bound schedule is generated. Heuristic choices are embedded into the algorithm to reduce the search space by prioritizing branches which are searched (29).

3.4.1 Lower Bound Metrics Informed search methods work to find an optimal solution without an exhaustive search of all possible solutions. Tests for optimality help the searching process determine when an optimal schedule is found. An example which illustrates the necessity for detecting an optimal schedule is shown for the one-pass system in Figure 3.5:

Assume that the 9 tasks shown in Figure 3.5 are to be scheduled onto 8 processors. How many ways are there to derive an optimal schedule for these 9 tasks?

There are 8 ways to schedule task 1 in timeslot 1.

There are 8 ways to schedule task 2 in timeslot 2.

There are 8 ways to schedule task 3 in timeslot 3.

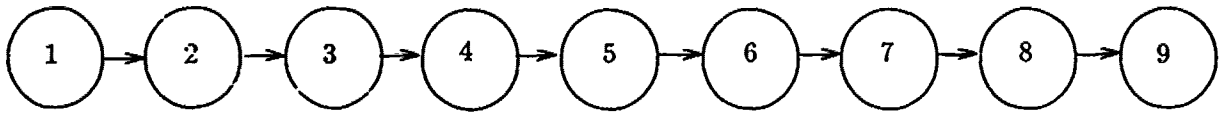


Figure 3.5. Chain of 9 Tasks

⋮

There are 8 ways to schedule task 9 in timeslot 9.

Since all these combinations are possible, there are $8^9 = 134,217,728$ optimal schedules.

Therefore, even the BEST search would have to generate 134,217,728 schedules.

If the problem is further complicated by adding one independent task to the system shown in Figure 3.5, the number of optimal schedules becomes even greater: For each of the 8^9 optimal schedules from above, there are $(8 \times 9) - 9 = 63$ vacant time slots. This means that there are $63 \times 8^9 = 8,455,716,864$ optimal schedules!

Although there are more than 134 billion possible optimal schedules for assigning a chain of 9 tasks to 8 processors, it is impossible for any of these schedules to be less than 9 time units long. If the searching process were able to test for a lower bound on the length of any given schedule, then the *first* schedule of length 9 to be generated would be accepted as an optimal schedule, and the search process could stop.

3.4.1.1 Tests for Lower-bound There are several tests for lower bounds which can be used to limit the search for an optimal schedule in classical (one-pass) systems. If these tests are implemented as part of the search, it is possible to terminate the search as soon as a lower-bound solution is found. Some of these tests for optimality can be directly applied to iterative scheduling; however, most are limited to one-pass applications.

- **Lower-bound test:** A schedule can be no shorter than its critical path (longest chain of tasks) (10). For example, suppose the task system in Figure 3.6 must be

scheduled onto 3 processors:

$t1 \prec t2 \prec t3 \prec t4 \prec t5$

t6 has no constraints

t7 has no constraints

All tasks take 1 time unit.

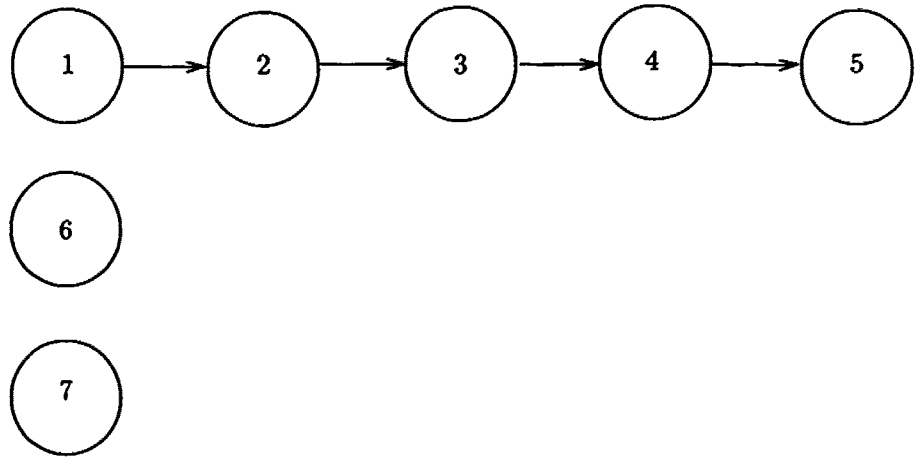


Figure 3.6. Longest Path = 5

In this case, the shortest amount of time needed to complete all tasks is 5 time units: The chain of precedence-constrained tasks dominates other considerations; even though there are idle processors, the following is an optimal schedule:

P1	T1	T2	T3	T4	T5
P2	T6				
P3	T7				

This test does not apply to iterative task systems. Using the Level Algorithm, the tasks shown in Figure 3.6 can be scheduled so that a complete iteration is output on average every 3 time units:

P1	$T1$	$T1'$	$T1''$	$T4$	$T4'$	$T4''$	$T7$	$T7'$	$T7''$
P2		$T2$	$T2'$	$T2''$	$T5$	$T5'$	$T5''$		
P3			$T3$	$T3'$	$T3''$	$T6$	$T6'$	$T6''$	

$\frac{9 \text{ time units}}{3 \text{ iterations}} = 3 \frac{\text{time units}}{\text{iteration}}$

- **Lower-bound test:** If all tasks have been scheduled, and there is no idle time in the schedule (or if the amount of idle time is less than

  smallest task execution time x *number of processors*)

  then the schedule is optimal (10).

  Example:

P1	T1	T2	T4
P2	T6		T3
P3		T5	T7

smallest task execution time = 1 *time unit*

number of processors = 3

1 x 3 = 3; *idle time* = 2 *time units*, thus the schedule is optimal.

Lower-bound test: If all tasks in a multiple-iteration system have been scheduled, and there is no idle time in the schedule (or if the amount of idle time is less than

smallest task execution time x *number of processors*

then the schedule is optimal.

- **Lower-bound test:** A schedule can be no shorter than the sum of task execution times divided by the number of processors (10).

  An example of this is shown for 4 independent tasks:

  t1: 2 time units

  t2: 2 time units

  t3: 1 time unit

  t4: 1 time unit

  No precedence constraints

  These tasks are scheduled on 2 processors.

  The shortest schedule requires

$$\frac{\sum \textit{execution times}}{\textit{number of processors}} = \frac{2+\ 2\ +\ 1+\ 1}{2} = \ 3\ \textit{time units}$$

  An optimal schedule for this task system is found by the search graph in Figure 3.7.

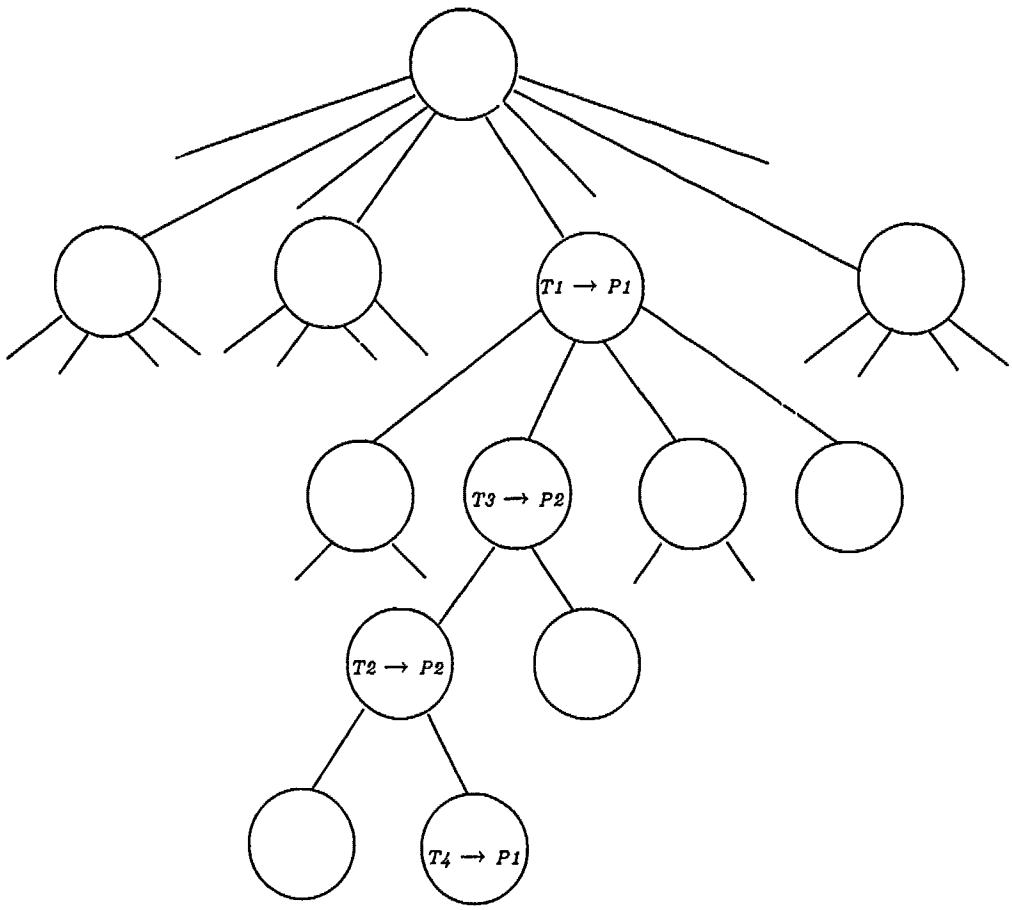
  (This test also works for multiple-iteration systems.)

- **Lower-bound test:** If every 'ready' task is assigned to a processor at the time it becomes ready, then the schedule is optimal.

  This test does not apply to iterative scheduling. If subsequent iterations are counted as 'ready' tasks, then the number of tasks must be less than or equal to the number of processors for this test to pass.

- **Lower-bound test:** If (at any point in a schedule) all ready tasks have been scheduled as soon as available, then the minimum schedule can be no less than:

$$\sum \textit{of current partial schedule} + \left\lceil \frac{\sum \textit{remaining execution times}}{\textit{number of processors}} \right\rceil$$



P1	T1	T1	T4
P2	T3	T2	T2

Figure 3.7. Search Graph and Optimal Schedule for Independent Task Example

This test does not apply to iterative scheduling. If subsequent iterations are counted as 'ready' tasks, then the number of tasks must be less than or equal to the number of processors for this test to pass.

- **Lower-bound test:** The minimum schedule can be no less than

$$\sum \left\lceil \frac{all\ tasks\ (execution\ times)\ with\ successors\ on\ critical\ path}{number\ of\ processors} \right\rceil$$

summed over all levels of the critical path.

This test does not apply to iterative scheduling.

Of all these tests for optimality, only the following apply to iterative scheduling:

- **Lower-bound test:** A schedule can be no shorter than the sum of task execution times divided by the number of processors (10).
- **Lower-bound test:** If all tasks have been scheduled, and there is no idle time in the schedule (or if the amount of idle time is less than

  smallest task execution time $\times$ *number of processors*),

  then the schedule is optimal.

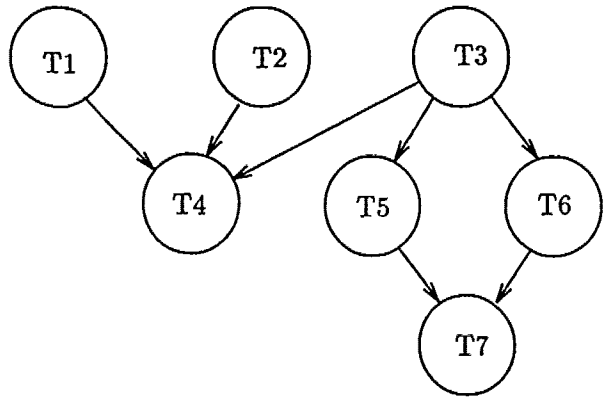
3.5 Heuristics for Schedule-Building

Infor searching, which eliminates certain branches from the search tree, can be used to shorten the amount of time to find an optimal solution. However, the search must be guided by rules which lead to good branches for selection. Heuristics are rules which can help prune the search space so that lower-cost branches are given priority for selection. The following heuristics can be used to decide which of several ready tasks should be added to the schedule under construction:

- Task with most successors.

  An algorithm for generating an optimal schedule of unit-execution-time tasks on two

processors uses the number of successors for each ready task to build schedules. Intuitively, this makes sense: by scheduling a task with many succesors, more tasks become ready in the next time step. This provides the opportunity for more processors to be in-use at that time step, indicating that processor utilization is relatively high. An example of this is shown in Figure 3.8.



P1	T2		T4	T6	T7	Cost = 5
P2	T1	T3	T5			

P1	T3	T6	T4	T7	Cost = 4
P2	T1	T2	T5		

Figure 3.8. Scheduling Tasks with Most Successors

In the task system shown in Figure 3.8, scheduling Task 3 before Task 2 yields a schedule of length 4, rather than a schedule of length 5. The reason for this improvement is that Task 3 has 3 successors, 2 of which can begin processing as soon as Task 3 completes execution.

The number-of-successors heuristic can be incorporated into an informed search in the following manner: If costs are assigned to each node in the search tree, tasks with many successors can be given a lower cost than tasks with fewer successors. This ensures that first consideration is given to tasks with many successors. Figure 3.9 shows part of the search tree which would produce the schedule shown in Figure 3.8.

- Farthest task from terminal. At time step 5 in Figure 3.8, either task 10 or 16 can be scheduled next. Since task 16 is only one step from the terminal and task 10 is two levels above the terminal, task 10 should be scheduled in time slot 5.

  This factor may also be cost-weighted to give higher-level tasks earlier scheduling preference than lower-level tasks.

3.6 Summary

This chapter has introduced the factors which must be considered in order to design a software solution to the iterative task scheduling problem:

- Execution time (equal or variable).
- Precedence of task system.
- Number of successors for each task.
- Length of each task from terminal.

Before an attempt is made to solve the specific problem, the structure of the underlying task system must be classified. If the system contains no level-traversing arcs, and if all tasks are of equal execution time, then assignments can be made based on the level strategy. If, however, the tasks have variable execution times, then an informed search which avoids redundant search paths should be used in order to obtain an optimal solution. Appendix C contains a more detailed explanation of the A* search algorithm.

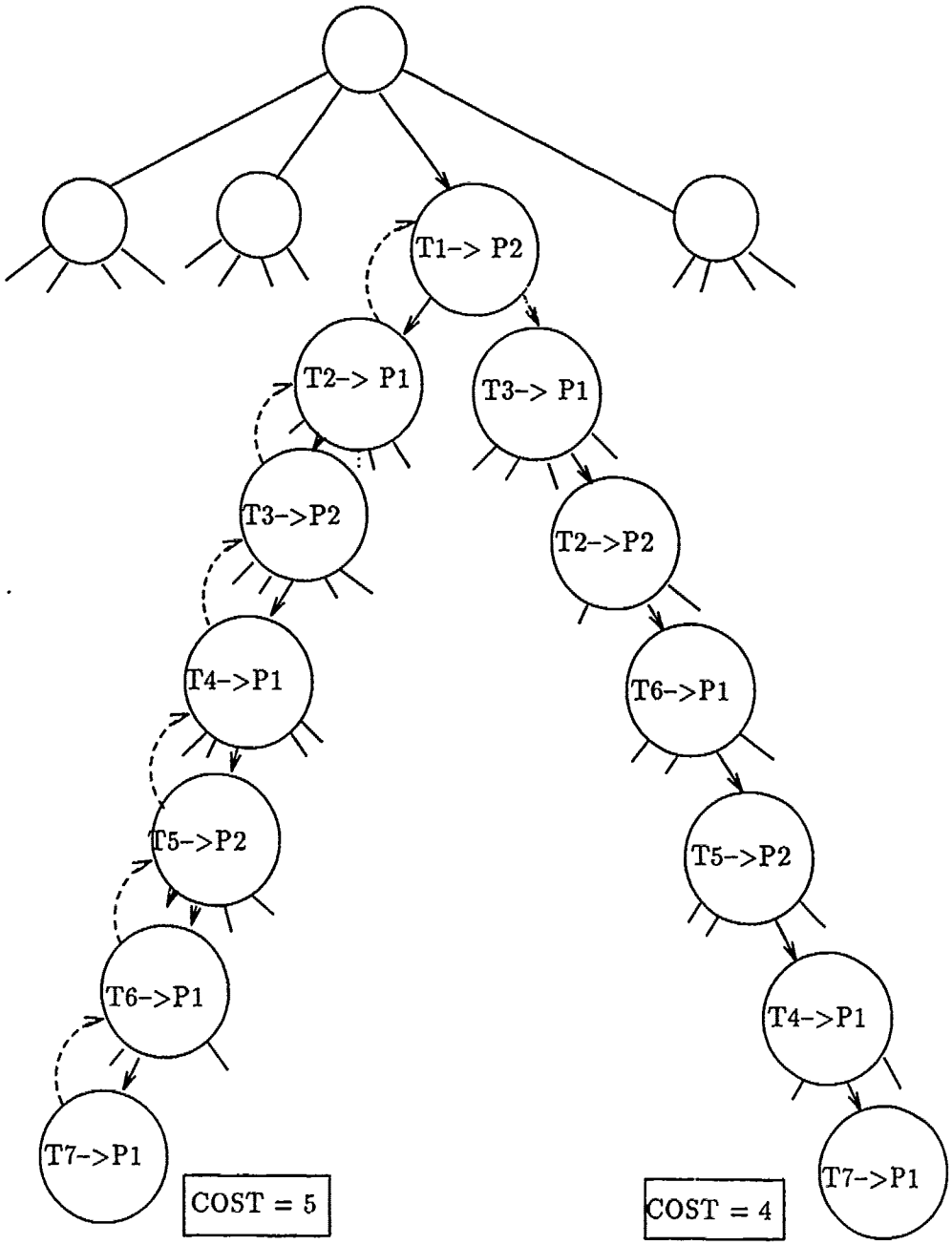


Figure 3.9. Search Tree for Tasks with Most Successors

IV. Low-level Analysis

The level strategy for assigning iterative tasks to minimize overall execution time is based on the level algorithm for one-pass task systems with a forest-structure precedence. This greedy algorithm provides an optimal latency schedule in polynomial-time when all tasks in the system have equal length. The iterative problem, like the classical scheduling problem (10), is shown in this chapter to be NP-complete when variable execution times are allowed. Since the time required to generate an exhaustive-search optimal solution may be prohibitive (7), an informed search can be used to create optimal schedules for large task systems with variable-execution times.

This chapter is divided into four sections. The first section describes the concept of **latency** as a measure of optimality for iterative task systems. In the second section, the decision strategy, which selects the task to schedule at a given time point within a processor, is developed. The third section contains results pertaining to Unit Execution Time task systems; the final section has results for Variable Execution Time systems. Variable-execution time results include proofs that the scheduling problem for iterative task systems does not produce results at the optimal latency when the level strategy is used; and that the minimal-latency problem is NP-complete.

4.1 Theoretical Design of Schedule

In many cases, a simulation iterates through hundreds of executions of each task. In such simulation, it becomes impractical to derive schedules based on a static task graph. With hundreds of nodes to consider, simply generating such a graph is a non-trivial task! For example, Figure 4.1 shows a task system of 33 tasks. A task graph which represented 10 iterations of this system would have 330 nodes (33 nodes $\times$ 10 iterations). In addition to the dependency arcs shown in the single-iteration graph, dependency arcs between successive iterations of a task would be required as well. In addition to the complexity of the task graph, These obstacles to analysis can be overcome if a new measure (latency) is used to define the effectiveness of the mapping. *The latency measures the average delay between successive iterations of the same task.* For example, a task graph and a mapping

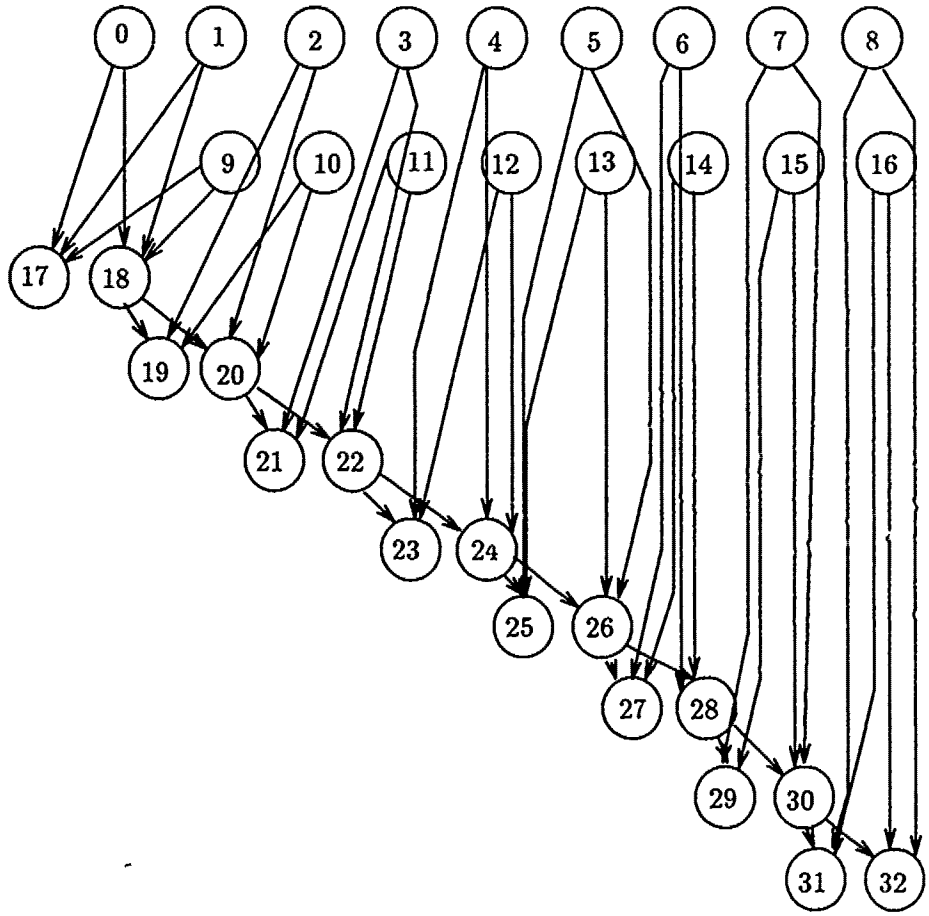
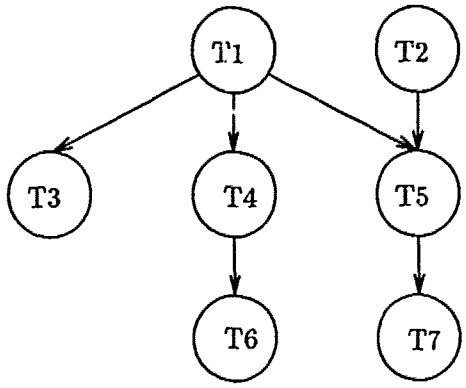


Figure 4.1. Precedence Graph: 8-bit Adder

with two iterations of the task system are shown in Figure 4.2. In this case, latency is 3. The search graph which assigns these tasks to processors is shown in Figure 4.3. Since the



P1	T1	T4	T7	$T1'$	$T4'$	$T7'$
P2	T2	T5	$T2'$	$T5'$		
P3		T3	T6	$T3'$	$T6'$	

LATENCY: $\lceil \frac{n}{m} \rceil = \lceil \frac{7}{3} \rceil = 3$

Figure 4.2. Latency

Level Strategy is a greedy method, the first assignment of tasks to processors is selected.

Although latency measures ignore the startup time when the 'pipeline' of processors is filling with tasks, it is a valid measure of performance if the number of iterations is large. For example, if a system with 10 precedence-constrained tasks is executed 1,000 times, (Figure 4.4), the latency reflects the rate at which complete iterations are produced. In the case where the system is to be mapped onto 10 processors, the startup time would require 10 time units before the first result is produced. However, the system produces a complete iteration *every time unit* once the pipeline is filled.

4.1.1 Justification for Latency Measure The scheduling problem has traditionally been associated with measurements which relate to a single execution of all tasks in the system. Two of the performance measures are to minimize schedule length and to optimize weighting factors (10). These measures are not appropriate tests for iterative task systems for the following reasons:

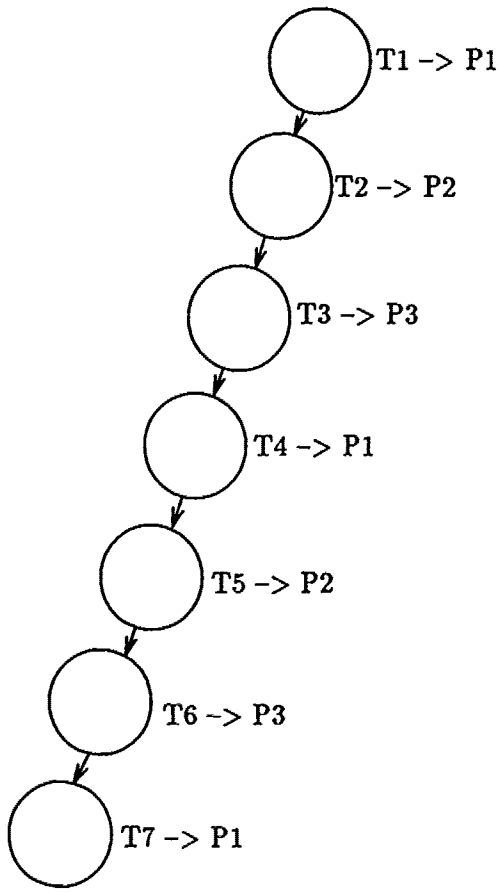


Figure 4.3. Search Tree (Level-Strategy Assignment)

P1	$T1$	$T1'$	$T1''$	$T1'''$	$T1''''$	$T1'''''$	$T1''''''$	...				
P2		$T2$	$T2'$	$T2''$	$T2'''$	$T2''''$	...					
P3			$T3$	$T3'$	...							
P4				$T4$	...							
P5					$T5$	...						
P6						$T6$	...					
P7							$T7$	...				
P8								$T8$	...			
P9									$T9$	...		
P10										$T10$	$T10'$	$T10''$

Figure 4.4. Mapping for 10 tasks and 10 processors; (latency = 1)

1. Minimize schedule length

   An optimal mapping for a single-execution system does not necessarily produce an optimal schedule for an iterative system. For example, an optimal one-pass schedule for the task system in Figure 4.4 maps all 10 tasks to one processor. This allows the first iteration to complete at the same time as the iterative mapping, but subsequent iterations would be output every 10 time units, rather than every time unit, as in the iterative mapping.

2. Optimize weighting factors

   In general, weighting factors (such as tardiness) are associated with individual tasks, and algorithms to optimize weighting factors rely heavily on the values associated with each individual task. For example, if the goal is to *minimize tardiness*, tasks with higher penalties for tardiness are scheduled before tasks with lower penalties. Since the goal of the iterative-task scheduling problem is to minimize execution time, tardiness (and other weighting factor) considerations for individual tasks do not apply.

Scheduling techniques for periodic tasks (32) also fail to map iterative simulation tasks in an optimal manner. In periodic scheduling, each task is associated with an execution time and a maximum period. For example, a 10-millisecond (execution time) task may be required to execute every 200 milliseconds (period). Since there is no periodic requirement associated with tasks in iterative simulation systems, periodic scheduling considerations do not apply.

4.2 Restrictions

In order to show that a minimal latency can be achieved, the task system must be constrained with the following assumptions:

- homogeneous processors
- unit-execution-time tasks

- queued message passing (no bounds assumed on buffer size)
- task exists on only one processor

  The parallel VHDL simulation has been constructed so that all iterations of a task execute on the same processor as the initial iteration of that task (30). Tasks are loaded onto computing nodes at the start of the simulation and remain on those nodes until simulation is complete, that is, no dynamic load-balancing. Since VHDL simulations are the primary application of this effort, scheduling considerations reflect the task structure of the existing VHDL simulations.

- no feedback arcs

  The level algorithm developed as a product of this research is based on the assumption that all task systems have no cycles. This limits the simulations to 'forward flow' applications which do not have feedback loops between tasks. Applications of simulations without cycles include various **gaming simulations, logic circuit simulations,** and **assembly-line simulations** (11, 22, 30).

Counterexamples to the optimal nature of the scheduling algorithm can be found if any of these assumptions are violated. For example, the task system in Figure 4.5 contains tasks 1 and 3 with execution time of 1 and tasks 2, 4, and 5 which require 2 units of execution time. For this system, the level strategy would produce the assignment shown in Figure 4.5.

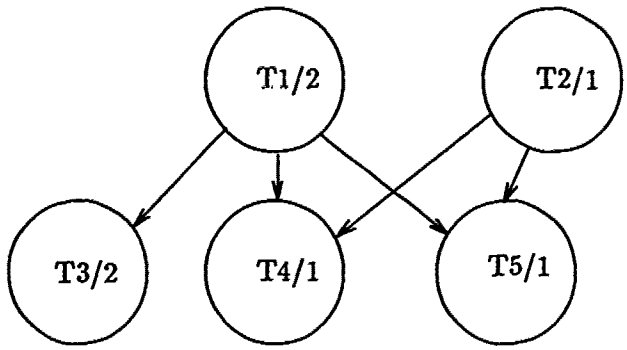
However, exhaustive search generates an optimal schedule with latency of 4:

P1	$T1$	$T1$	$T3$	$T3$		...
P2	$T2$		$T4$	$T5$		

4.3 Level Strategy

In order to prove that a minimal schedule can be achieved, task assignments are made using the LEVEL ALGORITHM:

> Let n be defined as the number of tasks and m the number of processors.
> Let the *level* of a node x in a task graph be the maximum number of nodes



P1	T1	T1	T3	T3	T5
P2	T2		T4		

Figure 4.5. Task System with Non-optimal Mapping

(including x) on any path from x to a terminal task. In a forest, there is exactly one such path. A terminal task is at level 1. Let a task be *ready* when all its predecessors have been executed.

Figure 4.6 shows the general idea between assigning levels to tasks: beginning with terminal tasks, each task is given a level. Tasks at the highest level represent the start of the longest chain of dependent tasks.

assign each task (first iteration only) to a processor in the following manner:

last-assigned-processor = m
while levels remain
    calculate number-ready
    for i in 1..number-ready
        assign task i to processor $((last - assigned - processor + 1) \bmod m)$
        last-assigned-processor = $((last - assigned - processor + 1) \bmod m)$
    end for
end while

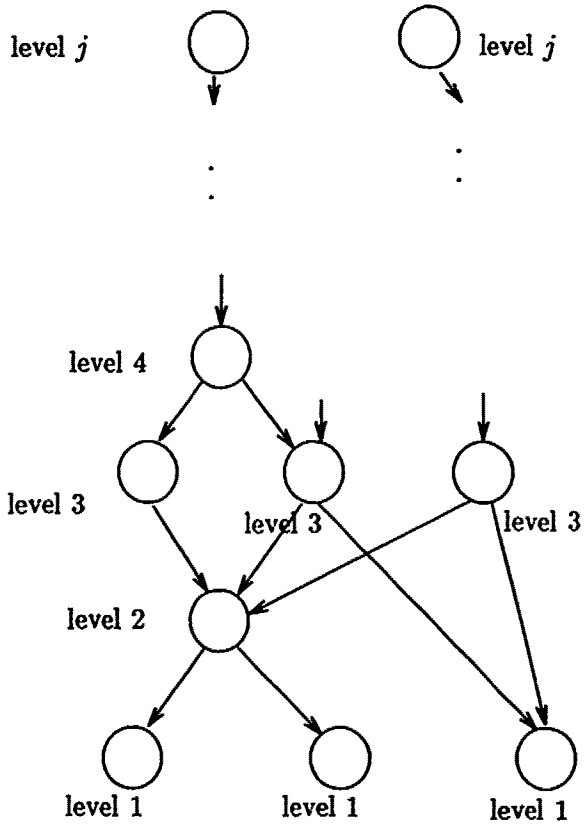


Figure 4.6. Assigning Levels to Tasks

When all of the ready tasks are assigned to processors, the precedecence matrix is restructured. Assigned tasks are deleted from the matrix; this enables the immediate successors of the already-scheduled tasks to enter a "ready" state. The tasks which are ready after the first 'level' tasks have been scheduled are now available for scheduling. The level strategy assigns these tasks, beginning with the next processor in rotation (i.e. If P3 was given the last task at level 1, then P4 gets the first task from level 2).

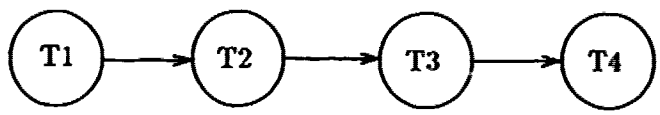
This process (scheduling ready tasks and then modifying the precedence graph) is continued until the last task (T_n) is scheduled on processor k, where k is the remainder of n/m. Once each task is assigned to a processor, all iterations of that task execute on that processor (by constraint). A detailed example of the level assignment strategy is given in Appendix B.

4.4 Scheduling Within a Processor

Although the level strategy provides an assignment of tasks to processors, the algorithm does not provide a schedule for all iterations of the task. Scheduling within each processor obeys the following constraints:

- If only one task (or iteration of a task) is ready at a given time, that task is executed.
- If a processor contains a choice of tasks to execute, the choice is selected based on a decision strategy.

4.4.1 Decision Strategy To ensure that there is minimal idle time on the most-heavily-loaded processor, an appropriate decision strategy must be employed. Otherwise, if the "wrong" task is chosen in a conflict between tasks, idle time may result, increasing latency. For example, Figure 4.7, shows the results of an arbitrary decision strategy when 4 iterations of a task system are mapped to 3 processors. In this case, Processor 1 is forced to be idle at $time_7$ because $task_1$ has completed all iterations, and all remaining iterations of $task_4$ are waiting for tasks on other processors to complete.



P1	$T1^0$	$T1^1$	$T1^2$	$T4^0$	$T4^1$	$T4^2$	$T1^3$			$T4^3$
P2		$T2^0$	$T2^1$	$T2^2$				$T2^3$		
P3			$T3^0$	$T3^1$	$T3^2$				$T3^3$	

Idle Time Due to Arbitrary Decision Strategy

P1	$T1^0$	$T1^1$	$T1^2$	$T4^0$	$T1^3$	$T4^1$	$T4^2$	$T4^3$
P2		$T2^0$	$T2^1$	$T2^2$		$T2^3$		
P3			$T3^0$	$T3^1$	$T3^2$		$T3^3$	

Optimal Schedule

Figure 4.7. Effects of Decision Strategy

Figure 4.7 shows that an arbitrary decision strategy can result in unnecessary idle time, generated when a processor waits for predecessor tasks to complete. In order to minimize this delay, a formal decision strategy must select which task to execute whenever a conflict occurs. The decision strategy should evaluate factors such as *number of predecessor iterations*, which impact delay. Some suggestions include an all-iterations-first strategy, which tends to consolidate delay time at the start of the schedule and an iteration-number strategy, which attempts to keep enough predecessor iterations buffered so that unnecessary delay is avoided.

1. SCHEDULE I: All iterations of the first task assigned to each processor will execute before any iteration of the second assigned task. In a similar manner, all of the second task's iterations must execute before any of the third assigned task, and so forth.

2. SCHEDULE II: Each task is numbered with the task number and iteration number: $task_\alpha^\beta$ is used to represent the $(\beta + 1)^{th}$ iterat of $task_\alpha$. For example, the first iteration of $task_1$ is represented as $task_1^0$; the second iteration of $task_3$ is represented as $task_3^1$; and so forth. The scheduling decision is made by comparing iteration numbers. If the difference between iterations is less than the number of processors, then the highest-level task (task with the highest iteration number) should be scheduled since it may be a predecessor of $(m - 1)$ tasks on other processors. If the difference between iterations is greater than or equal to the number of processors, then the lower-level task should be scheduled, since the higher-level task has stored up enough iterations to keep the system from generating idle time.

SCHEDULE II is conjectured to produce minimal delay time (Appendix D). SCHEDULE I, however, is the less complex scheduling strategy. Therefore, SCHEDULE I has been chosen as the decision strategy for this investigation.

4.5 Lower Bound on Latency

The minimum latency that can be achieved occurs when all tasks in the system are completely independent. In this case, n tasks assigned to m processors can complete one

iteration every $\lceil \frac{n}{m} \rceil$ time units. Figure 4.8 shows an example of this for 5 independent tasks mapped onto 3 processors.

P1	T1	$T1'$	T4	$T4'$
P2	T2	$T2'$	T5	$T5'$
P3	T3	$T3'$		

LATENCY: $\lceil \frac{n}{m} \rceil = \lceil \frac{5}{3} \rceil = 2$

Figure 4.8. 5 independent tasks; 2 iterations

Theorem 1 *(UET Lower Bound)*

The lower bound for latency on task systems with the following restrictive assumptions is $\lceil \frac{n}{m} \rceil$ *:*

- *n unit-execution-time tasks*
- *m homogeneous processors*
- *precedence relationship* $\prec$ *defined between tasks*
- *level-strategy assignment*
- *all-iterations-first decision strategy*
- *queued message passing (no bounds assumed on buffer size)*
- *task exists on only one processor*

This proof uses the simplest task system – completely independent tasks – to show that the minimum latency is $\lceil \frac{n}{m} \rceil$.

PROOF: Independent Tasks

Case 1: $n \leq m$ (by construction)

In this case, $(n \leq m)$, each task can be scheduled on a separate processor. Since all tasks

are independent, each task can begin another iteration every timestep. This results in a latency of $\lceil \frac{n}{m} \rceil = 1$.

Case 2: $n > m$ (by construction)

If there are more tasks than processors, ($n > m$), then each processor contains more than one task (by the level strategy.) Furthermore, the level strategy attempts to balance the amount of tasks assigned to each processor, with no more than $\lceil \frac{n}{m} \rceil$ tasks assigned to any processor and with $\lceil \frac{n}{m} \rceil$ tasks to at least one processor, Processor k. By the all-iterations-first decision strategy, the first task on Processor k executes all iterations before any other task executes any itertions. Thus the latency becomes

$$\frac{number\ of\ tasks\ on\ one\ processor \times number\ of\ iterations}{number\ of\ output\ values}$$

$$= \frac{\lceil \frac{n}{m} \rceil \times i}{number\ of\ iterations} = \left\lceil \frac{n}{m} \right\rceil$$

Therefore, the average time between iterations, over the life of the schedule, for at least one processor is $\lceil \frac{n}{m} \rceil$. Since the processor with the heaviest load dominates execution time, $\lceil \frac{n}{m} \rceil$ is the minimal latency achieved by the system.

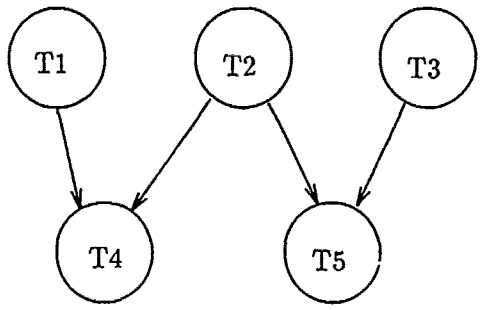
□

4.6 Upper Bound on Latency

The derivation of an upper bound for iterative task systems is not intuitively obvious. In order to prove that $\lceil \frac{n}{m} \rceil$ is an upper bound, as well as a lower bound on latency for UET iterative task systems, it is necessary to categorize the impact of delay time, time periods in which a processor has not completed all tasks but must remain idle because no tasks can execute. It is also necessary to prove that (in a balanced mapping), an optimal latency is achieved when at least one processor has no idle time in a time frame equivalent to the pipelined portion of the schedule. This is done in lemmas which bound the amount

of time a processor may be delayed while predecessor tasks execute.

An example is given in Figure 4.9 which shows the maximum latency achieved when a UET task system is mapped to 2 Processors using the level strategy.



P1	T1	T1′	T3	T3′	T5	T5′
P2		T2	T2′	T4	T4′	

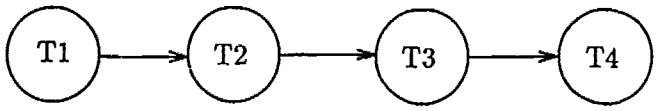
$$\text{latency} = \frac{6 \text{ time units}}{2 \text{ iterations}} = 3 = \left\lceil \frac{5}{2} \right\rceil = \left\lceil \frac{n}{m} \right\rceil$$

Figure 4.9. Maximum Latency for Task System

In this example, the largest latency is determined by the processor which contains the most tasks, P_1. The number of time units required by the most-heavily-loaded processor is equal to the number of tasks on that processor multiplied by the number of iterations required by the task system. For example, in Figure 4.9, Processor 1 contains $\left\lceil \frac{n}{m} \right\rceil = \left\lceil \frac{5}{2} \right\rceil = 3$ tasks. Since the entire system must perform 2 executions, this value is multiplied by 2, indicating that the most-heavily-used processor is active for 6 time units during the schedule.

The concept of *delay t*[illegible] closely associated with the concept of latency. The next section discusses the maximum delay time which can occur when tasks are assigned by the level strategy and are scheduled with the all-iterations-first strategy.

4.6.1 Processor Delay Time If the level algorithm were **not** used to assign tasks to processors, the following could occur with SCHEDULE 1:



P1	$T1^0$	$T1^1$	$T1^2$	$T1^3$	$T2^0$	$T2^1$	$T2^2$	$T2^3$					
P2						$T3^0$	$T3^1$	$T3^2$	$T3^3$	$T4^0$	$T4^1$	$T4^2$	$T4^3$

Arbitrary Assignment (5 units of delay time)

P1	$T1^0$	$T1^1$	$T1^2$	$T1^3$	$T3^0$	$T3^1$	$T3^2$	$T3^3$	
P2		$T2^0$	$T2^1$	$T2^2$	$T2^3$	$T4^0$	$T4^1$	$T4^2$	$T4^3$

Level-Strategy Assignment (1 unit of delay time)

Figure 4.10. Comparison: Assignment Strategies with Schedule 1

Intuitively, the level strategy assignment seems to produce 'better' schedules than an arbitrary assignment, as demonstrated in Figure 4.10. The next question to arise involves the amount of delay time that can be generated when the level strategy with SCHEDULE I is used for assignment.

4.6.2 Chain of Tasks If the task system is a chain of related tasks

$$(T_1 \prec T_2,\ T_2 \prec T_{3,},\ \prec \ \ldots \ \prec T_n),$$

such that the number of tasks is equal to the number of processors, then the level strategy will generate at most $m - 1$ units of delay time on any given processor, as shown in Figure 4.11. In fact, each $processor_k$ will have exactly $k - 1$ units of delay time during the entire simulation.

Suppose that there are $m + 1$ tasks to be scheduled on m processors. For example, Figure 4.12 shows 3 iterations of 4 tasks to be mapped onto 3 processors. In this example, P_1 has no delay time; P_2 has 1 unit of delay time and P_3 has 2 units of delay time. This implies that the *most-heavily-loaded processor* will have the least idle time throughout the

P1	$T1$	$T1'$	$T1''$			
P2		$T2$	$T2'$	$T2''$		
P3			$T3$	$T3'$	$T3''$	
P4				$T4$	$T4'$	$T4''$

Figure 4.11. Level Strategy Assignment: chain of m tasks on m processors

simulation.

P1	$T1$	$T1'$	$T1''$	$T4$	$T4'$	$T4''$
P2		$T2$	$T2'$	$T2''$		
P3			$T3$	$T3'$	$T3''$	

Figure 4.12. Level Strategy Assignment: chain of $m + 1$ tasks on m processors

Lemma 1 *(Delay Time: chain)*
For a system with the following constraints, the maximum amount of delay time (on any processor) is equal to $m - 1$ *time units:*

- *chain of* n *unit-execution-time tasks*
- m *homogeneous processors*
- *precedence relationship* $\prec$ *defined between tasks*
- *level-strategy assignment*
- *all-iterations-first decision strategy*
- *number of iterations,* i*, is greater than the number of processors*
- *queued message passing (no bounds assumed on buffer size)*
- *task exists on only one processor*

PROOF: (by construction)

BASE STEP: Show that P_1 has no more than $m-1$ units of delay time.

DELAY TIME AT THE START OF SCHEDULE: Since tasks are assigned by the level strategy, each processor begins execution 1 time unit after the previous processor: 1 begins execution of the task at level n at $time_0$; 2 begins executing the task at level $n-1$ at $time_1$; ... Since the all-iterations-first decision strategy is used, P_1 is busy in the first i time units; 2 executes all iterations of its first task between $time_1$ and $time_{i+1}$; 3 is busy between $time_2$ and $time_{i+2}$; ... m receives its first task at $time_{m-1}$, and thus has a delay time of $m-1$ time units at the start of the schedule. This pattern is shown in Figure 4.13.

P1	$T1$	$T1'$	$T1''$	...	T_{m+1}	...		T_{2m+1}	...
P2		$T2$	$T2'$	$T2''$	...	T_{m+2}	...		
⋮									
P_m			T_m	T'_m	T''_m	...	T_{2m}	...	

$time_m$ *to* $time_{i+m}$: i *time units*

Figure 4.13. Level Strategy Assignment: Serial Precedence (chain)

DELAY TIME DURING THE SCHEDULE: At time i, P_1 has completed all executions of T_1. The second task assigned to P_1 is $task_{m+1}$, which has m predecessors, each of which are assigned to different processors. Since $i > m$, and since m predecessor tasks have each performed their first execution by $time_m$, $task_{m+1}$ can begin execution at or after $time_m$. Since $time_i > time_m$, $task_{m+1}$ begins execution at $time_i$, generating no delay time.

At time $2i$, P_1 has completed all executions of T_{m+1}. The third task assigned to P_1 is $task_{2m+1}$, which has $2m$ predecessors. Since $2i > 2m$, and since the $2m$ predecessor tasks have each performed their first execution by $time_{2m}$, $task_{2m+1}$ can begin execution at or after $time_{2m}$. Since $time_{2i} > time_{2m}$, $task_{2m+1}$ begins execution at $time_{2i}$, generating no delay time.

At time ji, ($j <$ *last task assigned to* P_1,) P_1 has completed all executions of T_{jm+1}. The next task assigned to P_1 is $task_{(j+1)m+1}$, which has $(j+1) \times m$ predecessors.

Since $(j+1) \times i > (j+1) \times m$, and since the $(j+1) \times m$ predecessor tasks have each performed their first execution by $time_{(j+1)m}$, $task_{(j+1)m+1}$ can begin execution at or after $time_{(j+1)m}$. Since $time_{(j+1)i} > time_{(j+1)m}$, $task_{(j+1)m+1}$ begins execution at $time_{(j+1)i}$, generating no delay time.

DELAY TIME AT END OF SCHEDULE: Once P_1 executes the last iteration of its last task, no further delay time can be added to P_1. This is indicated in Figure 4.14.

P1	...	T_{n-m}	...	T^i_{n-m}		
P2		...	$T_{n-(m-1)}$	...	$T^i_{n-(m-1)}$	
⋮						
P_l		...	T_n	T'_n	...	T^i_n
⋮						

Figure 4.14. Last task execution on all processors

Induction Step:

If P_n has no more than $n-1$ units of delay time, then P_{n+1} has no more than n units of delay time.

P_n can accumulate delay time at the start of the schedule and during the schedule. In a chain of k tasks, each processor (other than P_1) to recieve a task is delayed from beginning execution by 1 time unit, since all tasks are unit-execution-time. Therefore, P_n is delayed by $n-1$ time units; P_n can thus begin its first task at $time_{n-1}$.

P_n delays P_{n+1} from beginning execution by 1 additional time unit. (Since the predecessor to T_{n+1} is scheduled on P_n.) Therefore, P_{n+1} can thus begin its first task at $time_n$.

By the all-iterations-first strategy, P_n executes its first task, T_n, from $time_{n-1}$ to $time_{n-1+i}$.

P_{n+1} executes its first task, T_{n+1}, from $time_n$ to $time_{n+i}$.

Between the first iteration of T_{n+1} on P_{n+1} and the first iteration of the second task on P_{n+1}, T_{n+1+m}, there can be at most one additional delay for each of the $m-1$ other

processors which contain tasks. Therefore, the longest delay between the first iteration of T_{n+1} and the first iteration of T_{n+1+m} is $m-1$ time units. But, since P_{n+1} performs all iterations of T_{n+1}, P_{n+1} is not idle before $time_{n+1-1+i}$. Since $i > m, i > m-1$ Therefore the i time units between the first execution of T_{n+1} and the first execution of T_{n+1+m} are completely filled with iterations of T_{n+1}, and no delay time is generated.

Thus P_{n+1} has n units of delay time at the start of the schedule and no delay time internal to the schedule. Therefore P_{n+1} has n units of delay time.

□

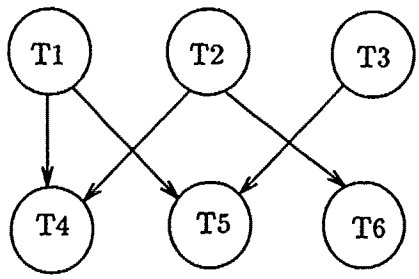
At this point, a question may arise: *What happens if the system is not a simple chain of tasks?*

4.6.3 Arbitrary Precedence Intuitively, it appears that a chain of tasks provides a more restricted case than an arbitrary precedence graph – In a chain, only one new task becomes ready in each time slot; arbitrary precedence can have many tasks become ready at the same time. This seems to imply that an arbitrary system can have less delay time before all processors are busy, as demonstrated in Figure 4.15: As each processor receives a task by the level strategy assignment, an idle time slot is created whenever the level changes. For example, Figure 4.16 shows a precedence graph and the level strategy assignment for the first task on each processor. In this example, P_4 delayed until $task_4$ is ready to execute. Each of these idle time slots is called a *stagger point.* Before P_k begins processing, the greatest number of stagger points which can occur is $k-1$. (Because, if each task is at a different level, $task_1$ begins processing on P_1 at $time_0$; $task_2$ begins processing on P_2 at $time_1$ ... $task_k$ begins processing on P_k at $time_{k-1}$.)

A proof is given to show that arbitrary precedence produces *no more than* the amount of delay time in a task system with serial precedence:

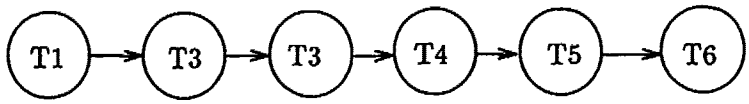
Lemma 2 *(Delay Time: arbitrary precedence)*

In a system with the following constraints, P_{k+1} can be delayed by at most 1 time unit by P_k:



(a) Arbitrary Precedence

P1	$T1$	$T1'$	$T1''$	$T4$	$T4'$	$T4''$
P2	$T2$	$T2'$	$T2''$	$T5$	$T5'$	$T5''$
P3	$T3$	$T3'$	$T3''$	$T6$	$T6'$	$T6''$



(b) Serial Precedence

P1	$T1$	$T1'$	$T1''$	$T4$	$T4'$	$T4''$		
P2		$T2$	$T2'$	$T2''$	$T5$	$T5'$	$T5''$	
P3			$T3$	$T3'$	$T3''$	$T6$	$T6'$	$T6''$

Figure 4.15. Level Strategy Assignment: arbitrary precedence *vs.* serial precedence

- *n unit-execution-time tasks*
- *m homogeneous processors*
- *precedence relationship $\prec$ defined between tasks*
- *level-strategy assignment*
- *all-iterations-first decision strategy*
- *number of iterations, i, is greater than the number of processors*
- *queued message passing (no bounds assumed on buffer size)*

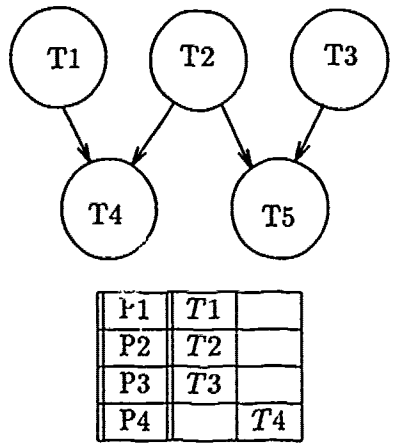


Figure 4.16. Idle time due to level change

- *task exists on only one processor*

PROOF: (by contradiction)

Assume that P_{k+1} is delayed more than 1 time unit by P_k. Since delays are only caused by change in level between tasks, the level between tasks assigned to P_k and P_{k+1} must change at least twice.

- Let $time_u$ denote the time when P_{k+1} must first be idle, waiting for the first iteration of a predecessor task on P_k.
- Let T_i denote the first predecessor task assigned to P_k, and T_h denote the next task assigned to P_k by the level strategy.
- Let T_j denote the first succcessor task assigned to P_{k+1}, and T_l denote the next task assigned to P_{k+1} by the level strategy.

At $time_u$, P_k begins its first iteration of T_i.

At $time_{u+1}$ P_{k+1} begins its first iteration of T_j (since all tasks take 1 time unit). This creates 1 time unit of delay.

At $time_{u+i}$, P_k completes its final iteration of T_i.

P_{k+1} completes its last iteration of T_j at $time_{u+1+i}$ (i time units later).

At $time_{u+i}$, P_k begins the first iteration of T_h, the next task assigned.

If T_h is a not predecessor to T_l, then T_h cannot impact the schedule on P_{k+1}.

If T_h is a predecessor to T_l, then T_l cannot begin execution until $time_{u+i+1}$. But, P_{k+1} completes its last iteration of T_j at $time_{u+1+i}$. Thus there is no delay incurred between the last iteration of T_j and the first iteration of T_l.

P_{k+1} is delayed 1 time unit by P_k.

Contradiction.

□

Theorem 2 *(Maximum Delay Time)*
If the following constraints are observed, the maximum amount of delay time on any processor is $m - 1$ *time units*

- *n unit-execution-time tasks*
- *m homogeneous processors*
- *precedence relationship* $\prec$ *defined between tasks*

- *level-strategy assignment*
- *all-iterations-first decision strategy*
- *number of iterations, i, is greater than the number of processors*
- *queued message passing (no bounds assumed on buffer size)*
- *task exists on only one processor*

PROOF:

By Lemma 2, P_{k+1} can be delayed by at most 1 time unit by P_k. Since there are $m-1$ processors which can delay any other processor, the greatest possible delay is $\frac{1\ time\ unit}{processor} \times\ m-1\ processors\ =\ m-1\ time\ units.$

□

Theorem 3 *(UET Upper Bound)*

Task systems which meet the following assumptions have a latency upper bound of $\lceil \frac{n}{m} \rceil$ *:*

- *m homogeneous processors*
- *n unit-execution-time tasks*
- *precedence relationship* $\prec$ *defined between tasks*
- *assignment by level algorithm*
- *all-iterations-first decision strategy*
- *queued message passing (no bounds assumed on buffer size)*
- *task exists on only one processor*

This theorem (proved by construction) uses Theorem 2 to show that the MAL is achieved if no processor has more than $m-1$ units of idle time.

PROOF:

The maximum delay time on any processor is $m-1$ time units (Theorem 2).

The maximum startup cost is $m-1$ time units.

The in-use time on a processor is equal to
number of tasks on the processor × number of iterations for each task + delay time.

For the most-heavily-loaded processor, this is equivalent to

$$\left\lceil \frac{n}{m} \right\rceil \times i + (m-1)$$

Since the startup cost must be subtracted to compute latency, the latency for the most-heavily-loaded processor is

$$\frac{\textit{in use time} - \textit{startup cost}}{\textit{number of iterations}} = \left\lceil \frac{n}{m} \right\rceil = MAL$$

□

Theorem 4 *(Firm Bound on UET Latency)*
Task systems which meet the following conditions and which are scheduled using the level algorithm have a latency of $\lceil \frac{n}{m} \rceil$ *:*

- *m homogeneous processors*
- *n unit-execution-time tasks*
- *precedence relationship* $\prec$ *defined between tasks*
- *level-strategy assignment*
- *all-iterations-first decision strategy*
- *number of iterations, i, greater than the number of processors*
- *queued message passing (no bounds assumed on buffer size)*

- *task exists on only one processor*

PROOF:

Theorem 1 showed that $\lceil \frac{n}{m} \rceil$ is a lower bound on latency. Theorem 3 showed that $\lceil \frac{n}{m} \rceil$ is an upper bound on latency. Let latency $\equiv \sigma$.

Then the following are true:

$$\sigma \leq \lceil \frac{n}{m} \rceil$$
$$\sigma \geq \lceil \frac{n}{m} \rceil$$

Therefore $\sigma = \lceil \frac{n}{m} \rceil$.

□

4.7 Equal Execution Time Task Systems

These results can be extended to task systems where all tasks have the same execution time: In UET systems, the length of each task is 1, and the sum of all task execution times $= \sum_{i=1}^{n} 1 = n$. Therefore the latency, $\lceil \frac{n}{m} \rceil$, can be written as $= \left\lceil \frac{k \times n}{m} \right\rceil$, where k is the execution time of each task ($k = 1$, in the UET case). This reasoning is expanded in the proof of Theorem 5.

Theorem 5 *(Equal Execution Time)*

If the following conditions are met:

- *n equal-execution-time tasks, each of length k*
- *m homogeneous processors*
- *precedence relationship $\prec$ defined between tasks*
- *level-strategy assignment*
- *all-iterations-first decision strategy*
- *number of iterations, i, is greater than the number of processors*

- *queued message passing (no bounds assumed on buffer size)*
- *task exists on only one processor then the MAL* $= \left\lceil \frac{k \times n}{m} \right\rceil$

PROOF:

Since all tasks are of length k, and since no preemption is allowed, the only time points which have task start- or end-times are multiples of k. These 'interesting' time points can be mapped to the positive integers by dividing by a scaling factor of k:

- *task length k becomes task of length* $\frac{k}{k} = 1$.
- *time k becomes time* 1
- *time* $2k$ *becomes time* 2

  $\vdots$
- *time* nk *becomes time* n

At this point, the tasks in the EET system are mapped to a UET system. The MAL for a UET system, (Theorem 4), is $\lceil \frac{n}{m} \rceil$.

In order to restore the converted system to the original configuration, it is necessary to multiply the MAL by k, the scaling factor. Thus the MAL becomes:

$$\frac{k \text{ } EET \text{ } time \text{ } units}{1 \text{ } UET \text{ } time \text{ } unit} \times \left\lceil \frac{n}{m} \right\rceil \text{ } UET \text{ } time \text{ } units = k \times \left\lceil \frac{n}{m} \right\rceil = \left\lceil \frac{k \times n}{m} \right\rceil$$

□

4.8 Variable Execution Times

4.8.1 Problem Description The Level Strategy generates schedules at the maximum latency for Unit Execution Time (UET) tasks. However, schedules for tasks with variable execution times produced by the Level Strategy are not guaranteed to iterate at the optimal latency, as shown in Figure 4.5. This section discusses factors which impact the following result:

If task lengths = $\{1,2\}$, the *minimum* achievable latency is no longer guaranteed to be found by the Level Strategy.

4.8.2 Reasons for Level-Strategy Failure When task execution times $\in \{1,2\}$, the Minimum Achievable Latency is not longer guaranteed to be the sum of task execution times divided by the number of processors, $\left\lceil \frac{\sum \textit{task execution times}}{m} \right\rceil$. There are several reasons that the minimum latency differs from the unit execution time case:

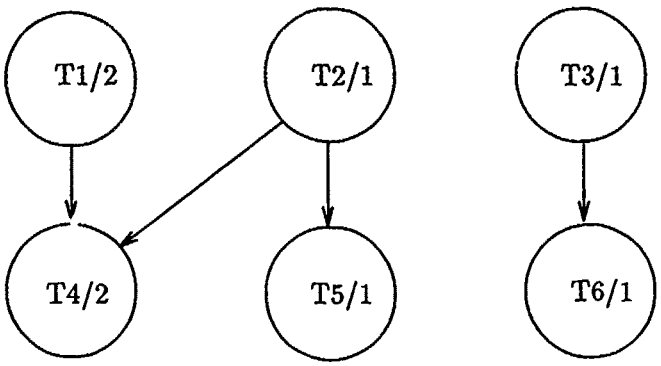
- Since tasks are assigned in one-time-unit *and* two-time-unit blocks, there is no guarantee that there will be a maximum of $\left\lceil \frac{\sum \textit{task execution times}}{m} \right\rceil$ unique tasks on each processor.
- In addition, tasks of length-2 tasks must be assigned in blocks of 2 time units. This seems to be an obvious point; however, it places a limiting factor on the available time slots for subsequent iterations of tasks which take 2 time units to execute.

In order to show that the Level Strategy fails to generate mappings at the minimal achievable latency for variable-execution-time task systems, counterexamples are used. Although it is not possible to prove the truth of a general statement by example, a single counterexample is sufficient to *disprove* a statement (19).

4.8.2.1 Maximum Tasks/Processor In order to demonstrate that the Level Strategy produces non-optimal latency mappings for variable-execution-time tasks, it is necessary to consider the method the Level Strategy uses to assign tasks: In the UET system, $\frac{\sum \textit{task execution times}}{m}$ produces a number, say x, plus a *remainder*, r. Assigning x tasks to each processor allows the r remaining tasks to be each given to a separate processor, with no more than

$$\left\lceil \frac{\sum \textit{task execution times}}{m} \right\rceil$$

tasks on any processor. If variable execution times are allowed, the Level Strategy could assign more than $\left\lceil \frac{\sum \textit{task execution times}}{m} \right\rceil$ unique task-time-slots to one processor. For example, in Figure 4.17 $\frac{\sum \textit{task execution times}}{m} = 3$; however, the tasks assigned to Processor 1 consume 4 time units on each iteration.



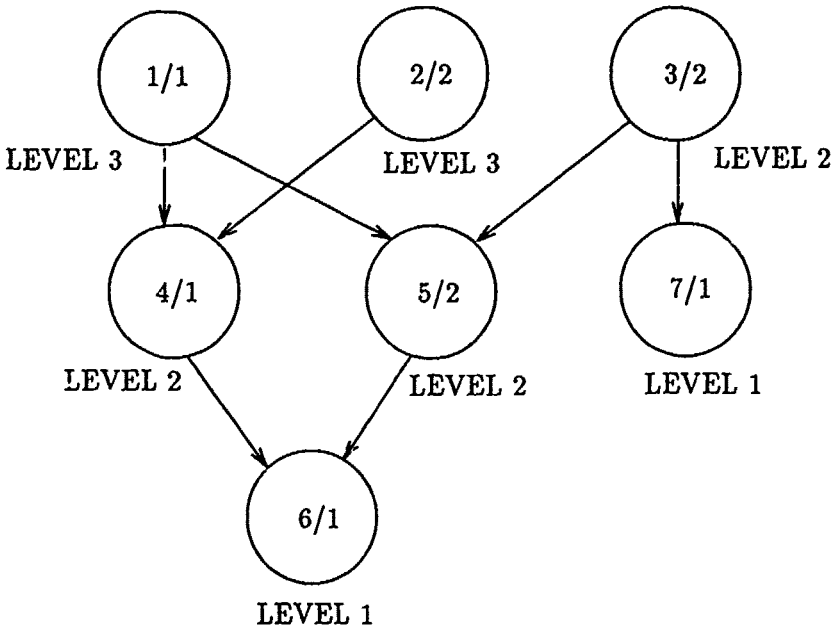
P1	T1	T1	T4	T4
P2	T2	T5		
P3	T3	T6		

Figure 4.17. Variable execution time task system

Although the level strategy assignment produces output once every 4 time units, the smallest possible latency is 3 time units. $\left\lceil \left\lceil \frac{\sum \textit{task execution times}}{m} \right\rceil = \left\lceil \frac{8}{3} \right\rceil = 3 \right\rceil$ Exhaustive search generates a schedule which produces outputs every 3 time units:

P1	$T1$	$T1$	$T1'$	$T1'$			
P2	$T2$	$T3$	$T6$	$T2'$	$T3'$	$T6'$	
P3		$T5$	$T4$	$T4$	$T5'$	$T4'$	$T4'$

4.8.2.2 Blocks of Size 2 Figure 4.18 shows a task system which contains tasks with execution time of 1 or 2 time units. Since $T2'$, for example, requires 2 time units to execute, it cannot be inserted in timeslot 1. This barrier is not present when UET tasks are scheduled; any subsequent iteration in that system can fill any hole in the schedule.



P1	$T2$	$T2$	$T4$		$T6$	$T2\prime$	$T2\prime$	$T4\prime$	
P2	$T3$	$T3$	$T7$			$T3\prime$	$T3\prime$		
P3	$T1$		$T5$	$T5$	$T1\prime$			$T5\prime$	$T5\prime$

Figure 4.18. Task System with Variable Execution Time

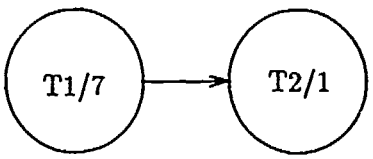
Since the execution time = $\{1,2\}$ scenario can have gaps of 1 time unit with only 2-time-unit tasks ready to iterate, the test for a completely-filled processor does not produce the same information as in the UET case.

4.8.3 Bounds on Variable-Execution-Time Latency Although the unit-execution-time problem differs from the variable-execution-time problem, it is possible to apply some of the same criteria as a measure of optimality. Theorem 6 is used to prove that a lower

bound for latency is at least as large as the following:[1]

$$max\left(\left\lceil\frac{\sum\ task\ execution\ times}{m^*}\right\rceil,\ length\ of\ the\ longest\ task\right)$$

Intuitively, this makes sense. Figure 4.19 shows a 2-task system with a MAL dominated by the task of length 7. Although $\left\lceil\frac{\sum\ task\ execution\ times}{m}\right\rceil = \left\lceil\frac{7+1}{2}\right\rceil = 4$, the best achievable latency is 7.



P1	T1	T1	T1	T1	T1	T1	T1	T1′	T1′	T1′	T1′	T1′	T1′	T1′	...
P2								T2							T2′

Figure 4.19. Variable execution time task system

Theorem 6 *(Variable-execution-time Lower Bound) If variable execution times are allowed in an iterative task system, with the following:*

- *n variable-execution-time tasks*
- *m homogeneous processors*
- *level-strategy assignment*
- *precedence relationship $\prec$ defined between tasks*
- *all-iterations-first decision strategy*
- *number of iterations, i, is greater than the number of processors*
- *queued message passing (no bounds assumed on buffer size)*

[1] where m^* denotes processors-in-use.

- *task exists on only one processor*

a lower bound on latency is the larger of the following:

- $\left\lceil \frac{\sum \textit{task execution times}}{m^*} \right\rceil$
- *length of the longest task*

This theorem is proved in two steps: First, it is shown that the length of the longest task provides a lower bound for latency when

$$\left\lceil \frac{\sum \textit{task execution times}}{m^*} \right\rceil < \textit{length of longest task}$$

After that, it is shown that

$$\left\lceil \frac{\sum \textit{task execution times}}{m^*} \right\rceil$$

provides a lower bound when

$$\left\lceil \frac{\sum \textit{task execution times}}{m^*} \right\rceil > \textit{length of longest task}$$

PROOF: (by contradiction)

Case 1:

Assume that

- $\left\lceil \frac{\sum \textit{task execution times}}{m^*} \right\rceil$ < length of longest task, **and**
- latency < length of longest task.

Let r = length of longest task. The *best case* occurs when the longest task, say t_k does not share a processor with any other tasks. In this case, t_k performs a new iteration every r time units, and the processor is finished $i \times r$ time units after startup. Since latency is measured by the throughput of all tasks in the system, system latency cannot be less than

the latency for any processor in the system. In particular, the processor containing t_k has a latency of r; therefore overall latency $\geq r$ = length of longest task, which violates the assumption that latency < length of longest task.

□

Case 2:

Assume that

- $\left\lceil \frac{\sum \text{task execution times}}{m^*} \right\rceil$ > length of longest task and
- latency < $\left\lceil \frac{\sum \text{task execution times}}{m^*} \right\rceil$

If $\left\lceil \frac{\sum \text{task execution times}}{m^*} \right\rceil$ > longest task, then the *best case* occurs when $\left\lceil \frac{\sum \text{task execution times}}{m^*} \right\rceil$ exceeds the longest task by 1 time unit (since all tasks can be scaled to integer lengths.)

Assume the least restrictive case, where all tasks are independent. Then, for $\left\lceil \frac{\sum \text{task execution times}}{m^*} \right\rceil$ to exceed the length of the longest task, all tasks may be assigned to processors so that there is no idle time in the schedule.
As long as $\left\lceil \frac{\sum \text{task execution times}}{m^*} \right\rceil$ > length of longest task, there must be at least one processor, say P_k, which has $\left\lceil \frac{\sum \text{task execution times}}{m^*} \right\rceil$ time slots filled with the first iteration of its tasks. In that case, P_k cannot iterate at a frequency greater than $\left\lceil \frac{\sum \text{task execution times}}{m^*} \right\rceil$. Contradiction.

□

Theorem 7 *(Variable-execution-time Upper Bound)*
If variable execution times are allowed in an iterative task system, with the following constraints, an upper bound on latency is defined as $\left\lceil \frac{n \times \text{length of longest task}}{m^*} \right\rceil$:

- *n variable-execution-time tasks*
- *m homogeneous processors*
- *precedence relationship $\prec$ defined between tasks*

- *level-strategy assignment*
- *all-iterations-first decision strategy*
- *number of iterations, i, is greater than the number of processors*
- *queued message passing (no bounds assumed on buffer size)*
- *task exists on only one processor*

PROOF: (by construction) Let $\mathcal{G}$ represent the graph of the task system; and let $\mathcal{G}'$, be the task system of $\mathcal{G}$, adjusted so that all task lengths are equal to the length of the longest task.

Then $latency(\mathcal{G}) \leq latency(\mathcal{G}')$.[2]

Let k be the length of the longest task.

Then, by Theorem 5, an upper bound for latency on $\mathcal{G}'$ is $\left\lceil \frac{k \times n}{m} \right\rceil$.

Since $\left\lceil \frac{k \times n}{m} \right\rceil \geq latency(\mathcal{G}') \geq latency(\mathcal{G})$, then $\left\lceil \frac{k \times n}{m} \right\rceil$ is an upper bound on latency.

□

4.8.4 Minimizing Number of Processors In some cases, the task system may not require all available processors in order to achieve the MAL. In that case, it would be useful to select an optimal number of processors so that remaining (idle) processors can be used for other applications. An obvious *upper bound* on the number of processors is n, where $n \equiv$ the number of tasks in the sytem. This bound allocates one task to each processor, which allows tasks to iterate at the minimum latency.

It is possible to place a bound on the number of processors required to obtain the optimal latency in a task system. For example, the parameters of a task system could indicate that a task system requires *at least* 4 processors (minimum lower bound) and *at most* 6 processors (maximum upper bound) to achieve the optimal latency. Although the optimal number of processors is not completely determined, the bounded range of values

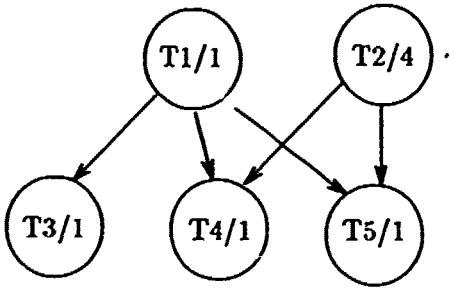
[2] If this were not true, then it would be necessary to have a space of length 2 to map every iteration of a task of length 1.

can be used when deciding on the number of processors required by the system. Figure 4.20 shows the relationship between lower and upper-bound notations used in this section.

Figure 4.20. Relationship between bounds

Intuitively, the idea of minimizing the number of processors can be seen in Figure 4.21. In this figure, Task 2, the longest task, requires as much time for execution as all other tasks in the system combined! Therefore, there is no benefit, in terms of latency, in assigning this task system for iterative execution to more than two processors.

Example:



$$\left\lceil \frac{\sum \textit{execution times}}{\textit{length of longest task}} \right\rceil = \left\lceil \frac{1+4+1+1+1}{4} \right\rceil = 2$$

P1	$T2$	$T2$	$T2$	$T2$			
P2	$T1$	$T1'$	$T1''$	$T1'''$	$T3$	$T4$	$T5$

Figure 4.21. Lower Bound on Processors

This value for the minimum value of the lower bound gives the least number of processors required to obtain the optimal latency:

Theorem 8 *(Number of Processors: Variable-execution-time)*

In an iterative task system with

- *n variable-execution-time tasks*
- *m homogeneous processors*
- *precedence relationship $\prec$ defined between tasks*
- *level-strategy assignment*
- *all-iterations-first decision strategy*
- *number of iterations, i, is greater than the number of processors*
- *queued message passing (no bounds assumed on buffer size)*
- *task exists on only one processor*

the minimum value for a lower bound on the number of processors can be computed by the following formula:[3]

$$\left\lceil \frac{\sum \textit{execution times}}{\textit{length of longest task}} \right\rceil$$

PROOF: An upper bound on the minimum achievable latency (MAL) is defined as

$$max\left(\left\lceil \frac{n \times \textit{length of longest task}}{m^*} \right\rceil, (\textit{length of longest task})\right)$$

(Theorem 7).

[3]This theorem was derived by Professors Lamont and Hammell of the Air Force Institute of Technology

In non-preemptive scheduling, the length of the longest task must remain constant. Therefore, the only way to reduce the MAL is to reduce

$$\left\lceil \frac{n \times \textit{length of longest task}}{m^*} \right\rceil$$

This can be done by 'throwing more processors at the problem!' Thus, if

$$\left\lceil \frac{n \times \textit{length of longest task}}{m^*} \right\rceil < (\textit{length of longest task}),$$

the MAL can be decreased by increasing m^*, the number of processors in use.

□

4.9 NP-Complete Aspects

4.9.1 Background The level strategy presented in Section 4.3 generates a minimal-latency mapping of tasks to processors in $O(n^3)$ time. However, the level strategy only guarantees optimal latency for task systems with the following restrictions:

- homogeneous processors
- no preemption
- equal-execution-time tasks
- queued message passing
- task exists on only one processor
- no feedback loops

As constraints are relaxed, the level strategy is no longer guaranteed to generate an optimal mapping. Furthermore, it is not obvious whether any p-time algorithm can produce minimal-latency schedules. At present, there are numerous problems which have no known polynomial time solutions (7, 15). The class *NP-complete* contains problems of this type. A primary characteristic of the class NP-complete is that every NP-complete problem can be mapped to every other NP-complete problem with a polynomial-time

transformation (1, 7, 15). This implies that *if a polynomial-time solution is found for one NP-complete problem, a p-time solution can be derived for all NP-complete problems.*

At first glance, NP-complete problems often *appear* to be no more complex than problems with polynomial-time solutions. Closer scrutiny, however, reveals intractable aspects of the problem. Since the one-pass scheduling problem is NP-complete when variable-execution-time tasks are allowed (10), a reasonable conjecture is that the iterative scheduling problem also becomes NP-complete in the variable-execution-time case.

4.9.2 Proving NP-Completeness Knowing whether a given problem is NP-complete is useful for several reasons:

- If a problem can be shown to be NP-complete, then there is no currently-known polynomial-time algorithm to solve it (7, 15). Although this does not preclude the possibility that a p-time algorithm will eventually be discovered, it emphasizes the difficulty of deriving such an algorithm.
- If an optimal solution cannot be acquired, it may be possible that an approximation algorithm will generate an acceptable solution (21, 25).
- Once a problem is known to be NP-complete, effort can be redirected into refining the graph search heuristics, rather than attempting to derive a p-time solution.

As a first step toward proving a problem NP-complete, the problem is often restated as a *decision problem,* which can be answered with a 'yes' or 'no' (7). For example, the one-pass scheduling problem. "Find the minimum execution time for a given task system," may be restated as "Is there a schedule for task system $\mathcal{G}$ which completes by time x?"

In order to prove that an open problem is NP-complete two conditions must be met (1, 7, 15):

1. A nondeterministic Turing machine (NDTM)[4] must be able to solve the problem in polynomial time. This condition demonstrates that the open problem is *no larger then* NP-complete.

[4]The concept of a NDTM is an abstraction; such a machine does not exist in the physical world.

2. A known NP-complete problem must be polynomially transformable to the open problem. The idea behind this transformation is shown in Figure 4.22 (20). This shows that the open problem is *at least* as large as an NP-complete problem.

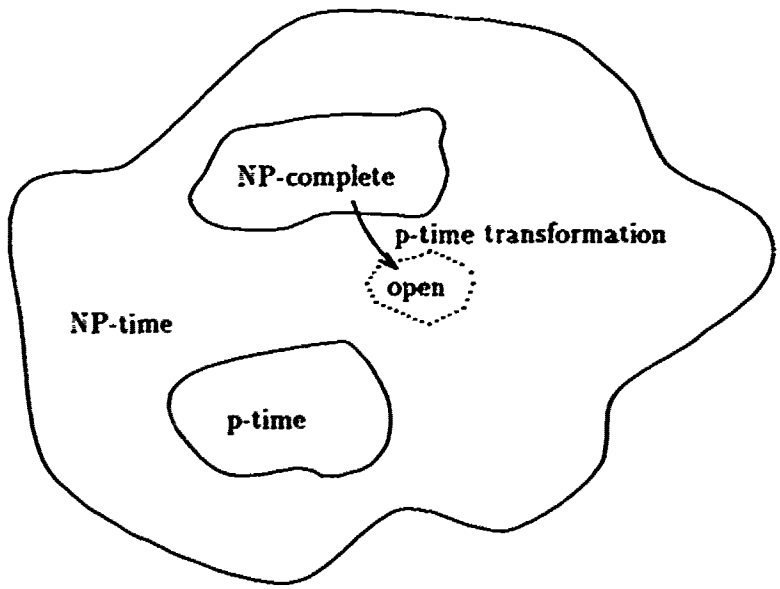


Figure 4.22. Transformation to Show NP-Completeness

4.9.3 Variable Execution Time Systems Results concerning the iterative scheduling problem of Section 4.3 were dependent on several constraints, such as equal-execution time and homogeneous processors. In general, relaxing these constraints increases the difficulty of generating a minimal-latency solution. This section discusses results when the equal-execution-time restriction is removed.

Theorem 9 *(NP-completeness)*

The minimal-latency iterative scheduling (MLIS) problem with

- *n variable-execution-time tasks,* $\in \{1, 2\}$
- *m homogeneous processors*
- *precedence relationship* $\prec$ *defined between tasks*

- *level-strategy assignment*
- *all-iterations-first decision strategy*
- *number of iterations, i, is greater than the number of processors*
- *queued message passing (no bounds assumed on buffer size)*
- *task exists on only one processor*

is NP-Complete.

This theorem is proved by showing that MLIS problem with task execution times $\in \{1,2\}$ can be solved in polynomial-time by a NDTM and that a known NP-complete problem can be mapped to the MLIS problem with a polynomial-time transformation.

PROOF:

Problem $\in$ NP.

In order to show that the minimal-latency iterative scheduling (MLIS) problem with execution times of $\{1,2\} \in$ NP, the problem is restated as a decision problem. It is then necessary to show that the question "Does the MLIS problem produce a schedule with latency $\mathcal{L}$?" can be answered in polynomial time on a nondeterministic Turing machine (NDTM) (15).

For a NDTM to answer this question in polynomial time, each copy of the NDTM makes an assignment of tasks to processors, as shown in Figure 4.23. Since there are a finite number of ways to assign tasks to processors, there are a finite number of NDTM's. In Figure 4.23, a copy of the NDTM is generated to produce each schedule beginning with a certain assignment of tasks to processors. For example, if $task_3$ is assigned to $P1$ at $time_1$, separate copies of the NDTM generate schedules beginning with $P2$ at $time_1$. This procedure is continued until the last task is scheduled on all copies of the NDTM. At this point, the latency of each assignment can be checked in at most $O(nk)$ time, where n is the number of tasks in the system and k is the number of iterations. Thus, the NDTM produces an answer in polynomial time.

Polynomial Transformation.

The second step in proving an open problem to be NP-complete requires that a known

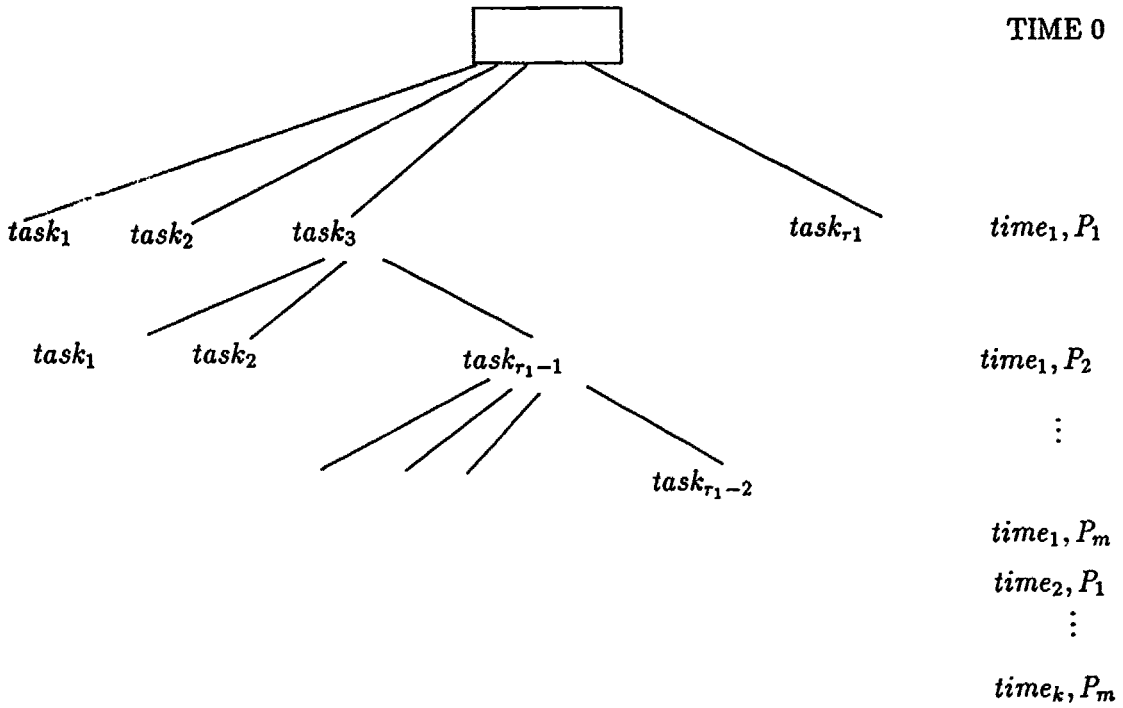


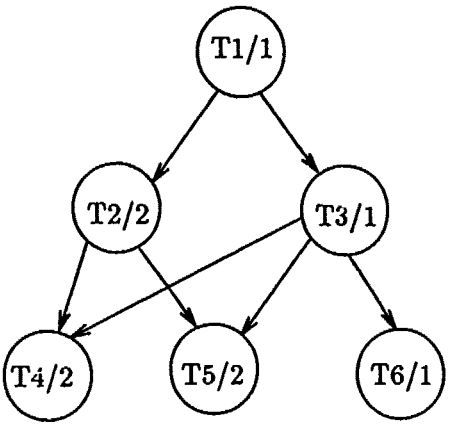
Figure 4.23. Nondeterministic Turing Machine Assignment

NP-complete problem be transformed to the open problem in polynomial time (15). Let Π represent the non-iterative scheduling problem with tasks of one- and two-time units, an NP-complete problem (34), and let Π' represent the iterative problem with tasks of one- and two-time units. Then the goal becomes *Transform* Π *to* Π'.

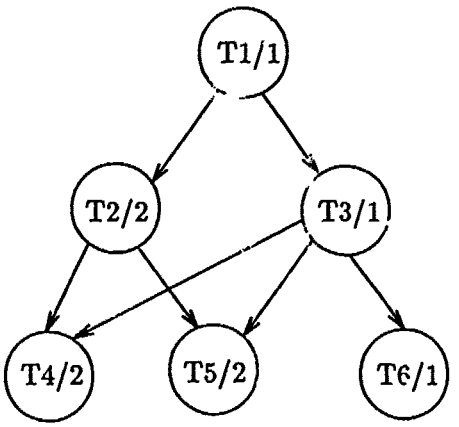
The task graph for Π and the task graph for 1 iteration of Π' can be represented as shown in Figure 4.24. These graphs have a 1-1 correspondence, as shown in Figure 4.24 (c). Thus, Π can be directly transformed to Π'.

□

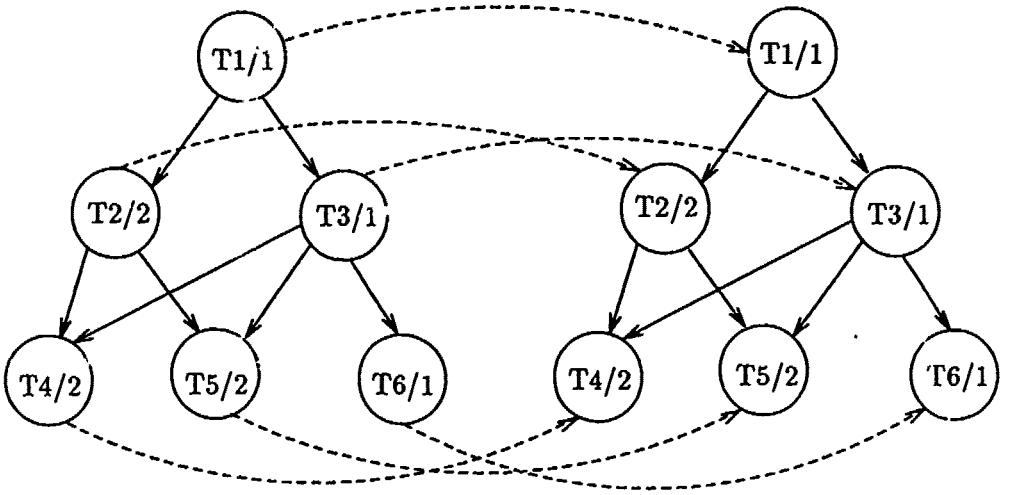
4.9.3.1 Transformation to Other NP-Complete Problems A primary characteristic of NP-complete problems is that any NP-complete problem can be transformed to any other in polynomial time (1, 7, 15). Since the MLIS problem has been proved NP-complete, it is possible to transform the MLIS problem to any other NP-complete problem with a polynomial-time transformation. The simplest method of transformation uses the



(a) task graph representation
for classical problem



(b) task graph representation
for 1 iteration of the
iterative problem

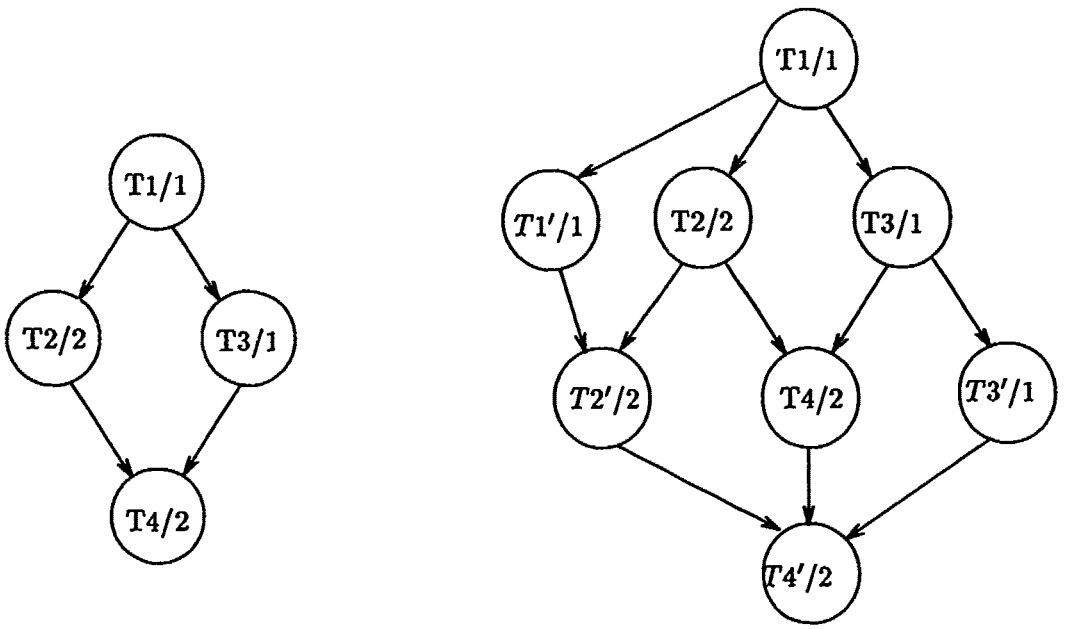


(c) Mapping classical problem to iterative problem
for # iterations = 1

Figure 4.24. Mapping NP-Complete Problem into Open Problem

relationship between the MLIS problem (Π') and the noniterative scheduling problem (Π): once Π' has been transformed to Π, all previous transformations which apply to Π also apply to the transformed version of Π'.

The idea behind the transformation from Π' (the iterative problem) to Π (the general scheduling problem) is shown in Figure 4.25 for 2 iterations of Π'. In this transformation, both iterations of the MLIS problem are mapped into a precedence graph. This precedence graph is identical to the task graph which represents the non-iterative scheduling problem with tasks of one- and two-time units (10). Therefore, the MLIS problem can be mapped into the non-iterative scheduling problem.



(a) Task graph for iterative system

(b) Precedence graph for 2 iterations of task system

Figure 4.25. Transforming the Iterative Problem to the Classical Problem

Since every known NP-complete problem is classified by proof and transformation, there is a finite number of known NP-complete problems. In addition, there are numerous cases where the class of a particular problem is unknown (10, 15). Thus the border between NP-complete problems and problems with polynomial-time solutions is not a well-defined threshold. For example, the classical scheduling problem with arbitrary precedence has a

p-time solution for UET tasks mapped to 2 processors, but the problem is NP-complete when variable-execution-times are allowed (10). In the same manner, there is a polynomial-time algorithm for the iterative scheduling problem with equal-execution-time tasks on a fixed number of processors, but the problem of deriving an optimal-latency schedule for iterative systems with variable-execution-time is NP-complete (Theorem 9).

Since the iterative problem with equal-execution-times has a polynomial-time solution, an additional result can be conjectured for the noniterative problem:

> If, in a noniterative UET system, the task graph can be collapsed to represent k iterations of an iterative task system, then the level strategy produces an optimal assignment in polynomial time.

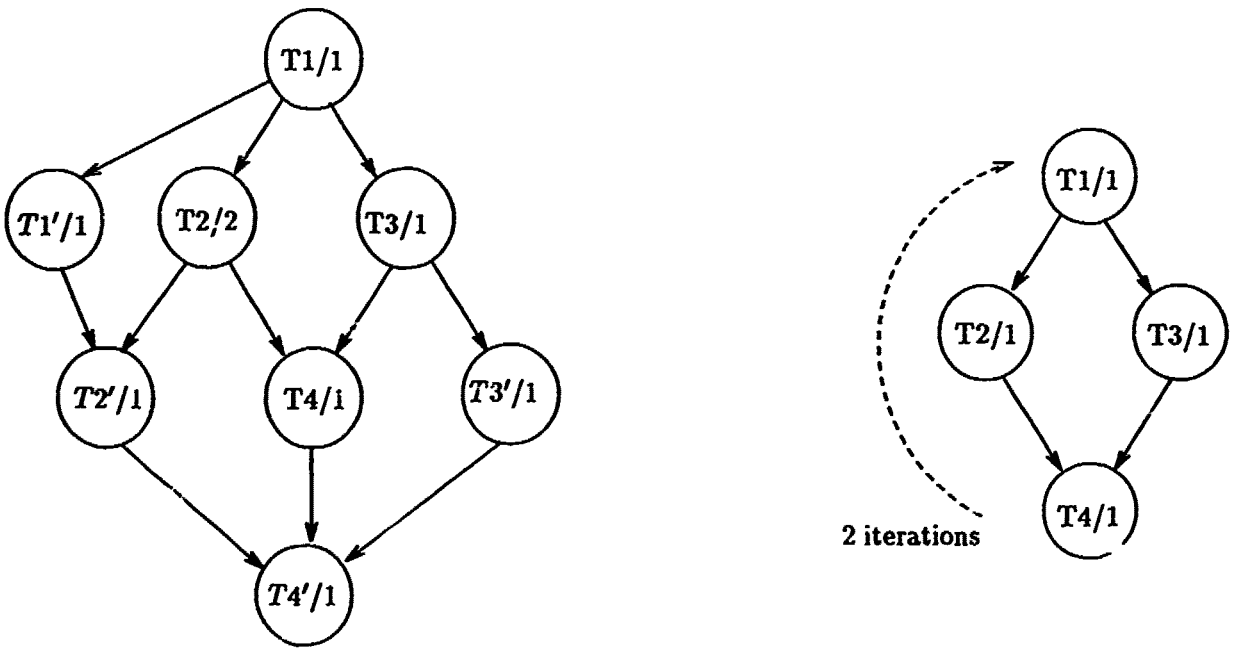
For example, Figure 4.26 shows a UET task graph reduced to an iterative task graph which must be repeated twice.

4.10 Summary

The mathematical development for the iterative scheduling problem is presented in this chapter. Proofs are given to show that the Level Strategy produces an optimal mapping for UET systems, and that systems with variable execution time tasks are NP-complete. An additional proof is shown to demonstrate that the *number of processors* can be minimized in an iterative system with variable execution times. These results are summarized in Table 4.1.

Task Lengths	*Precedence*	*Problem complexity*
equal	arbitrary	p-time
1 *or* 2	arbitrary	NP-complete

Table 4.1. Results for Minimizing Execution Time: Iterative Task Systems (non-preemptive; no resource constraints)



(a) 1-pass task graph

(b) Representation as iterative system

Figure 4.26. Transforming the Classical Problem to the Iterative Problem

V. *Application/Experimental Results*

Computer simulation of electronic circuit behavior can be used to streamline the process of circuit and chip design. The VHDL (VHSIC High-level Design Language) was created to enable designers to model circuit behavior on a computer, rather than actually building the circuit to be tested. As designs become larger and more complex, sequential computer simulations of circuit behavior take a disproportionate amount of time. The parallel simulations which were implemented in previous research (30) are used to test the mapping strategies developed in Chapters III and IV, with the objective of minimizing overall execution time.

5.1 VHDL Application

The precedence graph in Figure 5.1 shows the simulation to be modeled. This simulation uses a queued-message protocol for passing data between tasks. For example, Task 1 can execute several times before Task 18 executes once. Every time a data value necessary for Task 18 is changed in Task 1, a message is sent to Task 18. These messages are queued in an input buffer until Task 18 is ready to respond to each message.

5.1.1 VHDL Parameters and Results Tests were conducted on VHDL simulations to determine the effects of various mapping strategies on execution time. Table 5.2 shows the matrix of tests which were used to test the various mapping strategies. The simulation parameters were set to model circuit simulation for different amounts of time: the shortest simulation modeled 1000 nanoseconds of circuit behavior (approximately 10 iterations of the complete task system); the longest simulation modeled 64,000 nanoseconds of operation (640 iterations of the system).

The LEVEL8 strategy uses the level algorithm developed in Section 5; the GREEDY mappings assign $\lceil \frac{n}{m} \rceil$ tasks to each processor so that the first $\lceil \frac{n}{m} \rceil$ tasks in the system are assigned to processor 0, the 2nd $\lceil \frac{n}{m} \rceil$ tasks to processor 1, etc. The 1-PASS OPT strategy assigns tasks to processors based on one iteration of the task system (classical scheduling

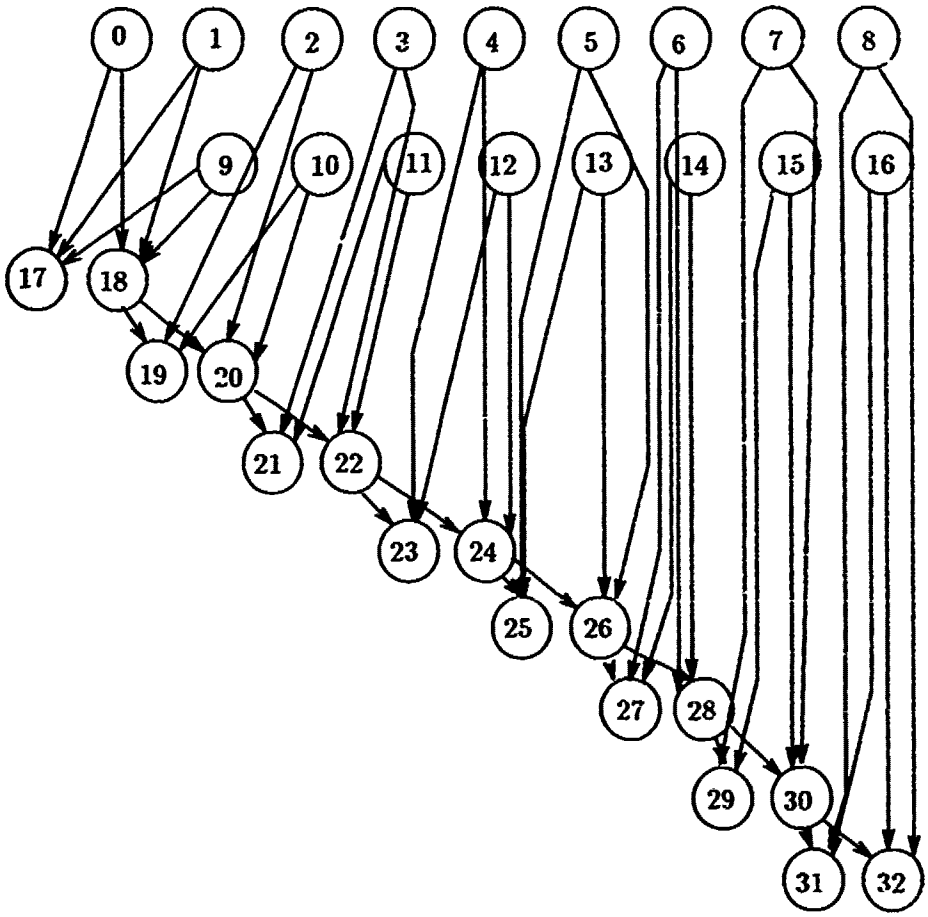


Figure 5.1 Precedence Graph: VHDL tasks

method). Since this mapping allocates all tasks to only 4 processors, poor performance is expected. The UNBALANCE8 strategy assigns one or two tasks to most processors and overloads the remaining processor with all remaining tasks. These assignments (for 8-node mappings) are shown in Figure 5.1.

In all test cases, a linear relationship holds for simulated-time increments. (i.e. If a mapping for a 10-nanosecond simulation takes 10 seconds, a 20-nanosecond simulation with the same mapping will take 20 seconds.)

Figure 5.2 shows the data from Table 5.2 in graphical form.

In the 8-node test cases, the level8 strategy consistently out-performed all other mapping schemes. In runs which simulated 64,000 nsec of circuit behavior, the level8 strategy takes only 56% of the time of the next-best mapping (greedy8). When simulation runs

Task Number	*Processor #*			
	Level	1-Pass Optimal	Greedy	Unbalanced
0	0	0	0	0
1	1	1	0	1
2	2	2	0	2
3	3	2	0	3
4	4	2	0	4
5	5	2	1	5
6	6	2	1	6
7	7	2	1	7
8	0	2	1	0
9	1	3	2	0
10	2	3	2	0
11	3	3	2	0
12	4	3	2	0
13	5	3	3	0
14	6	3	3	0
15	7	3	3	0
16	0	3	3	0
17	1	0	4	0
18	2	1	4	0
19	3	0	4	0
20	4	1	4	0
21	5	0	5	0
22	6	1	5	0
23	7	0	5	0
24	0	1	5	0
25	1	0	6	0
26	2	1	6	1
27	3	0	6	1
28	4	1	6	1
29	5	0	7	1
30	6	1	7	1
31	7	1	7	1
32	0	0	7	1

Table 5.1. 8-node Mapping Strategies

4-node mappings	*Simulated time*		
	1000 nsec	*8000 nsec*	*16000 nsec*
level4	34	253	511
1-pass optimal	51	381	780
greedy4	60	447	897
unbal4	72	541	1051

8-node mappings	*Simulated time*				
	1000 nsec	*8000 nsec*	*16000 nsec*	*32000 nsec*	*64000 nsec*
level8	18 sec	126 sec	252 sec	509 sec	1108 sec
1-pass optimal	51 sec	381 sec	779 sec	1621 sec	3358 sec
greedy8	42 sec	314 sec	595 sec	1086 sec	1973 sec
unbal8	79 sec	573 sec	1147 sec	2272 sec	4404 sec

Table 5.2. VHDL Test Cases

are based on the level strategy of assignment, the VHDL simulation completes in approximately 56% of the time needed by the greedy8 strategy, the second-best mapping. When compared with an optimal schedule generated by the one-pass scheduling algorithm, the level strategy simulation required only 33% of the time needed by the classical mapping scheme. This is due to the nature of the VHDL task system: an optimal 1-pass mapping assigns the longest chain of tasks to one processor, in an attempt to minimize communication time. Finally, comparison with a deliberately unbalanced mapping shows that the level strategy executes in 25% of the time.

As the number of processing nodes is increased from 4 to 8, the time for execution decreases to half of the 4-node amount; therefore, linear speedup is obtained, as shown in Figure 5.3.

5.2 Gaming Simulation Results

In order to further substantiate the results found in VHDL simulation mappings, a gaming simulation developed in previous research [23] was tested with the same mapping strategies as used in the VHDL test runs. The precedence graph for this simulation is shown in Figure 5.4.

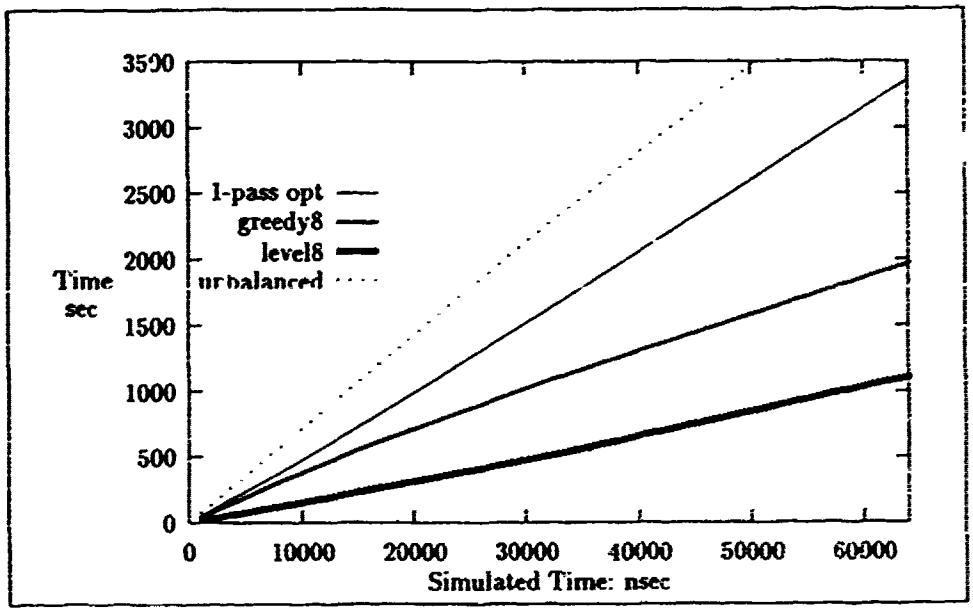


Figure 5.2. Varied Mapping Strategies

Table 5.3 and Figure 5.5 summarize the results of these tests.

4-node mappings	*Simulated time*				
	1000	*2500*	*5000*	*10000*	*15000*
level4	42 sec	114	260	601	1179
1-pass optimal	47	130	286	751	1375
greedy4	42	116	267	695	1272
unbal4	67	157	346	833	1478

Table 5.3. Gaming Test Cases (Spin Loops = 10.000)

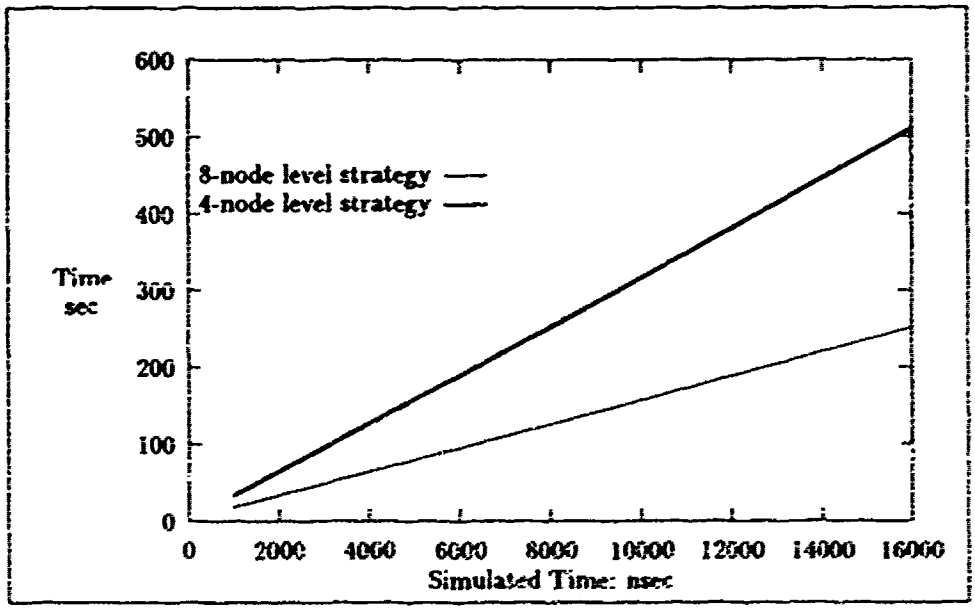


Figure 5.3. Speedup

When spin loops of 100,000 are input, the greedy 4 strategy (which assigns tasks so that no two *communicating* tasks are on the same processor) produces results in less time than the level strategy mapping. This may be due to some unspecified simulation parameter or to the small number of tasks on each processor–if a sending task and receiving task are both blocked for communication on the same processor, then the node operating system has no other tasks which can be working on computation.

4-node mappings	*Simulated time*		
	5000	*10000*	*15000*
level4	562	1238	2102
3-pass optimal	719	1605	2660
greedy4	524	1138	2057
unbal4	915	1973	3130

Table 5.4. Gaming Test Cases (Spin Loops = 100,000)

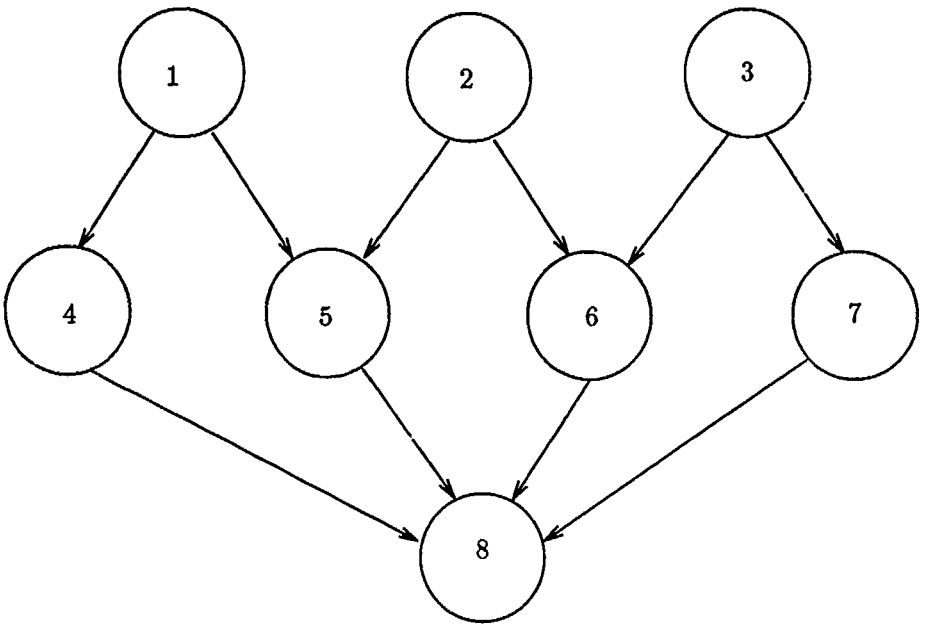


Figure 5.4. Gaming Simulation Precedence Graph

5.3 Summary

This chapter discusses the results when various mapping strategies are used to assign VHDL simulation tasks. When simulation runs are based on the level strategy of assignment, the VHDL simulation completes in approximately 56% of the time needed by the second-best mapping, which also assigns a balanced load to each processor. When compared with the optimal one-pass mapping, the results appear counterintuitive: the level strategy simulation executes in 33% of the time needed by the one-pass optimal simulation; however, an optimal one-iteration mapping for this simulation assigns all tasks to 4 processors; this minimizes communication time, but does not take advantage of all available processors.

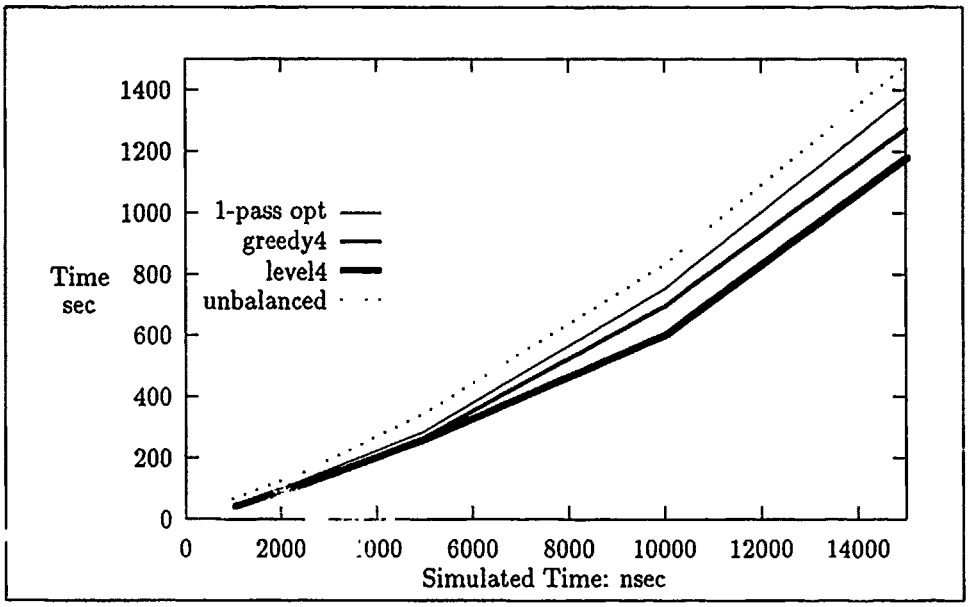


Figure 5.5. Varied Mapping Strategies (Spin Loops 10,000)

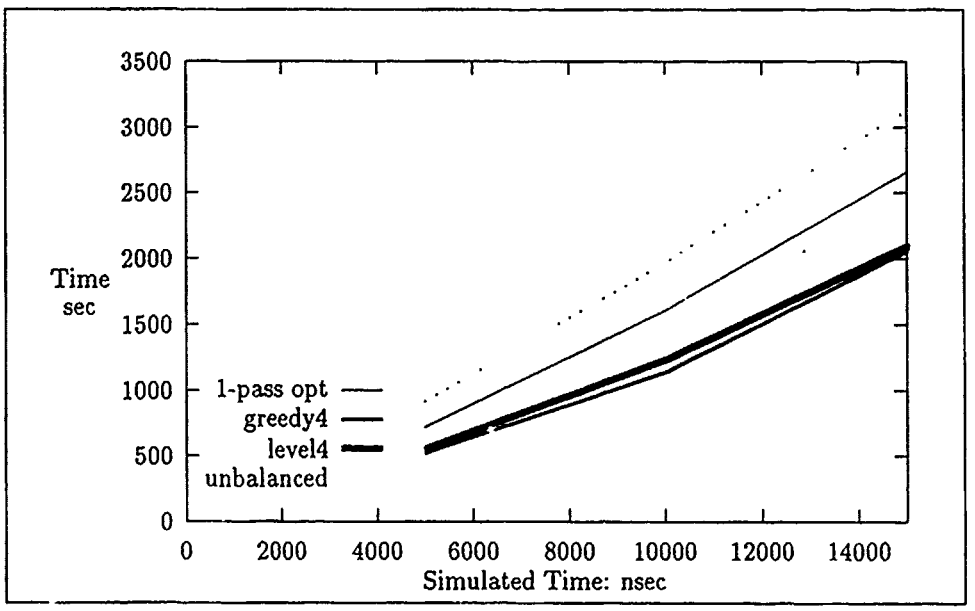


Figure 5.6. Varied Mapping Strategies (Spin Loops 100,000)

VI. Conclusions and Recommendations

This research has explored the problem of mapping parallel simulation tasks to processors, with emphasis on VHDL circuit simulations. In the process, some interesting results have been uncovered – in particular, the tendency for a single-execution optimal mapping in the classical sense to produce suboptimal timing results for parallel simulations. In addition to producing an algorithm for optimal iterative schedules, the mathematical foundation for the simulation task scheduling problem has been developed, and the set of NP-complete now includes the iterative scheduling problem.

6.1 Conclusions and Contributions

This research has approached the science of Scheduling Theory from a new perspective; as a result, several contributions to the field have been made. The most important of these are the following:

- This investigation has exposed the fallacy that an optimal mapping for a single execution is optimal for multiple iterations as well. Schedules which are optimal in the classical sense are shown to produce less-than-optimal results for simulation tasks. The critical factor which produces this anomoly is shown to be the iterative nature of simulation tasks.
- Having determined that the iterative nature of simulations presents significant differences from classical multiprocessor scheduling, a polynomial-time algorithm, the *level strategy*, is developed for scheduling iterative task systems with properties corresponding to VHDL simulations. This algorithm is shown to produce speedup in applications other than circuit simulation when the specified constraints are met.
- In order to formalize the theory of iterative task scheduling, a mathematical foundation is developed for iterative task systems. The characteristics of parallel VHDL simulations are used to form the basis of this foundation, leading to the following theorems which are proved for task systems conforming to these conditions:

n unit-execution-time tasks [1]

m homogeneous processors

precedence relationship $\prec$ defined between tasks

level-strategy assignment

all-iterations-first decision strategy

number of iterations, i, is greater than the number of processors

queued message passing

task exists on only one processor

- The lower bound for latency, the time between successive iterations of a given task, on task systems with the above assumptions is $\lceil \frac{n}{m} \rceil$ (i.e. lower bound $\geq \lceil \frac{n}{m} \rceil$) .
- Task systems which meet the above assumptions have a latency upper bound of $\lceil \frac{n}{m} \rceil$ (i.e. upper bound $\leq \lceil \frac{n}{m} \rceil$) ..
- Task systems which meet the above conditions have an absolute bound on latency of $\lceil \frac{n}{m} \rceil$ (because the upper-bound is equal to the lower-bound).
- If the above conditions are met, and all tasks have equal execution time k, then an absolute bound on latency is $\left\lceil \frac{k \times n}{m} \right\rceil$.

This theoretical foundatation is further expanded by removing the equal-execution-time restriction and examining the variable-execution-time case. Theorems are proved about variable-execution-time systems with the following constraints:

n variable-execution-time tasks

m homogeneous processors

precedence relationship $\prec$ defined between tasks

level-strategy assignment

all-iterations-first decision strategy

number of iterations, i, is greater than the number of processors

[1] This constraint is later scaled in the time domain and results are proved for *equal-execution-time* tasks.

queued message passing

task exists on only one processor

- A lower bound on latency for iterative task systems with the above conditions is [2]:

$$max \left\lceil \frac{\sum task\ execution\ times}{m^*} \right\rceil, length\ of\ the\ longest\ task$$

- An upper bound on latency for iterative task systems with the above constraints, is defined as

$$upper\ bound \leq \left\lceil \frac{n \times length\ of\ longest\ task}{m^*} \right\rceil .$$

- In an iterative task system with the constraints listed above, the minimum value for a lower bound on the number of processors can be computed by the following formula:[3]

$$lower\ bound \geq \left\lceil \frac{\sum execution\ times}{length\ of\ longest\ task} \right\rceil$$

- In addition to the mathematical and simulation contributions listed above, the set of NP-complete problems is expanded to include iterative task systems. When classical repetitive problems, *periodic scheduling* and *fixed cycle scheduling*, are examined, it becomes evident that simulation task scheduling requires different parameters and goals (Section 2.2). Thus the category of *iterative scheduling* is introduced. Problems which conform to the iterative scheduling parameters and which have task systems with variable execution times are shown to be NP-complete; problems with the restrictions outlined above and with equal-execution-time can be scheduled to minimize latency in polynomial-time.
- In addition to developing and proving the theoretical foundation, experimental results, based on current parallel implementations, are used to validate the use of

[2] where m^* denotes processors-in-use

[3] This theorem was derived by Professors Lamont and Hammell of the Air Force Institute of Technology

latency as a measure of optimality. When parallel circuit simulations are executed with different mappings, the level strategy consistently out-performed the other test cases. A sample of the results is shown below:

Mapping Technique	*Execution Time*
level strategy	126 seconds
1-pass optimal	314 seconds
greedy method	381 seconds
unbalanced	573 seconds

Timing runs which use the level algorithm to schedule tasks ran significantly faster than any other test case; the closest competitor was the greedy strategy which took 2.49 times as long to execute the same circuit simulation.

6.2 Recommendations for Further Research

The iterative scheduling problem is proven to be NP-complete when the variable execution-times are allowed. The following topics are suggested for future research into the iterative scheduling problem:

- Expand the theoretical foundation of iterative task scheduling by relaxing the constraints on the basic Equal Execution Time (EET) system:
  - Task systems with dynamic load balancing
    (goal: minimize latency)
    n EET tasks
    m homogeneous processors
    no limits on buffer size
  - Heterogeneous processor systems
    (goal: minimize latency)

n EET tasks

Processor speeds 1 and 2

no limits on buffer size

- Simulations with feedback loops

  (goal: minimize latency)

  n EET tasks

  m homogeneous processors

  no limits on buffer size

The feedback constraint requires dependent tasks to iterate before subsequent iterations of the first task. For example, in Figure 6.1, Task A is allowed to perform all iterations consecutively, but Task B must receive an input from Task C before it can perform a subsequent iteration.

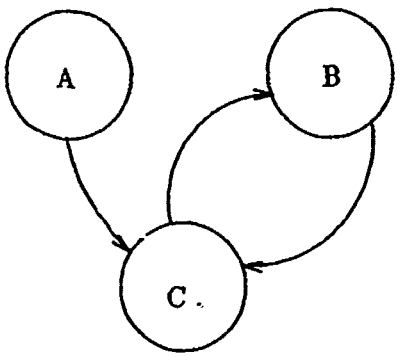


Figure 6.1. Precedence Graph with Feedback

- Task systems with variable number of processors

  (goal: minimize number of processors)

  n EET tasks

  homogeneous processors

  no limits on buffer size

- Expand the decision strategy results for the iteration-number decision strategy (Appendix D).
- Examine results when variable execution times (i.e. spin loops of 50,000; 100,000; and 150,000) are incorporated as part of the VHDL task sytem, corresponding to the variable-execution-time problem.
- As larger VHDL simulations become available, use the level ... assign tasks to processors. Validate conclusions.
- Explore mapping strategies for simulations which contain feedback loops and cycles. Methodologies exist for transforming precedence graphs containing cycles into acyclic graphs (24). Determine and implement algorithms for this transformation process.

Appendix A. *Combinatoric Complexity Example*

In Chapter 1, the following problem is considered:

> Find the number of combinations which must be examined to derive an optimal schedule for 60 independent tasks assigned to 2 processors.

When the problem was simplified so that exactly 30 tasks were scheduled on each processor, there were still 60! possible combinations.

A.1 Detailed Example

In order to explore the worst-case ramifications of an unsimplified version of the problem, an system of independent tasks is chosen. (Dependent, or precedence-constrained, systems have fewer possibilities for combinations of tasks in each time slot.) In this case, a system of 5 tasks is to be scheduled on 2 processors. All combinations of assignments must be considered:

5 tasks on Processor 1; 0 tasks on Processor 2

(There are 5! x 0! ways to arrange these tasks.)

4 tasks on Processor 1; 1 task on Processor 2

(There are 4! x 1! ways to arrange these tasks.)

3 tasks on Processor 1; 2 tasks on Processor 2

(There are 3! x 2! ways to arrange these tasks.)

2 tasks on Processor 1; 3 tasks on Processor 2

(There are 2! x 3! ways to arrange these tasks.)

1 tasks on Processor 1; 4 tasks on Processor 2

(There are 1! x 4! ways to arrange these tasks.)

0 tasks on Processor 1; 5 tasks on Processor 2

(There are 0! x 5! ways to arrange these tasks.)

Therefore, the following equation applies:

$$Number\ of\ combinations\ =\ 5!0! + 4!1! + 3!2! + 2!3! + 1!4! + 0!5!\ =\ 312\ combinations$$

In order to determine an optimal schedule by exhaustive search, 312 combinations must be examined. For the simple problem of 5 tasks assigned to 2 processors, exhaustive search is not an unreasonable option. However, if the problem is increased by just one task, the number of combinations that must be checked in an exhaustive search is

$$Number\ of\ combinations\ =\ 6!0! + 5!1! + 4!2! + 3!3! + 2!4! + 1!5! + 0!6!\ =\ 1812\ combinations.$$

Table A.1 indicates the growth rate for number of combinations as the number of tasks is incremented[1]

n	$n!$	number of combinations
7	5040	$> 10,080$
8	40320	$> 80,640$
9	362,880	$> 725,760$
10	3,628,800	$> 7,257,600$

Table A.1. Growth Rate for $n!$

[1]The number of combinations for each schedule of n tasks includes $n!$ tasks on P1 and 0! task on P2, as well as the mirror-image combinations (0! tasks on P1 and $n!$ on P2).

Appendix B. *Level Strategy Example*

This appendix shows a step-by-step example of a task schedule generated by the level strategy. Figure B.1 contains the precedence graph for a system where $t1 \prec t2 \prec t3$. Assume these tasks are to be scheduled on 2 processors.

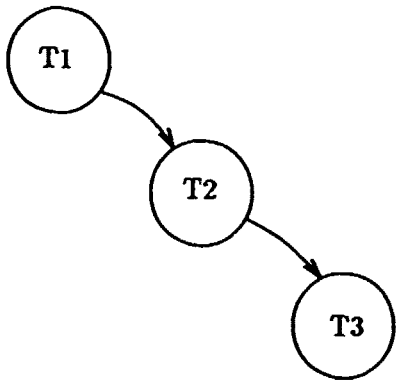


Figure B.1. Precedence Graph

The first step assigns a level to each task in the system:

> Let n be defined as the number of tasks and m the number of processors. Let the *level* of a node x in a task graph be the maximum number of nodes (including x) on any path from x to a terminal task. A terminal task is at level 1. Let a task be *ready* when all its predecessors have been executed.

Figure B.2 shows the assignment of levels to each task.

The computer representation of the precedence graph is a precedence matrix, where a 1 in the $(i,j)^{th}$ position of the matrix indicates that $task_j \prec task_i$, as shown below:

	1	2	3
1	0	0	0
2	1	0	0
3	0	1	0

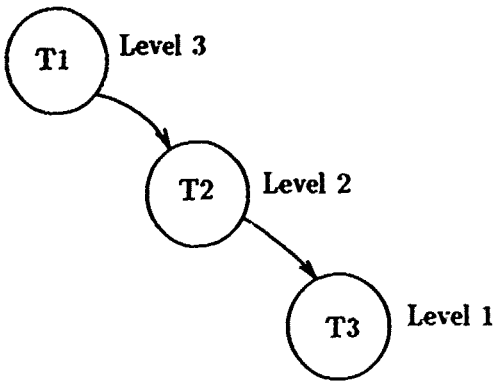


Figure B.2. Level Assignment

$(2,1) = 1 \Rightarrow t1 \prec t2$

The schedule is generated as follows: Each row in the precedence matrix is scanned. If a row contains all zeros, then that task is ready. For example, at $time_0$, row_1 is all zeros. Therefore, $task_1$ is the only task which is ready for scheduling, and $task_1$ is assigned to P_1:

P1	T1
P2	

After all ready tasks (in this case, only $task_1$) are assigned, the time is incremented (so that the next ready task will be scheduled at $time_1$), and the precedence matrix is recomputed to indicate that

- All assigned tasks are no longer available for scheduling.
- Tasks whose dependencies are satisfied are now ready:

	1	2	3
1	-1	0	0
2	0	0	0
3	0	1	0

At $time_1$, $task_1$ is finished processing, and $task_2$, the only ready task, is assigned to P_2. The Gantt chart becomes

P1	T1	
P2		T2

The same procedure is followed at $time_2$: The precedence matrix is recomputed, and ready tasks are assigned to processors.

	1	2	3
1	-1	0	0
2	-1	0	0
3	0	0	0

Precedence Matrix

P1	T1		T3
P2		T2	

At this point, all tasks are assigned to processors.

If there are more ready tasks than available processors, the time is incremented when all processors are given a task; however, the precedence matrix is not recomputed until the last ready task is assigned to a schedule. Figure B.3 shows a case where 4 tasks are ready at $time_0$, but there are only 3 available processors. In this case, the assignment of tasks to processors continues until all ready tasks are scheduled.

B.1 Level Strategy

The level strategy is concerned with the assignment of tasks to processors; however, the operating system of the parallel processor is assumed to do the bookkeeping associated with iterations of different time slots. This assumption allows the scheduling algorithm to determine what set of tasks is assigned to each processor without knowledge of operating

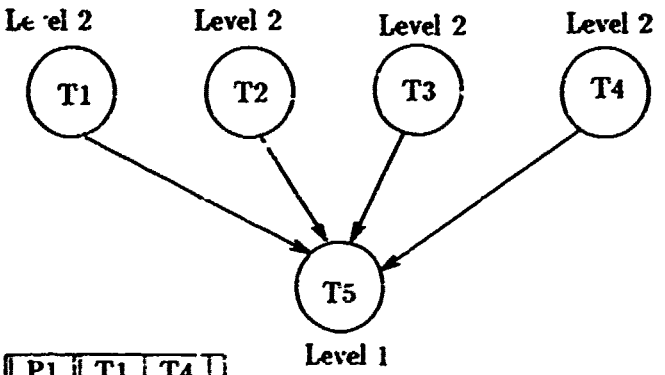


Figure B.3. More ready tasks than processors

system parameters such as the time quantum used for multitasking on each processor. An algorithmic description of the level strategy is given below:

last-assigned-processor = m
while levels remain
    calculate number-ready
    for i in 1..number-ready
        assign task i to Processor $((last - assigned - processor + 1) \bmod m)$
        last-assigned-processor = $((last - assigned - processor + 1) \bmod m)$
    end for
end while

In addition to the level strategy defined above, tasks must be scheduled within a processor by the decision strategy given in Chapter 4 in order to ensure minimal latency.

Since the last-assigned-processor is used to determine the next-processor for assignment, the amount of tasks assigned to each processor is roughly equivalent. If the queued-message paradigm is used, this allows predecessor tasks to iterate in 'vacant' time slots. Experimental results (Chapter 5) confirm the optimal nature of task assignments made with this strategy.

Appendix C. *A* Search*

The problem of generating an optimal schedule is essentially a search problem. An exhaustive search requires all possible schedules to be generated. From that list of all schedules, the one with the shortest execution time is chosen.

Since, however, the problem of generating all possible schedules for real-life problems requires more computing power than is available, *heuristics* are used to drive the search toward the optimal, by rejecting high-cost branches of the search tree.

C.1 A overview*

The A* search uses two functions to guide the search (26, 29):

- g: A function which is used to calculate the cost of the path which has been traversed so far.
- h: An estimator function, which is used to guess the cost from the current node to the goal state.

These functions are added to obtain the node-selection function f, ($f = g + h$), which leads the search to attempt lowest-cost paths before higher-cost paths.

C.2 Sample A Search*

Figure C.1 shows a graph which represents the problem to be solved by A* search (26).

- s is the start node.
- $n2$ is the goal node.
- Arcs are labeled with traversal costs.

From this the cost and estimator functions are developed:

- g = cost of all arcs traveled on the current path, (for example, $g(n2) = 7$).

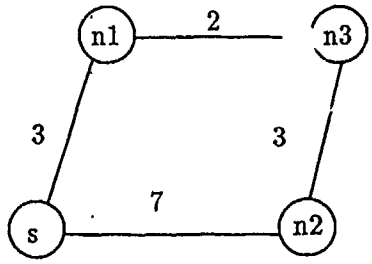


Figure C.1. Graph for A* Example

- Since information to estimate the distance to the goal is not available, the optimistic estimator function, $h = 0$ is used. This degenerates to a breadth-first search which ensures that the optimal solution is reached eventually; however, a tighter bound on h tends to expand fewer nodes in the search (26, 29).

If $n2$ is the goal node, the search proceeds as follows:

- The successors of s, $n1$ and $n2$ are generated.
- The estimates are 3 and 7, respectively.
- The lowest-cost estimate is selected, and the successors of $n1$, ($n2$ and $n3$) are generated.
- The estimates become

n1	n2	n3
3	6	5

- Since n2 is the goal state, the search is complete, with a cost of 6.

C.3 Evaluation Functions

Evaluation functions are the additive cost function g and the estimator function h, which are used to guide the A* search (29).

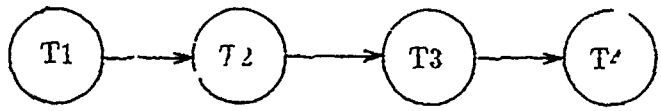
Within the context of the scheduling problem, the following heuristics can be used to guide the search so that low-cost schedules tend to be derived without the necessity of traversing all branches of the search tree. Although these heuristics do not guarantee that the first solution is an optimal solution, when combined with delayed termination, these heuristics will produce an optimal schedule.

- Greedy Heuristics for Single-Execution Problem
  - longest chain of processes (higher-level tasks scheduled first)
  - number of successors (greatest number scheduled first)
  - execution time (longest execution time first)
- Heuristics for Iterative Problem

  The iterative problem is NP-complete when execution times of 1 and 2 are allowed. Since the goal of the iterative problem is to *minimize latency* rather than to measure overall execution time, optimal solutions tend to occur when all processors have equivalent task loads (Chapter 4). Therefore, the following are suggested as heuristics to drive the search in the iterative problem:
  - longest execution time first (Greedy method)
  - keep processor loads balanced as tasks are assigned. For example, if a 1-time-unit task and a 2-time-unit task are available at $time_k$, and if P_1 has 3 time-slots used, while P_2 is only loaded in 1 time-slot, then it would be reasonable to place the 1-time-unit task on P_1 and the 2-time-unit task on P_2.

Appendix D. *Iteration-Number Decision Strategy*

To ensure that there is minimal delay time on the most-heavily-loaded processor, an appropriate decision strategy must be used. Otherwise, if the "wrong" task is chosen in a conflict between tasks, idle time may result, increasing latency. For example, Figure D.1. shows the results of an arbitrary decision strategy when 4 iterations of a task system are mapped to 3 processors. In this case. Processor 1 is forced to be idle at $time_7$ because $task_1$ has completed all iterations, and all remaining iterations of $task_4$ are waiting for tasks on other processors to complete.



P1	$T1^0$	$T1^1$	$T1^2$	$T4^0$	$T4^1$	$T4^2$	$T1^3$			$T4^3$
P2		$T2^0$	$T2^1$	$T2^2$				$T2^3$		
P3			$T3^0$	$T3^1$	$T3^2$				$T3^3$	

Figure D.1. Delay Time Due to Arbitrary Decision Strategy

To avoid this problem, a decision strategy must guarantee that a choice between tasks for a particular time slot always chooses the task most likely to cause delay. For example, the schedule for the graph in Figure D.1 can be scheduled so that no idle time occurs on Processor 1, as shown in Figure D.2

The decision strategy which yields the above schedule is based on the longest time a task might have to wait for predecessor tasks to execute: In the previous example, the worst case occurs for $task_3$ and $task_4$. Each of these tasks is bound by predecessors which execute on other processors. Since there are *at most* $m-1$ processors which must iterate before any given task can execute, no task will be blocked from execution if $m-1$ iterations of predecessor task are stored in the buffer between tasks.

P1	$T1^0$	$T1^1$	$T1^2$	$T4^0$	$T1^3$	$T4^1$	$T4^2$	$T4^3$
P2		$T2^0$	$T2^1$	$T2^2$		$T2^3$		
P3			$T3^0$	$T3^1$	$T3^2$		$T3^3$	

Figure D.2. Optimal Schedule

The choice is based on the *iteration number* of each task. Each task is numbered with the task number and iteration number: $task_\alpha^\beta$ is used to represent the $(\beta+1)^{th}$ iteration of $task_\alpha$. For example, the first iteration of $task_1$ is represented as $task_1^0$; the second iteration of $task_3$ is represented as $task_3^1$; and so forth.

The scheduling decision is made by comparing iteration numbers. If the difference between iterations is less than the number of processors, then the highest-level task (task with the highest iteration number) should be scheduled since it may be a predecessor of $(m-1)$ tasks on other processors. If the difference between iterations is greater than or equal to the number of processors, then the lower-level task should be scheduled, since the higher-level task has stored up enough iterations to keep the system from generating idle time.

In general, if the choice is between $task_i^k$ and $task_j^l$, the following tests are done:

- If $|k \leq l|$ and $|k-l| \geq m$, then $task_i^k$ should be scheduled.
- If $|k \leq l|$ and $|k-l| < m$, then $task_j^l$ should be scheduled.

In the example above, the first decision point occurs at $time_3$. In order to choose which task should be selected for the schedule, $task_1^3$ and $task_4^0$ are evaluated:

$$|3-0| = 3 = m \Rightarrow task_4^0 \textit{ should be selected.}$$

At the next decision point, $time_4$, $task_1^3$ is selected instead of $task_4^1$ because:

$$|3 - 1| = 2 < m \Rightarrow task_1^3 \textit{ should be selected.}$$

This strategy is appropriate because it allows predecessor tasks to build up results in buffers so that successor tasks never have to wait for predecessors after the first iteration.

Conjecture 1 *If the following conditions are met, then no processor has more than $m-1$ units of delay time*

- *n unit-execution-time tasks*
- *m homogeneous processors*
- *precedence relationship $\prec$ defined between tasks*
- *level-strategy assignment*
- *all-iterations-first decision strategy*
- *queued message passing (no bounds assumed on buffer size)*
- *task exists on only one processor*

PROOF SKETCH: (by construction)

Part I: Level Changes for First Task on each Processor

Assume that the above conditions are met and that the first m tasks each cause a delay of 1 time unit. It can be assumed, without loss of generality, that P_k is assigned 2 tasks, $task_i$ and $task_j$, where $i < j$.

- Let $time_l$ denote the time when all processors have a task.
- Let $time_u$ denote the time when the last iteration of $task_i$ occurs.

- Let $time_v$ denote the time when the last iteration of $task_j$ occurs. Since tasks are assigned by the level strategy, $time_v$ also represents the end of the simulation.

Idle time cannot occur on P_k between $time_t$ and $time_{t+m}$, since the iteration-difference strategy gives priority to $task_i$ in this time frame.

In $time_{t+m+1}$, the first iteration of $task_j^0$ occurs because $task_i$ has executed m times. Therefore $task_j^0$ is given priority by the decision strategy:
$|0 \leq m|$ and $|0 - m| = m$, therefore $task_j$ is scheduled.

At $time_{t+m+2}$, $|k - l| = |m - 1| < m$, and $task_i$ is scheduled.

This alternating pattern continues, until $time_u$, when $task_i$ completes its last iteration. At this point, there are exactly m iterations of $task_j$ which have not executed. From $time_u$ until $time_v$, $task_j$ is the only task remaining on P_k, so no scheduling decision is required. Furthermore, since all predecessors of $task_j$ are scheduled in the same manner as tasks on P_k, and since there are at most $m - 1$ iterations of predecessor tasks which have not executed, no delay is generated on the processor. Therefore, the last predecessor of $task_j$ executes by $time_{v-1}$, and $task_j$ completes its last iteration at $time_v$.

Part II: Level Changes After Start of Schedule (incomplete proof sketch)

The remainder of the proof must show that delay time during the schedule will not exceed $m - 1$ time units on any processor, even if one or more processors encounters its first level change during the pipelined portion of the schedule. Using the same logic as the all-iterations-first decision strategy (Lemma 2), it becomes necessary to show that *at most* $m - 1$ time units of delay can occur internal to the schedule.

□

approximation algorithms, 2-5
Assumptions
    iterative UET systems, 2-14

chordal complement graphs, 2-10
cyclic forest, 2-11

general scheduling problem, 2-1
    formal definition, 2-1

heuristics, 1-2, 3-14, 4-36

Latency, 1-8, 2-9, 4-1, 4-3, 4-5, 4-11–4-13, 4-22, 4-23, 4-26, 4-27, 4-30, 4-31, 6-2, 6-3

near-optimal solutions, 2-5
NP-complete, 1-1, 1-2, 1-4, 1-7, 1-9, 1-10, 2-4–2-6, 2-9, 2-11, 4-1, 4-35–4-42, 6-3, C-3
NP-complete problems, 1-7, 3-1, 6-4

optimal schedules, 1-1, 1-2, 1-4, 1-5, 1-7–1-9, 2-6, 2-9, 2-11, 2-12, 2-14, 3-3, 3-5, 3-8, 3-10–3-12, 3-14, 4-1, 4-6, A-1
optimal solutions, 3-14

pipelined mapping, 2-14
polynomial-time algorithms, 2-9, 3-1, 3-3
    2-processors, 2-9
    chordal complement, 2-10
    heterogeneous processors, 2-11
    UET forest, 2-10
polynomial-time solutions, 2-4
precedence constraints, 2-6
Precedence-constrained schedules, 4-1
precedence-constraints, 1-4, 1-8, 1-9, 2-1, 2-4, 2-9, 3-10

Theorem
    Latency for UET Systems, 4-23, 6-2
    Lower bound, 4-11, 6-2
    Number of processors, 4-34, 6-3
    Upper bound, 4-22, 6-2
    Variable execution time tasks, 4-29, 4-31, 6-3

unit execution time systems, 2-4, 2-15

Bibliography

1. Aho, A.V., J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Bell Telephone Laboratories. 1974.
2. Armstrong, James R. *Chip-Level Modeling with VHDL.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.
3. Barhen, Edith et al. *ROSES, A Robot Operating System Expert Scheduler: Methodological Framework.* ORNL/TM-9987. CESAR-86-09. Oak Ridge National Laboratory. August 1990.
4. Beard, R. Andrew. "Determinism of Algorithm Parallelism in NP-complete Problems for Distributed Architectures." Master's thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, December 1989.
5. Bertsekas, Dimitri P. and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, Inc. Englewood Cliffs, New Jersey. 1989.
6. Bokhari, Shahid H. *Assignment Problems in Parallel and Distributed Computing.* Kluwer Academic Publishers. Norwell, Massachusetts. 1987.
7. Brassard, Giles and Paul Bratley. *Algorithmics: Theory and Practice.* Prentice Hall, Englewood Cliffs, New Jersey, First edition, 1988.
8. Beard, R. Andrew and Gary B. Lamont. *Compendium of Parallel Programs.* Electrical Engineering Department, Air Force Institute of Technology. November 1989.
9. Casavant, T.L. and J.G. Kuhl. "Taxonomy of Scheduling in Distributed Computing Systems." *IEEE Transactions on Software Engineering.* Vol 14. No 2. February 1988.
10. Coffman, E.G., et al. *Computer and Job Shop Scheduling Theory.* New York: John Wiley & Sons, Inc., 1976.
11. *Corps Battle Analyzer (CORBAN) Operations Guide.* Volume IV. April 1986.
12. El-Rewini Hesham and T.G. Lewis. "Scheduling Parallel Program Tasks onto Arbitrary Target Machines." *Journal of Parallel and Distributed Computing,* 9, 138-153. June 1990.
13. Fox, G., M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors.* Englewood Cliffs, New Jersey: Prentice-Hall, 1988.
14. Garey M.R. and D.S. Johnson Miller Papadimitriou. "Complexity of Coloring Circular Arcs and Chords." *SIAM J. Alg. Disc. Meth.* Vol 1, No 2. June 1980.
15. Garey, M.R. and D.S. Johnson. *Computers and Intractability.* W.H. Freeman & Co., San Francisco, Ca. 1979.
16. Garfinkel, R.S. and W.J. Plotnicki. "Fixed-cycle Scheduling: A Solvable Problem with Empty Precedence Structure." *Operations Research.* Vol 38. No. 4. July-August 1990.
17. Garfinkel, R.S. and W.J. Plotnicki. "A Solvable Cyclic Scheduling Problem with Serial Precedence Structure." *Operations Research.* Vol 28. No. 5. September-October 1980.

18. Goyal, Deepak K. *Scheduling Processor Bound Systems,* Report No. CS-76-036, Computer Science Department, Washington State University, Pullman, Washington. November 1976.

19. Grimaldi, Ralph P. *Discrete and Combinatorial Mathematics.* Addison-Wesley. June 1989.

20. Harel, David. *Algorithmics, the Spirit of Computing.* Addison-Wesley Publishing Company. 1987.

21. Kruatrachue, Boontee. "Static Task Scheduling and Grain Packing in Parallel Processing Systems." PhD thesis, Oregon State University. June 1987.

22. Lamont, Gary B. *et. al. Compendium of Parallel Programs.* Electrical Engineering Department, Air Force Institute of Technology. December 1990.

23. Lee, Ann K. "An Empiracal Study of Combining Communicating Processes in a Parallel Discrete-Event Simulation." MS thesis, AFIT/GCS/ENG/90D-08. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

24. Lee, Edward Ashford and David G. Messerschmitt. "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing." *IEEE Transactions on Computers.* Vol C-36, No. 1, January 1987. pp. 24-35.

25. Lewis, T.G. *et al.* "Task Grapher: A Tool for Scheduling Parallel Program Tasks." Fifth Distributed Memory Computing Conference. Charleston, South Carolina. April 1990.

26. Nilsson, Nils J. *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill Publishing. 1971.

27. Orlin, James B. "Minimizing the Number jof Vehicles to Meet a Fixed Periodic Schedule: An Application of Periodic Posets. *Operations Research* Vol 30, No 4, July-August 1982.

28. Papadimitriou, C.H. and M. Yannakakis, *Scheduling Interval-Ordered Tasks,* Report No. TR-11-78, Center for Research in Computing Technology, Harvard University, Cambridge, MA. 1978.

29. Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison Wesley, Reading, Massachusetts, 1984.

30. Proicou, Michael C. "Distributed Kernel for Simulation of the VHSIC Hardware Description Language". MS thesis, AFIT/GCS/ENG/89D-14. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989.

31. Ragsdale, Susann, *et. al. Parallel Programming Primer.* Intel Corporation. March 1990.

32. Seward, Walter D. "Optimal Multiprocessor Schedulingj of Periodic Tasks in a Real-Time Environment." PhD Dissertation, AFIT/DS/EE/79-2. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 1979.

33. Ullman, J.D. "NP-Complete Scheduling Problems." *Journal of Computer and System Sciences 10*, 384-395, 1975.

34. Ullman, J.D. "Polynomial Complete Scheduling Problems." *ACM Operating Systems Review.* Vol 7, No 4. 1973.