[illegible]

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Task Assignment in a DSP Multiprocessor Environment

by

Marco Lavoie, B.Sc.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of

Master of Science

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario

August 3, 1990

©copyright

1990, Marco Lavoie

NATIONAL LIBRARY
of Canada

BIBLIOTHÈQUE NATIONALE
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

ISBN 0-315-60494-8

The undersigned recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis
Task Assignment in a DSP Multiprocessor Environment
submitted by
Marco Lavoie, B.Sc.
in partial fulfilment of the requirements for
the degree of Master of Science

Thesis Supervisor

Chair, Department of Systems and
Computer Engineering

Carleton University

September 14, 1990

Abstract

To exploit the parallelism in multiprocessor architectures, *Digital Signal Processing* (DSP) problems are partitioned into smaller interconnected *tasks*. The pattern of assignment of the tasks to the processors directly affects the length of time needed to transfer data between communicating tasks. This has a major impact on the performance of the resulting DSP systems. This thesis describes research towards the optimal assignment of tasks to processors, a difficult NP-Complete problem. The optimal assignment of tasks allows data streams to be processed with *minimum delay*. A provably optimal *task-to-processor assignment method* (TPAM) based on *depth-first branch-and-bound search* is described, as well as several near-optimal heuristics. All TPAMs are tested on randomly generated and actual DSP problems. While the optimal TPAM exhibits good performance for small DSP problems, a heuristic is found which performs better on larger problems than existing TPAMs.

Acknowledgements

I wish to thank the many people whose comments and assistance have been so helpful in this work. In particular my supervisor, John Chinneck, who taught me what little I know about “organized” research, and who was patient enough to go through this thesis over and over, always promptly. Thanks also to Gerald Karam and Rafik Goubran who provided me with much guidance during our numerous Friday meetings, and who also helped me in the writing of some sections of this thesis. A big thanks also to Scott Kerr for his programming skills.

I want to express my special thanks to the system wizard, Naren Mehta, who answered all my questions (mostly stupid ones) and was always willing to explain the Sun network’s mysteries. I must also thank the Department of National Defence of Canada (DND) and the Natural Sciences and Engineering Research Council of Canada (NSERC) for their joint funding.

Finalement, je remercie sincèrement Sylvie, qui m’a toujours soutenu et encouragé pendant ces deux dernières années. Sans elle ce travail m’aurait été beaucoup plus pénible à réaliser.

Contents

1	Introduction	1
1.1	A DSP Design Environment	1
1.2	Research Orientation	3
1.3	Thesis Outline	4
2	Basic Concepts in Digital Signal Processing	6
2.1	DSP Problems and Task Graphs	6
2.2	The Multiprocessor Architecture	7
2.3	Task Graphs and Total Delay	11
2.4	Constraints on the Assignment of Tasks to Processors	12
3	The State of the Art in TPAMs	14
3.1	Graph Theoretical Approaches	16
3.2	Zero-One Integer Programming Approaches	16
3.3	Heuristic Approaches	17
3.3.1	Task Clustering	18
3.3.2	List Scheduling	19
3.3.3	Ngo's Scheduling for Transputers	21
3.4	A Depth-First Branch-and-Bound Heuristic	23
3.5	The RDAD Project	24
3.5.1	A Genetic Algorithm Approach	24

3.5.2	A Logic Programming Approach	25
4	The Minimum Total Delay Problem	27
4.1	Algorithmic Complexity	27
4.2	Thesis Problem Statement	28
4.3	Handling Feedback Loops	30
5	An Optimal TPAM	32
5.1	The Principles of TPAMs for Minimum Total Delay	32
5.1.1	Branch-and-Bound Search	33
5.1.2	Depth-First Branch-and-Bound	34
5.1.3	The Bounding Function	35
5.2	The Optimal TPAM in Detail	35
5.2.1	Preprocessing	36
5.2.2	Depth-First Branch-and-Bound Search	38
5.2.3	Finding the Minimum Number of Processors	48
5.2.4	Summary of the TPAM	49
5.3	Example: Optimal Solution to RDAD06	51
6	Near-Optimal TPAMs	54
6.1	An Extended Elimination Rule	55
6.2	Automatic Clustering of Tasks	56
6.3	Another Task Ordering Procedure: CP/MISF	57
6.4	Best-First Branch-and-Bound Combined with Beam Search	59
7	Test Results	61
7.1	Test Problems	62
7.1.1	Problem Classification	63
7.1.2	Random Graph Generation	64
7.2	Experiments	65

7.2.1	Comparing Optimal TPAMs	66
7.2.2	DFBB _c [*] and DFBB _{heuristic} [*]	69
7.2.3	Performance of CP/LPPF and CP/MISF as TPAMs	71
7.2.4	BFBB/bs _c with Various Queue Lengths	71
7.3	Actual DSP Applications	74
7.4	Worst Case Performances	77
7.5	Discussion of Results	80
8	Conclusions and Future Research	82
8.1	Summary	82
8.2	Conclusions	83
8.3	Future Research	84
8.4	Summary of Contributions	85
A	Task Graphs for DSP Applications	93
B	Program Listings	105

List of Tables

7.1	Average characteristics of the random task graphs generated.	65
7.2	Comparing ordering schemes for the DFBB (with $U = +\infty$).	67
7.3	Comparing delays from list scheduling heuristics.	68
7.4	Results using $DFBB_{\epsilon}^*$ with $\epsilon = 20\%$	69
7.5	Results using $DFBB_{\epsilon}^*$ with $\epsilon = 40\%$	70
7.6	Results using $DFBB_{\text{heuristic}}^*$	70
7.7	Assignments found with CP/MISF alone.	71
7.8	Assignments found with CP/LPPF alone.	72
7.9	BFBB/ bs_x with maximum queue length of $x = 100$	72
7.10	BFBB/ bs_x with maximum queue length of $x = 1000$	73
7.11	BFBB/ bs_x with maximum queue length of $x = 10000$	74
7.12	RDAD problems.	75
7.13	Optimal solutions to RDAD problems using $DFBB_{\text{optimal}}^*$	77
7.14	Solutions found to RDAD problems with near-optimal TPAMs. . . .	78
7.15	Performance of heuristics on delay.	79
7.16	Worst performances on CPU requirements.	79

List of Figures

2.1	Task graph for the RDAD06 problem.	8
2.2	Multiprocessor architecture.	9
2.3	Data transfer synchronization.	10
5.1	RDAD06 labeled using CP/LPPF.	37
5.2	Initial solution to RDAD06 using CP/LPPF.	38
5.3	Task-to-processor assignment tree.	39
5.4	Calculation of bounding function value for partial assignment.	41
5.5	Partial solution with intermediate unused processor.	43
5.6	Isomorphic assignments.	44
5.7	TPAM result file for RDAD06.	53
6.1	RDAD06 Labeled Using CP/MISF.	58
7.1	Examples from task graph classes.	63
B.1	Input file for the RDAD06 problem.	106

Chapter 1

Introduction

The recent generations of powerful and inexpensive Digital Signal Processing (DSP) chips have generated a great deal of interest in multiprocessor solutions to DSP problems. Unfortunately, numerous processors interconnected in sophisticated architectures introduces an element of complexity which makes it difficult to achieve an optimal (or even acceptable) multiprocessor solution in the general case. For this reason, there is great interest in computer assistance in the task of DSP multiprocessor system design.

This thesis reports research on a first step towards the development of a functional environment for the optimal design of solutions to complex DSP problems. The core of such an environment resides in the mapping of the various modules which compose a given DSP problem onto the processors of the target architecture. This thesis thus focuses on optimal and near-optimal *task-to-processor assignment methods* (TPAMs).

1.1 A DSP Design Environment

A comprehensive environment to assist in the design of solutions to DSP problems should allow the graphical specification of the DSP problem via a task diagram

and the non-real-time simulation of the specification, should automatically perform an optimal mapping of the tasks to the processors in the multiprocessor target architecture, should generate code based on task libraries, and should download the code to the appropriate processors. The three key components of such an overall environment are thus

1. a graphical DSP problem specification interface,
2. an extensible general purpose DSP target architecture,
3. a task-to-processor assignment method (TPAM).

The graphical interface permits the specification of a DSP application by placing icons representing DSP functions in a graphics modelling area and declaring the parameters (e.g. name, processing time, required memory, etc.) associated with each icon. The flow of signals through the diagram is then specified simply by joining icons with directed arcs. Commercially available graphical tools include BOSS [5], DADiSP [13], DSPlay [15], Gabriel [40], and SPW [50]. In the more complete tools, once the task graph is completed, a non-real-time simulation of the system can be requested. Data at various points in the graph can be plotted in the frequency or the time domains. Statistical properties of the signal at a given point, such as histograms, can also be plotted.

In order to accommodate various types of DSP applications, the multiprocessor architecture to which the applications must eventually be downloaded must be extensible (i.e. processors can be added as required) so that no physical burden can hamper the performance requirements. It must also have a bounded communication time for interprocessor data transfers, so that minimum input-to-output processing delay can be an attainable objective.

The next step in the design process is implementation on the target architecture. Coupled with the architecture, the automatic TPAM must address two issues: (1) the number of processors and (2) a reasonable assignment of tasks to processors so as to meet some design objective. Specifically, the assignment is based on the time and space requirements of each task as well as the time and space resources

of the various processors in the multiprocessor system. The goal of the assignment process could be to minimize the total delay introduced by the system, given a fixed number of processors, or just to minimize the number of processors. The first case is suitable for applications requiring fast signal processing, whereas the latter case concerns implementation costs. Both depend on the final use of the DSP system according to the design specifications.

1.2 Research Orientation

The task-to-processor assignment problem considered here is different from similar problems previously addressed in the literature. DSP problems deal with the transformation of input signals to output signals. Once an optimal (or near-optimal) assignment is determined for a given DSP problem, the application is downloaded only once in the lifetime of the system. Once installed, a *DSP system* is fed with an infinite stream of data (the input signal), which is processed continuously, producing an uninterrupted output stream. Each task in the system is thus processed continuously, on a non-stop basis. DSP systems are usually used on a continuous basis and are not modified unless to be reconfigured.

On the other hand, the task-to-processor assignment problem previously considered in the literature deals with the "one-time run" of a large computational task (divided into *modules*) on a set of interconnected identical processors. Unlike DSP systems, once a module is processed, it can be retrieved from the processor (all resources released) and another module may then be assigned to the processor. This problem is identical to the assembly-line balancing problem so well-known in operations research [32, 38, 39]. In the assembly-line balancing problem, the total delay of a complete assignment (the time required to process all modules) is determined by the order in which modules are assigned to newly available processors.

Unlike the assembly-line balancing problem, the task-to-processor assignment problem considered here deals with permanently assigning each task to a specific processor. A task can never be swapped from its processor since it is processed continuously, without interruption. Furthermore DSP systems are not only affected

by the order in which tasks are processed, but also by the position of tasks in the architecture, since intertask communication times now depend on the architecture topology.

This thesis presents functioning implementations of task-to-processor assignment methods (TPAMs) which take into consideration the interprocessor communication times. The TPAM is the core of any comprehensive DSP design environment, and directly determines its performance. Both optimal and near-optimal assignments of the DSP tasks have been considered. The optimal assignment of tasks to processors is an extremely difficult combinatorial problem. There are two possible criteria for optimality: *minimum total delay*, or *minimum number of processors*. Minimum total delay is attained when all tasks can be processed in minimal time. This thesis concentrates on the minimum total delay problem for a number of reasons: the version of the problem including communication delays has not been previously addressed in the literature, it is of great practical interest, and reasonable heuristics exist for the solution of the second problem.

The optimal TPAM reported here uses a depth-first branch-and-bound algorithm coupled to a task ordering heuristic. Solutions achieved are provably optimal. Various strategies have been explored to speed the solution. Near-optimal heuristics have also been derived and compared to existing heuristics proposed in the literature.

1.3 Thesis Outline

The body of the thesis presents the main points of the research. Chapter 2 is a primer on DSP problems and their associated task graph representation, and also describes the multiprocessor architecture considered. Chapter 3 reviews the state of the art in algorithms which have been suggested previously for the task-to-processor assignment problem. Chapter 4 gives a formal statement of the problem considered here and discusses side issues such as complexity. Chapter 5 gives a thorough description of the research which led to the optimal TPAM presented herein. Faster near-optimal strategies considered (both derived from the optimal TPAM and adapted from the literature) are described in chapter 6. Chapter 7

summarizes test results and compares all the TPAMs herein, and finally chapter 8 gives conclusions and future directions.

Appendix A lists the problems derived from actual DSP applications on which the TPAMs are tested. Complete listings of the procedures for the optimal TPAM are presented in Appendix B.

Chapter 2

Basic Concepts in Digital Signal Processing

The first step in designing sophisticated DSP applications is the construction of the *task graph*. The task graph is a directed graph where each node or *block* represents a DSP function and arcs denote data flow paths between these functions. The resulting graph represents the desired algorithm or system.

2.1 DSP Problems and Task Graphs

The task set T is represented by the task graph $G = (N, A)$, which is a finite directed graph (section 4.3) where N stands for the set of n nodes representing DSP functions, and A stands for the set of arcs representing data flow paths among these functions. The graph is assumed to have exactly one *entry node* and one *exit node*; dummy tasks can be added so that this assumption is always valid. It is also assumed that the exit node is reachable from all nodes in the graph. Each node is characterized by four parameters, which correspond to the DSP function's unique name, and its resource requirements : *processing time*, *program memory* and *data memory*. If the task graphs are acyclic, they can be viewed as PERT diagrams [53]. Note however that DSP problems often contain *feedback loops*, as discussed in section 4.3.

The example task graph of figure 2.1 depicts a task diagram for test problem

RDAD06. Section 3.5 introduces RDAD (*Real-time DSP Applications Design*) problems and their relationship to this thesis. RDAD06 represents a baseband land mobile channel simulator [27], capable of simulating multipath propagation, fading, and Doppler effects. The input signal is split into in-phase and quadrature components using a Hilbert transform (task *a*). Appropriate delays are introduced to align the phases of the two components (tasks *b1* and *b2*) which are then filtered (tasks *c1* and *c2*). Each component is then modulated with a shaped white Gaussian noise signal which represents the multiplicative impairments of the channel. Tasks *k* and *n* are white Gaussian noise generators, tasks *l* and *m* are shaping filters, and tasks *d1* and *d2* are balanced modulators. The additive impairments of the channel which represent the line of sight propagation are then introduced (tasks *e1* and *e2*). The two components are then combined (task *f*), scaled (task *g*), and filtered (task *h*).

2.2 The Multiprocessor Architecture

The multiprocessor architecture considered here is described thoroughly in [7]. It has two major features which support the design method:

1. It is extensible, thus the number of processors can be varied to achieve the performance requirements.
2. It has a bounded communication time for interprocessor data transfers, thus minimum input-to-output processing delay can be a design objective for TPAMs. It is extremely difficult to create a useful TPAM where the communication delays are stochastic.

The three main aspects which characterize the architecture are (1) the *interconnection topology*, which specifies how processors are linked together, (2) the *data transfer strategy*, which determines how interprocessor communications are performed, and (3) the *transfer synchronization*, which specifies when interprocessor communications can occur.

Figure 2.1: Task graph for the RDAD06 problem.

Interconnection Topology and Data Transfer

The multiprocessor architecture, depicted in Figure 2.2, is loosely-coupled; that is, each processor has its own local memory and its own clock. The interprocessor communications are performed through shared memories. The topology of the architecture includes a master processor connected to each processor in a pipeline of p processors (a p -stage pipeline). This organization can be alternately viewed as: (1) a single bus architecture, augmented with a direct link connecting all adjacent processors, or (2) a one-dimensional systolic array, or ring architecture, augmented with a master processor having access to all other processors.

In this configuration, the master processor can communicate with any processor in the pipeline whereas a processor in the pipeline can only communicate with the

master, read from its predecessor, and write to its successor. In most applications, the master processor handles the I/O processing and performs the global functions required by the application, while the p processors in the pipeline perform the numerically intensive computations. The master processor can also be used as part of the pipeline and can therefore contribute to computations in addition to handling I/O and global functions.

Figure 2.2: Multiprocessor architecture.

The memory in each processor card in the pipeline is configured into four sections: one for local memory, two for memory shared with the master, and one for memory shared with its predecessor.

Consider processor P_i in the pipeline. The shared memory with the master processor (processor $\#0$) includes two sections: the first is read-only by P_i and write-only by the master — its purpose is the transfer of information from the master to P_i , and the second is write-only by P_i and read-only by the master — its purpose is the transfer of information from P_i to the master. The last memory section is shared between adjacent processors. This shared memory is read-only by P_i and write-only by P_{i-1} , where the successor of the master is P_1 and the successor of P_p is the master.

In this multiprocessor architecture, more communications flexibility is introduced by having a common bus and a master along with the one-directional ring pipeline. A transfer from non-master P_i to non-master P_j (where j is not the successor of i),

is performed as follows: P_i sends the data to the master, then the master sends it to P_j . Thus, data can be moved between any two processors in at most two transfers (it can be moved from a processor to its successor in one transfer).

Transfer Synchronization

Each processor in the system has its own clock. The processors are synchronized by a synchronization clock, corresponding to the sampling rate of the incoming signal (data) for the system.

Figure 2.3: Data transfer synchronization.

Data transfer in the system is synchronized to the common synchronization clock. Figure 2.3 shows the synchronization clock cycle and its decomposition into the read sub-cycle and write sub-cycle. During the read sub-cycle, each processor copies its input buffers into its local memory and processors are not allowed to write into their output buffers. During the write sub-cycle, which represents most of the clock cycle, the processors are allowed to modify their output buffers. This scheme eliminates the need for any handshaking by requiring all transfers to occur only at the clock edge (no transfers can be done between clock edges). Consequently, all the information to be transferred must be available at the clock edge; i.e., all tasks must be performed within the synchronization period.

2.3 Task Graphs and Total Delay

As mentioned in the previous section, all processors are synchronized by the sampling rate of the incoming data signal to the DSP system. Since data transfers among tasks can only be performed at clock cycle edges, communication times vary according to task assignments. Specifically, data moved from one task to another requires

no transfer — if both tasks are assigned to the same processor;

one transfer — if tasks are assigned to adjacent processors;

two transfers — if tasks are assigned to non-adjacent processors;

For each transfer, if the data is ready to be sent before the end of the current clock cycle, the receiving task must wait until the next clock cycle before the data can be transferred. Therefore each data transfer from one processor to another may extend the total time required to process all tasks. For example, if tasks *a*, *b1* and *c1* of RDAD06 (figure 2.1) are all assigned to a single processor, then they can all be processed within one clock cycle. On the other hand, if they are assigned to processors #0, #1 and #3 respectively (of figure 2.2), then only task *a* is processed in the first clock cycle, then task *b1* is processed at the next clock cycle, then finally task *c1* is processed two clock cycles later (an extra clock cycle is required for routing data from processor #1 to processor #3 through processor #0 or #2). Thus the latter assignment requires four clock cycles.

The *total delay* in a DSP system is the time difference between the acceptance of a signal at the system input port and the appearance of a good output signal at the system output port. In the simplest case (acyclic task graph), it can be compared to the “pipeline fill time”, i.e. the time required for all tasks to process the signal.

The critical path is a well-known feature of acyclic task graphs, such as PERT diagrams (see [53], for example). Briefly, where there are precedence relationships in the task graph, the critical path is the minimum time from the input to the output, which corresponds to the total delay. Because a task cannot begin until all of its predecessor tasks have finished, the critical path amounts to the longest duration

path from the input to the output in an acyclic task graph. Good algorithms exist for finding the critical path in any acyclic graph [53].

The problem of finding the minimum total delay is complicated by the fact that the intertask communication times are not known until the tasks have been assigned to the processors, hence the critical path length (and total delay) cannot be calculated before the assignment is complete. This fact is crucial to the design of the TPAM. In addition, DSP systems may have feedback loops, and so will not be acyclic. This complicates the calculation of the total delay, as discussed in section 4.3.

As the delay is strictly an input-to-output calculation, individual tasks or streams of tasks which are solely for the generation of internal signals (such as sine wave generators) are not included in the calculation of the delay. Such *generator streams* can be removed from the task graph before the TPAM is applied to find the minimum total delay. The assumption is that these tasks are switched on long before external inputs are introduced so that their ultimate outputs are available to the other tasks in the graph. Tasks in generator streams may then be packed onto processors after the minimum-delay assignment for the reduced problem has been found. Task diagram RDAD06 of figure 2.1 contains two generator streams, tasks $k-l$ and tasks $n-m$, which are removed before searching for the assignment giving the minimum total delay.

2.4 Constraints on the Assignment of Tasks to Processors

A feasible task-to-processor assignment must satisfy a set of constraints imposed by each of the identical processors in the architecture. The most obvious of these is the real-time requirement imposed by transfer synchronization, as described in section 2.2: the total processing time for all of the tasks assigned to a particular processor cannot exceed the duration of the synchronization clock cycle. The processing time of a task (which includes any data transfer time) is the projected

execution time of the task on the target architecture. As mentioned earlier in section 1.2, a DSP application is to be downloaded only once in the lifetime of the system. Once installed, the DSP system will be fed with a steady stream of data (the input signal), to be processed continuously, producing an uninterrupted output stream (the output signal). Each task in the system is thus processed *in every clock cycle*. Since the architecture is synchronized, all tasks assigned to a processor must thus all be processed within one clock cycle.

There are other constraints imposed by the processors as well, depending on their type. As mentioned in section 2.2, since tasks are assigned definitively to processors (i.e. never swapped afterward), one common set is a limit on the total program memory (in bytes) and a separate limit on the total data memory (in bytes). Again, the sum of the program memory requirements for all of the tasks assigned to a processor must not exceed the total program memory available for the processor. A similar condition applies to data memory. Note that according to figure 2.2, the data memory is actually split into 3 parts for communication purposes (see section 2.2 for details). However in this thesis we simplify the problem by unifying all 3 parts into a single resource (data memory). They could be considered separately by introducing new task parameters and processor resources.

For generality, the task requirements in terms of processing time, program memory and data memory are expressed in normalized form. That is, each requirement for a task is listed as the percentage of the processor limit. For example, a data memory requirement listed as 75 implies that the task requires 75 percent of the total data memory available on a single processor.

Chapter 3

The State of the Art in TPAMs

Efficient multiprocessor task assignment algorithms are especially important for real-time applications, where optimal or near-optimal solutions are required. A TPAM may be used to solve various problems:

- Given a partially ordered set of tasks represented by the nodes of a directed graph with their associated resource requirements, find an assignment in which all tasks are processed with *minimum total delay*.
- Find the *minimum number of processors* required to process all tasks (without considering the associated delay).
- Finally, combine the above problems, i.e. find the minimum number of processors required to achieve minimum delay.

While all three problems are equally important for a wide range of applications, solution strategies may differ considerably. This thesis concentrates on solving the first problem, that is to find the minimum total delay without concern for the number of processors used. The minimum number of processors problem is a variant of the well-studied bin-packing problem [31], so heuristics with acceptable bounds are readily available. Preliminary work has been conducted on the third problem and is also presented in this thesis.

The minimum total delay problem has been studied extensively and different approaches for solving it have been suggested [9]. They usually fall into one of

three categories: (1) *graph theory*, (2) *mathematical programming*, or (3) *heuristics*. Of course these approaches are not mutually exclusive, as each may use techniques from other areas. This classification does however distinguish the concepts in most approaches.

While the first two types of approach attempt to find an optimal assignment of tasks, heuristics are intentionally less complicated and usually return near-optimal solutions. By avoiding the computational complexity of optimum strategies, heuristic methods are faster, more extensible and simpler than graph theoretical and mathematical programming approaches, but cannot guarantee that the solutions found will be optimal.

This chapter surveys the literature on the task-to-processor assignment problem and outlines the ideas behind the different major techniques. Note that no TPAM was found which solves the DSP oriented task-to-processor assignment problem considered in this thesis. Lee and Messerschmitt [38, 39] show that when communication times are not considered, the task-to-processor assignment problem in DSP applications is equivalent to the assembly-line balancing problem, a standard problem in operations research. Most authors looked at problems similar to the assembly-line balancing problem, ignoring communication times. The only reported research which considers interprocessor communication times was conducted by Fernandez and Ngo [21, 46], but no test results were published. Section 3.3.3 gives details on their approach. While some more general TPAMs can be adapted to the problem herein, others are designed for specific problems and do not offer the required flexibility for adaptation.

The next three sections present the TPAMs proposed in the literature according to the classification introduced above: section 3.1 discusses the unsuccessful attempts using graph theoretical approaches, section 3.2 discusses the earlier trend to mathematical programming, and section 3.3 presents more recent work on TPAMs based on heuristics, including Ngo's heuristic. Then section 3.4 presents Kasahara and Narita's TPAM, an algorithm which combines some of the above heuristics, and is actually optimal (the optimal TPAM presented in this thesis is partly based on their work). Finally, section 3.5 presents preliminary research conducted in the

scope of the RDAD project (section 1.2), which encompasses this thesis work.

3.1 Graph Theoretical Approaches

There is an extensive literature on algorithms proposed for problems related to the minimum-delay task-to-processor assignment problem. The early attempts to solve this problem exploited graph theory and combinatorial algorithms. Most researchers in this field have used a *graph-based minimum-cut* procedure [48, 51, 52], in which a minimum-cut algorithm is used to partition the task graph into as many partitions as there are processors in the architecture. The minimum cut corresponds to the assignment with the least amount of interprocessor communications.

Unfortunately, such an approach can only be used efficiently for assignment among two to three processors, since the extension of a minimum cut to n processors calls for a *n-dimensional minimum-cut* procedure, which quickly becomes intractable. Furthermore, minimum-cut algorithms can only be applied to undirected graphs; precedence relationships among tasks cannot be considered. Preserving the precedence relationships in the task graphs is essential in DSP systems, so these graph theoretical methods are not sufficiently powerful tools for a TPAM.

Some authors have used graph theoretical techniques with limited success to transform task graphs into simpler problems by adding precedence relations among tasks [6, 20]. The main idea is the transformation of a graph into sets of independent nodes by the addition of arcs between tasks in the graph. This corresponds to a *clustering strategy* which does not prune optimal solutions. Fernandez and Lang [20] concluded that this approach did not lead to better scheduling algorithms, but only gave insight into means of formulating new heuristic methods for scheduling problems.

3.2 Zero-One Integer Programming Approaches

Zero-one integer programming has also been applied with relative success to a problem similar to minimum total delay assignment, the *file allocation problem* [8]. The

file allocation problem deals with the distribution of several files among a number of geographically separated computers, so that the allocation yields minimum overall operating costs. Once the problem is expressed as a zero-one integer problem, the formulation is solved using conventional mathematical programming techniques such as the *cutting planes* method or the *branch-and-bound* method.

A zero-one integer programming based TPAM provides much greater flexibility in modelling than graph theoretical approaches. Unlike the graph theoretical methods, processor resources (such as memory) can be modelled. Precedence relationships among tasks can also be modelled at the expense of adding new variables and a large number of constraints to the model, which quickly makes the problem intractable. Moreover, there is no provision for representing a specific multiprocessor architecture; only a fully connected architecture (implicit in the zero-one integer model) may be considered.

Like the graph theoretical methods, mathematical programming alone does not provide the required modelling tools for the task-to-processor assignment problem of interest. Used in conjunction with other techniques, zero-one integer programming may however lead to efficient TPAMs. Section 3.4 describes a depth-first branch-and-bound algorithm combined with a heuristic, which is able to find optimal solutions to assembly-line balancing problems.

3.3 Heuristic Approaches

More recent research work has focused on developing *heuristic* approaches which exploit simplifying assumptions about the task-to-processor assignment problem to rapidly narrow down the search space. Even though they cannot guarantee that solutions found will be optimal, many heuristics will very often find the optimum for small problems, and near-optimal solutions for large problems. Furthermore, efficient heuristics require only a fraction of the processing time required by other methods. A number of heuristics for *clustering tasks* to reduce the size of the search space have been adapted for TPAMs. Other efficient heuristics have been designed to exploit particular topologies found in task graphs, such as *Hu's heuristic* for tree-

like task graphs. *List scheduling heuristics* assign priorities to tasks (i.e. arrange tasks in an ordered list), and then assign these tasks to processors according to the priorities.

This section also presents an heuristic which is a combination of some of the above heuristics. Fernandez and Ngo's heuristic (section 3.3.3) takes into consideration interprocessor communication times by assigning tasks to *time slots*, clustering the time slots, and then assigning the clusters to processors. Although the authors claim the heuristic is optimal, no satisfactory proof is presented and it must therefore be classified as a near-optimal heuristic.

3.3.1 Task Clustering

The main idea shared by many early heuristics is task clustering [17, 21, 25, 22, 46, 28]. The goal is to form task clusters such that when assigned to different processors, interprocessor communications are reduced as much as possible. Task clustering heuristics thus attempt to minimize intercluster data transfers, as done by the minimum-cut algorithm presented in section 3.1. However heuristics use clustering strategies which avoid the algorithmic complexity encountered in graph theory.

Clustering heuristics usually proceed by iteratively pairing tasks (or already formed clusters) so as to reduce the number of communication links at each iteration [22]. Tasks or clusters to be paired are selected according to *pairing rules* based on criteria such as the number of adjacent tasks [46], cluster size, amount of data transfer between tasks [17], load balancing of processors, or multi-parameter distance functions between tasks [28]. The clustering process ends when no more tasks are "eligible" for pairing. Eligibility is usually specified by user defined *constraints*, such as maximum cluster size, the number of processors considered, or minimum concurrency requirements. Clustering heuristics differ on pairing rules and clustering constraints employed.

Two important limitations of clustering algorithms are evident. First, the number of resulting clusters is often more than the number of processors in the archi-

ture, so *reassignment heuristics* [17] must be introduced to divide and assign the remaining clusters among the processors. And second, once clusters are formed, there is still need for an assignment procedure to assign clusters to the processors. This is specially true when interprocessor communication times must be considered. For these reasons, clustering heuristics are better used in conjunction with other heuristic approaches, where they serve as a problem reduction scheme to shrink the task graphs into fewer tasks.

3.3.2 List Scheduling

List scheduling heuristics [1, 12, 34, 43, 47] are a class of scheduling heuristics for assembly-line balancing problems. They assign priorities to tasks according to particular ordering rules, and place the tasks in a list ordered in decreasing magnitude of priority values (ties being broken randomly). Given such an ordered list, the first task is assigned to the first idle processor, the next task to the next idle processor, and so on. Whenever a processor becomes idle, the next task in the list is assigned to it. This process continues until all tasks have been processed.

Adam, Chandy and Dickson describe the state of the art in list scheduling heuristics in [1]. Their paper classifies deterministic list scheduling heuristics into five categories:

HLFET (Highest Levels First with Estimated Times) are list schedules in which the priority of a task is its *level*, the level being the longest path from the exit task up to the task. The longest path is computed according to the processing times of the tasks.

HLFNET (Highest Levels First with No Estimated Times). In these list schedules all tasks are assumed (possibly incorrectly) to have the same execution time; the priority assigned to a task is its level computed under this assumption.

RANDOM list schedules are those in which tasks are assigned priorities randomly.

SCFET (Smallest Co-levels with Estimated Times). In these list schedules the priority of a task is the negative of its *co-level*. The co-level of a task is the

longest path (with processing times) from the entry task down to the task.

SCFNET (Smallest Co-levels with No Estimated Times), where all tasks are assumed (again possibly incorrectly) to have the same execution time.

Tests on randomly generated graphs and graphs extracted from real problems have shown HLFET schedules to be better than other schedules in most cases (in over 900 simulations there were 847 cases in which HLFET was the best).

One special case where HLFNET outdoes HLFET is pointed out by Hu [32]. He proposed a list scheduling heuristic which is optimal when reversed tree-like task graphs are considered. His simple HLFNET heuristic assumes all tasks to have an execution time of one unit. By labelling tasks from the exit task up to the entry task(s), the resulting priority list generates an optimal schedule. The schedules are provably optimal only if tasks are linked in an inverse tree topology, where leaves stand for entry tasks and the root stands for the unique exit task. Lee and Messerschmitt [38] recommended Hu's heuristic as a TPAM for DSP problems (without considering communication times however). Since Hu's heuristic also requires the multiprocessor architecture to be fully connected in order to achieve optimality, it cannot be exploited in general architecture topologies such as the one considered in this thesis (figure 2.2).

More recently, Kasahara and Narita [34] proposed a list scheduling heuristic based on interconnectivity of tasks, CP/MISF (Critical Path/Most Immediate Successors First). Similar to HLFET heuristics, CP/MISF however differs on the longest path computation. Instead of using the processing time of tasks to establish priorities among tasks, the number of immediate successors is used. Therefore, among tasks at the same level in the graph, CP/MISF assigns higher priority values to tasks which have more immediate successors. The performance of CP/MISF is comparable to HLFET heuristics. However, the real strength of the heuristic is exploited by Kasahara and Narita [34] in DF/IHS, an optimal procedure which uses the priority list to reduce the search space. See section 3.4 for more details on DF/IHS, and see section 6.3 for a performance analysis of CP/MISF.

Although list scheduling heuristics are only used to generate priority lists and do not take into consideration the multiprocessor architectures on which tasks are to be assigned, they can still generate good assignments for any architecture. Section 5.2.1 describes a simple procedure to transform an ordered list of tasks into a complete assignment for any architecture. Furthermore, section 5.2.1 also presents a new list scheduling heuristic which is better than CP/MISF and is used in the optimal TPAM described in this thesis.

3.3.3 Ngo's Scheduling for Transputers

Ngo and Fernandez recently proposed a TPAM which takes into consideration interprocessor communication times [21, 46]. In fact, their problem is substantially similar to the one considered herein. Their goal is to map a set of communicating Occam-based modules [44] onto a set of interconnected *transputers* [3] in order to achieve minimum total delay. The Occam language is based on the concept of communicating sequential processes and the transputer is the physical realization of the computational model defined by the language. The transputer contains a processor, some memory, and a number of communication links which allow direct connection to other transputers. A system is constructed from a collection of transputers which operate concurrently and communicate through links. The proposed TPAM can consider any transputer network topology. The only major difference between their problem and the one considered here is in the synchronization. In contrast to Fernandez and Ngo's problem, the one here deals with a clocked multiprocessor architecture where interprocessor communications can only occur at clock cycle edges (see section 2.2). Thus apart from delays due to interprocessor communications, additional delays are introduced when tasks terminates before the end of a clock cycle (i.e. such a task must wait until the current cycle ends for its output signal to be forwarded to tasks assigned to other processors). In Fernandez and Ngo's transputer network, once a task terminates, its output signal can be forwarded immediately to the next tasks on different processors.

Ngo's algorithm is divided into the following steps:

1. Establish the earliest and latest total delays.
2. Compute lower bounds on total delay and number of processors.
3. Determine the desired processor and time constraints.
4. Assign the tasks to time slots according to the constraints.
5. Group tasks into clusters.
6. Map each cluster of tasks to a processor in the architecture.
7. Assign communication channels between communicating tasks on different processors.

The approach is similar to a clustering heuristic where tasks are clustered so that the number of intertask communication links within each cluster is maximized. A major difference between this heuristic and others is that the tasks are assigned to time slots and to processors separately in different steps rather than being assigned directly to processors. Interprocessor communication times can thus be considered. Their heuristic is implemented in Prolog, which implies an implicit depth-first search (Prolog does it automatically). The search procedure is coupled to a lower bound estimation procedure [19], which is used to restrict the search for assigning tasks to time slots. The mapping of task clusters to processors is carried out so that clusters with the highest number of intercluster communication links are assigned to processors with the highest number of interprocessor links.

The authors claim this heuristic reaches optimality, but their proof is not convincing. Since the approach is based on a depth-first search, it is subject to a combinatorial explosion which prevents its use for solving larger DSP problems. Furthermore, the heuristic has not been tested thoroughly. No simulation runs were performed except on a few problems, and no results have been forwarded to compare the heuristic's performance to other heuristics.

Nevertheless, Fernandez and Ngo's approach has a positive advantage: the basic idea of splitting the assignment into time groupings of tasks and then assigning

clusters to processors is a concept that could perhaps be used to improve many proposed TPAMs.

3.4 A Depth-First Branch-and-Bound Heuristic

Kasahara and Narita [34] describe an interesting combination of list scheduling and depth-first branch-and-bound. Called *Depth-First/Implicit Heuristic Search* (DF/IHS), the algorithm is able to find optimal solutions to small assembly-line balancing problems (it is still subject to combinatorial explosion). The main advantage of DF/IHS lies in the fact that, unlike conventional depth-first branch-and-bound, it does not require the computation of the values of the lower bound function for all active nodes with the largest depth in order to find, as the next branching node, the node having the smallest lower bound. Prior to the search procedure, priorities are assigned to those nodes that may be generated during search by using the priority list of the CP/MISF method (see section 3.3.2). In this manner, the memory requirements and the computing time required for search can be reduced markedly, since the choice of the proper next branching node can be made without computing the heuristic values. Moreover, CP/MISF always finds a reasonable solution quickly so that a complete assignment is always available even if the branch-and-bound process is interrupted before completion.

The optimal method presented in this thesis is similar in many aspects to Kasahara and Narita's proposal: it also combines a list scheduling heuristic with branch-and-bound. The fundamental difference between the method herein and theirs is in the list heuristic used and in the generation of partial solution nodes in the search tree. Their method uses a list scheduling heuristic based on the interconnectivity of tasks. The list scheduling heuristic adopted here considers pipelined groups of tasks according to their length in processing time. Also, when generating child nodes during the search, Kasahara and Narita examine all possible assignments of ready tasks to available processors (a task is ready if it's unassigned but all its predecessors are assigned); we look at the possible assignments of only the next task in the ordered list. This restriction reduces the memory requirements considerably.

3.5 The RDAD Project

This thesis is part of a broader project titled in general terms *Real-time DSP Applications Design* (RDAD). The ultimate goal of the RDAD project is the development of a functional environment for the optimal design of solutions to complex DSP problems. The complete design environment will be useful for producing the final implementation of many DSP applications. It will also be used to develop a real-time prototype for systems to be implemented using application specific VLSI chips. For example, the environment is to be used eventually as part of a fast-prototyping test-bed for Integrated Services Digital Network (ISDN) applications, Background Acoustic Noise Cancellers (BANC), and high-speed modems. It will allow algorithms to be tested in real-time, trimmed, and evaluated with respect to each other in field tests before the final costly VLSI implementation.

In the early stages of the RDAD project, some original approaches to the task-to-processor assignment problem (with communication times) were investigated by members of the research team. Resnick [49] implemented a TPAM based on *Genetic Algorithms*. The next section describes her work. Several *Logic Programming* approaches were also considered and TPAMs implemented in Prolog tested. Section 3.5.2 describes the different approaches considered and results obtained.

3.5.1 A Genetic Algorithm Approach

Genetic algorithms (GA) have been developed by Dr. John Holland at the University of Michigan [30]. They are a set of heuristics exploiting the principles of evolution and natural selection to navigate through a search space. Similar to algorithms such as *simulated annealing* [36], GAs emulate a process occurring in nature in order to converge toward optimal solutions. An analogy is made between solutions and *chromosomes*. Just as chromosomes undergo genetic operations which gradually improve them over time, so too can populations of solutions and the operations on them be modelled.

The idea is to randomly sample the solution space and *evolve* towards optimal regions. To do so, three main genetic operators are simulated, *reproduction*, *crossover*

and *mutation*. Reproduction models the “survival of the fittest” in the population, while crossover generates new solutions by combining the best existing solutions in various ways. These two operators guide the evolving population towards optimal solutions, while mutation increases the span of the search, allowing the search to tunnel out of local optima.

Preliminary work on a GA-based TPAM has been conducted by Resnick [49]. She proposed a straightforward string representation of the task-to-processor assignment problem: an assignment is represented by a one-dimensional array where indices represent task labels and array elements represent processor numbers to which the corresponding tasks are assigned. Her Pascal implementation of the GA-based TPAM was successful in solving small problems, and proved GAs to be suitable for larger ones also. Her TPAM was also implemented in C for comparing its performance to the TPAMs herein, but no thorough testing was done due to time restrictions. Preliminary results did corroborate Resnick’s conclusions: solutions are often optimal for small problems and close to the optimum for larger ones; but tests also revealed that very long computation times are most often required in order to obtain good solutions.

While initial experiments proved GA-based TPAMs worthwhile, further testing needs to be performed before well-founded conclusions can be drawn and an acceptable TPAM produced. Several combinations of crossover and mutation probabilities need to be investigated, and other advanced reproduction operators and techniques to be tested, such as *dominance*, *diploidy* and *abeyance* [24].

3.5.2 A Logic Programming Approach

The task-to-processor assignment problem considered herein was also attacked using *logic programming*, and in particular Prolog. Dr. Gerald M. Karam (Carleton University) made several attempts at solving this problem, some of which are summarized below. The main attraction of a logic programming approach is the possibility of obtaining a symbolic solution, which eliminates the problem of multiple functional isomorphs of essentially the same assignment which are produced by searching the

tree of possible solutions (further details on isomorphic assignments in section 5.2.2). The general idea is to map the optimum symbolic solution onto an actual solution.

Basic Best-First Branch-and-Bound

In this approach, a *Best-First Branch-and-Bound* (BFBB) TPAM was implemented which assigned tasks to processors for a fixed minimum number of processors (as calculated by a bin-packing algorithm) so as to optimize the total delay. Critical path calculations were used to find the total delay for a partial or complete assignment. The Prolog implementation was simple to construct and understand (because of the declarative nature of the language), but it exhibited poor time and memory performance, due to memory swapping and queuing of partial solutions. Section 6.4 gives more details on BFBB.

Symbolic Equation Solving

The first approach attempted to characterize contributions to the total delay of a tasks assignment in a symbolic equation. For example, if task t_1 is assigned to some processor P_i then the delay resulting from this assignment can be computed in terms of the placement of other tasks in processors positioned *relative* to P_i . These equations are then used to solve the task assignment problem. Unfortunately, generating the information used in this approach is exponential in complexity, precluding its use in practice.

Constraint Solving Using CHIP

Dr. Pascal Van Hentenryck has attempted to formulate a solution to the task-to-processor assignment problem herein using the CHIP (Constraint Handling in Prolog) language [14, 29]. CHIP has symbolic and numeric (integer) constraint solving methods in addition to the basic Prolog capability, notably a built-in branch-and-bound facility. It produces lucid solutions to optimization problems such as task-to-processor assignment. While some preliminary results for test problems demonstrate that it can produce minimum total delay assignments in acceptable computation times, some tests also displayed much worse performance in other cases.

Chapter 4

The Minimum Total Delay Problem

Most TPAMs found in the literature deal with task-to-processor assignment problems similar to the assembly-line balancing problem (chapter 3), in which inter-processor communication times are ignored. The minimum total delay problem considered here is thus more complicated by the fact that the total delay of a given assignment is not only a function of the order in which tasks are assigned to available processors, but is also a function of the available processors to which they are assigned. For example, the communication delay between two tasks assigned to adjacent processors is less than if they are assigned to non-adjacent processors, because data transfers must be routed through a third processor.

Section 4.1 examines the complexity of the minimum total delay problem herein. Section 4.2 formally states the problem considered here. Finally, section 4.3 looks at some pathological cases which may add complexity to DSP problems, and ways they can be overcome.

4.1 Algorithmic Complexity

The problem of finding the best assignment of tasks to processors is combinatorially explosive. Assuming that each task could potentially be assigned to any processor,

then given t tasks and p processors, there are p^t possible assignments of tasks to processors. Not all of these assignments are feasible, but the solution must be generated before feasibility can be tested. The solution space to be searched is extremely large. Where the number of processors is not known initially, then the worst case assumption is that the number of processors equals the number of tasks, so the solution space has size t^t .

The solution to the minimum delay problem depends critically on the inter-task communication times, which are themselves a product of the task-to-processor assignments. Minimizing the total delay differs fundamentally from the task-to-processor assignment problem discussed by Lee and Messerschmitt [38, 39] and Lee et al. [40]. While the underlying synchronous data flow approach is identical, Lee and Messerschmitt ignore the interprocessor communication times, so their problem is identical to the assembly-line balancing problem [32, 38, 39].

As Coffman [10], Lee and Messerschmitt [39], and Lenstra and Rinnooy Kan [41] point out, the assembly-line balancing problem is NP-complete. Since the minimum total delay problem has the added complexity of intertask communication times that depend on the task-to-processor assignment, it is even more difficult.

4.2 Thesis Problem Statement

The goal of this thesis is to solve the minimum total delay task-to-processor assignment problem stated as follows:

Given any DSP problem represented by a task graph (as described in section 2.1), and the fixed processor architecture described in section 2.2 (with an arbitrary number of processors),

- 1. derive an algorithm which is able to find the task-to-processor assignment having the minimum total delay in all cases;*
- 2. derive an alternate heuristic which will find near-optimal assignments within a fraction of the time required by the optimal algorithm.*

All proposed methods are to be based on the following assumptions: (1) the DSP applications are deterministic with respect to the data stream (there are no data dependent tasks); (2) the task-to-processor assignment can be done once in the lifetime of the system for fixed systems or once for each configuration for reconfigurable systems; (3) the DSP applications do not contain feedback loops; (4) the system is not subject to bus contention; and (5) no communication time is involved between tasks on the same processor.

The near-optimal heuristic is to be used when the optimal TPAM will not be able to reach the optimum after a “reasonable” amount of processing time. This is usually the case when relatively large problems are considered.

The five stated assumptions aim at focusing research on basic TPAMs for DSP problems. The first assumption is common to most of the literature on this subject, and its purpose is to concentrate work on what is commonly called *deterministic scheduling*. TPAMs in general can be categorized as *deterministic* or *stochastic* [25]. In a deterministic TPAM, process execution times and precedence ordering are fixed, and tasks are assigned in a static manner, whereas in a stochastic TPAM, execution time is treated as a random variable with a given distribution and the assignment is done dynamically through queues [2]. It can be argued that the stochastic approach is more appropriate in the general case, however in a DSP system where the tasks are loaded once and remain in the architecture for some time, running continuously, a deterministic approach is more appropriate.

The second assumption stated above specifically restricts the application of the proposed TPAMs to such DSP systems. The third assumption is only included in order to simplify TPAM implementations, since feedback loops can be handled easily, as described in the next section. Most published work on the task-to-processor assignment problem ignores the difficulties presented by feedback loops. A method is presented next for handling loops in such a way that the static assignment of tasks is still possible, whatever the number of iterations required by the design specifications.

The minimum delay assignment problem is not deterministically tractable if factors such as bus contention cause random variations in the interprocessor communication times. The fourth assumption limits the problem to the case of bounded communication times; fortunately many practical DSP applications fall into this class. Finally, the last assumption states that no communication delay is incurred between tasks assigned to a single processor. While this assumption may not hold technically (memory transfers within a processor do require some processing), it may still be valid if a small proportion of the processing time available in each processor is retained for “in-processor” communications. A *dummy task* assigned to each processor could retain the required processing time for such communications. Ignoring in-processor communication times does not invalidate the proposed TPAMs.

As mentioned previously, this problem is different from most task-to-processor assignment problems considered in previous literature, because of the interprocessor communication times involved. All previous papers based on graph theoretical and mathematical programming approaches dealt exclusively with problems similar to the assembly-line balancing problem. Previous work on task clustering and list scheduling heuristics also ignored interprocessor communication times; so did Kasahara and Narita in their depth-first branch-and-bound approach. The only authors who did consider interprocessor communication times are Fernandez and Ngo (section 3.3.3), but they did not provide performance estimations for their proposed heuristic approach, and Karam’s experiments (section 3.5.2) tend to indicate that such a logic programming approach may not perform very well in many cases. Furthermore, since Fernandez and Ngo’s heuristic is still subject to combinatorial explosion (as the authors explicitly admit), there is still a need to develop good heuristic approaches to solve larger problems, which is done in this thesis.

4.3 Handling Feedback Loops

When a DSP problem contains feedback loops, the corresponding task graph representation is not acyclic. In this case, the meaning of total delay is unclear: What marks the first appearance of a good output signal? How many iterations of the

feedback loops are needed before the output signal is good enough? How do we interpret nested loops? Most previously published work on task-to-processor assignment problems either explicitly or implicitly ignored feedback loops by assuming that an entire loop could be assigned to a single processor. While this assumption may be acceptable for assembly-line balancing problems, it is invalid when processor resources (such as processing time) are limited. Furthermore, shrinking a large loop into a single task may reduce concurrency, and hence lengthen the minimum total delay.

Cyclic task graphs can be handled as follows. First determine the total delay in the usual way by stipulating that data flow can traverse each arc in the task graph once at most; this is designated as *CP1*. Next determine the total delay under the condition that each arc can be traversed at most twice, and designate this as *CP2*. The difference between the durations of *CP2* and *CP1* gives the duration of all feedback loops in the graph. The designer is then free to specify the number of iterations of the feedback loops that are necessary to achieve a good output signal, and the total delay is then calculable. This method is effective for multiple and nested loops.

Since handling feedback loops can be reduced to solving two task-to-processor assignment problems, we assume that all task graphs to be processed are acyclic from now on.

Chapter 5

An Optimal TPAM

The task-to-processor assignment method (TPAM) implemented in the prototype software is described in this chapter. The primary objective of the TPAM is to find the assignment which gives the minimum total delay of the signal, and secondary emphasis has been placed on minimizing the number of processors used. Section 5.1 describes the general principles of TPAMs, and section 5.2 gives the details of the particular implementation used in the software. Section 5.3 gives an example of the use of the TPAM.

5.1 The Principles of TPAMs for Minimum Total Delay

The branch-and-bound technique is appropriate for finding optimal solutions to problems which have a large tree-like solution space. Branch-and-bound reduces the size of the search space by pruning developing search branches without eliminating the optimal solution. Since the minimum delay task-to-processor assignment problem has this combinatorial tree-like growth of the search space, branch-and-bound is a good search strategy to use. Dynamic programming is ruled out because it becomes quickly intractable as the state vector grows exponentially (Winston [53] points out that branch-and-bound performs much better in such cases). Finally, a linear programming solution (especially in a network assignment version) is not

possible because the time cost of assigning a particular task to a particular processor cannot be specified beforehand since the associated intertask communication time depends on where the other tasks are assigned.

5.1.1 Branch-and-Bound Search

The problem of assigning tasks to processors is best viewed as a sequence of assignments of a single task. Since each task can be assigned to any processor, the sequence of assignments develops in the form of a tree, as shown in Figure 5.3. This assignment tree represents a small section of the search tree to find a minimum delay assignment for RDAD06 (figure 2.1). Each node represents the assignment of one task to one processor, and every level (lv) of the tree represents the assignment of one particular task to all available processors. As calculated above, there are either p^l or t^l leaves to the tree.

The branch-and-bound technique can be applied to combinatorial problems which exhibit this tree-like growth of the solution space. The technique limits the explosive growth of the tree by pruning the less-promising branches of the tree at an early stage, thereby reducing the number of developing branches which remain to be searched.

Branch-and-bound works by optimistically estimating the best value of the objective function (minimum delay in this case) attainable by continuing the growth of a partial solution in the decision tree — this is the bound on the partial solution. The partial solution with the best bound (lowest estimated total delay in this case) is selected for further growth. Infeasible partial solutions are pruned as they are discovered.

Whenever a complete feasible solution is found (a leaf of the decision tree is reached) any partial solutions with equal or worse bounds are pruned. Partial solutions with better bounds continue to be followed up. Additional complete feasible solutions with better values may be generated. The process halts when the value of a feasible solution is better than or equal to the bound on any partial solution. This complete feasible solution must be the optimum since no partial solution can

be developed into a better complete solution, as verified by the bound values.

General information on the branch-and-bound technique can be found in any standard text on operations research (such as [53]).

5.1.2 Depth-First Branch-and-Bound

The standard branch-and-bound algorithm, as described in the previous section, continues the growth of the partial solution which has the best bound of any partial solution in the entire decision tree. As even a well-pruned decision tree can still be very large, such a *global best-first* branch-and-bound implementation can consume huge amounts of computer memory. To avoid this problem, a *depth-first* version of the branch-and-bound algorithm can be used.

In depth-first branch-and-bound, the partial solution chosen for further growth is selected from the set of partial solutions just created; i.e., partial solutions elsewhere on the decision tree are temporarily ignored. This technique rapidly generates a complete feasible solution. The method then backtracks through the tree created so far, developing any partial solutions which have lower bounds than the best complete feasible solution attained so far. These are again treated in a depth-first manner. The algorithm backtracks every time it reaches a dead end, such as no feasible partial solutions, and terminates as usual when the lowest delay of a complete feasible solution is less than or equal to the lowest bound for a partial solution.

There are significant advantages to the depth-first scheme. First, the reduced memory requirements make implementation of the algorithm practical. Second, the early achievement of a complete feasible solution helps in pruning the search tree effectively. Finally, if the algorithm must be halted before completion, the depth-first search has a better chance of terminating with at least one feasible solution in hand.

5.1.3 The Bounding Function

A good bounding function is essential to the success of the branch-and-bound technique. In this case, the bounding function must provide an optimistic estimate of the minimum delay attainable by completing a partial solution. It must be optimistic to avoid pruning a partial solution that could grow to be the optimum, but it should be as accurate as possible to give the greatest reduction in the search space. The bound should also become progressively more accurate as partial solutions are gradually completed.

As discussed in Section 2.3, the total delay incurred by an assignment of tasks to processors is directly related to the critical path through an acyclic task diagram which includes the intertask communication times. The critical path is partly calculable in a partial assignment because some of the interprocessor communication times are known, but not completely calculable because the communication times are not known for the unassigned tasks. It is the bounding function's job to optimistically estimate the total critical path length (and hence delay) attainable by a partial solution when the currently unassigned tasks are finally assigned.

A bounding function provides an optimistic estimate through simplifying the problem by relaxing one or more of the constraints on task assignment. As described in detail in section 5.2.2, good bounding functions for this problem are created by simplifying the calculation of intertask communication times in various ways.

5.2 The Optimal TPAM in Detail

The TPAM considered here has two major parts. The first part is an initial *preprocessing* of the task diagram which uses a list scheduling heuristic to put the tasks into an ordered list for use by the branch-and-bound algorithm. This preprocessing step also provides an initial complete solution, which helps to prune the search tree during the branch-and-bound. The second part of the method, the *depth-first search*, finds an optimal assignment by depth-first branch-and-bound (DFBB).

5.2.1 Preprocessing

The sequence of tasks for assignment during the DFBB is specified by an ordered list. The list should begin at either the input or output task in the task graph and should be arranged so that the next task on the list is connected directly to tasks previously assigned. This prevents the creation of disconnected groupings of assigned tasks, which would hamper the calculation of the partial solution bounds.

List scheduling heuristics are ordering schemes in which tasks are assigned priorities and placed in a list ordered by decreasing priority. Whenever tasks must be assigned to a processor, the selection is done on the basis of priority with the higher priority tasks still available (or *ready*) being assigned first. If there is more than one task of a given priority, ties are broken randomly. Section 3.3.2 reviews list scheduling heuristics suggested in the literature.

Intuitively, an appropriate task ordering scheme should order tasks such that a set of pipelined tasks are assigned to one or more successive processors, in which case the communication delays between these tasks are minimized (in Figure 5.1, tasks *b1*, *c1*, *d1* and *e1* form a pipeline, and tasks *b2*, *c2*, *d2* and *e2* form another one). That is the assumption underlying the *Critical Path/Longest Parallel Pipeline First* (CP/LPPF) task ordering algorithm proposed in this thesis. CP/LPPF assigns priority values to tasks as follows:

1. Each task has a unique label value. The label counter starts at 1 and is incremented every time a task is labeled.
2. Labeling starts from the exit task and progresses backwards in a depth-first manner until the entry task is reached.
3. A task is labeled only if all its successors are labeled, otherwise the labeling procedure backtracks.
4. The order in which pipelines are considered is based on their total processing times. Pipelines in which the sum of processing times is the highest are labeled first.

Since no two tasks have the same priority value and pipelined tasks have successive priority values, tasks in a pipelined set will appear one after the other in the ordered list. Furthermore, the fact that the longest pipelines of tasks are assigned first reduces the search space in the branch-and-bound. This is due to two reasons: primarily, assigning the longest pipeline first gives the shorter pipelines the time to communicate around the longer one in the multiprocessor architecture, i.e. it's easier to assign a shorter pipeline such that it does not increase the total delay within which the assigned longest pipeline may be processed. The second reason is linked to the elimination rule *E* described in section 5.2.2, and has to do with intermediate unused processors in assignments. The elimination rule *E* of the DFBB must first be introduced in order to further demonstrate why assigning the longest pipelines of tasks first reduces the search space of the DFBB.

Figure 5.1 depicts the RDAD06 task graph labeled using CP/LPPF. Note that tasks *k-l* and *n-m* have been removed since they represent generator streams. Since both pipelines in RDAD06 have the same length in processing time ($\sum pt = 100$), the higher priority pipeline is selected arbitrarily.

Figure 5.1: RDAD06 labeled using CP/LPPF.

The main use of this ordered list is to choose the next task for assignment during the DFBB. However, it is also used to quickly find a complete feasible assignment. This is extremely valuable as an early pruning tool for the decision tree in the depth-first search. One simple way to find a complete assignment using the ordered list is to assign the tasks, in order, to successive processors, beginning with the master

processor. The processors are packed as much as possible without violating their limits on processing time, program memory or data memory. An initial solution for RDAD06 is given in figure 5.2, with a total delay of 5 clock cycles. This delay is near-optimal, since the minimum delay for RDAD06 is 4 clock cycles, as discussed in section 5.3. Recall that tasks k , l , m and n are part of generator streams, which are removed from the task graph prior to calling CP/LPPF. Once an assignment is obtained for the reduced graph, generators are packed onto the processors, so as to use as few processors as possible; an attempt is made to pack generators on processors which already hold non-generating tasks, such as processor #3 in figure 5.2.

Task	a	$b2$	$c2$	$d2$	$e2$	$b1$	$c1$	$d1$	$e1$	f	g	h	k	l	m	n
Level	12	11	10	9	8	7	6	5	4	3	2	1	-	-	-	-
Processor	0	0	0	1	1	1	2	2	2	2	3	3	3	3	4	4

Figure 5.2: Initial solution to RDAD06 using CP/LPPF.

5.2.2 Depth-First Branch-and-Bound Search

In order to describe the features of the *Depth-First Branch-and-Bound* method (DFBB) developed below, we use Kohler's general representation of a branch-and-bound process [37]:

$$BB(Bp, S, E, F, D, L, U, \epsilon, RB).$$

The branching rule Bp is used to generate search nodes, for which lower bound values are estimated using the lower bound function L . The elimination rule E then uses these bounds in conjunction with the current upper bound cost U to either accept or eliminate search nodes. The selection rule S then selects one of the valid nodes for further search. The acceptance approximation error ϵ specifies a maximum bound on deviation from the true optimum that is acceptable in terminating the

search. The resource bound RB fixes an upper limit on the amount of memory that the branch-and-bound algorithm may use. The characteristic function F and the dominant relation D are not used in the DFBB.

Each of these parameters is discussed in detail below. The discussion presupposes that the objective of the search is minimization.

Branching rule B_p

The branching rule is used to generate a new set of active search nodes by expanding from the current node. These new nodes are generated by assigning the next task from the ordered list to the available processor(s).

Figure 5.3: Task-to-processor assignment tree.

Consider the partial search tree of figure 5.3, which represents the search for a minimum delay assignment to the RDAD06 problem (figure 2.1). Each node represents the assignment of one task to one processor. The list R at each level lv is the unassigned tasks ordered using CP/LPPF during preprocessing. At each level, the first task from R is assigned successively to all processors which can accommodate it.

Lower bound function L

As mentioned previously, the total delay of a complete feasible solution is calculated using the *critical path method* (CPM). To accommodate the critical path calculations, the intertask communication times (known when the task-to-processor assignment is completed) are inserted as labels on the corresponding arcs, and the total delay then corresponds to the critical path between the entry node (input task) and the exit node (output task).

Because the intertask communication delays are not known before an assignment is completed, the CPM cannot be used directly for estimating lower bounds on delay for partial solutions. An optimistic estimate with potential for use as a bounding function can be calculated by assuming that no interprocessor communication delays are incurred in assigning the remaining tasks. This amounts to assuming perfect packing of the remaining tasks into processors in the same order as on the critical path. The complete critical path length using this assumption constitutes a valid bounding function value.

A better bounding estimate is obtained by improving on the unrealistic assumption of perfect packing along the critical path. Instead, we assume that unassigned task sequences are packed in pipeline order, while respecting processor limitations on the time and memory resources. The total critical path length using this assumption is a more accurate, though still optimistic estimate of the smallest total delay attainable by the growth of the partial solution. This bounding function is implemented in the prototype software.

Note that with this packing scheme the bounding function never takes into account the extra clock cycle required for the final transfer of the signal from the exit task to the master processor for output (when the exit task is not assigned to the master processor). The packing of unassigned tasks on successive processors is performed in such a way that the exit task will seldom be assigned to the master processor — the packing of unassigned tasks starts in the processor where the last assigned task on the current critical path is found, which is often not the master processor. Since the best assignment of tasks on the critical path is to pack on successive processors with the entry task and the exit task in the master processor

(so that no extra clock cycles are required to route the signal from and to the I/O port), the bounding function always ignores the extra clock cycle to transfer the signal from the exit task to the output port, so that no better packing of the tasks exists.

An example calculation for a partial solution to RDAD06 generates a lower bound on the total delay of 3 clock cycles as shown in figure 5.4. When large task graphs are considered, the difference between this bounding function and the first bounding function proposed above may be quite significant.

Task	a	b2	c2	d2	e2	b1	c1	d1	e1	f	g	h	k	l	m	n
Status*	A	A	A	A	Cr	-	-	-	-	Cr	Cr	Cr	-	-	-	-
Processor	0	0	0	1	1	-	-	-	-	1	2	2	-	-	-	-

* Status = A : task assigned.

Cr : task unassigned but on current critical path.

Figure 5.4: Calculation of bounding function value for partial assignment.

Selection rule S

Selection of the next search node to branch from is made using a *Least Lower Bound* rule (LLB). With this rule, the search node with the lowest lower bound value (calculated using function L) is selected from the set of active nodes. This same rule is applied to the remaining search nodes in the active set when backtracking. In figure 5.3, where the lower bound value L_{ij} is computed for every search node i at level j , the node selected for further search at level $lv = 2$ is search node $i \mid L_{2i} \leq L_2, \forall j$.

Upper bound cost U

The upper bound cost U is the lowest total delay of a complete assignment found so far during the search. U is theoretically set to $+\infty$ at the start of the algorithm, since no initial solution is known a priori. However, since the CP/LPPF ordering

is used to find an initial solution during preprocessing, U is initially set to the total delay of this solution. U is reset whenever a complete solution with a lower total delay is found during the search. U is used in conjunction with the lower bound function L to prune the search tree, as described next.

Elimination rule E

E stands for the set of rules used to eliminate search nodes from the set of active nodes. It is these rules which reduce the size of the search tree, giving branch-and-bound algorithms their power. The DFBB makes use of three rules, applied consecutively until one succeeds in rejecting the search node or until all fail, in which case the search node is retained for further growth. The first one uses the lower bound function L and the upper bound cost U :

E.1 An active search node is eliminated if its estimated lower bound value (computed with L) is greater than or equal to the current upper bound cost U .

This rule is referred to by Kohler [37] as *upper bound tested for dominance of descendants of branching node and members of currently active set* (U/DBAS). Since the current upper bound U is always reset to the total delay of the current best complete assignment found so far, no partial solution with estimated lower bound value not lower than U will lead to a better complete feasible assignment.

The second elimination rule employed by the DFBB is directly related to the properties of the ordered list of tasks generated by CP/LPPF. The labeling procedure of CP/LPPF is such that for any task in the ordered list, all predecessors of the task (in the task graph) appear before it in the list (i.e. they have higher priority values), while all successors appear after it in the list (i.e. lower priority values). Therefore at any node of the search tree, all tasks which precede the active task under consideration (called the *current task*) have already been assigned, while all tasks which succeed the current task are not assigned yet. Note however that no such information can be assumed about tasks which do not have a precedence relationship with the current task, i.e. tasks in parallel streams.

Based on this observation, the second elimination rule used by the DFBB is stated below. An example and a full description follow.

E.2 Eliminate the search node if the following criteria are all true:

- All tasks which do not have a precedence relationship with the current task (i.e. tasks on any other parallel pipelines) are already assigned.
- The current processor considered is not the master processor.
- The processor preceding the current processor in the architecture is unused.

In other words, no partial solution may leave intermediate unused processors in the developing assignment unless absolutely required. Figure 5.5 depicts a partial solution for RDAD06 in which intermediate processor #2 is unused. According to rule E.2, this partial assignment is invalid since all of the tasks with no precedence relationship with task *b1* (see figure 2.1) have already been assigned (tasks *b2*, *c2*, *d2* and *e2*), and processor #2 is unused. All similar partial assignments with one or more intermediate unused processors are not considered.

Figure 5.5: Partial solution with intermediate unused processor.

The basic purpose of rule E.2 is to eliminate the problem of *multiple functional isomorphs* of essentially the same solution which are produced in the search tree. Since the depth-first branch-and-bound process has no tools for remembering the assignments previously considered in the current search, several copies of the same assignment which differ only by a shift of tasks in the multiprocessor architecture are considered. Figure 5.6 presents such a case: the assignment in 5.6b is identical to the

one in 5.6a, except for the processor shift. In both cases the total delay is 5 cycles. In order to avoid such isomorphism in partial or complete assignments considered

Figure 5.6: Isomorphic assignments.

during search, partial assignments with intermediate unused processors must be avoided when possible, *without pruning the optimal solution*. If isomorphism can be eliminated in partial assignments in particular, the search space will be reduced considerably, since many subtrees will not be searched.

One obvious case where intermediate unused processors can be avoided without pruning the optimum is explicitly specified in rule E.2: *when all tasks which do not have a precedence relationship with the current task are already assigned*. When this situation occurs during search, there is no need to leave an unused processor between the previously assigned tasks and the current task since:

1. the unused processor may cause an unnecessary communication delay for data transfers to the current task,

2. all unassigned tasks are successors of the current task, so it's impossible to reduce the anticipated total delay by assigning one or several of the remaining unassigned tasks ahead of the current task in the architecture; such an attempt will also impose an unnecessary communication delay among the remaining tasks (backward communications in the architecture require two data transfers).

Therefore, when all unassigned tasks are successors of the current task, there is no need to leave one or several unused processors ahead of the current task in the architecture. Furthermore, no optimal solution is pruned, since both interprocessor communications through an unused processor or backward in the architecture require two transfers (see section 2.3).

What happens when there are unassigned tasks which are not directly or indirectly dependent on the current task? In such case there is a possibility that the optimal solution lies in the assignment of these tasks *ahead* of the current task. An example is when such a task is directly linked to an already assigned task; in such a situation the search procedure must take into consideration the eventual complete assignments where the unrelated task (which will be assigned later on) is ahead of the current task. So the search procedure must consider a partial assignment where there is one unused processor right ahead of the current task, to leave room for the unrelated task which is to be assigned later on in the search. This situation is addressed by the last rule, E.3.

The rule E.3 is used when rule E.2 does not apply because there are unassigned tasks with no precedence relationship with the current task. E.3 determines how many intermediate unused processors must be left in a partial solution in order to ensure that the optimal solution is not pruned, while minimizing redundancy in the search tree.

E.3 Eliminate the search node if the following criteria are all true:

- The current processor considered is not the master processor.
- If η is the number of *unassigned* tasks which do not have a precedence relationship with the current task (i.e. on parallel pipelines), then $\eta \geq 1$.

- If φ is the number of unused processors preceding the current processor in the assignment, then $\varphi > \eta$.

This rule leaves enough processor resources in a partial solution so that tasks without a precedence relationship with the current task may be placed before or after the current task in any complete solutions derived from this search node. Referring again to the partial assignment of figure 5.5, suppose tasks are assigned up to $c2$ inclusively, the next task to assign being $d2$. According to rule E.3, since there are four unassigned tasks unrelated to the current task $d2$ (tasks $b1$ through $e1$), partial assignments with intermediate unused processors $\#1$, $\#2$, $\#3$ and $\#4$ will be considered, i.e. a partial solution such as $(a, \#0)$, $(b2, \#0)$, $(c2, \#0)$, and $(d2, \#5)$ is valid (according to E.3), and therefore considered for further growth in the search.

In the example of figure 5.5, it is obvious that only two intermediate unused processors should be considered instead of four (since tasks $b1$ through $e1$ can be packed in two processors). However, when larger problems are considered, such a set of tasks with no precedence relationship with the current task may not always be grouped in a single pipeline but instead spread over the graph. In such cases, finding the number of processors required for packing these tasks in the longest possible way (in number of processors) is a bin-packing problem, which is in itself a combinatorial problem. Therefore we simply fix the number of intermediate unused processors considered at the number of tasks with no precedence relationship with the current task, i.e. to η . The search process is thus prevented from considering too many isomorphic partial assignments, but is still able to consider all complete assignments where one, several or all of the remaining unassigned tasks (with no precedence relationship with the current task) are assigned anywhere in the architecture, i.e. before, with or after the current task.

As introduced in section 5.2.1, CP/LPPF gives higher priority to assigning the longest pipelines of tasks first, which reduces the search space of the DFBB. The primary reason is that assigning the longest pipelines first gives the shorter pipelines the time to branch around the longer ones in the multiprocessor architecture. The second reason is directly linked to the rules E.2 and E.3 introduced before. Because

of rule E.2, the DFBB must consider partial assignments with intermediate unused processors. The number of unused processors to consider is determined by rule E.3. This rule states that the number of intermediate unused processors to consider must be large enough to hold the set of all unassigned tasks which do not have a precedence relationship with the current task. By giving highest priority to tasks in the longest pipeline, this set of unassigned tasks is smaller than if a shorter pipeline was assigned first.

Acceptance approximation error ϵ

The DFBB can obtain a near-optimal solution with a provable relative error of ϵ or less compared to the optimal solution. Lengthy search can therefore be avoided by terminating the search process as soon as the predetermined approximation error limit is reached. The terminating condition is checked by comparing the lowest bound value L of a partial solution to the current upper bound cost U . The search is terminated when $L \geq U/(1 + \epsilon)$ for all partial solutions. The solution returned corresponds to the complete feasible solution with value U .

Resource bound RB

In conventional best-first branch-and-bound, the set of active search nodes includes all partial solutions waiting for possible completion by further search. For large problems, this set quickly outgrows the memory capacity of the host system. In the DFBB, since the set of active nodes is limited to the newly created nodes while partial solutions elsewhere in the tree are temporarily ignored, the memory requirements are very limited, and linearly proportional to problem size. Since resource bounding is not an issue in the DFBB, RB is consequently not used in the prototype software implementation of the design environment. Chapter 6 describes a best-first branch-and-bound procedure with *beam search*, which fixes the resource bound RB in order to limit the length of the partial solution queue, where all partial assignments are queued in waiting for later growth.

5.2.3 Finding the Minimum Number of Processors

Finding the minimum number of processors has not been the major thrust of the research. However, in many cases assignments which satisfy both the minimum delay and minimum number of processors criteria simultaneously can be found.

This is done using a minor change to the algorithm described above. Instead of terminating with the first complete solution having a delay lower than or equal to the bounds on all partial solutions, the algorithm continues until a feasible solution has a delay that is strictly lower than the bounds on all partial solutions. This generates all of the complete solutions having the minimum delay. The algorithm counts the number of processors used in each of these solutions and retains the one using the fewest. Elimination rule *E* must also be reduced to rule E.1 alone, since rules E.2 and E.3 may prune out solutions having the same delay but a smaller number of processors. The removal of generator streams must also be disabled since their assignment may have a negative effect on the number of processors used.

This approach finds an assignment using the minimum number of processors *for the minimum total delay*. This is different than searching for an assignment with the minimum number of processors: the two solutions may differ, since the assignment using the minimum number of processors (without concern for total delay) may not have minimum total delay. However, while the original DFBB presented earlier cannot guarantee finding the true minimum number of processors, it is usually close (and often actually optimal) since the total delay is roughly proportional to the number of processors used in an assignment.

It is sometimes possible to check whether a solution returned by this revised TPAM uses the minimum number of processors or not. This is done by comparing to a theoretical (but perhaps unattainable) minimum number of processors calculated as follows: calculate separate totals for processing time, data memory, and program memory over all of the tasks in the task diagram. Divide each of these totals by the corresponding maximum processing time, data memory and program memory in one processor. Round each division up to the next integer. The maximum of these figures is the theoretical minimum number of processors. The theoretical minimum

for RDAD06 is 4.05, rounded up to 5 processors. Thus the minimum-delay solution reported in section 5.3 for RDAD06, which uses five processors, is also optimal in the number of processors used.

Standard approaches to this three dimensional bin-packing problem (the three dimensions are the task resource requirements: processing time, data memory and program memory), such as the *first-fit decreasing* and *best-fit decreasing* algorithms [31], have also been considered in the implemented optimal TPAM. Since these are easily implemented, the first-fit decreasing algorithm is included in the software to provide an estimate of the minimum number of processors needed. This heuristic orders the tasks in decreasing order of task parameters (the processing time parameter being the dominant one), then uses this ordered list to pack tasks into processors, a task being assigned as follows: starting with processor #0 (the master processor), attempt to assign the task to successive processors until one has enough resources to accommodate it. This process is repeated until all the tasks are assigned. The first-fit decreasing bin-packing algorithm guarantees that the number of processors used will be within 22% of the true optimum. Thus, to get an optimistic estimate of the true lower bound, the bin-packing result is reduced by 22%. The 22% bound actually comes from the fact that the first-fit decreasing bin-packing is performed according to the processing time parameter of tasks only. This is the dominant parameter (its value is usually larger than the other parameters). The other two parameters are only considered when testing if a partially filled processor can handle a task.

5.2.4 Summary of the TPAM

The complete basic TPAM is summarized below:

[INPUT] (input to the TPAM):

- a. the complete set of tasks,
- b. the precedence relationships among the tasks,

- c. the resource requirements for the tasks (processing time, data memory, program memory) in percent of processor limits,
- d. (optional) a maximum number of processors to be used.
- e. (optional) an acceptance approximation error ϵ .

[STEP 1] (precalculations): Calculate the theoretical minimum number of processors for a feasible assignment. Estimate the minimum number of processors for a feasible assignment by the adjusted first-fit decreasing bin-packing. Count the number of tasks in the problem, find and count the number of tasks in generator streams. If a maximum number of processors was not specified as input, set this equal to the total number of tasks. If no acceptance approximation error is specified, set ϵ to 0.

[STEP 2] (preprocessing): Remove the generator streams. Order the remaining tasks using CP/LPPF. Use the CP/LPPF ordering to generate an initial feasible solution; the delay of this solution is used as the initial upper bound value U for the DFBB.

[STEP 3] (DFBB): Select tasks from the CP/LPPF list and assign them using the DFBB.

[STEP 4] (assignment completion): Pack the tasks in the generator streams into the optimum assignment found in Step 3.

[STEP 5] (postprocessing): Collect or calculate the output data.

[OUTPUT] (output by the TPAM):

- a. The number of tasks in the original diagram, and the number of tasks when the generator streams are removed.
- b. The theoretical minimum number of processors needed for a feasible solution.
- c. The minimum number of processors needed as calculated by a first-fit decreasing bin-packing algorithm, decreased by 22%.

- d. The theoretical minimum delay as computed by the lower bound function L before any tasks are assigned.
- e. The maximum number of processors considered.
- f. The initial delay obtained with CP/LPPF.
- g. The optimal assignment found using the DFBB.
- h. The number of processors used in the optimal DFBB assignment.
- i. The mean fraction of the resources used in each processor.
- j. Computation statistics: total CPU processing time, size of the solution tree generated and searched.

Most of the various versions of the TPAM discussed in the next chapter have the same algorithmic structure except for minor changes in steps 2 and 3. The TPAM implementation can also generate an optional *trace file* in which every improved assignment found during the search is dumped. The file can also include a partial or full trace of the search tree. This allows the user to inspect the current best assignment found by the TPAM while it is still searching for the optimum.

5.3 Example: Optimal Solution to RDAD06

The TPAM was used to find an optimal assignment of RDAD06 (figure 2.1) to a multiprocessor architecture in which the amounts of resources available to each processor are all fixed at 100 units, i.e. 100 units of data memory, 100 units of program memory, and a clock cycle of 100 units of time.

The initial solution based on the CP/LPPF ordering is given in figure 5.2; it has a total delay of 5 clock cycles. The optimal solution found with the DFBB is given by the following task-processor assignment tuples: $(a, \#0)$, $(b1, \#1)$, $(b2, \#3)$, $(c1, \#1)$, $(d1, \#1)$, $(e1, \#2)$, $(f, \#0)$, $(c2, \#3)$, $(d2, \#3)$, $(e2, \#4)$, $(g, \#0)$, $(h, \#0)$, $(k, \#4)$, $(l, \#4)$, $(n, \#2)$, and $(m, \#2)$. This assignment uses 5 processors and the corresponding minimum delay is 4 clock cycles. The DFBB generated 26189 branching nodes during the depth-first search process, of which 7138 nodes were

selected for further growth (i.e. were not eliminated by the three rules in *E*). Overall, 203 complete feasible solutions were compared before the best solution found was identified as optimal. Figure 5.7 shows the result file generated by the TPAM for RDAD06.

These results were obtained from a C implementation of the TPAM on a *SUN SPARC station 1* workstation with 16 Mbytes of RAM under SunOS 4.0.3. The solution to the RDAD06 problem was found in less than 6.39 CPU seconds. Chapter 7 presents test results obtained by applying the TPAM on large sets of problems.


```

Task Graph
*****

Filename                                RDAD06

Number of tasks in original graph        16
Number of tasks after removing generator streams  12

Bin Packing
*****

Theoretical minimum # of processors      4.06
Bin packing minimum # of processors      5

Theoretical minimum delay                 2 cycles
*****

Maximum number of processors             12 processors
*****

CP/LPPF minimum delay                   5 cycles
*****

B&B Solution :  MINIMUM DELAY
*****

Minimum delay                          4 cycles (on 5 processors)
Assign
task a to processor #0
task b1 to processor #1
task b2 to processor #3
task c1 to processor #1
task d1 to processor #1
task e1 to processor #2
task f to processor #0
task c2 to processor #3
task d2 to processor #3
task e2 to processor #4
task g to processor #0
task h to processor #0
task k to processor #4
task l to processor #4
task n to processor #2
task m to processor #2

Mean processor load :  Processing time   = 81%
                       Data memory     = 70%
                       Program memory  = 72%

Statistics
*****

Processing time                        6.383333 CPU seconds

Total number of nodes generated        26189
Total number of nodes visited          7138
Total number of solutions evaluated    203

```

Figure 5.7: TPAM result file for RDAD06.

Chapter 6

Near-Optimal TPAMs

The TPAM described in this thesis is the core of the complete DSP design environment that is the goal of this research. Often, design specifications limit the time available for the TPAM to execute, at the expense of accepting near-optimal solutions rather than true optima. For this reason, a number of heuristic solution methods, strategies, and approaches for the TPAM have been investigated. The objective is to propose a fast procedure which can find assignments close to the optimum in a fraction of the time required by the optimal TPAM (a formal objective was stated in section 4.2). Some of these are briefly reviewed in this chapter.

In looking at these alternate strategies for TPAMs, an important distinction must be drawn between TPAMs which return a provably optimum solution, and heuristics. Heuristics cannot guarantee that the solution is optimum, but use assumptions and simplifications so that the solution is usually very good, and sometimes actually optimum, while the processing time is reduced significantly. To test the worth of a heuristic it is necessary to compare the solution returned by the heuristic to the true optimum solution over a set of test problems. Accordingly, in research such as this, it is necessary to have on hand a TPAM which returns the provably optimum assignment.

For this reason, efforts were initially concentrated on the branch-and-bound solution (which returns a provably optimum solution), and various modifications which reduce the size of the search space without pruning the optimum. Thus one thrust

in terms of alternate strategies has been experimentation aimed at improving the efficiency of the branch-and-bound TPAM. In some cases, attempts at efficiency improvements have later been found to potentially prune the optimum under certain conditions, and so have been moved into the class of heuristics.

The ordering of tasks as performed by the CP/LPPF algorithm is classified as a heuristic if the assignment so generated is used directly as the solution. However, when the ordered list generated by the method is used as the input to the DFBB, then CP/LPPF is only a preprocessing step, and does not prune the true optimum. However, the efficiency of the DFBB can be affected by the task ordering. For this reason another task ordering algorithm has been investigated, the *Critical Path/Most Immediate Successors First* (CP/MISF) heuristic [34], described in section 6.3.

Elimination rules E.2 and E.3 are also improvements on the basic branch-and-bound method which help to limit the size of the search tree without pruning the optimum solution. They are precursors of the heuristic described in section 6.1. Another attempt to reduce the processing time of the DFBB involves the *clustering* of tasks so that the total number of tasks considered in the DFBB is reduced, which in turn reduces the size of the search tree. Care must be taken however so that the clustering of tasks does not prune out optimal solutions. Section 6.2 summarizes research performed on clustering.

A *best-first branch-and-bound* (BFBB) approach for the TPAM was investigated. While the basic BFBB method exhibited poor performance, a revised BFBB method with *beam search* (BFBB/bs) displayed appreciable improvement over the original BFBB. Section 6.4 gives more details on the work accomplished in this area.

6.1 An Extended Elimination Rule

The elimination rule *E* described in section 5.2.2 can be made more restrictive (thereby eliminating more search nodes) by disallowing intermediate unused processors in all partial and complete solutions considered during the search. As explained in section 5.2.2, isomorphic partial and complete assignments are considered when

intermediate unused processors are allowed. By preventing the DFBB from considering such search nodes, much redundancy is eliminated in the search tree. However, the cost of such reduction in the search tree size is the risk of pruning the optimum, since in some cases attaining the optimum requires an earlier partial assignment with intermediate unused processors (again, see section 5.2.2 for details).

The revised DFBB algorithm is implemented by replacing rules E.2 and E.3 of the elimination rule *E* by the following unique rule:

E.2' Eliminate the search node if the following criteria are both true.

- The current processor considered is not the master processor.
- The processor preceding the current processor in the architecture is unused.

The increase in performance attributable to this new rule depends on the task graph topology. Rule E.2' produces a major reduction in computing time when the task graph contains several parallel sets of tasks, such as the two parallel pipelines of RDAD06 in figure 5.1. On the other hand, when the task graph has little parallelism, the improvement is less significant.

Tests have shown that the solutions attained using rule E.2' are optimal for most task graphs, while the non-optimal solutions attained are very close to optimum. See chapter 7 for the experimental results supporting these conclusions.

6.2 Automatic Clustering of Tasks

Another strategy considered for reducing the search tree is clustering the original tasks into "supertasks", thereby reducing the number of tasks considered. Consequently the search tree considered by the DFBB is reduced; e.g. given t tasks and p processors, there are p^t possible assignments to consider (section 4.1), so reducing t results in an exponential decrease in the number of assignments to consider. Obviously, a cluster must not exceed the resources available to a single processor. Moreover, clustering should be performed such that the true optimum assignment is not pruned in the reduced search tree. Using these guidelines, some general clustering rules have been established. The first is that clustering should be restricted

to groups of pipelined tasks, to keep clustering procedures simple. Intuitively, the best way to assign pipelined tasks is to pack them into successive processors. Thus, by grouping these tasks into one or more clusters and assigning these clusters to successive processors, no optimal solution should be pruned from the search tree. Unfortunately, this argument is sometimes misleading. The problem is this strategy does not take into consideration the relationship of the tasks at both ends of the pipeline with tasks outside the pipeline. When not clustered, tasks at both ends of a pipeline may be assigned with adjacent tasks not part of the pipeline. However when clustered, assigning these to the same processor may not be possible any more. This may result in pruning the optimal assignment from the search.

To prevent such pathological cases, we have to resort to *dynamic clustering*, which considers several clustering patterns during the search. The DFBB algorithm must therefore work on the original task graph, and when it encounters a set of pipelined tasks during the search, it must consider several ways to cluster these tasks, pursuing the search toward complete assignments for every clustering strategy considered. Implementation of dynamic clustering is complicated by the fact that the task graph is continuously modified during search as clusters are formed during growth of partial solutions, and then broken down to the original tasks when backtracking.

Preliminary work was conducted on dynamic clustering, but no implementation was done. A deterministic clustering procedure was however implemented, where clustering of pipelined tasks is performed at the preprocessing stage of the TPAM, thus reducing the number of tasks considered in the DFBB. The revised TPAM is therefore not optimal, and so is classified as a heuristic. Chapter 7 analyses the improvement in performance when deterministic clustering is combined with the extended elimination rule of section 6.1 in the DFBB.

6.3 Another Task Ordering Procedure: CP/MISF

Kasahara and Narita [34] suggested the *Critical Path/Most Immediate Successors First* (CP/MISF) heuristic. This ordering heuristic considers the precedence rela-

tionships in the task graph while favouring the early assignment of tasks having more immediate successors. In CP/MISF, a level number is assigned to each task, beginning at the output task (exit node), which is given level number 1. Tasks preceding this one are given a level number equal to the maximum of the levels of their immediate successors, plus the number of these successors. CP/MISF consists of the following steps:

1. Determine the *level* for each task, the level l_i of task T_i being defined as the maximum of the levels assigned to successors of T_i plus the number of immediate successors of T_i . In mathematical terms,

$$l_i = \max_{j \in \pi_i} (l_j) + |\pi_i|$$

where π_i is the set of tasks which are immediate successors of task i , and $|\pi_i|$ is its size.

2. Construct a priority list in descending order of l_i ; ties are broken randomly.

Figure 6.1 depicts the RDAD06 task graph labeled using CP/MISF. Note again that tasks $k-l$ and $n-m$ have been removed since they are generator streams.

Figure 6.1: RDAD06 Labeled Using CP/MISF.

Since CP/MISF is the best list scheduling heuristic proposed so far in previous literature, it has been implemented in order to compare its performance to CP/LPPF, proposed in this thesis. Chapter 7 describes experiments where CP/MISF replaces

CP/LPPF in the preprocessing stage of the optimal TPAM presented in the previous chapter. Both the initial solutions returned by CP/MISF and the use of its ordering lists by the DFBB are tested, in order to determine if CP/MISF should be used instead of CP/LPPF in the optimal TPAM.

6.4 Best-First Branch-and-Bound Combined with Beam Search

In depth-first branch-and-bound, the partial solution chosen for further growth is selected from the set of partial solutions just created, partial solutions elsewhere in the search tree are temporarily ignored until considered later while backtracking. The partial solution selected is thus referred to as the *local best*. Conversely, *best-first branch-and-bound* (BFBB) selects the partial solution for further growth from the set of all partial solutions generated so far since the search started. The partial solution selected is thus the *global best*. To keep track of all partial solutions generated so far, BFBB makes use of a partial solution *queue*, in which all partial solutions generated so far are stored in decreasing order of closeness to the optimum (i.e. in increasing order of lower bound values). The BFBB search thus consists of an iteration step in which the best partial assignment in the queue is unqueued (i.e. the one at the head of the queue), is expanded one step further, then all newly created partial assignments are queued. This iteration process continues until a complete assignment is unqueued; it is the optimum. For more information on best-first branch-and-bound, refer to any standard text on operations research, such as [53].

While the DFBB has significant advantages, such as no queuing requirements and early achievement of a complete assignment, the BFBB has the major advantage of generating each search node only once, cutting off more of the search tree than the DFBB. It was therefore decided to implement a BFBB-based TPAM in order to compare the performance of the DFBB-based TPAM to a standard branch-and-bound approach. However, in order to avoid memory overflow problems and long queuing delays such as encountered by Karam (section 3.5.2), a BFBB procedure

with *beam search* (BFBB/ bs_x) was implemented. In BFBB/ bs_x , the queue length is limited to x partial assignments, thus restricting the memory requirements of the procedure. When a partial assignment is to be inserted in the queue, the queue is scanned in order to determine the position of the partial assignment in it (to keep the queue ordered). Once its position is determined, all subsequent elements of the queue are shifted and the new partial assignment inserted (to avoid excessive memory swapping, most data queued are actually pointers to much bigger data structures). If the queue is already full, the last element of the queue is shifted out, and never reconsidered.

A short queue length thus limits the memory required by the BFBB, thus making the technique practical. It also reduces the processing time involved in queuing partial solutions, since scanning is limited to the maximum queue length. The optimum cannot however be guaranteed any more, since many promising partial solutions are discarded when the queue is full. Chapter 7 compares the performance of the optimal TPAM of chapter 5 to BFBB/ bs_x with several queue lengths.

It should be mentioned that the best-first branch-and-bound algorithm can be improved by including the length of partial solutions in the lower bound function. When considering two partial assignments of equal lower bound value but with different lengths, the longer partial assignment should be given higher priority since it is more likely to lead to a better complete assignment. How the lower bound function should take the length of partial solutions into consideration must be defined carefully; the length of a partial assignment should not override its estimated total delay. The heuristic BFBB/ bs_x implemented here does not take into consideration the length of partial assignments in its lower bound function.

Chapter 7

Test Results

The various versions of the TPAM have been tested on a large set of randomly generated task graphs. They have been tested also on a set of 14 test problems derived from actual DSP applications. Throughout this chapter, the TPAM implementations are designated as follow:

DFBB_{CP/LPPF} - The basic TPAM, combining CP/LPPF with the original DFBB.

DFBB_{CP/MISF} - The basic TPAM, but using CP/MISF instead of CP/LPPF.

DFBB _{ϵ} * - The basic TPAM using the CP/LPPF ordered list for its selection rule S , but also considering the CP/MISF initial solution in initializing the upper bound cost value U . ϵ indicates the acceptance approximation error value used; $\epsilon = 0\%$ is replaced by the term "optimal". More details in section 7.2.

DFBB_{heuristic}* - Similar to DFBB_{optimal}*, but where the extended elimination rule in section 6.1 and the deterministic clustering procedure in section 6.2 are used.

BFBB/bs _{x} - The BFBB/bs beam search heuristic with a queue limited to x partial assignments.

The first three implementations, $DFBB_{CP/LPPF}$, $DFBB_{CP/MISF}$, and $DFBB_{optimal}^*$, guarantee the optimal assignment, while $DFBB_{\epsilon}^*$ with $\epsilon \neq 0\%$, $DFBB_{heuristic}^*$ and $BFBB/bs_{\epsilon}$ are heuristics which cannot guarantee the optimum. The performance of the list scheduling heuristics, CP/LPPF and CP/MISF, are also tested when used alone as TPAMs or combined with the DFBB for task ordering.

All of the above TPAMs were implemented in C and test results were obtained from simulations on a *SUN SPARC station 1* with 16 Mbytes of RAM¹ under SunOS 4.0.3.

The body of this chapter presents comparisons of the performance of the various TPAMs listed above. Section 7.1 classifies DSP problems according to their design topology, and describes how test problems were obtained with the use of a *random graph generator*. Section 7.2 lists test results and computation statistics gathered through multiple simulations with the different TPAMs, and also compares their performance. Section 7.3 briefly discusses results obtained for actual DSP applications. Section 7.4 discusses the worst performances encountered during testing and briefly examines the testing of TPAMs on very large problems. Finally, section 7.5 summarizes these analyses.

7.1 Test Problems

Preliminary experiments showed that the performance of a TPAM is sometimes affected by the various topologies that a DSP application task diagram may exhibit. Thus test results accumulated in this study have been classified according to task graph topologies. For each class of topology, a random graph generator was implemented and a large set of problems generated, which were subsequently solved with the various TPAMs.

¹Random Access Memory.

7.1.1 Problem Classification

In this study, DSP problems are classified into three categories according to their topology: *random* (R), *merging tasks* (MT) and *parallel pipelines* (PP). The R class includes all problems which do not exhibit a particular topology. MT problems exhibit tasks which decrease in number from one level to the other, tasks at one level merging into fewer tasks at the next level. Finally, some DSP problems contain pipelines of tasks which can be executed concurrently; these are classified as PP problems. Figure 7.1 displays an example for each class of topology.

Figure 7.1: Examples from task graph classes.

This classification aims at establishing a relation between the performance of the various TPAMs and problem topologies. While one topology may be beneficial to a particular TPAM, it may cause a dramatic decrease in performance with another TPAM. The analysis aims to determine if some of the TPAMs presented herein are

best suited for particular topologies of DSP problems.

7.1.2 Random Graph Generation

The random graph generating procedure accepts as input:

- a. the minimum and maximum number of levels in the graph (its height);
- b. the minimum and maximum number of nodes at each level (the graph's width);
- c. the maximum number of nodes in the graph;
- d. the average number of predecessors and successors of a node;
- e. averages for task parameters (processing time, program memory and data memory).

A task graph is then generated by randomly selecting the height of the graph, its width at each level (i.e. number of nodes at each level), and the interconnections between the nodes. For simplicity, all random variables are based on a discrete uniform distribution $U[a,b]$. The topology of the generated graphs can be controlled by the use of appropriate input average values and by minor modifications in the program:

R – no modification to the program, and the $U[a,b]$ distribution is used for all random variables.

MT – the average number of nodes at each level is decreased linearly according to the level value.

PP – the number of nodes generated for level i is propagated up to level $i+x$ (where x is a random variable with small mean), and nodes from level i through $i+x$ are connected in a pipeline manner (i.e. one incoming arc and one outgoing arc for each node).

Figure 7.1 displays a task graph in each class, generated with the different versions of the random graph generator. Note that all graphs generated may have several entry

nodes but one and only one exit node. Also, no generator streams are included in generated graphs, since they are always removed by the TPAMs in the preprocessing stage (see section 4.3).

In the current study, random task graphs have been limited to a maximum of 16 tasks, since larger problems require excessive computation time to find the minimum total delay with the optimal TPAMs. For each class of problems, over 100 graphs were generated and solved with each TPAM. Table 7.1 lists the average number of nodes for the random task graphs in each class, and the average minimum delay for these problems. Note that MT problems are on average smaller than others because the optimal TPAMs require longer computation times to solve them. MT problems usually have a dense pattern of interconnections, which causes the CPM (critical path method) to undergo more computations to estimate the lower bound values for the partial assignments in the DFBB.

Random Task Graphs		
Graph class	Average size	Average minimum delay
PP	10.55	4.10
MT	8.26	4.14
R	11.20	4.86

Table 7.1: Average characteristics of the random task graphs generated.

7.2 Experiments

Experiments were performed and test results gathered in order to determine which of the various TPAMs considered exhibit the best performance, and also to correlate the TPAMs' performance with the topology classes. Section 7.2.1 looks at how $DFBB_{CP/LPPF}$ and $DFBB_{CP/MISF}$ are combined into $DFBB_{optimal}^*$. Section 7.2.2

analyses the performance of DFBB^* with various ϵ values, and also looks at the performance of $\text{DFBB}_{\text{heuristic}}^*$. Section 7.2.3 considers whether assignments generated with CP/LPPF and CP/MISF should be used directly as final solutions. Finally, section 7.2.4 analyses the performance of BFBB/bs_x for several maximum queue lengths.

7.2.1 Comparing Optimal TPAMs

Initial experiments were conducted in order to determine which of the CP/LPPF or CP/MISF task orderings is the most beneficial to the DFBB search. To do so, both $\text{DFBB}_{\text{CP/MISF}}$ and $\text{DFBB}_{\text{CP/LPPF}}$ were applied on the test problems and their respective performances compared. However, since the primary objective is to compare task ordering strategies alone, the initial solution returned by the list scheduling heuristic must be ignored by the DFBB, i.e. the upper bound cost U is reset to $+\infty$ prior to initiating the search procedure. Therefore the DFBB only uses the task ordering for its selection rule S .

Table 7.2 lists average results² obtained by comparing the solutions and performances of $\text{DFBB}_{\text{CP/MISF}}$ versus $\text{DFBB}_{\text{CP/LPPF}}$. Note that returned total delays are not compared since both TPAMs are optimal. While the solutions in terms of number of processors used are similar, $\text{DFBB}_{\text{CP/LPPF}}$ is much faster than $\text{DFBB}_{\text{CP/MISF}}$. The search spaces considered by $\text{DFBB}_{\text{CP/MISF}}$ are up to more than 3 times larger (for PP problems) than the ones considered by $\text{DFBB}_{\text{CP/LPPF}}$.

The reasons for such a difference of the search space size can be seen in the RDAD06 task graph labeled using CP/MISF, in figure 6.1 of the previous chapter. Tasks in both parallel pipelines (tasks b_1 through e_1 and b_2 through e_2) are labeled in a symmetric fashion; for example, tasks b_1 and b_2 are of equal priority, so are tasks c_1 and c_2 , and so on. Since the DFBB assigns the tasks in decreasing order of priority, tasks are assigned alternately from each pipeline. Again referring to figure 6.1, the DFBB will assign tasks successively as follows: a , b_1 , b_2 , c_1 , c_2 , etc.

While this ordering scheme may seem acceptable for task graphs with few par-

²Values presented in all tables are averages of ratios, except where mentioned otherwise.

Ratios $\frac{DFBB_{CP/MISF}}{DFBB_{CP/LPPF}}$			
Graph class	Number of processors	Number of search nodes	CPU time
PP	1.0167	3.8833	3.4339
MT	0.9963	1.7361	1.7314
R	1.0000	1.6602	1.5949

Table 7.2: Comparing ordering schemes for the DFBB (with $U = +\infty$).

allel pipelines, such as RDAD06, it is rather inefficient when the DFBB procedure must process larger task graphs. Intuitively, it seems much more appropriate to assign such a set of pipelined tasks to one or more successive processors, so that the communication delays between these tasks are minimized. CP/LPPF does exactly that. Furthermore, since tasks in the longest pipeline are assigned higher priority values by CP/LPPF, the use of the ordered list of tasks in conjunction with the elimination rule E.3 further reduces the search tree considered by the DFBB (see section 5.2.2 for details). When the DFBB uses the CP/MISF task orderings, the rule E.3 does not reduce the search space since tasks are assigned alternately from each parallel pipeline. According to results summarized in table 7.2, the DFBB should always use the CP/LPPF priority list for its branching rule B_p .

Since the DFBB uses the initial assignment found with the list scheduling heuristic as an initial upper bound cost value U (for early pruning of the search tree), the initial total delays found by CP/LPPF and CP/MISF must also be compared in order to determine which one provides the lowest delay value on average. Test results listed in table 7.3 suggest that both list scheduling heuristics offer equivalent performance on average, except for PP problems where CP/LPPF is about 28% better than CP/MISF. Note that computation times are not compared since they are negligible for both heuristics. These results concur with the previous analysis on the performance of CP/MISF. When an initial solution is generated using the

Ratios $\frac{CP/MISF}{CP/LPPF}$		
Graph class	Delay	Number of processors
PP	1.2878	1.0150
MT	1.0078	1.0091
R	1.0689	1.0005

Table 7.3: Comparing delays from list scheduling heuristics.

CP/MISF ordering list, tasks which appear one after the other in the task graph are not necessarily assigned on successive processors in the multiprocessor architecture. The initial delay obtained may then be higher than if pipelines were assigned to successive processors.

According to the results in tables 7.2 and table 7.3, an improved optimal TPAM can be derived, which always uses the CP/LPPF priority list as the branching rule Bp for the DFBB, but which uses both CP/LPPF and CP/MISF to get an initial solution for early pruning of the search tree. The new TPAM, named $DFBB^*_{optimal}$, proceeds as follow:

1. Set $U_{CP/LPPF}$ to the initial total delay returned by CP/LPPF.
2. Set $U_{CP/MISF}$ to the initial total delay returned by CP/MISF.
3. Set the upper bound cost $U = \min(U_{CP/LPPF}, U_{CP/MISF})$, and use the corresponding assignment as the current best solution.
4. Set the branching rule Bp to the priority list built with CP/LPPF.
5. Run the DFBB.

$DFBB^*_{optimal}$ makes use of both CP/LPPF and CP/MISF since both require negligible computation time. Further testing has shown that $DFBB^*_{optimal}$ is about 57% faster than $DFBB_{CP/MISF}$ but only about 2% faster than $DFBB_{CP/LPPF}$, on average.

7.2.2 $DFBB_{\epsilon}^*$ and $DFBB_{heuristic}^*$

The performance of $DFBB_{\epsilon}^*$ was then compared to $DFBB_{optimal}^*$ for two acceptance approximation error values: $\epsilon = 20\%$ and $\epsilon = 40\%$. Because of the low minimum delays involved in the test problems (see table 7.1), ϵ values lower than 20% are irrelevant, thus not considered here. Table 7.4 lists average ratios of $DFBB_{20\%}^*$ versus $DFBB_{optimal}^*$, except for the last column, which indicates how often optimality is reached with $DFBB_{20\%}^*$. While the optimum is often reached, $DFBB_{20\%}^*$ does not yield major improvement in CPU time requirements. Table 7.5 considers the

Ratios $\frac{DFBB_{\epsilon}^*}{DFBB_{optimal}^*}$ with $\epsilon = 20\%$				
Graph class	Delay	Number of search nodes	CPU time	Optimum reached
PP	1.0037	0.9436	0.9033	98%
MT	1.0100	0.8674	0.9021	95%
R	1.0244	0.6640	0.6359	87%

Table 7.4: Results using $DFBB_{\epsilon}^*$ with $\epsilon = 20\%$.

performance of $DFBB_{40\%}^*$, which requires much less processing time than $DFBB_{20\%}^*$, but reaches the optimum less frequently.

The use of $DFBB_{\epsilon}^*$ involves the appropriate selection of the acceptance approximation error value ϵ , considering the trade off between processing time requirements and the closeness of solutions to the minimum total delay.

Combining $DFBB_{optimal}^*$ with the extended elimination rule in section 6.1 and the deterministic clustering procedure in section 6.2 creates $DFBB_{heuristic}^*$, which exhibits very good performance. While requiring less than one third of the CPU time consumed by $DFBB_{optimal}^*$, $DFBB_{heuristic}^*$ still returns optimal solutions for more than 90% of the test problems, on average (see table 7.6). As expected, the search tree considered by $DFBB_{heuristic}^*$ for PP problems is much smaller than for MT and R

Ratios $\frac{DFBB^*_\epsilon}{DFBB^*_{optimal}}$ with $\epsilon = 40\%$				
Graph class	Delay	Number of search nodes	CPU time	Optimum reached
PP	1.0378	0.1747	0.2274	86%
MT	1.1282	0.1358	0.1359	56%
R	1.0748	0.1337	0.1473	66%

Table 7.5: Results using $DFBB^*_\epsilon$ with $\epsilon = 40\%$.

problems, due to the use of CP/LPPF in conjunction with the extended elimination rule (see section 6.3 for more details), which results in pruning more parts of the search tree. Also, due to the clustering procedure, $DFBB^*_{heuristic}$ performs better with pipelined tasks, but there is consequently a higher probability of pruning optimal solutions in PP problems. Nevertheless, $DFBB^*_{heuristic}$ proved to be an acceptable alternative to $DFBB^*_{optimal}$ when larger problems are considered, even if it is still subject to combinatorial explosion, though at a much slower rate.

Ratios $\frac{DFBB^*_{heuristic}}{DFBB^*_{optimal}}$					
Graph class	Delay	Number of processors	Number of search nodes	CPU time	Optimum reached
PP	1.0270	1.0819	0.0576	0.0765	90%
MT	1.0110	0.9936	0.2709	0.3321	95%
R	1.0054	0.9974	0.3137	0.2989	97%

Table 7.6: Results using $DFBB^*_{heuristic}$.

7.2.3 Performance of CP/LPPF and CP/MISF as TPAMs

Section 7.2.1 established that CP/LPPF offers better performance than CP/MISF when these heuristics are used directly as TPAMs, instead of just as list-ordering schemes for the DFBB. Comparing the list scheduling heuristics as TPAMs to $DFBB^*_{\text{optimal}}$ (tables 7.7 and 7.8), outlines once again the superiority of CP/LPPF over CP/MISF. For PP problems, CP/LPPF reaches the optimum 56% of the time, while CP/MISF finds optimum solutions in only 21% of tests performed. Computation times are not compared since they are negligible for both heuristics.

Ratios $\frac{CP/MISF}{DFBB^*_{\text{optimal}}}$			
Graph class	Delay	Number of processors	Optimum reached
PP	1.4427	1.0556	21%
MT	1.1958	1.0688	35%
R	1.2442	1.0671	29%

Table 7.7: Assignments found with CP/MISF alone.

While CP/LPPF is a fast polynomial heuristic (each test problem was processed within a few milliseconds), its assignments may often be unacceptably far from optimum. It is best used in conjunction with the DFBB for early pruning of the search space, such as in $DFBB^*_{\text{optimal}}$ and $DFBB^*_{\text{heuristic}}$.

7.2.4 BFBB/ bs_x with Various Queue Lengths

The BFBB/ bs_x heuristic has been tested with three different maximum queue lengths: $x = 100$, $x = 1000$ and $x = 10000$ partial assignments. Results are compiled in tables 7.9, 7.10 and 7.11, respectively. Each table compares the performance of BFBB/ bs_x to $DFBB^*_{\text{optimal}}$, and also indicates how often the optimum was reached and the proportion of partial assignments discarded because of a saturated queue

Ratios $\frac{CP/LPPF}{DFBB^*_{optimal}}$			
Graph class	Delay	Number of processors	Optimum reached
PP	1.1222	1.0431	56%
MT	1.1907	1.0671	34%
R	1.1699	1.0696	40%

Table 7.8: Assignments found with CP/LPPF alone.

(column titled **Pruning**). Note that the last column of table 7.9 (and tables 7.10 and 7.11) must not be interpreted as the the proportion of nodes discarded (from the search tree) compared to a full best-first branch-and-bound, since discarding a single partial solution from the search tree implicitly discards an undetermined

Ratios $\frac{BFBB/bs_{100}}{DFBB^*_{optimal}}$				Pruning
Graph class	Delay	CPU time	Optimum reached	Search nodes discarded
PP	1.0475	0.3630	81%	46%
MT	1.1337	0.9369	50%	52%
R	1.0872	0.3449	63%	3%

Table 7.9: BFBB/ bs_x with maximum queue length of $x = 100$.

number of subsequent search nodes. These values may rather be interpreted as a lower bound on the proportion of search nodes discarded compared to a full best-first branch-and-bound.

According to the results in table 7.9, a queue limited to 100 partial assignments is clearly insufficient; the performance of BFBB/ bs_{100} is much worse than $DFBB^*_{heuristic}$

(table 7.6), both in returned total delays and in processing time requirements. As shown in table 7.10, BFBB/bs₁₀₀₀ yields the optimum solution much more often than BFBB/bs₁₀₀, however the processing time requirements are even worse than with DFBB_{optimal}^{*}, on average. Finally, BFBB/bs₁₀₀₀₀ returns the optimum in proportions closer to those of DFBB_{heuristic}^{*}, but the processing time requirements may be up to 43 times what DFBB_{optimal}^{*} uses (see table 7.11). Consequently, whatever the maximum queue length may be, BFBB/bs_{*x*} does not offer better performance than DFBB_{optimal}^{*} or DFBB_{heuristic}^{*}.

Ratios $\frac{\text{BFBB/bs}_{1000}}{\text{DFBB}_{\text{optimal}}^*}$				Pruning
Graph class	Delay	CPU time	Optimum reached	Search nodes discarded
PP	1.0290	1.7538	89%	15%
MT	1.0468	6.1845	78%	28%
R	1.0635	2.1972	72%	33%

Table 7.10: BFBB/bs_{*x*} with maximum queue length of *x* = 1000.

The reason for such higher computation times is mainly due to two implementation factors: *queuing time* and data swapping. In order to keep the partial solutions queue in decreasing order of closeness to the optimum (technically speaking, in increasing order of lower bound values), it must be scanned every time a partial assignment must be queued. When the queue is very long (as in BFBB/bs₁₀₀₀₀), this scan time becomes significant. Data swapping occurs when a partial assignment is queued and unqueued; it corresponds to the data transfers which occur when the information which characterizes a partial assignment is moved back and forth between the global data structures and the queue's temporary storage. While data transfers may be reduced by swapping pointers instead of the data itself, not all data can be handled dynamically (creating and killing large data structures dynamically may take as much time as swapping memory blocks). Instead, only some parts

of the data structures are handled dynamically, and the rest is handled through memory swapping, which also contributes to the computation overheads involved in managing the queue.

Ratios $\frac{\text{BFBB}/bs_{10000}}{\text{DFBB}^*_{\text{optimal}}}$				Pruning
Graph class	Delay	CPU time	Optimum reached	Search nodes discarded
PP	1.0145	41.3242	94%	2%
MT	1.0225	43.7289	90%	17%
R	1.0315	36.9340	84%	19%

Table 7.11: BFBB/ bs_x with maximum queue length of $x = 10000$.

7.3 Actual DSP Applications

Table 7.12 lists a set of actual DSP applications (refer to section 3.5 for the origin of RDAD problems) of various sizes and topologies (including RDAD06) which have been solved with the TPAMs. Corresponding task diagrams for these problems are listed in appendix A. The first problems are small and have a simple topology, usually consisting of a single pipeline of tasks. On the other hand, some problems such as RDAD08 and RDAD40 are much larger and show more complex topologies. The main characteristics of each problem are also listed in table 7.12: the size (in number of tasks) of the task graph, the number of tasks in generator streams, and the dominant aspect of its topology (together with its classification in brackets). RDAD00 through RDAD02 and RDAD05 are simple, all tasks forming a single pipeline. Some problems show a single pipeline splitting into two or more parallel pipelines which rejoin later into a single one (e.g. RDAD06). RDAD03 and RDAD20 have a topology similar to the multiprocessor architecture described in section 2.2, i.e. it consists of a set of pipelined tasks plus one or more additional tasks which

are connected to each task within the pipeline. RDAD08 is a large MT problem and RDAD40 a large PP problem.

RDAD Problems					
Problem	Characteristics			Theoretical Minima	
	Size	Generators	Topology	Delay	Processors
RDAD00	5	0	Pipeline [PP]	3	2
RDAD01	9	0	Pipeline [PP]	3	2
RDAD02	9	0	Pipeline [PP]	3	2
RDAD03	6	0	Architecture-like [R]	5	3
RDAD04	19	3	Parallel pipelines [PP]	2	5
RDAD05	12	4	Pipeline [PP]	2	3
RDAD06	16	4	Parallel pipelines [PP]	2	4
RDAD07	12	4	Parallel pipelines [PP]	2	3
RDAD08	15	0	Merging tasks [MT]	2	4
RDAD09	9	6	Many generators [R]	2	4
RDAD10	15	4	Parallel pipelines [PP]	4	4
RDAD20	13	6	Architecture-like [R]	3	4
RDAD40	25	0	Parallel pipelines [PP]	3	6
RDAD41	25	11	Parallel pipelines [PP]	3	6

Table 7.12: RDAD problems.

Table 7.12 also gives the theoretical lower bounds for the minimum total delay and the minimum number of processors used by any complete feasible assignment. Although usually not attainable, these bounds are however useful in determining the order of magnitude of the optimal solution for a given problem. Details on the computation of these bounds are given in sections 5.2.4 and 5.2.3, respectively. Briefly, the bound on the minimum total delay is:

the length of the critical path when tasks are packed on successive pro-

processors. A depth-first search procedure scans the task graph, packing tasks on processors as they are visited (and unpacking while backtracking), and retains the path (from entry node through exit node) which yields the longest total delay.

The lower bound on the minimum number of processors is the maximum of:

1. the bin-packing solution minus 22%;
2. the sum of the processing time of all the tasks divided by the maximum processing time of a processor, rounded to the next integer;
3. the sum of the data memory of all the tasks divided by the maximum data memory of a processor, rounded to the next integer; and
4. the sum of the program memory of all the tasks divided by the maximum program memory of a processor, rounded to the next integer;

The maximum of the last three values is called the theoretical lower bound in section 5.2.3.

Table 7.13 lists the optimal assignments attained by $DFBB^*_{\text{optimal}}$. Values displayed are actual results and statistics obtained from a single run of $DFBB^*_{\text{optimal}}$ on the corresponding problems. CPU time is expressed in seconds of processor utilization. Note that the number of processors used in most of the assignments is close to the corresponding theoretical lower bounds. The RDAD40 problem is too large to be solved with $DFBB^*_{\text{optimal}}$; the CPU time limit (48 hours) was exceeded before reaching the optimum.

Table 7.14 lists solutions reached by CP/LPPF, $DFBB^*_{\text{heuristic}}$ and BFBB/bs₁₀₀, the heuristics which displayed best performance on tests with random graphs. Note again that computation times for CP/LPPF are negligible, so they are not included in table 7.14. All optimal delays are emphasized with a star symbol in superscript. Note that $DFBB^*_{\text{heuristic}}$ reached the optimum for all problems except one, exhibiting a better performance than the two other heuristics. This agrees with previous analyses on randomly generated problems.

Optimal Assignments using DFBB _{optimal} *				
Problem	Solution		Statistics	
	Minimum delay	Number of processors	CPU time (sec)	Number of search nodes
RDAD00	4	3	0.0000	8
RDAD01	4	3	0.0167	105
RDAD02	4	3	0.0167	101
RDAD03	6	5	0.0000	18
RDAD04	4	6	51.700	148727
RDAD05	4	4	0.0167	124
RDAD06	4	5	6.3000	26189
RDAD07	3	4	0.0500	220
RDAD08	4	5	969.13	2999529
RDAD09	3	6	0.0167	8
RDAD10	5	5	0.0333	129
RDAD20	4	5	0.0167	73
RDAD40	-	-	-	-
RDAD41	5	8	0.1000	339

Table 7.13: Optimal solutions to RDAD problems using DFBB_{optimal}*

7.4 Worst Case Performances

While average results are suitable for comparing the performance of the various TPAMs, more information is needed when examining the performance of a specific heuristic. Because the minimum total delays involved are fairly low (as listed in table 7.1), the average ratio values are somewhat misleading (e.g. a 5 clock cycle solution represents a 25% increase over a 4 clock cycle minimum). For these reasons, the results are expressed here as the *number of clock cycles above the optimum*. This also explicitly outlines the worst performance of the heuristic according to the total

Heuristic Solutions					
Problem	CP/LPPF	DFBB _{heuristic} *		BFBB/bs ₁₀₀	
	Delay	Delay	CPU time (sec)	Delay	CPU time (sec)
RDAD00	4*	4*	0.0000	4*	0.0000
RDAD01	4*	4*	0.0167	4*	0.0333
RDAD02	4*	4*	0.0000	4*	0.0333
RDAD03	6*	6*	0.0000	6*	0.0000
RDAD04	5	4*	0.0000	5	1.6667
RDAD05	4*	4*	0.0167	4*	0.0833
RDAD06	5	4*	0.0333	4*	0.2167
RDAD07	4	3*	0.0000	3*	0.0500
RDAD08	5	4*	29.067	5	1.7333
RDAD09	3*	3*	0.0000	3*	0.0167
RDAD10	7	6	0.0000	6	0.2000
RDAD20	4*	4*	0.0000	4*	0.0333
RDAD40	8	6*	1.7333	8	8.2333
RDAD41	5*	5*	0.0333	5*	0.2000

Table 7.14: Solutions found to RDAD problems with near-optimal TPAMs.

delay values obtained.

Table 7.15 displays the performance of two heuristics according to the delay values returned for all 300 random problems considered previously. It clearly shows the good performance of DFBB_{heuristic}*, since the heuristic only misses the optimum by one clock cycle on eight occasions. On the other hand, CP/LPPF often fails to find the optimum, and may even sometimes return unacceptable solutions (for a random problem with minimum total delay of 5 clock cycles, it returned an assignment of 9 clock cycles!). CP/LPPF may behave adequately on average, but it should only be employed when there are no other alternatives, such as when the problems are too

Performance on Delay					
TPAM	No. Clock Cycles from Min.				
	+0	+1	+2	+3	+4
DFBB [*] _{heuristic}	292	8	0	0	0
CP/LPPF	130	146	20	3	1

Table 7.15: Performance of heuristics on delay.

large to be solved with the other TPAMs.

Table 7.16 displays the worst performances encountered according to the CPU requirements for DFBB^{*}_{optimal} and DFBB^{*}_{heuristic} (CP/LPPF is not considered since its CPU requirements are negligible). Worst cases are presented for both random problems and actual DSP problems. As mentioned in section 7.3, the worst per-

Worst Performance on CPU		
TPAM	Random problems	RDAD problems
DFBB [*] _{optimal}	2.4 hrs	exceeded 48 hrs
DFBB [*] _{heuristic}	30 sec	29 sec

Table 7.16: Worst performances on CPU requirements.

formance of DFBB^{*}_{optimal} was in failing to find the optimal solution to RDAD40 after 48 hours of processing. The worst performance of DFBB^{*}_{heuristic} was encountered in several cases when the heuristic required the same amount of CPU time as DFBB^{*}_{optimal} for solving a problem. However in all such cases the heuristic did return the optimum.

Since the optimal TPAM (DFBB^{*}_{optimal}) is subject to combinatorial explosion, it cannot be employed to solve very large problems. It actually could not solve RDAD40, with 25 tasks (as reported in table 7.13). Consequently, there is no way

of testing the performance of heuristics on very large problems since the minimum total delays cannot be obtained. Neither can the CPU requirements of the various proposed heuristics be compared since they are also subject to exponential growth, except for list scheduling heuristics. Both CP/LPPF and CP/MISF were compared on problems of up to 100 tasks, and results similar to the ones reported in table 7.3 were obtained. Both list scheduling heuristics processed each problem within a few seconds (with CP/MISF slightly faster than CP/LPPF).

7.5 Discussion of Results

The analysis of the various TPAMs on real and randomly generated DSP problems can be summarized by the following observations.

- $DFBB_{\text{optimal}}^*$ is a practical TPAM for small and moderate size DSP problems (15–20 tasks); it can provide a near-optimal solution quickly, even if it's interrupted before the optimum is reached.
- $DFBB_{\text{heuristic}}^*$ is a suitable alternative to $DFBB_{\text{optimal}}^*$. It's able to find near-optimal assignments (when not optimal) while requiring only a fraction of the processing time used by $DFBB_{\text{optimal}}^*$. It outdoes all other heuristics considered in this study.
- When used in conjunction with the DFBB (either $DFBB_{\text{optimal}}^*$ or $DFBB_{\text{heuristic}}^*$), CP/LPPF is a better list scheduling heuristic than CP/MISF.
- When used by themselves as TPAMs, CP/LPPF still exhibits a better performance than CP/MISF. CP/LPPF seldom finds optimal assignments but often finds one within a close range of the optimum. It should be considered for very large problems, since it's a polynomial heuristic.
- To have a substantial decrease in processing time requirements for $DFBB_{\epsilon}^*$, ϵ must be quite large, which often prevents $DFBB_{\epsilon}^*$ from finding assignments with delays close to optimum.

- For BFBB/bs_r to find near-optimal solutions, r must be so large that the processing time requirements exceed to a large extent what DFBB_{optimal}^{*} uses.

The only correlation between the performance of the TPAMs and the task diagram topology classes is that most heuristics find better solutions for PP problems than for MT or R problems, except DFBB_{heuristic}^{*}, which however requires significantly less processing time for PP problems than for other problem classes. This is attributed to the clustering heuristic, which clusters more tasks in PP graphs, thus reducing the problem size considerably more than with MT or R graphs (but pruning the optimum more often). To attain the optimum more often for PP problems, clustering must be turned off in DFBB_{heuristic}^{*}; it then requires processing times comparable to MT and R problems, and the optimum is reached in 98% cases.

Therefore, when a DSP application is too large to use DFBB_{optimal}^{*}, DFBB_{heuristic}^{*} should be considered. If the processing time used by DFBB_{heuristic}^{*} still exceeds a predefined limit, then CP/LPPF can be employed to obtain an acceptable solution to the task-to-processor assignment problem. This conclusion holds independently of the task diagram topology, whether it's a PP, MT or R problem.

Chapter 8

Conclusions and Future Research

The goal of this thesis has been to develop good TPAMs for the assignment of tasks to processors. What distinguishes the problem here from most task-to-processor assignment problems considered previously is the introduction of interprocessor communication times, which make the problem much more difficult to solve.

8.1 Summary

The list-directed depth-first branch-and-bound algorithm presented in this thesis (DFBB_{optimal}^{*}) is very effective for small to medium DSP problems. Of the two list scheduling heuristics implemented, CP/LPPF seems an adequate choice for both the generation of a good initial solution and for providing the DFBB with a useful ordered list of tasks.

Although the DFBB can find optimal assignments for problems of moderate sizes, it is still a branch-and-bound search which only ameliorates the combinatorial explosion of the solution space, it does not eliminate it altogether. The maximum size of task graph that can be solved this way is still limited. The occasionally poor performance of DFBB_{optimal}^{*} may sometimes be attributed to the task graph topology rather than to the number of tasks. Densely connected task graphs are very difficult to solve, even for the heuristic versions of the TPAM.

By combining the extended elimination rule and the clustering procedure to-

gether with the DFBB, $DFBB_{\text{heuristic}}^*$ considerably improves the performance of the TPAM. When not optimal, the assignments found have a total delay which is very close to the minimum delay. Furthermore, this sub-optimal TPAM requires much less processing time than its optimal counterpart; it is an alternative to lengthy optimal search when large task diagrams are considered.

The optimal and near-optimal minimum delay assignments found with the TPAMs usually require a number of processors close to the theoretical lower bound. Thus, searching for the minimum delay alone is often a reasonably effective and much quicker substitute for the longer search for both the minimum number of processors for a given minimum delay (which is different from searching for an assignment with the minimum number of processors alone).

8.2 Conclusions

This thesis presents novel approaches to the assignment of DSP problems on a specific multiprocessor architecture involving interprocessor communication times. The main objectives stated in section 4.2 are thus attained:

1. The task-to-processor assignment problem with interprocessor communication times is solved.
2. The proposed optimal TPAM, $DFBB_{\text{optimal}}^*$, is able to find the minimum total delay assignment for any DSP problem, *given sufficient computation time*. The TPAM is provably optimal. Practically speaking, minimum total delay assignments can be attained for problems of moderate size.
3. The proposed heuristic, $DFBB_{\text{heuristic}}^*$, makes larger problems solvable, returning near-optimal (and often optimal) solutions in much less time than $DFBB_{\text{optimal}}^*$. $DFBB_{\text{heuristic}}^*$ is a valuable TPAM which will be useful in practice.
4. For very large problems, CP/LPPF is able to find good near-optimal solutions very quickly (in polynomial time).

Complex multiprocessor DSP problems can thus be solved. A minimum total delay assignment can be found quickly for small problems, while near-optimal assignments can be found quickly for larger problems. This was the main goal of this research to date.

While a significant amount of work remains, there is confidence that the proposed TPAMs will be of real use in building comprehensive environments for the practical design of complex DSP systems.

8.3 Future Research

While the objectives have been attained, a great deal of work is still left to be done:

1. One priority is to improve the current implementations of $DFBB^*_{\text{optimal}}$ and $DFBB^*_{\text{heuristic}}$. It is possible to better assist the algorithms by improving the clustering procedure. The current clustering procedure sometimes prunes out the true optimum, but by introducing dynamic clustering, optimal solutions will be reached much faster.
2. The genetic algorithm TPAM proposed by Resnick [49] can be improved. A C version of her TPAM has already been implemented, but testing remains to be done. It can be enhanced in several ways, such as by providing list scheduling solutions (CP/LPPF, CP/MISF and others) as part of the initial population. These would provide the GA with a good starting ground for its search process. Other genetic operators can also be tested for the TPAM, such as *dominance*, *diploidy* and *abeyance*. Finally, various combinations of input parameters must be tested, in order to determine which parameter tuples are best for solving the task-to-processor assignment problem with communication times.
3. Fernandez and Ngo's heuristic should be implemented and thoroughly tested, since the authors claim that it provides optimal solutions. A great deal of testing must be performed to compare the performance of their heuristic to the TPAMs proposed herein and in previous literature. While the heuristic itself

may not exhibit better performance (it is subject to combinatorial explosion), it proposes basic ideas which may help improve the TPAMs herein. One such interesting approach is the assignment in two steps: assigning the tasks to time slots, and then the time slots to the processors.

4. More testing should be done with CHIP (section 3.5.2), since it exhibited good performance in some preliminary tests. The use of logic programming tools in TPAMs must thus be further investigated.
5. Since the TPAMs in this thesis are all based on the single-bus architecture described in section 2.2, research should be performed on adapting the TPAMs to various other multiprocessor architecture topologies. Modifications will be minimal as long as interprocessor communication delays are bounded (as with the architecture in section 2.2). If the communication delays cannot be bounded, then fundamental changes in the TPAMs will be needed.

8.4 Summary of Contributions

The work reported in this thesis contributed in many ways to the current research on TPAMs:

1. A new variation of the minimum total delay task-to-processor assignment problem with assignment-dependent task communication times for DSP applications is introduced. Fernandez and Ngo [21] is the only other paper which considers a related problem. But the problem here involves additional communication delays due to the clocked architecture (see section 3.3.3).
2. A provably optimal algorithm for solving all DSP problems is given (no convincing proof of optimality has been given by Fernandez and Ngo for their heuristic). The DFBB procedure also introduces an elimination rule *E* which includes isomorphism elimination schemes not considered previously.
3. A new list scheduling heuristic is presented, CP/LPPF, and proved to generate better assignments than list scheduling heuristics proposed previously in the

literature. This is of great practical value for handling very large DSP problems. CP/LPPF also proved to generate better task orderings than previous list scheduling heuristics.

4. In contrast to many previous works on TPAMs, a random task graph generator is described and used to compare methods. Furthermore, the graph generator is able to generate several classes of topologies. It allows the testing of algorithms and heuristics on large sets of different problems.
5. A procedure is proposed to handle feedback loops, which are generally explicitly or implicitly ignored in most previous publications in the literature.

It should also be mentioned that by combining the proposed TPAMs with the extensible multiprocessor architecture and a graphical interface, the concept of a complete computer-assisted optimum design environment for solving complex multiprocessor DSP problems is thus seen to be achievable.

2

of/de

2

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

Bibliography

- [1] T.L. Adam, K.M. Chandy, and J.R. Dichson, "A Comparison of List Schedules for Parallel Processing Systems," *Communications of the ACM*, Vol. 17, No. 12, December 1974, pp. 685-690.
- [2] J.L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing," *Computing Surveys*, Vol. 5, No. 1, March 1973, pp. 31-80.
- [3] I.M. Barron et al., "The Transputer," *Electronics*, Vol. 17, November 1983, p. 109.
- [4] G. Bolch, F. Hofmann, B. Hoppe, C.U. Linster, R. Polzer, H.W. Schussler, G. WackerSreuther, and F.X. Wurm, "MUPSI: A Multiprocessor for Signal Processing," *Proceedings of the IEEE*, Vol. 75, No.9, September 1987, pp. 1211-1219.
- [5] *Block Oriented Systems Simulator (BOSS)*, Comdisco Systems Inc., San Francisco, CA.
- [6] B. Bussell, E.B. Fernandez, and H.C. Levy, "Optimal Scheduling for Homogeneous Multiprocessor," *Information Processing '74*, Vol. 2, North Holland, 1974, pp. 286-290.
- [7] J.W. Chinneck, R.A. Goubran, G.M. Karam, and M. Lavoie, "A Design Approach for Real-Time Multiprocessor DSP Applications," Report SCE-90-05, Carleton University, Ottawa, CANADA, February 1990.

- [8] W.W. Chu, "Optimal File Allocation in a Multiple Computer System," *IEEE Transactions on Computers*, Vol. C-18, No. 10, Oct. 1969, pp. 885-889.
- [9] W.W. Chu, L.J. Holloway, M.T. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *Computer*, Vol. 13, No. 11, Nov. 1980, pp. 57-69.
- [10] E.G. Coffman Jr., *Computer and Job-Shop Scheduling Theory*, John Wiley, New York, NY, 1976.
- [11] C.D. Covington, G.E. Carter, and D.W. Summers, "Graphic Oriented Signal Processing Language GOSPL," *International Conference on Acoustics, Speech, and Signal Processing*, Dallas, TX, 1987, pp. 1879-1882.
- [12] R.W. Conway, W.L. Maxwell, and L.W. Miller, *Theory of Scheduling*, Addison-Wesley, Reading, Mass., 1967.
- [13] *DADiSP*, DSP Development Corporation, Cambridge, MA.
- [14] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Bertheir, "The Constraint Logic Programming Language CHIP," *Proceedings of International Conference on Fifth Generation Computer Systems (FGCS-88)*, Tokyo, Jap., 1988.
- [15] *DSPlay*, Burr-Brown Inc., DSP Systems Division, Livingston, Scotland.
- [16] *EC-3 Echo Canceled*, Coherent Communications Systems Corp., 60 Commerce Drive, Hauppauge, NY.
- [17] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *IEEE Computer*, Vol. 15, No. 6, June 1982, pp. 50-56.
- [18] M. Feder, A.V. Oppenheim, and E. Weinstein, "Maximum Likelihood Noise Cancellation using the EM algorithm," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 37, No. 2, February 1989, pp. 204-216.

- [19] E.B. Fernandez and T. Lang, "Computation of Lower Bounds for Multiprocessor Schedules," *IBM Journal of Research and Development*, Vol. 19, No. 5, September 1975, pp. 435-444.
- [20] E.B. Fernandez and T. Lang, "Scheduling as a Graph Transformation," *IBM Journal of Research and Development*, November 1976, pp. 551-559.
- [21] E.B. Fernandez and T. Ngo, "Scheduling of Task Graphs into Transputer Networks," *Congressus Numerantium*, Vol. 59, 1987, pp. 79-88.
- [22] M. Fujii, T. Kasami, and K. Ninomiya, "Optimal Scheduling of Two Equivalent Processors," *SIAM Journal of Appl. Math.*, Vol. 17, No. 4, July 1969, pp. 784-789.
- [23] N.P. Galatsanos and R.T. Chin, "Digital Restoration of Multichannel Images," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 37, No. 3, March 1989, pp. 415-421.
- [24] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, Mass., 1989.
- [25] M.J. Gonzalez Jr., "Deterministic Processor Scheduling," *ACM Computing Surveys*, Vol. 9, No. 3, September 1977, pp. 173-204.
- [26] R.A. Goubran and H.M. Hafez, "Background Acoustic Noise Reduction in Mobile Technology," *Proceedings of the 36th IEEE Vehicular Technology Conference*, Dallas, Texas, 1986, pp. 72-76.
- [27] R.A. Goubran, H.M. Hafez, and A.U.H. Sheikh, "Real-Time Programmable Land Mobile Channel Simulator," *Proceedings of the 36th IEEE Vehicular Technology Conference*, Dallas, Texas, 1986, pp. 215-218.
- [28] V.B. Gylys and J.A. Edwards, "Optimal Partitioning of Workload for Distributed Systems," *Proceedings of Compcon, Fall 76*, 1976, pp. 353-357.
- [29] P. Van Hentenryck, "Constraint Satisfaction in Logic Programming," MIT Press, Cambridge, Mass., 1989.

- [30] J.H. Holland, *Adaptation in Natural and Artificial Systems*, Univ. of Michigan Press, Ann Arbor, MI, 1975.
- [31] T.C. Hu, *Combinatorial Algorithms*, Addison-Wesley, Reading, MASS, 1982.
- [32] T.C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, Vol. 9, No. 6, 1961, pp. 841-848.
- [33] D.A. Kandle, "A Systolic Signal Processor for Signal Processing Applications," *IEEE Computer*, Vol. 20, No. 7, July 1987, pp. 94-95.
- [34] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Transactions on Computers*, Vol. C-33, No. 11, November 1984, pp. 1023-1029.
- [35] S. Kerr, "The Graphical Interface and Task-to-Processor Assignment Software - User's Manual," Carleton University, Ottawa, CANADA, March 1990.
- [36] S. Kirkpatrick, C.D. Gelatt, and M.D. Vecchi, "Optimization by Simulated Annealing," *Science*, Vol. 220, May 1983, pp. 671-680.
- [37] W.H. Kohler, "Characterization and Theoretical Comparison of Branch and Bound Algorithms for Permutation Problems," *Journal of the ACM*, Vol. 21, January 1974, pp. 140-156.
- [38] E.A. Lee and D.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.
- [39] E.A. Lee and D.G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, Vol. 75, No. 9, September 1987, pp. 1235-1245.
- [40] E.A. Lee, W.H. Ho, E.E. Goei, J.C. Bier, and S. Bhattacharyya, "Gabriel: A Design Environment for DSP," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 37, No. 11, November 1989, pp. 1751-1762.

- [41] J.K. Lenstra and A.H.G. Rinnooy Kan, "Complexity of Scheduling Under Precedence Constraints," *Operations Research*, Vol. 26, No. 1, January 1978, pp. 22-35.
- [42] S.F. Lundstrum, "Applications Considerations in the System Design of Highly Concurrent Multiprocessors," *IEEE Transactions on Computers*, Vol. C-36, No. 11, November 1987, pp. 1292-1309.
- [43] G.K. Manacher, "Production and Stabilization of Real-Time Task Schedulers," *Journal of the ACM*, Vol. 14, No. 3, July 1967, pp. 439-465.
- [44] D. May, "Ocamm," Inmos Technical Note 6, 1983.
- [45] J.G. Nash, K.W. Przytula, and S. Hansen, "The Systolic/Cellular System for Signal Processing Applications," *IEEE Computer*, Vol. 20, No. 7, July 1987, pp. 96-97.
- [46] T. Ngo, "Optimal Scheduling of Processes for a Network of Transputers," M.Sc. Thesis, Department of Electrical and Computer Engineering, Florida Atlantic University, FL. December 1986.
- [47] C.V. Ramamoorthy, K.M. Chandy, and M.J. Gonzalez Jr., "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE Transactions on Computers*, Vol. C-21, No. 2, February 1972, pp. 137-146.
- [48] G.S. Rao, H.S. Stone, and T.C. Hu, "Assignment of Tasks in a Distributed Processor System with Limited Memory," *IEEE Transactions on Computers*, Vol. C-28, No. 4, April 1979, pp. 291-299.
- [49] N. Resnick, *Network Allocation Using Genetic Algorithms*, B. Eng. Project Report, Carleton University, Ottawa, CANADA, 1989.
- [50] *Signal Processing WorkSystem (SPW)*, Comdisco Systems Inc., San Francisco, CA.

- [51] H.S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January 1977, pp. 85-93.
- [52] H.S. Stone and S.H. Bokhari, "Control of Distributed Processes," *IEEE Computer*, Vol. 11, No. 7, July 1978, pp. 97-106.
- [53] W.L. Winston, *Operations Research: Applications and Algorithms*, PWS/Kent Publishers, Boston, MASS, 1987.

Appendix A

Task Graphs for DSP Applications

This appendix presents the task graphs representing the actual DSP problems listed in section 7.3. The diagrams were created using a graphical interface linked to the TPAMs, in order to provide users with a comprehensive environment for the optimal design of solutions to DSP problems (see section 1.1 for details on such environments). Accordingly, the *Block Oriented Systems Simulator* (BOSS) [5] was adapted for use as the graphical interface. BOSS provides a complete interactive graphical environment for the specification of the DSP problems in a task diagram format, and also has facilities for problem simulation and analysis.

Appendix B

Program Listings

This appendix presents the C implementation of both $DFBB^*_{optimal}$ and $DFBB^*_{heuristic}$. Both TPAMs are combined in the same implementation, and may be called specifically by the use of command line arguments. Therefore, given the input file “rdad06.in” which contains the task graph description of a DSP problem, $DFBB^*_{optimal}$ may be run as follow:

```
rdad -g -i rdad06.in
```

while $DFBB^*_{heuristic}$ may be run with the following command line:

```
rdad -g -c -f -i rdad06.in
```

The description of all the command line arguments is included in the program documentation (in the heading of the main file, **RDAD_grf.c**). Figure B.1 lists the contents of “rdad06.in”, the description file for the RDAD06 problem of figure 2.1. Each line in the file is a Prolog-like definition of a task where the entries represent respectively: its label, the program memory, data memory and processing time requirements, the list of incoming communication links, and the list of outgoing communication links (a communication link is identified by the adjacent task’s label). The entry task is identified by the “INPUT” label in its incoming arcs list, while the exit task is identified by the “OUTPUT” label in its outgoing arcs list.

The source code is divided into eight files. Each file contains a heading which describes the file contents, and each function within a file also has a heading to

```

process("a",30,20,40,["INPUT"],{"b1","b2"}).
process("b1",25,15,30,{"a"},{"c1"}).
process("c1",40,30,20,{"b1"},{"d1"}).
process("d1",15,20,20,{"c1"},{"e1"}).
process("e1",10,40,30,{"d1"},{"f"}).
process("b2",25,15,30,{"a"},{"c2"}).
process("c2",40,30,20,{"b2"},{"d2"}).
process("d2",15,20,20,{"c2"},{"e2"}).
process("e2",10,40,30,{"d2"},{"f"}).
process("f",30,10,10,{"e1","e2"},{"g"}).
process("g",10,40,20,{"f"},{"h"}).
process("h",10,10,15,{"g"},{"OUTPUT"}).
process("k",25,15,30,[],{"l"}).
process("l",25,15,30,{"k"},{"d1"}).
process("n",25,15,30,[],{"m"}).
process("m",25,15,30,{"n"},{"d2"}).

```

Figure B.1: Input file for the RDAD06 problem.

describe its purpose and structure. In-code documentation is also added to provide more insights on the functions. The eight files are:

RDAD_grf.c – contains the main function and all required functions to read in task graph files and output results to standard output. Most global data structures are also defined in this file.

RDAD_crp.c – contains all the functions which implement the *critical path method* (CPM).

RDAD_sch.c – contains the functions required to compute the total delay of complete feasible assignments.

RDAD_bnd.c – contains the code for the list scheduling heuristics (CP/MISF and CP/LPPF) and the implementation of the lower bound function for the DFBB.

RDAD_trc.c – contains all the functions implementing the trace utility.

RDAD_BnB.c - contains the DFBB functions.

RDAD_Bin.c - contains the bin-packing functions and the theoretical lower bound functions for estimating the minimum number of processors required to assign the task graph.

RDAD_clu.c - contains the clustering functions and the functions for handling generator streams.

File listings are provided in the following pages, in the same order as above.

For details on how to recompile and use the TPAM, refer to the heading of the main file, **RDAD_grf.c**.

<p>Jul 13 15:54 1990 depth ROAD/ROAD.grf.c Page 1</p>	<pre> FILE: ROAD.grf.c DESCRIPTION: This file contains the main procedure, together with the I/O functions for reading a task graph from a specified file and outputting results on default console. Most global data structures are also defined in this file. For more details on each function, see function headings. For compiling the whole program, you must perform the following UNIX command. cc ROAD.grf.c -o road This command compiles ROAD.grf.c and all included files with the optimization parameter enabled. Since all other ROAD *.c files are included in ROAD.grf.c, only this file has to be compiled. The resulting executable file is named "road". Running "road" can then be done by the following UNIX call. road -i (input file) [-p] [-q] [-c] [-f] [-t] [-n] [-e %] [-a %] The first parameter is always required, but all others are optional. Since that parameters can be specified in any order. The function of each parameter is the following: -i (input file) to specify the input file which contains the task graph definition. This parameter must always be specified -p when this option is enabled, the program searches for the solution to the assignment problem which has the minimum number of processors for the minimum completion delay. Note that by enabling this option, the search process will take considerably more processing time. -q this option is to be used when the input task graph has one or several generator streams to be removed prior to the search. The program will thus remove all generator streams from the task graph, find a solution to the assignment problem, then back the removed tasks on the resulting assignment solution. This procedure allows to solve the task graph much more quickly without pruning any optimal solutions. If task graph has no generator streams, this option has no effects. -c when enabled, the program will indicate the number of clusters which can be created in order to shrink pipelined tasks into a single vertex in the task graph, and actually perform the clustering. This clustering process often allow the program to reduce task graph size in order to accelerate the search process. Once an assignment is found, all tasks within a same cluster are assigned to the same processor. NOTE THAT IF NO CLUSTER IS FOUND, THE PROGRAM STOPS WITHOUT SEARCHING FOR A SOLUTION. -f when enabled, the branch & bound process will not consider </pre>	<pre> partial solutions which include intermediate empty processors. This option greatly accelerates the search process but may result in pruning optimal solutions in some cases. For most assignment problems however, it will find an optimal solution very quickly. So, you may use this option to find a near-optimal solution more quickly but remember that the solution is not a guaranteed optimum. t (N) when this option is enabled, the program creates a trace file into which all better solutions found during the search are written. This allows to have a good solution without having to wait for the program to terminate. The trace file is named according to the input file (see option -i). Input filename is of the form f.t. The extension f is first removed from the filename, and the extension t is added. The trace file will be named f.t. With the option, an optional integer parameter in the range [0..20] may be specified. This value corresponds to the proportion of the search tree to be dumped in the trace file. For example, if a problem has 10 tasks, and the option -t 20 is added to the road call, then 20% of the search tree will be dumped, which corresponds to the assignment of the first 2 tasks. The default dump size is 0%, i.e. no search tree is dumped. -o % this option is used with an approximation acceptance error. It is to be specified for the branch and bound procedure. The approximation error must be in the integer range [0..100] -o %d this option is used to set a limit on the number of processors a feasible assignment may contain. If the specified limit is too low or too high, the program will reset it to a valid limit. By specifying a low number of processors, the search tree may be reduced considerably. Be careful: if the processor limit is too low, the optimal assignment may be pruned. : INSTANTS: MAX TASKS: MAX PROCESSORS, maximum sizes for task graphs and processors architectures respectively. MAX TASKNAME: maximum task name size (in character.) NO VALUE, INFINITE: numeric constants representing the absence of value and the largest integer value, respectively. MARKED: used to mark a task in the task graph. 1 indicating this task has been visited. TRUE, FALSE: represent boolean constants. PROCESSOR PT, PROCESSOR DN, PROCESSOR PM, constants which represent the maximum processing time, data memory and program memory of a processor. These constants are available for use by task graphs. Task's resource requirements must then be specified according to these constants. BOOLEAN: used to define boolean variables (to be used in the if-then-else with constants TRUE and FALSE). BEGIN, END: to replace block definition and to respect the order of </pre>
---	---	--

*** NOTE THAT THESE CONSTANTS MAY BE USED IN ANY INCLUDED FILES ***

DATA TYPE

Label: labels to be assigned to each task during search

Link, *PrtLink: defines linked lists associated to each task of the task graph. These lists contain all defined arcs from the task to adjacent tasks. For each element of a linked list, SuccTask indicates to which task the arc leads, Delay is the delay value of the arc, and Next is a pointer to the next element (i.e. next arc) of the linked list.

TTInfo: defines the task and processor resources (i.e. processing time, data memory and program memory).

Task: defines all the informatics of a task of the task graph. It includes resource requirements, label value, linked list of outgoing arcs

Task_Graph, *PrtTaskGraph: task graph structure. It defines a task graph: the number of tasks in it, its entry and exit tasks, and a vector of tasks. Note that arcs are defined within tasks themselves (see Task data type).

Proc_Graph, *PrtProcGraph: processor architecture structure: it specifies the number of processors in the architecture, the default resources available when a processor holds no task, and the current resources available for each processor (in vector).

TCbar: this vector makes the correspondence between the task name as read in the task graph, and the task numbers used in the programs

CurSol: this data structure is used to hold a solution to a task to processor problem. It indicates the assignment vector, the total completion delay for the solution, and the number of processors required by that solution.

Schedule, *PrtSchedule: this data structure is used to record partial and complete assignments. The vector Assign holds the current partial assignment; the vector SList holds the list of schedule generated by CP/MSF (or CP/LPFF) and used in the selection rule of B&B

OrdList, *PrtOrdList: data structure used to hold a list of schedule used by the B&B procedure for selecting branches of the search tree. The list in OrdList is copied in SList of Schedule before calling the B&B procedure.

State: this data structure is used to hold processing statistics. Total_Nodes counts the # nodes visited in search tree, Nb_Solutions counts the # of solutions generated; Nb_Pursued counts the # of nodes not rejected during search, i.e. the number of subnodes searched

MACROS

talloc: used to allocate memory space for dynamic structures.

Get_Delay: returns the delay value assigned to the specified task graph arc _a_.

Put_Delay: assigns the value _b_ to the specified task graph arc _a_.

All_Adjacent: the macro is to be used as the conditional of a for loop

It iteratively assigns to ptr each outgoing arc from task via. Note that via must be the integer label corresponding to the task (not its name specified in the input file), and ptr must be declared as PTrLink (see AIA type).

Initialize_TGraph: initializes the specified task graph ptr to its simplest form, i.e. one task with no resources requirements and no output. This macro is called prior to reading a task graph from a file. Note that ptr must be of type PTrTaskGraph.

Terminate_TGraph: resets the specified task graph structure ptr to default values, i.e. the empty task graph. Note that ptr must be of type PTrTaskGraph.

Min, Max: return respectively the minimum and the maximum of the two arguments a and b.

Get_Label: returns the label structure value assigned to the specified task via. The returned label is assigned to argument lbi; b is a task via must be the integer label corresponding to the task (not its name specified in the input file), and lbi must be of type Label.

Put_Label: assigns the specified label value to the specified task. See Get_Label macro for details on arguments via and ptr.

Reset_Main_DB: resets all data structures to default values. It is called prior to calling CP/MSF (or CP/LPFF) and prints the B&B search. It initializes the Best_Solution to None, the schedule to None, and the processor architecture as empty.

PRXIDONES Init PGraph, fgetatr, Get ID, Read Tests Main

AUTOUR Mouchon Lavoue

DATE March 31, 1990

COUVRANT (C) B&B MAJESTY THE QUEEN IN RIGHT OF CANADA (1990.)
represented by the minister of Defence

History

Date Changed By Reason

Constants

```

#define MAX_TASKS 100 /* maximum number of tasks */
#define MAX_PROCESSORS 50 /* maximum number of processors */
#define MAX_TILDSAME 50 /* maximum task name size */
#define NO_VALUE -1 /* represent no value assigned to _a_ */
#define INFINITE 99999 /* large integer to _a_ */

```

```

#define MAXED 1 /* used to mark a vertex in graph */
#define TRUE 1 /* for boolean variables */
#define FALSE 0 /* for boolean variables */

#define PROCESSED_PT 100 /* normalised processing time */
#define PROCESSED_ON 100 /* normalised data memory size */
#define PROCESSED_PM 100 /* normalised program memory size */

int /* to define boolean variables */
{ /* begin of block */
} /* end of block */

.....
Macros:
.....
#define malloc(type) (type *) malloc(sizeof (type)) /* to allocate heap memory */
#define Get_Delay(a) a->Delay /* returns delay on the arc a */
#define Put_Delay(a,b) a->Delay = b /* assigns delay b to arc a */

#define All_Adjacent(vts, ptr) /* for all adjacent vertices */
    ptr = TGraph->Vertices[vts].Adjacent; ptr != NULL; ptr = ptr->Next

#define Initialize_TGraph(ptr) /* initialise task graph */
    ptr->Nb_Tasks = 1;
    ptr->Entry_VerTEX = 0;
    ptr->Exit_VerTEX = NO_VALUE;
    ptr->Vertex[0].Adjacent = NULL;
    ptr->Vertex[0].TConfig.ProcTime = 0;
    ptr->Vertex[0].TConfig.NbData = 0;
    ptr->Vertex[0].TConfig.NbData = 0

#define Terminate_TGraph(ptr) /* add dummy exit vertex */
    ptr->Vertex[ptr->Nb_Tasks].Adjacent = NULL;
    ptr->Vertex[ptr->Nb_Tasks].TConfig.ProcTime = 0;
    ptr->Vertex[ptr->Nb_Tasks].TConfig.NbData = 0;
    ptr->Vertex[ptr->Nb_Tasks].TConfig.NbData = 0

#define min(a,b) ((a) > (b)) ? (a) : (b) /* returns maximum of arguments */
#define max(a,b) ((a) < (b)) ? (a) : (b) /* returns minimum of arguments */

.....
Included C libraries:
.....
#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>

Global data structures:

typedef struct etcl
begin
    int Value;
/* represents labels of vertices */
/* value of label */

```

```

Jul 13 15:54 1990 depth ROM/ROMD.grf c Page 6

end CurrSol;

typedef struct atc {
    int Assign (MAX_TASKS);
    int SList (MAX_TASKS);
    CurrSol Best_Solution;
    end Schedule, "PrtSchedr";

typedef struct atcd {
    begin
        int Delay;
        int Processor;
        end OrdLst, "PrtLst";

typedef struct atcl {
    begin
        int Total_Nodes;
        int No_Solutions;
        int No_Failed;
        end Stats;

Global variables.
.....
Boolean Product;
Boolean MultiProc;
Boolean OutTree;
float TrcDepth;
int TrcNbNodes;
float ApproxError;
Boolean UseCPUPPF;
Boolean LimitProc;
int Limit = MAX_PROCESSORS;
State State = {0, 0, 0};

char "TraceSet = ".Ans";
char "TraceFile = "

.....
.. Name : Init_PcGraph
.. Function : This function initializes the processor architecture to default
.. values. All processors are reset to initial resource limits. Note that
.. both the input and the output processors are set to processor 80, since it

```

```

.. .. is the only processor with and I/O port in the architecture
.. Arguments : PGraph (t) : pointer to processor architecture.
.. Func Value : None
.. ..
init PGraph (PGraph)
PcGraph PGraph;
begin
    int i;
    PGraph > PGraph.ProcTime = PROCESSOR_PT;
    PGraph > PGraph.HwData = PROCESSOR_HW;
    PGraph > PGraph.HwData = PROCESSOR_HW;
    for i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849, 850, 851, 852, 853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874, 875, 876, 877, 878, 879, 880, 881, 882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1010, 1011, 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019, 1020, 1021, 1022, 1023, 1024, 1025, 1026, 1027, 1028, 1029, 1030, 1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039, 1040, 1041, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 1050, 1051, 1052, 1053, 1054, 1055, 1056, 1057, 1058, 1059, 1060, 1061, 1062, 1063, 1064, 1065, 1066, 1067, 1068, 1069, 1070, 1071, 1072, 1073, 1074, 1075, 1076, 1077, 1078, 1079, 1080, 1081, 1082, 1083, 1084, 1085, 1086, 1087, 1088, 1089, 1090, 1091, 1092, 1093, 1094, 1095, 1096, 1097, 1098, 1099, 1100, 1101, 1102, 1103, 1104, 1105, 1106, 1107, 1108, 1109, 1110, 1111, 1112, 1113, 1114, 1115, 1116, 1117, 1118, 1119, 1120, 1121, 1122, 1123, 1124, 1125, 1126, 1127, 1128, 1129, 1130, 1131, 1132, 1133, 1134, 1135, 1136, 1137, 1138, 1139, 1140, 1141, 1142, 1143, 1144, 1145, 1146, 1147, 1148, 1149, 1150, 1151, 1152, 1153, 1154, 1155, 1156, 1157, 1158, 1159, 1160, 1161, 1162, 1163, 1164, 1165, 1166, 1167, 1168, 1169, 1170, 1171, 1172, 1173, 1174, 1175, 1176, 1177, 1178, 1179, 1180, 1181, 1182, 1183, 1184, 1185, 1186, 1187, 1188, 1189, 1190, 1191, 1192, 1193, 1194, 1195, 1196, 1197, 1198, 1199, 1200, 1201, 1202, 1203, 1204, 1205, 1206, 1207, 1208, 1209, 1210, 1211, 1212, 1213, 1214, 1215, 1216, 1217, 1218, 1219, 1220, 1221, 1222, 1223, 1224, 1225, 1226, 1227, 1228, 1229, 1230, 1231, 1232, 1233, 1234, 1235, 1236, 1237, 1238, 1239, 1240, 1241, 1242, 1243, 1244, 1245, 1246, 1247, 1248, 1249, 1250, 1251, 1252, 1253, 1254, 1255, 1256, 1257, 1258, 1259, 1260, 1261, 1262, 1263, 1264, 1265, 1266, 1267, 1268, 1269, 1270, 1271, 1272, 1273, 1274, 1275, 1276, 1277, 1278, 1279, 1280, 1281, 1282, 1283, 1284, 1285, 1286, 1287, 1288, 1289, 1290, 1291, 1292, 1293, 1294, 1295, 1296, 1297, 1298, 1299, 1300, 1301, 1302, 1303, 1304, 1305, 1306, 1307, 1308, 1309, 1310, 1311, 1312, 1313, 1314, 1315, 1316, 1317, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1327, 1328, 1329, 1330, 1331, 1332, 1333, 1334, 1335, 1336, 1337, 1338, 1339, 1340, 1341, 1342, 1343, 1344, 1345, 1346, 1347, 1348, 1349, 1350, 1351, 1352, 1353, 1354, 1355, 1356, 1357, 1358, 1359, 1360, 1361, 1362, 1363, 1364, 1365, 1366, 1367, 1368, 1369, 1370, 1371, 1372, 1373, 1374, 1375, 1376, 1377, 1378, 1379, 1380, 1381, 1382, 1383, 1384, 1385, 1386, 1387, 1388, 1389, 1390, 1391, 1392, 1393, 1394, 1395, 1396, 1397, 1398, 1399, 1400, 1401, 1402, 1403, 1404, 1405, 1406, 1407, 1408, 1409, 1410, 1411, 1412, 1413, 1414, 1415, 1416, 1417, 1418, 1419, 1420, 1421, 1422, 1423, 1424, 1425, 1426, 1427, 1428, 1429, 1430, 1431, 1432, 1433, 1434, 1435, 1436, 1437, 1438, 1439, 1440, 1441, 1442, 1443, 1444, 1445, 1446, 1447, 1448, 1449, 1450, 1451, 1452, 1453, 1454, 1455, 1456, 1457, 1458, 1459, 1460, 1461, 1462, 1463, 1464, 1465, 1466, 1467, 1468, 1469, 1470, 1471, 1472, 1473, 1474, 1475, 1476, 1477, 1478, 1479, 1480, 1481, 1482, 1483, 1484, 1485, 1486, 1487, 1488, 1489, 1490, 1491, 1492, 1493, 1494, 1495, 1496, 1497, 1498, 1499, 1500, 1501, 1502, 1503, 1504, 1505, 1506, 1507, 1508, 1509, 1510, 1511, 1512, 1513, 1514, 1515, 1516, 1517, 1518, 1519, 1520, 1521, 1522, 1523, 1524, 1525, 1526, 1527, 1528, 1529, 1530, 1531, 1532, 1533, 1534, 1535, 1536, 1537, 1538, 1539, 1540, 1541, 1542, 1543, 1544, 1545, 1546, 1547, 1548, 1549, 1550, 1551, 1552, 1553, 1554, 1555, 1556, 1557, 1558, 1559, 1560, 1561, 1562, 1563, 1564, 1565, 1566, 1567, 1568, 1569, 1570, 1571, 1572, 1573, 1574, 1575, 1576, 1577, 1578, 1579, 1580, 1581, 1582, 1583, 1584, 1585, 1586, 1587, 1588, 1589, 1590, 1591, 1592, 1593, 1594, 1595, 1596, 1597, 1598, 1599, 1600, 1601, 1602, 1603, 1604, 1605, 1606, 1607, 1608, 1609, 1610, 1611, 1612, 1613, 1614, 1615, 1616, 1617, 1618, 1619, 1620, 1621, 1622, 1623, 1624, 1625, 1626, 1627, 1628, 1629, 1630, 1631, 1632, 1633, 1634, 1635, 1636, 1637, 1638, 1639, 1640, 1641, 1642, 1643, 1644, 1645, 1646, 1647, 1648, 1649, 1650, 1651, 1652, 1653, 1654, 1655, 1656, 1657, 1658, 1659, 1660, 1661, 1662, 1663, 1664, 1665, 1666, 1667, 1668, 1669, 1670, 1671, 1672, 1673, 1674, 1675, 1676, 1677, 1678, 1679, 1680, 1681, 1682, 1683, 1684, 1685, 1686, 1687, 1688, 1689, 1690, 1691, 1692, 1693, 1694, 1695, 1696, 1697, 1698, 1699, 1700, 1701, 1702, 1703, 1704, 1705, 1706, 1707, 1708, 1709, 1710, 1711, 1712, 1713, 1714, 1715, 1716, 1717, 1718, 1719, 1720, 1721, 1722, 1723, 1724, 1725, 1726, 1727, 1728, 1729, 1730, 1731, 1732, 1733, 1734, 1735, 1736, 1737, 1738, 1739, 1740, 1741, 1742, 1743, 1744, 1745, 1746, 1747, 1748, 1749, 1750, 1751, 1752, 1753, 1754, 1755, 1756, 1757, 1758, 1759, 1760, 1761, 1762, 1763, 1764, 1765, 1766, 1767, 1768, 1769, 1770, 1771, 1772, 1773, 1774, 1775, 1776, 1777, 1778, 1779, 1780, 1781, 1782, 1783, 1784, 1785, 1786, 1787, 1788, 1789, 1790, 1791, 1792, 1793, 1794, 1795, 1796, 1797, 1798, 1799, 1800, 1801, 1802, 1803, 1804, 1805, 1806, 1807, 1808, 1809, 1810, 1811, 1812, 1813, 1814, 1815, 1816, 1817, 1818, 1819, 1820, 1821, 1822, 1823, 1824, 1825, 1826, 1827, 1828, 1829, 1830, 1831, 1832, 1833, 1834, 1835, 1836, 1837, 1838, 1839, 1840, 1841, 1842, 1843, 1844, 1845, 1846, 1847, 1848, 1849, 1850, 1851, 1852, 1853, 1854, 1855, 1856, 1857, 1858, 1859, 1860, 1861, 1862, 1863, 1864, 1865, 1866, 1867, 1868, 1869, 1870, 1871, 1872, 1873, 1874, 1875, 1876, 1877, 1878, 1879, 1880, 1881, 1882, 1883, 1884, 1885, 1886, 1887, 1888, 1889, 1890, 1891, 1892, 1893, 1894, 1895, 1896, 1897, 1898, 1899, 1900, 1901, 1902, 1903, 1904, 1905, 1906, 1907, 1908, 1909, 1910, 1911, 1912, 1913, 1914, 1915, 1916, 1917, 1918, 1919, 1920, 1921, 1922, 1923, 1924, 1925, 1926, 1927, 1928, 1929, 1930, 1931, 1932, 1933, 1934, 1935, 1936, 1937, 1938, 1939, 1940, 1941, 1942, 1943, 1944, 1945, 1946, 1947, 1948, 1949, 1950, 1951, 1952, 1953, 1954, 1955, 1956, 1957, 1958, 1959, 1960, 1961, 1962, 1963, 1964, 1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025, 2026, 2027, 2028, 2029, 2030, 2031, 2032, 2033, 2034, 2035, 2036, 2037, 2038, 2039, 2040, 2041, 2042, 2043, 2044, 2045, 2046, 2047, 2048, 2049, 2050, 2051, 2052, 2053, 2054, 2055, 2056, 2057, 2058, 2059, 2060, 2061, 2062, 2063, 2064, 2065, 2066, 2067, 2068, 2069, 2070, 2071, 2072, 2073, 2074, 2075, 2076, 2077, 2078, 2079, 2080, 2081, 2082, 2083, 2084, 2085, 2086, 2087, 2088, 2089, 2090, 2091, 2092, 2093, 2094, 2095, 2096, 2097, 2098, 2099, 2100, 2101, 2102, 2103, 2104, 2105, 2106, 2107, 2108, 2109, 2110, 2111, 2112, 2113, 2114, 2115, 2116, 2117, 2118, 2119, 2120, 2121, 2122, 2123, 2124, 2125, 2126, 2127, 2128, 2129, 2130, 2131, 2132, 2133, 2134, 2135, 2136, 2137, 2138, 2139, 2140, 2141, 2142, 2143, 2144, 2145, 2146, 2147, 2148, 2149, 2150, 2151, 2152, 2153, 2154, 2155, 2156, 2157,
```

```

** Func Value = <0> if succeeds,
**               <1> if a fatal error occurs
**
.....
int Read_Tasks (TGraph, In File)
PrtTGraph TGraph
char *In File

begin
    int Vtx;
    int OutDeg;
    int J, k, n;
    int Counters;
    int Heading;
    int Mediate;
    int ProcTime;
    char str (MAX_TASKNAME);
    PrtLine header;
    FILE *Input File;
    boolean Is Entry (MAX TASKS);

    if ((Input File = fopen (In File, "r")) == NULL) /* open input file */
        begin
            fprintf(stderr, "ERROR can't find input file %s\n", In File);
            return (1)
        end

    for (j = 0; j < MAX TASKS; j++)
        begin
            Is.Entry(j)
                = TRUE;
            TGraph->Vertex(j).Generator = FALSE;
            TGraph->Vertex(j).No.Is = 0;
        end

    Initialize TGraph (TGraph);

    /* read all task graph from the input file, one task at a time */
    while (fscanf (Input File, "%d\n", &j) != EOF) /* read one task from file */
        begin
            if (file = fseekar (Input File, 0, 1)); /* read task name */
                return (1);

            if ((Vtx = Get ID (str, n)) == NO_VALUE) /* if it's not a numbered entry */
                begin
                    Vtx = TGraph->No Tasks++;
                    strcpy (TChar[Vtx].Name, str, n)
                        TChar[Vtx].Size = n;
                end

            /* read task configuration and initialize its arcs list */
            fscanf (Input File, "%d,%d,%d", &nHeading, &nMediate, &nProcTime);
            TGraph->Vertex(Vtx).Adjacent = NULL; /* initialization of adjacent list */
        end
end

```

```

JUL 13 15 54 1990  depch_READ/ROAD_gff.c Page 6

TGraph->Vertex[Vin].TConfig.ProcTime = ProcTime; /* initialize task parameters */
TGraph->Vertex[Vin].TConfig.NearProg = NearProg;
TGraph->Vertex[Vin].TConfig.NearData = NearData;

fclose(Input_File, "r"); /* read the list of incoming arcs */
if (getc(Input_File) == 93) /* task has no incoming arcs */
    TGraph->Vertex[Vin].Generator = TRUE;
else
    while (getc(Input_File) != 93) /* ship input list */
    {
        fscanf(Input_File, "%i", &n); /* read the list of outgoing arcs */
        while ((n = getc(Input_File)) != 93) /* create record for each neighbor */
        {
            begin
                ungetc(n, Input_File);
                if ((n = fgetc(Input_File, str))) /* read the neighbor's task name */
                    return (1);
                if (strcmp("OUTPUT", str, 6) == 0) /* check if the task is linked to Output */
                    begin
                        if ((n = getc(Input_File)) == 93)
                            ungetc(n, Input_File);
                        continue;
                    end
                neighbor = talloc(Link); /* create new arc for linked list */
                if ((neighbor->SuccTask = Get_ID(str, n)) == NO_VALUE) /* make sure the neighbor
                                                                    task has a task # */
                    begin
                        neighbor->SuccTask = TGraph->Nb_Tasks;
                        strategy(TChar(neighbor->SuccTask) Name, str, n);
                        TChar(neighbor->SuccTask).Size = n;
                    end
                neighbor->Next = TGraph->Vertex[Vin].Adjacent; /* add structure to linked list */
                TGraph->Vertex[Vin].Adjacent = neighbor;
                neighbor->Delay = NO_VALUE; /* initialize delay value to arc */
                Is_Entry(neighbor->SuccTask) = FALSE; /* neighbor is not an entry vertex */
                TGraph->Vertex[neighbor->SuccTask].Nb_In++; /* increment nbr of incoming arc */
                /* for neighbor */
                if ((n = getc(Input_File)) == 93)
                    ungetc(n, Input_File);
            end
        }
        fclose(Input_File, "r"); /* ship to the next line of input file */
    }
end

for (i = 1, j = TGraph->Nb_Tasks, j++) /* find the entry vertices of task graph */
    if (Is_Entry(j))
        begin
            neighbor = talloc(Link); /* create n+1 arc for linked list */

```

```

        neighbor->Next = TGraph->Vertex[0].Adjacent; /* add structure to linked list */
        TGraph->Vertex[0].Adjacent = neighbor; /* initialize of head of arc */
        /* initialize delay value of arc */
        TGraph->Vertex[j].Nb_In++;
    end

for (i = 0, j < TGraph->Nb_Tasks, j++) /* find the exit vertices of task graph */
    if (TGraph->Vertex[j].Adjacent == NULL) /* and link them to dummy exit vertex */
        begin
            neighbor = talloc(Link); /* create new arc for linked list */
            neighbor->Next = NULL; /* add arc from vertex to dummy exit vertex */
            neighbor->SuccTask = TGraph->Nb_Tasks;
            neighbor->Delay = NO_VALUE;
            TGraph->Vertex[j].Adjacent = neighbor;
            TGraph->Vertex[TGraph->Nb_Tasks].Nb_In++;
        end

Terminate_TGraph(TGraph); /* initialize dummy exit vertex */
TGraph->Exit_Ver = TGraph->Nb_Tasks;
TGraph->Nb_Tasks++;
fclose(Input_File);
return (0);
end

/******
Macros
*****
#define Get_Label(vtx, lbl) /* returns lbl, pair v, v
    lbl Value = TGraph->Vertex[vtx].Lbl_Value; \
    lbl Origin = TGraph->Vertex[vtx].Lbl_Origin

#define Put_Label(vtx, lbl) /* assigns lbl, pair v, v
    TGraph->Vertex[vtx].Lbl_Value = lbl_Value; \
    TGraph->Vertex[vtx].Lbl_Origin = lbl_Origin

#define Reset_Near_DB
Reset_Solution_Init (SOriginal_TGraph, APGraph, ASched); \
Schedule_Init(SOriginal_TGraph, APGraph, ASched); \
Init_PGraph(APGraph)

/******
Included files See each file for more information on their contents
*****
#include "ROAD_crp.c" /* critical path functions */
#include "ROAD_ach.c" /* scheduling functions */
#include "ROAD_bnd.c" /* lower bound functions */
#include "ROAD_trc.c" /* trace functions */
#include "ROAD_Brs.c" /* branch and bound functions */

```

```

Jul 13 15 54 1990 depth ROM/ROAD/ROAD_gff.c Page 7

#include "ROAD_bin.c"
#include "ROAD_clu.c"

int argc;
char *argv[];

begin
  TaskGraph TGraph, Original_TGraph, TGraphC; /* task graph structures */
  ProcGraph PGraph; /* processor graph structure */
  Schedule Sched; /* schedule structure */
  PtrLine ptr, Get_Arc(); /* pointers to arc structures */
  char *in_file; /* input filename */
  int i, j, k; /* counters */
  int nProc; /* argument number of filename */
  int SumPT, SumDM, SumPM; /* summation variables */
  time_t time1; /* returns processing time */
  time_t tot_time; /* processing time in CPU seconds */
  struct time1, tm 2; /* time blocks */
  float Theoretical_Min_Proc(); /* function */
  int Correspond1 [MAX_TASKS][2]; /* correspondence matrix for generator */
  int Correspond2 [MAX_TASKS][2]; /* correspondence matrix for clustered */
  int Clus [MAX_TASKS]; /* tasks recovery */
  boolean DoClustering; /* TRUE if clustering to be done */
  boolean NoGenerators; /* indicate if generators must be removed from task graph */

  char *Format = "Usage: road -i (input file) [-p] [-g] [-c] [-f] [-t] [%s] [-a %d] [-e %s]\n";
  /* default: do not optimise */
  /* default: do not remove generators */
  /* default: no clustering */
  /* default: full search */
  /* default: no auxilliary file */
  /* default: no limit on # processors */
  /* default: no dump of search tree */
  /* default: search for optimal */
  /* minimum of 2 arguments (e filename) must be specified */

  ProcOpt = FALSE;
  NoGenerators = FALSE;
  DoClustering = FALSE;
  NoOptProc = FALSE;
  AuxFile = FALSE;
  LLimitProc = 0.0;
  TrDepth = 0.0;
  ApproxError = 0.0;

  if (argc < 3)
  begin
    printf (stderr, Format);
    fflush (stdout);
    exit(1);
  end

  /* read in all call arguments and setup corresponding parameters */
  j = 1;
  while (j < argc)
  begin
    if (strcmp (argv[j], "-p", 2) == 0) /* perform processor optimisation */
      ProcOpt = TRUE;

```

```

    else if (strcmp (argv[j], "-g", 2) == 0) /* remove generator if ready */
      NoGenerators = TRUE;
    else if (strcmp (argv[j], "-c", 2) == 0) /* perform clustering */
      DoClustering = TRUE;
    else if (strcmp (argv[j], "-a", 2) == 0) /* set upper bound on # of processors */
      begin
        LimitProc = TRUE;
      end;
    if (j == argc)
    begin
      printf (stderr, Format);
      fflush (stdout);
      exit(1);
    end
    PLimit = atoi (argv[j]); /* read in the max # of processors specified */
    if ((PLimit > MAX_PROCESSORS) || (PLimit < 1)) /* validate processor limit */
    begin
      printf (stderr, "ERROR invalid number of processors specified\n");
      printf (stderr, "(min = 1, max = %d)\n", MAX_PROCESSORS);
      printf (stderr, Format);
      fflush (stdout);
      exit(1);
    end
    else if (strcmp (argv[j], "-e", 2) == 0) /* set approximation error */
    begin
      j++; /* get % value */
      if (j == argc)
      begin
        printf (stderr, Format);
        fflush (stdout);
        exit(1);
      end
    end
    ApproxError = (float) atoi (argv[j]); /* read in specified approximation error */
    if ((ApproxError > 100.0) || (PLimit < 1.0)) /* validate specified approximation error */
    begin
      printf (stderr, "ERROR approximation acceptance error is too high\n");
      printf (stderr, "in the integer range 1 to 100\n");
      printf (stderr, Format);
      fflush (stdout);
      exit(1);
    end
  end
end

```

```

Jul 13 15:54 1990 depth ROAD/ROAD.grf c Page 8

ApproxEror /= 100.0;
/* convert to ratio */
end
else if (strncmp (argv[3], "-l", 2) == 0) /* do not consider intermediate */
/* empty processors */
NoEmptyProc = TRUE;
else if (strncmp (argv[3], "-t", 2) == 0) /* generate an auxiliary trace file */
begin
OutTrace = TRUE;
/* determine trace depth in search */
if ((i) == argc) || (strncmp (argv[3], "-d", 1) == 0)
j = 1;
else
TrcDepth = ((float) atoi (argv[3])) / 100;
TrcDepth = max (min (TrcDepth, 1.0), 0.0);
end
else if (strncmp (argv[3], "-i", 2) == 0) /* input filename */
begin
INFile = j;
/* used again for trace file name */
/* read the task graph from the specified input file */
if ((i) >= argc) || (head_Tasks (sOriginalTGraph, argv[3]) == 0)
begin
fprintf (stderr, Format);
fflush (stdout);
exit(1);
end
else
begin
fprintf (stderr, Format);
fflush (stdout);
exit(1);
end
/* unknown parameter specified */
end
if (iProcOpt)
NoGenerators = FALSE;
/* cannot remove generators when */
/* optimizing on # of processors */
/* build trace file name. */
begin
strcpy (TraceFile, argv[INFile]);
strcat (TraceFile, TraceExt);
if ((iTrc = Create(1)) == 0)
/* create the trace file */
exit(1);
end

```

```

Copy TGraph (sOriginalTGraph, sTGraph); /* make a copy of task graph for graph */
/* as a working copy */
if (NoGenerators)
RemoveGenerators (sTGraph, Correspond); /* remove generator streams */
/* print input filename and the number of tasks read */
printf ("Task Graph\n\n-----\n\n");
printf ("Filename\n\n");
printf ("Number of tasks in original graph\n\n");
/* print the number of remaining tasks after removing generator streams */
if (NoGenerators)
printf ("Number of tasks after removing generator streams\n\n");
Init PGraph (sPGraph); /* initialize p-processor graph */
/* print the number of clusters identified */
if (NoClustering)
begin
Copy TGraph (sTGraph, sTGraphNC);
Clustering (sTGraph, sPGraph, Correspond);
printf ("Number of tasks after clustering\n\n");
if (TGraphC Nb Tasks == TGraph Nb Tasks)
begin
printf ("FATAL ERROR: NO TASKS CLUSTERED\n\n");
exit(1);
end
end
/* print search setups */
if (OutTrace)
begin
TrcDepth = (TGraph Nb Tasks - 2) * TrcDepth;
printf ("Trace enable with trace depth of tasks\n\n");
end
if (NoEmptyProc)
printf ("No empty processors are used in part of search\n\n");
printf ("Task graph\n\n");
fflush (stdout);
PGraph Nb Processors = min (MAX_PROCESSORS, TGraph Nb Tasks);

```



```

Jul 13 15.54 1990 depth ROAD/ROAD.grf.c Page 10

fflush(stdout);
/* flush prints to output */
tot_time = tot_2_time + tot_1_time;
/* compute used CPU time */
/* print statistics on required CPU time and size of search tree visited */
printf("\n Statistics\n\n");
printf(" Processing time\n\n");
printf(" Total number of nodes generated\n\n");
printf(" Total number of nodes visited\n\n");
printf(" Total number of solutions evaluated\n\n");
printf("\n");
fflush(stdout);
/* flush prints to output */
end

```

Jun 7 13:13 1990 depth ROAD/ROAD_sch.c Page 1

```

/*****
FILE:      ROAD_sch.c

DESCRIPTION:
The functions found in the file are used to compute the total
delay of a specified assignment. The assignment is given by the
Sched data structure, which contains the assignment vector. The
first two functions (Schedule_Init and Best_Solution_Init) are used to
initialise the Sched data structure. The function Compute_Delay computes
the delay between two adjacent tasks assigned according to the
respective processor. Finally, Eval_Assignment is used to compute the
total delay for a complete assignment. The total delay is computed
using the CPM (critical path method).
*****/

```

```

MACROS
Enough_Room: returns TRUE if there is enough room in the specified
processor to hold the specified task; returns FALSE otherwise

```

```

/* n_Task: assigns the specified task to the specified processor,
updating the assignment vector (in Sched) and reducing
correspondingly the amount of resources available in the
specified processor.

```

```

Deassign: unassigns the specified task from the specified processor,
updating the assignment vector (in Sched) and resetting
correspondingly the amount of resources available in the
specified processor.

```

```

Is_Assigned: returns TRUE if the specified task is assigned to some
processor; returns FALSE otherwise.

```

```

PROCEDURES:  Schedule_Init, Best_Solution_Init, Communic_Delay, Eval_Assignment

```

```

AUTHOR:      Marco Lavoie

```

```

DATE:        March 31, 1990

```

```

COPYRIGHT (c) HER MAJESTY THE QUEEN IN RIGHT OF CANADA (1990)
represented by the minister of Defence

```

```

----- History -----

```

```

Date:        Changed By:      Reason.

```

```

/*****
** Name      : Schedule_Init
** Function  : This function initializes the assignment vector of the schedule.
**            The assignment vector is represented by the data structure to which
**            is pointing the argument Sched. The Sched data structure must
**            always be initialized before the search is started in B&B
*****/

```

```

**
** At this early stage, no tasks are assigned, except the dummy entry and
** exit tasks, which represent I/O tasks. These two dummy tasks are assigned
** to the master processor (80) since it is the only processor in the
** architecture which has an I/O port. These dummy tasks do not affect
** the overall assignments (except when computing lower bounds and
** critical paths) since their resource requirements is nil.

```

```

** Arguments : TGraph (i) : pointer to a task graph structure
**            PGraph (i) : pointer to processor architecture structure
**            Sched (i) : pointer to a schedule structure

```

```

** Func. Value : None

```

```

Init
Schedule_Init (TGraph, PGraph, Sched)

```

```

PutTGraph TGraph
PutPGraph PGraph
PutSched Sched
begin

```

```

int i;
/* output */

```

```

for (i = 0; i < TGraph->Nb_Tasks; i++)
Sched->Max_en(i) = NO_VALUE;
/* initialize assignment vector */

```

```

Sched->Assign(TGraph->Entry_Vervec) = PGraph->Input_Processor; /* assign dummy tasks */
Sched->Assign(TGraph->Exit_Vervec) = PGraph->Output_Processor;

```

```

return;

```

```

end

```

```

/*****

```

```

** Name      : Best_Solution_Init

```

```

** Function  : This function is similar to Schedule_Init, except that it initializes
**            the Best_Solution section of the Sched data structure. This structure
**            holds the assignment information for the current best assignment
**            found during search. Every time a better solution is found, the
**            content of Best_Solution is updated. Like for Schedule_Init, the
**            two dummy tasks are assigned to the master processor (80). Furthermore
**            the best delay found (Delay) is set to NO_VALUE, since there is
**            none known a priori.

```

```

** Arguments : TGraph (i) : pointer to a task graph structure
**            PGraph (i) : pointer to processor architecture structure
**            Sched (i) : pointer to a schedule structure

```

```

** Func. Value : None

```

```

Init
Best_Solution_Init (TGraph, PGraph, Sched)

```

```

PutTGraph TGraph
/* pointer to a task graph */

```


J-n 7 13 1990 depth ROAD/ROAD sch.c Page 3

```

begin
  inc
  inc
  p:=link pt;
  for i:=0:3 < TGraph>Nb.Tasks: i++
    for (All_Adjacentes [i], pcr)
      p:=Delay (ptr, Communic_Delay (TGraph>Vertex[ptr>SuccTask].GenatInt,
        Sched>Assign[i]), Sched>Assign[ptr>SuccTask]);
  delay:=Critical Path (TGraph, TGraph>Entry.Vertex); /* evaluate completion delay */
  delay++;
  return (delay);
end

```

```

/* counter */
/* estimated delay between two tasks */
/* pointer to link structure */
/* put the estimated delay on each arc */
/* compute delay for all outgoing arcs */
p:=Delay (ptr, Communic_Delay (TGraph>Vertex[ptr>SuccTask].GenatInt,
  Sched>Assign[i]), Sched>Assign[ptr>SuccTask]);
/* correction for emitting processor 00 */

```

```
nt race_create (t)
begin
    FILE *out_file;
    /* file descriptor */
```

```

Mar 29 16 31 1990 depth ROAD/ROAD_trc.c Page 2

end

fprintf (Out_File, "  N SOLUTION:\n %s\n", task) /* print statistics */
fprintf (Out_File, "    Total number of nodes generated %d\n",
         BlkStats.Total_Nodes);
fprintf (Out_File, "    Total number of nodes visited %d\n",
         BlkStats.Nb_Pursued);
fprintf (Out_File, "    Total number of solutions evaluated %d\n",
         BlkStats.Nb_Solutions);

fprintf (Out_File, "  Delay = %d cycles\n",
         Sched->Best_Solution_Delay); /* print best delay */

fprintf (Out_File, "  Assign %s", task) /* print best assignment */

fprintf (Out_File, "task %s", task);
for (k = 0; k < TChar[1].Size; k++)
    fprintf (TChar[1].Name[k], Out_File);
fprintf (Out_File, " to processor %d\n",
         Sched->Best_Solution_Assign[1]);

for (j = 2; j < TGraph->Nb_Tasks; j++)
begin
    fprintf (Out_File, " task %s", task);
    for (k = 0; k < TChar[1].Size; k++)
        fprintf (TChar[1].Name[k], Out_File);
    fprintf (Out_File, " to processor %d\n",
            Sched->Best_Solution_Assign[j]);
end

fprintf (Out_File, "\n");
fclose (Out_File);

return (0);

end

/* close file for later use */

.....
Name : Trace_Tree
Function : This function prints the partial search tree into the auxiliary trace
          file (depending to the current one).
.....
The argument Depth indicates down to which level the search tree must
be traced, i.e. the trace will cover the assignments of the first
Depth tasks from the ordered list [Sched->SList].

The depth of the current search tree dumped is indicated by the
indentation character ">". For example, suppose the current search
node to be dumped is the fourth task (in the ordered list) named "j".
to be assigned to processor 83, the the following line will be
concatenated to the trace file:

```

```

>>> (1,0).

The brackets indicates to which level the search node is (0-1) and the
pair indicates that the current task (j) is assigned to processor
83).

NOTE: Be careful, is the depth is too large, the whole search tree will
be dumped, which may represent several Kbytes of information, and
consequently slow down the program significantly.

Arguments :
  TGraph (1) : pointer to a task graph structure
  Sched (1) : pointer to the assignment structure
  Depth (1) : indicates down to which depth the search tree will
              be dumped in the trace file
  Func_Value : <0> if no error,
              <1> if cannot access file
.....

int
Trace_Tree (TGraph, Sched, Depth)
/* pointer to task graph */
/* pointer to assignment structure */
/* depth of tree to print */
begin
    FILE *Out_File;
    int i, j, k, n;
    /* to dump file */
    /* counters */
    if ((Out_File = fopen (TraceFile, "a")) == NULL) /* open file for append */
        fprintf (stderr, "ERROR: cannot open auxiliary file %s\n", TraceFile);
    return (1);
end

fprintf (Out_File, " PARTIAL TREE %s",
         n = min (Depth, TGraph->NbTasks);
for (i = 1; i <= n; i++)
    fprintf (Out_File, ">");
j = Sched->SList[n];
fprintf (Out_File, " (%s)",
         for (k = 0; k < TChar[1].Size; k++)
             puts (TChar[1].Name[k], Out_File);
    fprintf (Out_File, " %d\n", Sched->Assign[j]);
fprintf (Out_File, "\n");
fclose (Out_File);

return (0);

end

```

Mar 29 15:36 1990 depth_RDAD/RDAD_crp.c Page 1

```

.....
FILE: RDAD_crp.c
.....

DESCRIPTION:
The functions found in this file constitute the CPM (Critical Path Method)
algorithm. It is based on a revised version of the famous Dijkstra's
algorithm for shortest paths (see headings of function Critical_Path for
details on the required modifications to the original algorithm in order
to search for critical paths).

Since Dijkstra's algorithm makes use of sets of vertices, the first few
functions are tools implementing the set of vertices M (called Mcp).
Mcp_Triang performs the triangular operation found in Dijkstra's
algorithm, and Critical_Path applies the triangular operation iteratively
until the critical path is found.

NOTE: In order to find a critical path, a delay must be assigned to each
arc of the task graph. Therefore these procedures cannot be used
to find a lower bound on the critical path length for a partial
assignment (unless all unknown delays are fixed to 0 externally).
For efficiency, no such validation is performed, so make sure
all delays are defined before calling Critical_Path.

DATA TYPE:
VSet: data structure which represents the set of vertices. It has an
element position for each vertex (vector Vertices), which is set
to TRUE if FALSE whether or not the vertex is in the set or not.
McpOut indicates the number of vertices not in the set (in the
algorithm, the search is pursued until all vertices are in the
Mcp, so that's why we count the number of tasks outside the set
instead of the number inside the set).

MACROS:
Add_Mcp: adds the specified vertex to the set Mcp.
Remove_Mcp: removes the specified vertex from the set Mcp.
In_Mcp: Returns TRUE if the specified vertex is in set Mcp; otherwise it
returns FALSE.

PROCEDURES:
Mcp_Init, Mcp_Max, Mcp_Triang, Critical_Path

AUTHOR:
Marco Lavoie

DATE:
March 31, 1990

COPYRIGHT (c) HER MAJESTY THE QUEEN IN RIGHT OF CANADA (1990)
represented by the minister of Defence

..... History .....
Date: Changed By: Reason:
.....

.....
/*
** Data structures for critical path algorithm
**
**.....
typedef struct sMcp {
    /* represents a set of vertices */
    boolean Vertices[MAX_TASKS];
    /* vertex flags (TRUE => in set) */
    int McpOut;
    /* number of vertices not in Mcp */
} Mcp;

/*
** Macros
**
**.....
#define Add_Mcp(vtx) \
    Mcp.Vertices[vtx] = TRUE; \
    Mcp.McpOut--;

#define Remove_Mcp(vtx) \
    Mcp.Vertices[vtx] = FALSE; \
    Mcp.McpOut++;

#define In_Mcp(vtx) (Mcp.Vertices[vtx]) /* returns TRUE is vtx in Mcp */

/*
** Global structure Mcp, to represent the set
**
**.....
VSet Mcp;

/*
** Name : Mcp_Init
** Function : This function initializes the set vector (Mcp) according to the task
graph. It is initialized such that the critical path is a path with
the source vertex Source. All vertices accessible from Source
(including Source) are retrieved from Mcp. Note that all vertices are
initialized as being in Mcp before the search starts (see if
Critical_Path).
** Arguments : TGraph (t) : pointer to a Task Graph structure
Source (i) : vertex which is considered as source of critical path
algorithm
** Func Value : None
**
**.....
int
Mcp_Init (TGraph Source)

```


Mar 29 15 34 1990 depth NCAD/ADAD.crp.c Page 2

```

/* pointer to graph */
/* Source vertex */

begin
  PtrToGraph TGraph;
  int Source;

  PtrLink ptr;
  if (In_Mcp (Source))
  begin
    Remove_Mcp (Source);
    for (All_Adjacent (Source, ptr))
      Mcp_Init (TGraph, ptr->SuccTask);
  end
end
return
end

/*.....*/
** Name      : Mcp_Max
** Function   : This function finds the highest label value among all vertices not
                included in Mcp. It returns the corresponding vertex number.
** Arguments  : TGraph (I) : pointer to a Task Graph structure.
** Func Value : Vertex number not in Mcp with highest label value
/*.....*/

int
Mcp_Max (TGraph)
PtrToGraph TGraph;
begin
  int j;
  int Max;
  Label p;
  int Vtx;

  Max = NO_VALUE;
  for (j = 0; j < TGraph->Nb_Tasks; j++)
    if (In_Mcp (j))
      begin
        Get_Label (j, p);
        if (p.Value > Max)
          begin
            Vtx = j;
            Max = p.Value;
          end
      end
  return (Vtx);
end

/*.....*/
** Name      : Mcp_Triang
** Function   : This function performs the triangular operation of Djikstra's algorithm.
                Given a vertex number, it goes through all reachable vertices and checks
                whether the corresponding label values can be improved (i.e., up)
                the current vertex. If so the labels are updated.
** Arguments  : TGraph (I) : pointer to a Task Graph structure
                Vertex (I) : vertex number on which must be labeled (i.e., a, b, c)
** Func Value : None
/*.....*/

int
Mcp_Triang (TGraph, Vtx)
PtrToGraph TGraph;
int Vtx;
begin
  int j;
  Label p;
  Label q;
  int w;
  PtrLink ptr;
  Get_Label (Vtx, q)
  for (All_Adjacent (Vtx, ptr))
    begin
      Get_Label (ptr->SuccTask, p);
      w = Get_Delay (ptr);
      if (p.Value < (q.Value + w))
        begin
          p.Value = q.Value + w;
          p-Origin = Vtx;
          Put_Label (ptr->SuccTask, p)
          if (In_Mcp (ptr->SuccTask))
            Remove_Mcp (ptr->SuccTask);
        end
      end
  return;
end

/*.....*/
** Name      : Critical Path
** Function   : This function finds the critical path of the graph. It uses Djikstra's algorithm multiple times. In each pass, only the
                modifications are made to the graph.
/*.....*/

```

```

**
**      - check for higher label instead of lower one in the
**      triangular operation.
**
**      - during triangular operation, check all vertices
**      (the ones already in set M as the ones outside):
**
**      - if the label of a vertex v in M is modified, remove
**      v from M for further considerations
**
** The routine can start from any vertex in the task graph. This algorithm
** perform in O(n^3).
**
**      Arg_verts : TGraph (t) : pointer to the task graph structure;
**      Source (t) : starting vertex.
**
** Func Value : critical path length from Source to exit vertex of task graph
**
*****
int
Critical_Path (TGraph, Source)
/* pointer to graph */
/* starting vertex */

begin
  int j;
  int MaxV;
  Ptl_rk pcr;
  Label p;

  for (j = 0; j < TGraph->Nb_Tasks; j++)
    begin
      Mcp_Vertex[j] = TRUE;
      p.Value = p.Origin = NO_VALUE;
      Put_Label (j, p);
    end

  Mcp_McOut = 0;

  Mcp_Init (TGraph, Source);

  p.Value = 0;
  p.Origin = Source;
  Put_Label (Source, p);

  for (all_Adjacents (Source, pcr))
    begin
      p.Value = Get_Delay (pcr);
      Put_Label (pcr->DestTask, p);
    end

  Add_Mcp (Source);

  while (Mcp_McOut > 0)
    begin
      MaxV = Mcp_Max (TGraph);
      Add_Mcp (MaxV);
      Mcp_Triang (TGraph, MaxV)

```

```

end
return (TGraph->Vertex[TGraph->Exit_Vertex] - 1).Value;
end

```

```

.....
FILE-      ROAD_civ.c
DESCRIPTION- This file contains two groups of functions
             - functions for clustering;
             - functions for handling generator streams

The functions for clustering do perform any clustering up to now, the
clustering procedure only consider as "clusterable" pipelines of
more than three tasks. Dynamic clustering algorithm will have to
be designed in order to cluster these tasks without pruning any
optimal solution. The current clustering procedure may prune the
optimal.

The other group of tasks are used to remove generator streams from
the task graph and to put them back in. Remove Generators is used
to remove the generator tasks before searching for an optimal
assignment. Once the optimal assignment is found by B&B, the
generator tasks are packed back on the optimal assignment by
Put_Generators_Schedule. These functions insure that the removal
and recovery of generator tasks is done without any pruning of
optimal assignments

DATA TYPE   Current_Label: current new label value.

NAMES       NoOnStream: Indicates whether the specified task is on a pipeline or not

PROCEDURES- Cluster_Init, Cluster_Labelling, Identify Clusters, Copy_TGraph,
              Mark_NoGenerators, Mark_Generators, Remove_Generators,
              Put_Generators_Schedule,

AUTHOR      Marco Laviole

DATE        March 31, 1990

COPYRIGHT (c) HER MAJESTY THE QUEEN IN RIGHT OF CANADA (1990)
represented by the Minister of Defence

----- History -----

Date:         Changed By:          Reason:

.....
%*%
%*%   Macros.
%*%
define NoOnStream(vtx)    ((TGraph->Vertex[vtx].No_In > 3) || \
                          (vtx == TGraph->Exit_Vertice)) || \
.....

```

Jul 13 16 16 1990 depth ROAD/ROAD clu.c Page 2

```

end

/*.....*/
** Name      : Cluster_Labeling
**
** Function   : This labelling algorithm identifies the tasks which
**              form a pipeline of three or more tasks (cluster). No
**              validation is performed according to critical path
**              restrictions.
**
** Arguments  : TGraph (I/O) : pointer to task graph structure;
**
** Func Value : None.
**
/*.....*/

int
Cluster_Labeling (TGraph)
PtrTGraph TGraph;

begin
    int      Vtx;
    boolean Found;

    int      Nb_In (MAX_TASKS);
    PtrLink ptx;
    Label    lbl;

    Current_Label++;
    TGraph->Vertexes[TGraph->Entry_Ver] [Nb_In] = Current_Label;

    for (Vtx = 0; Vtx < TGraph->Nb_Tasks; Vtx++)
        Nb_In[Vtx] = TGraph->Vertex[Vtx] Nb_In;

    for (.)
        begin
            Found = FALSE;

            for (Vtx = 0; Vtx < TGraph->Nb_Tasks; Vtx++)
                if (Nb_In[Vtx] == 0)
                    begin
                        Found = TRUE;
                        Nb_In[Vtx] = NO_VALUE;
                        break;
                    end

            if (!Found)
                break;

            for (All_Adjacentes (Vtx, ptx))
                begin
                    Nb_In[ptx->NextTask]--;
                    if (Nb_In[ptx->NextTask] == 0)
                        continue;
                end

            if (NotOnStream (Vtx))

```

```

        begin
            Current_Label++;
            lbl.Value = Current_Label;

            end

        else if (NotOnStream (ptx->NextTask))
            begin
                Current_Label++;
                lbl.Value = Current_Label;

            end

        else
            lbl.Value = TGraph->Vertex[Vtx] [Nb_In];
            /* adjacent task is a ready */
            keep same label.

            Put_Label (ptx->NextTask, lbl)
            /* put new label on adjacent task */

        end
    end
end

/*.....*/
** Name      : Clustering
**
** Function   : This function identifies valid clusters in the given task graph.
**              It does not perform any clustering of a y form, it only identifies
**              group of tasks which could be clustered, i.e. pipelines of three
**              or more tasks.
**
** Arguments  : TGraph (I/O) : pointer to task graph structure.
**
** Func Value : Number of clusters
**
/*.....*/

int
Identify_Clusters (TGraph)
PtrTGraph TGraph;

begin
    int      i, j, k;
    PtrLink ptx;
    int      In_Cluster (MAX_TASKS);

    Cluster_Init (TGraph);

    Cluster_Labeling (TGraph);

    for (i = 0; i < TGraph->Nb_Tasks; i++)
        begin
            In_Cluster[i] = 0;
            TGraph->Vertex[i] . Cluster = NO_VALUE;
        end

    for (i = 1; i < TGraph->Nb_Tasks; i++)
        In_Cluster[TGraph->Vertex[i] . Lbl_Value]++;

    k = 0

```

```
proper = prr;
```

```

.. Name                                Find Cluster Pair
.. Function
.. : This routine look at the tanks adjacent to Vtx i determine what
.. : tanks are clustered in a single cluster. It is an adjacent to the
.. : feature Join Tanks to build the Cluster Pair Mapping.
..
.. Acquires
.. Graph (1/0) : pointer to tank graph
.. Graph (1) : pointer to precursor data structure
.. Vtx (1) : current vertex
..
.. Stack, Dstack (0) : tank pair to be processed
..
.. Func. Value
.. : 0 if pair found.

```

Jul 13 16 16 1990 depth READ/READ.c Page 4

```

**
**      <0> otherwise.
**
**.....
int
Find_Cluster_Pair (TGraph, PGraph, Vtx, Stack, Dtask)
/*TGraph TGraph:
/*PGraph PGraph:
/*Vtx Vtx:
/*Stack Stack:
/*Dtask Dtask:
begin
  PLink ptr;
  for (All_Adjacent (Vtx, ptr))
  begin
    if (!SameCluster (ptr->SuccTask, Vtx)) &&
    begin {SameInTask (ptr->SuccTask, Vtx))
      *Stack = ptr->SuccTask;
      *Dtask = Vtx;
      return 1;
    end
  else
    Find_Cluster_Pair (TGraph, PGraph, ptr->SuccTask, Stack, Dtask);
  if (*Stack != NO_VALUE) return 1;
  end
  return 0;
end
/* no pair found */

end

/*.....
** Name : Clustering
**
** Function : The main routine for clustering tasks. It calls all other routines
** which themselves perform the clustering operations. Note that
** only pipelined tasks (two or more) are clustered. Furthermore,
** Clustering makes sure that cluster sizes do not exceed processor
** limits. A corresponding matrix is built in order to perform
** unclustering once an assignment is obtained for the clustered task
** graph.
**
** Arguments : TGraph (I/O) : pointer to task graph.
** PGraph (I) : pointer to processor arch.
** Correspond (O) : correspondence matrix
**
** Func. Value : None.
**
**.....
int
Clustering (TGraph, PGraph, Correspond)

```

```

/*TGraph TGraph:
/*PGraph PGraph:
/*Correspond MAX TASKS[12]
int
begin
  Stack, Dtask,
  int k;
  PLink ptr;
  /* vertex numbers */
  /* counters */
  /* pointer to linked list elements */
  if (!Identify_Clusters (TGraph)) == 0) return 0 /* identify what can be clustered */
  Stack = Dtask = NO_VALUE;
  for (j = 0; j < MAX TASKS; j++)
  begin
    RemTask[j] = FALSE;
    Clustered[j] = 0;
  end
  while (find Cluster Pair (TGraph, PGraph, 0, kStack, kUnstack))
  begin
    Join Tasks (TGraph, Stack, Dtask);
    RemTask[Stack] = TRUE;
    Clustered[Unstack] = TRUE;
    Stack = Dtask = NO_VALUE;
  end
  k = 0;
  for (j = 0; j < TGraph->Nb Tasks; j++)
  if (!RemTask[j])
  begin
    Correspond[j][0] = k;
    Correspond[k][j] = j;
    k++;
  end
  else
    Correspond[j][0] = NO_VALUE;
  Correspond[k][j] = NO_VALUE;
  k = 0;
  while (Correspond[k][j] != NO_VALUE)
  begin
    if (Correspond[k][j] != k)
    begin
      TGraph->Vertex[k] = TGraph->Vertex[Correspond[k][j]];
      TGraph->Vertex[Correspond[k][j]] = k;
    end
    k++;
  end
  TGraph->Nb Tasks = k;
  for (j = 0; j < TGraph->Nb Tasks; j++)
  for (All_Adjacent (j, ptr))
    ptr->SuccTask = Correspond[ptr->SuccTask][j];

```

```

.....
** Name      Copy_TGraph
** Function   This function creates an exact copy of a given task graph.
** Arguments  TGraph (I/O) : pointer to task graph.
              NewTGraph (I/O) : pointer to new task graph
** Func Value None
**
**
int Copy_TGraph (TGraph, NewTGraph)
/* pointers to task graph */
begin
  int j;
  FillIn pkr, nempt;
  /* counter */
  /* pointers to link structures */
  NewTGraph->Nb_Tasks = TGraph->Nb_Tasks; /* copy graph size and */
  NewTGraph->Entry_VerTEX = TGraph->Entry_VerTEX; /* dummy tasks */
  NewTGraph->Exit_VerTEX = TGraph->Exit_VerTEX;
  for (j = 0; j < TGraph->Nb_Tasks; j++) /* make a copy of each task */
  begin
    NewTGraph->VerTEX[j].TCConfig = TGraph->VerTEX[j].TCConfig;
    NewTGraph->VerTEX[j].Lib = TGraph->VerTEX[j].Lib;
    NewTGraph->VerTEX[j].Nb_CP = TGraph->VerTEX[j].Nb_CP;
    NewTGraph->VerTEX[j].Nb_In = TGraph->VerTEX[j].Nb_In;
    NewTGraph->VerTEX[j].Cluster = TGraph->VerTEX[j].Cluster;
    NewTGraph->VerTEX[j].Generator = TGraph->VerTEX[j].Generator;
    NewTGraph->VerTEX[j].Adjacent = NULL;
  end
  for (All_Adjacentes (j, pkr)) /* make a copy of the linked list */
  begin
    nempt = tailon (Link);
    nempt->SuccTask = pkr->SuccTask;
    nempt->Delay = pkr->Delay;
    nempt->Next = NewTGraph->VerTEX[j].Adjacent;
    NewTGraph->VerTEX[j].Adjacent = nempt;
  end
  return 0;
end
.....
** Name      Mark_Non_Generators
** Function   This function marks all vertices which are not part
              of a generator stream, and which have successors (direct
              or indirect) of via
.....

```

```

.....
TGraph->Entry_VerTEX = Correspond(TGraph->Entry_VerTEX)[0];
TGraph->Exit_VerTEX = Correspond(TGraph->Exit_VerTEX)[0];
return;
end
.....
** Name      UnClustering
** Function   This routine rebuild the original task graph which was clustered
              by a call to Clustering. It uses the correspondence matrix to
              rebuild the original tasks, and reset the solution in Sched to reflect
              the original tasks.
** Arguments  TGraph (I/O) : pointer to task graph;
              Correspond (I) : correspondence matrix.
              PCGraph (I) : pointer to processor arch;
              Sched (I/O) : pointer to schedule vector.
** Func Value None.
**
int UnClustering (TGraph, Correspond, PCGraph, Sched)
/* pointer to task graph */
/* matrix of correspondence */
/* processor architecture */
/* schedule structure */
/* counter */
/* total delay value */
/* initialize processors */
begin
  int j, k;
  int Delay;
  Init_PCGraph (PCGraph);
  for (j = 0; j < TGraph->Nb_Tasks; j++)
  begin
    k = Correspond (j) [0];
    if (RandTasks (j))
    k = Correspond (Clustered_In (j)) [0];
    Assign_Task (j, Sched->Best_Solution.Assign(k));
  end
  for (j = 0; j < TGraph->Nb_Tasks; j++) /* move back current solution */
  Sched->Best_Solution.Assign(j) = Sched->Assign(j); /*
  Delay = Eval_Assignment (TGraph, PCGraph, Sched);
  if (Delay != Sched->Best_Solution.Delay)
  return 1;
  return 0;
end
.....

```

```

J 1 13 16 16 1990 depth ROAD/ROAD.clu.c Page 6

** Arguments : TGraph (I/O) : pointer to task graph,
** Vtx (I) : vertex number where to start search.
** Func Value : None.
**
**.....
int Mark_Non_Generators (TGraph, Vtx)
/* pointer to task graph */
/* counter */
/* pointer to link structures */
begin
int j;
PtrLink ptr, nextptr;
if ((TGraph>Vtxes[Vtx].Lbl.Value == MARKED) ||
(TGraph>Vtxes[Vtx].Generators))
return 0;
TGraph>Vtxes[Vtx].Lbl.Value = MARKED; /* mark newly visited vertex */
for (All_Adjacentes (Vtx, ptr))
Mark_Non_Generators (TGraph, ptr>SuccTask); /* check successors */
return 0;
end

**.....
** Name : Mark_Generators
** Function : This function marks all vertices which cannot be accessed
** from dummy Entry_Vtxes of the task graph. These tasks are
** part of generator streams, which are not of concern in
** optimal solutions.
** NOTE: Mark_Generators proceed by marking non-generating tasks
** with Mark_Non_Generators and then swapping the marked
** labels in the graph.
** Arguments : TGraph (I/O) : pointer to task graph.
** Func Value : None.
**
**.....
int Mark_Generators (TGraph)
/* pointer to task graph */
/* counter */
/* initialization */
for (j = 0; j < TGraph>Nb_Tasks; j++)
TGraph>Vtxes[j].Lbl.Value = NO_VALUE;
begin
int j;
**.....
Mark_Non_Generators (TGraph, TGraph>Entry_Vtxes);
for (j = 0; j < TGraph>Nb_Tasks; j++)
if (TGraph>Vtxes[j].Lbl.Value != MARKED)
TGraph>Vtxes[j].Generator = TRUE;
return 0;
end

**.....
** Name : Remove_Generators
** Function : This procedure removes the generator streams from the graph
** task graph. The graph is then restructured in order to
** avoid non-used vertex elements in Vtxes (in TGraph). The
** correspondence matrix Correspond is setup to give the
** vertex correspondences between tasks in restructured
** graph and tasks in original task graph
** Arguments : TGraph (I/O) : pointer to task graph
** Correspond (O) : correspondence matrix
** Func Value : None.
**
**.....
int Remove_Generators (TGraph, Correspond)
/* pointer to task graph */
/* matrix of correspondences */
PtrTGraph TGraph;
int Correspond [MAX_TASKS][2];
begin
int j, k;
PtrLink ptr, nextptr;
/* pointers to link structures */
Mark_Generators (TGraph);
k = 0;
for (j = 0; j < TGraph>Nb_Tasks; j++)
if (TGraph>Vtxes[j].Generator)
begin
Correspond[j][0] = k;
Correspond[k][1] = j;
k++;
end
else
Correspond[j][0] = NO_VALUE;
Correspond[k][1] = NO_VALUE;
for (j = 0; j < TGraph>Nb_Tasks; j++)
if (TGraph>Vtxes[j].Generator)
begin
ptr = TGraph>Vtxes[j].Adjacent;
while (ptr != NULL)
begin;
end;
end
end
end

```



```

TCGraph->Vertices[ptr->SuccTask] Nb In =
nextptr = ptr->Next; /* release memory */
free(ptr);
ptr = nextptr;
end

TCGraph->Vertices[j].Adjacent = NULL;
end

while (Correspond[n][j] != NO_VALUE) /* restructure Vertices vector */
begin; i := Correspond[n][j]; i := h;
TCGraph->Vertices[h] = TCGraph->Vertices[Correspond[n][j]];
TCGraph->Vertices[Correspond[n][j]].Adjacent = NULL;
end;

h++;
end;

TCGraph->Nb_Tasks = h; /* new graph size */

for i = 0; i < TCGraph->Nb_Tasks; i++) /* renumber tasks in links */
for (All Adjacent {j, ptr})
ptr->SuccTask = Correspond[ptr->SuccTask][0];

TCGraph->Entry_Verxes = Correspond[TCGraph->Entry_Verxes][0];
TCGraph->Exit_Verxes = Correspond[TCGraph->Exit_Verxes][0];

for i = 0; i < TCGraph->Nb_Tasks; i++) /* delete unnecessary links left */
begin
nextptr = TCGraph->Vertices[j].Adjacent;
ptr = NULL;
while (nextptr != NULL)
if (nextptr->SuccTask == NO_VALUE)
begin
if (ptr == NULL)
TCGraph->Vertices[j].Adjacent = nextptr->Next;
else
ptr->Next = nextptr->Next;
free(nextptr);
if (ptr == NULL)
nextptr = TCGraph->Vertices[j].Adjacent; /* next link */
else
nextptr = ptr->Next;
end
else
begin
ptr = nextptr;
nextptr = nextptr->Next;
end
end

return 0;
end

```



```

int
    Main(TGraph TCgraph)
{
    TGraph TCgraph;

    begin
        PtrLink ptr, nnextptr;
        int j;

        for i = 0; i < TCgraph>Nb_Tasks; i++)
            begin
                ptr = TCgraph>Vertex[i].Adjacent;
                while (ptr != NULL)
                    begin
                        nnextptr = ptr->Nnext;
                        free(ptr);
                        ptr = nnextptr;
                    end
            end
        end

        .....

        Name          : Construct_List_Assignment
        Function       : This function constructs the ordered list according to the
                          priorities as, ned to the tasks by the list scheduling
                          heuristic (CF/MISF or CF/LPFF). After calling the list
                          scheduling heuristic, a call to this function will order the
                          tasks in the list Sched->List in decreasing order of
                          priorities as assigned in labels of vertices in the task
                          graph.

                          Furthermore, this function also constructs an initial assignment
                          by pecking the tasks in the ordered list to successive processors
                          in the processor architecture. This assignment corresponds to
                          the initial solution to be used by the branch-and-bound procedure.

        Arguments     : TCgraph (1) : pointer to original task graph structure-
                          BGraph (1) : pointer to reverse graph,
                          PCgraph (1) : pointer to processor structure,
                          Sched (1/0) : pointer to list schedule and assignment structure

        Func. Value   : None.

        .....

    int Construct_List_Assignment (TCgraph, BGraph, PCgraph, Sched)
    {
        PtrTCgraph TCgraph;
        PtrBGraph BGraph;
        PtrPCgraph PCgraph;
        PtrSched Sched;

        begin
            Labels [MAX_TASKS];
            int i, k;

            .....

            PtrTCgraph TCgraph; /* pointers to task graph */
            PtrBGraph BGraph; /* pointer to architecture */
            PtrPCgraph PCgraph; /* pointer to schedule structure */

            .....

            /* temporary array for label values */
            /* counters */
            .....
        }
}

```

```

04/25/14 03:14:00 depth_READ/ROAD bnd c Page 3
.....
** Name : CP_MISF
** Function : This function finds an initial assignment with the task graph
              TGraph and the processor architecture PGraph. The routine
              is based on CP/MISF (Critical-Path / Most Immediate
              Successors First), which is a list scheduling algorithm
              where priorities are assigned according to the number of
              immediate successors of tasks. The found solution is
              copied in Sched structure (argument).
** Arguments : TGraph (t) : pointer to task graph structure;
              PGraph (p) : pointer to processor architecture structure;
              Sched (l/o) : pointer to list schedule and assignment structure.
** Func. Value : None.
.....
int
CP_MISF (TGraph, PGraph, Sched)
{
    PGraph PGraph;
    TGraph TGraph;
    PGraph Sched;

    begin
        Task_Graph BGraph;
        int j, k;
        int Vmax;
        int Nmax;
        PGraph ptr, nnext;

        /*
        .....
        /* pointer to processor graph */
        /* pointer to task graph */
        /* pointer to schedule structure */

        /* reversed task graph */
        /* counters */
        /* vertex number */
        /* new label */
        /* pointers to arc structures */

        /*
        .....
        /* Initialisation of schedule structure and task graph labels.
        .....
        */

        Schedule_Init (TGraph, PGraph, Sched);

        for (j = 0; j < TGraph->Nb_Tasks; j++)
            TGraph->Vertex[j].Lbl.Value = NO_VALUE;

        /*
        .....
        /* constructing the reversed task graph BGraph.
        .....
        */

        Build_Reversed (TGraph, BGraph);

        /*
        .....
        /* Calculating priorities among vertices, according to the
        .....
        /* number of immediate successors in TGraph
        .....

```

```

Apr 25 14:01 1990 depth_R0AD/R0AD_hnd.c Page 4

**      Vtx (i) : vertex i to label, together with its descendants.
**      Level (i) : level value to assign to vertex
**
** Func Value : None.
**
int
LPFF_Labels (TGraph, Vtx, Level,
             /* pointer to original task graph */
             /* current vertex to label */
             /* current level value */)
begin
    PtrLink ptr;
    if ((TGraph->Vertex[Vtx].Lbl.Value != NO_VALUE) &&
        (TGraph->Vertex[Vtx].Lbl.Value <= Level)) return
    Level := TGraph->Vertex[Vtx].TConfig.ProcTime;
    TGraph->Vertex[Vtx].Lbl.Value = Level;
    for (All Adjacents (Vtx, ptr))
        LPFF_Labels (TGraph, ptr->SuccTask, Level)
end

/*
** Name : LPFF_Labels
** Function : This function visits the given task graph in a depth-first manner,
**            and assigns to each task a label which corresponds to the task's
**            level (depth) in the task graph. This function is used by CP_LPF
**            to calculate priorities to be assigned to tasks (priorities are
**            based partially on task depth).
** Arguments : TGraph (i) : pointer to task graph structure;
**            Vtx (i) : vertex i to label, together with its descendants;
**            Level (i/0) : level value to assign to vertex.
** Func. Value : None.
**
int
LPFF_Labels (TGraph, Vtx, Level)
/* pointer to original task graph */
/* current vertex to label */
/* current level value */
begin
    PtrLink ptr;
    TGraph->Vertex[Vtx].Lbl.Value = *Level;
    /* assign level to current task */

```

```

(*Level)**
for (All Adjacents (Vtx, ptr))
    LPFF_Labels (TGraph, ptr->SuccTask, Level)
end

/*
** Name : CP_LPF
** Function : This function finds an initial solution to a R0AD problem
**            given a task graph and a processor architecture. The solution
**            is based on CP/LPF (Critical Path / Longest feasible
**            Path First), which is a list scheduling algorithm
**            where priorities are assigned according to the current
**            longest path of unassigned tasks. The found solution is
**            copied in Sched structure (larg-mem).
** Arguments : TGraph (i) : pointer to task graph structure
**            PCGraph (i) : pointer to processor architecture structure
**            Sched (i/0) : pointer to list schedule and assignment structure
** Func. Value : None
**
int
CP_LPF (TGraph, PCGraph, Sched)
PtrGraph PCGraph;
PtrTGraph TGraph;
PtrSched Sched;
begin
    Task_Graph BGGraph;
    int i, j, k, l;
    int Vtx;
    int NewLbl;
    int VtxLbl = 0;
    int SortVec [MAX_TASKS];
    PtrLink ptr, nextr;
    /*
    ** Initialization of schedule structure and task graph labels
    **
    **
    Schedule Init (TGraph, PCGraph, Sched);
    for (i = 0; i < TGraph->NbTasks; i++)
        TGraph->Vertex[i].Lbl.Value = NO_VALUE
    */
    */

```

```

Apr 25 16 03 1990 depth ROAD/ROAD_bnd.c Page 5

** Assign levels to tasks.
****/

LPPF_Labels (TGraph, BkGraph, Entry_VerTEX, AVLbl).
**
** Constructing the list schedule according to priorities estimated
** (label), and the corresponding assignment
****/

/*****
**
** Construct List Assignment (TGraph, BkGraph, PGraph, Sched).
*****/
** Release memory used by BkGraph
*****/

Kill_TGraph (BkGraph).
return.
end

/*****
** Name : Low_Bound_Delay
** Function : This function estimates a lower bound on completion delay
** for a partial assignment. The partial assignment is given
** in Sched structure (argument). The routine finds the
** critical path in the partially assigned task graph and
** then attempt to assign all unassigned tasks and complete the
** critical path, by packing them on successive processors
**
** Arguments : TGraph (1) : pointer to task graph structure.
** PGraph (1) : pointer to processor architecture structure
** Sched (1/0) : pointer to list schedule and assignment structure
**
** Func Value : Estimated lower bound on delay for given partial assignment
*****/

Int Low_Bound_Delay (TGraph, PGraph, Sched)
**
** PGraph PGraph.
** TGraph TGraph.
** PGraph Sched.
begin
  PGraph ptr;
  Int delay;
  Int i, NO in MAX TASKS;
  Int PT in MAX TASKS;
  Int Travel;
  Int Vtx;
  Int Succor;
  Boolean found;
  Label lb;

  /* pointer to processor graph */
  /* pointer to task graph */
  /* pointer to schedule structure */
  /* pointer to arc structure */
  /* estimated delay */
  /* counters */
  /* processing time left */
  /* processing time available */
  /* vertex number */
  /* succeeding vertex of this vertex */
  /* indices of valid arcs */
  /* task structure */

```

```

Apr 25 14:03 1990  daphn ROM/ROMAD_bnd.c Page 6

/*****
** initialize counters;
** *****/
for (i = 0; i < TGraph->Nb_Tasks; i++) /* for all tasks */
begin
  if (Is_Assigned(i))
  PT(i) = PGraph->Processor[Sched->Assign(i)] ProcTime;
else
  PT(i) = PGraph->Config.ProcTime;
  TGraph->Vertex(i).TConfig.ProcTime;
end

Nb_In[i] = TGraph->Vertex(i).Nb_In; /* number of incoming arcs */
TGraph->Vertex(i).Lbl_Value = NO_VALUE; /* initialize task labels */
end

/*****
** find critical path;
** *****/
TGraph->Vertex(TGraph->Entry_VerTEX).Lbl_Value = 1;
TGraph->Vertex(TGraph->Exit_VerTEX).Lbl_Origin = TGraph->Entry_VerTEX;
for (i = 1; i < TGraph->Nb_Tasks; i++) /* repeat until all tasks visited */
begin
  Found = FALSE;
  for (Vtx = 0; Vtx < TGraph->Nb_Tasks; Vtx++) /* find a vertex with all incident */
  if (Nb_In[Vtx] == 0) /* neighbors already visited */
  begin
    Found = TRUE;
    Nb_In[Vtx] = NO_VALUE; /* found a task for which all incident */
    break; /* neighbors has been visited */
  end
  if (!Found)
  break; /* if all tasks visited, finish */
end
for (All_Adjacent (Vtx, ptr)) /* compute delay on each arc in graph */
begin
  Nb_In[ptr->SuccTask]--;
  if (ptr->SuccTask == TGraph->Exit_VerTEX) /* finish */
  begin
    Lbl_Value = TGraph->Vertex[Vtx].Lbl_Value;
    PTime = PT[ptr->SuccTask];
  end
  else if (Is_Assigned(ptr->SuccTask)) /* compute according to Communic Delay */
  begin
    Lbl_Value = TGraph->Vertex[Vtx].Lbl_Value +
      Communic Delay (TGraph->Vertex[ptr->SuccTask].Generator,
        Sched->Assign[Vtx], Sched->Assign[ptr->SuccTask]);
  end
end
PTavail = PTime;
PTavail = PT[ptr->SuccTask];
end
else if (PT[Vtx] > TGraph->Vertex[Vtx].Lbl_Value) /* task unassigned, pack on a */
begin
  Lbl_Value = TGraph->Vertex[Vtx].Lbl_Value; /* pack on same */
  PTime = PT[Vtx] - TGraph->Vertex[ptr->SuccTask].TConfig.ProcTime; /* processor */
end
else
begin
  Lbl_Value = TGraph->Vertex[Vtx].Lbl_Value + 1 /* task unassigned, pack on a */
  PTime = PGraph->Config.ProcTime; /* new processor */
  TGraph->Vertex[ptr->SuccTask].TConfig.ProcTime;
end
if ((Lbl_Value > TGraph->Nb_Tasks) || (Lbl_Value > 1) || (Lbl_Value > TGraph->Vertex[ptr->SuccTask].Lbl_Value) ||
  (PTime < PT[ptr->SuccTask]))
begin
  Lbl_Origin = Vtx;
  Put_Label(ptr->SuccTask, Lbl);
  PT[ptr->SuccTask] = PTime;
end
end
end
/****
**
** The label on the exit vertex corresponds to the length of the critical path
** *****/
return (TGraph->Vertex[TGraph->Exit_VerTEX].Lbl_Value);
end

```

Mar 29 21:27 1990 depth_BROAD/BROAD_Bin.c Page 1

```

/*****
FILE:      BROAD_Bin.c

DESCRIPTION:
The functions in this file are used to estimate a lower bound on the minimum
number of processors required to assign all tasks in the task graph.
The first function computes a bin packing minimum, which is within 21% of
the true minimum (i.e. to get a lower bound on the number of processors
required, reduce the bin-packing solution by 21%).

The last function computes the theoretical minimum number of processors for
a task graph. This minimum is a lower bound on the true minimum and is
usually lower than the bin-packing solution, except for small problems,
where it is often higher than the bin-packing solution.

These lower bound estimates are useful in giving an idea on what the
minimum number of processors might be.

DATA TYPE
TaskVect, *PtrTaskVect: this data structure is used to sort tasks in
decreasing order of processing time requirements. This ordered
list is then used for bin-packing.

PROCEDURES:
Compare_Sizes, Bin_Packing, Theoretical_Min_Proc.

AUTHOR:
Marco Laviole

DATE:
March 31, 1990

COPYRIGHT (c) HER MAJESTY THE QUEEN IN RIGHT OF CANADA (1990)
represented by the Minister of Defence

Date      Changed By:      Reason:
*****/

/*****
** Global Structure.
**
**
typedef struct atbl2
begin
    int Task_Mor;
    int Task_Size;
end TaskVect, *PtrTaskVect;

/*****
** Name      : Compare_Sizes
**
**
*****/

```

```

** function      : This function compares value *a* to value *b*. It is used by the quant
**                function call in Bin_Packing to sort the task vector in decreasing
**                order of task sizes.
**
** Arguments     : a, b (int) : values to compare
**
** Local Value   : 1 if a > b,
**                0 if a = b,
**                -1 if a < b
**
*****/

int
Compare_Sizes (a, b)
PtrTaskVect a, b
begin
    if (a->Task_Size > b->Task_Size)
        return (1);
    if (a->Task_Size == b->Task_Size)
        return (0);
    return (-1);
end

/*****
** Name      : Bin_Packing
**
** Function   : This function finds the minimum number of processors required for
**                the specified task graph by bin packing. The bin packing is based
**                on the processing time parameter of tasks only. First fit decreasing
**                bin-packing is used, which guarantees to give an assignment within 21%
**                of the optimum.
**
** Arguments  : TGraph (int) : pointer to task graph structure
**                PGraph (int) : pointer to processors structure
**                Sched (int) : pointer to schedule structure
**
** Local Value : < Minimum number of processors computed.
**
*****/

int
Bin_Packing (TGraph, PGraph, Sched)
PtrTGraph PGraph,
PtrPGraph PGraph,
PtrSched Sched;
begin
    int i, j;
    TaskVect List [MAX_TASKS];
    int Compare_Sizes();
    int Nb_Proc;

    for (i = 0; i < TGraph->Nb_Tasks; i++)
        List[i] = TGraph->Tasks[i];

```



```

101:
Header_Proc_Used (TGraph, PGraph, Sched)

PGraph TGraph;
PGraph PGraph;
PGraph Sched;

begin
  int sum = 0;
  int i;
  boolean Proc_Used (MAX_PROCESSORS);

  /* pointer to task graph structure */
  /* pointer to processor architecture */
  /* pointer to assignment structure */

  /* counter */
  /* counts the processors used */

```

1000

```

Jul 13 15 58 1990 depth_BDAD/BDAD_BnB.c Page 2

CP_MISF (TGraph, PGraph, Sched); /* find initial solution with CP/MISF */
/*
**
** consider the initial solution as the best found yet
**
***/
Sched->Best_Solution.Delay = Eval_Assignment (TGraph, PGraph, Sched);
Sched->Best_Solution.Lbound = Sched->Best_Solution.Delay / (1.0 + ApproxError);
for (i = 0; i < TGraph->Nb_Tasks; i++)
    Sched->Best_Solution.Assign[i] = Sched->Assign[i];
Sched->Best_Solution.Nb_Proc = Number_Proc_Used (TGraph, PGraph, Sched);

Schedule_Init (TGraph, PGraph, Sched); /* reinitialise assignment vector */
Init_PGraph (PGraph); /* reinitialise processor architecture */

CP_LPFF (TGraph, PGraph, Sched); /* find initial solution with CP/LPFF */
/*
**
** if CP/LPFF solution is better than CP/MISF solution, then consider the new initial
** solution as the best found yet. Note that CP/LPFF ordering list is always used
**
***/
if ((Eval_Assignment (TGraph, PGraph, Sched)) <= Sched->Best_Solution.Delay)
    begin
        Sched->Best_Solution.Delay = Eval_Assignment (TGraph, PGraph, Sched);
        Sched->Best_Solution.Lbound = Sched->Best_Solution.Delay / (1.0 + ApproxError);
        for (i = 0; i < TGraph->Nb_Tasks; i++)
            Sched->Best_Solution.Assign[i] = Sched->Assign[i];
        Sched->Best_Solution.Nb_Proc = Number_Proc_Used (TGraph, PGraph, Sched);
    end

Method = 1;
end

if (LimitProc)
    begin
        Limit = min (PLimit, Sched->Best_Solution.Nb_Proc);
        PGraph->Nb_Processors = min (PLimit, PGraph->Nb_Processors);
    end
/*
**
** Output the maximum number of processors, since it will not be changed anymore.
**
***/
printf ("%n\n MAX NUMBER OF PROCESSORS : %d processors.\n",
        PGraph->Nb_Processors);

```

```

Jul 13 15:58 1990 depth_READ/ROAD_Rdb.c Page 3

return;
end

.....
/* Name : Compare_Delays
...
Function : This function compares value ca_2 to value cb_2. It is
used by the sort routine in RB
...
Arguments : a, b [I] : values to compare
...
Func Value : -1 if a < b;
0 if a = b;
+1 if a > b.
...
*/
.....
int Compare_Delays (a, b)
PtrList a, b;
begin
if (a->Delay < b->Delay)
return (-1);
if (a->Delay == b->Delay)
return (0);
return (1);
end
.....
/* values to compare */
.....
/* Name : Path_Successors
...
Function : This function marks all vertices which are descendants
of the given vertex Vtx. It performs a depth first search
starting at the specified vertex Vtx.
...
Arguments : TGraph [I] : pointer to task graph structure.
Vtx [I] : starting vertex number
...
Func Value : None.
...
*/
.....
int Mark_Successors (TGraph, Vtx)
PtrTGraph TGraph;
int Vtx;
begin
int j;
PtrLink pLk;
/* counter */
/* pointer to Lnk structures */
/* pointer to task graph structure */
/* starting vertex ID */
.....
for (j = 0; j < TGraph->nNbTasks; j++)
if (!TGraph->Vertex[j]) continue;
Mark_Successors(TGraph, Vtx);
if (j == 0) j = TGraph->nNbTasks; j++;
if ((!TGraph->Vertex[j]) || (TGraph->Vertex[j] != MARKED))
Sum += TGraph->Vertex[j];
}
return Sum;
end
.....

```

Jul 13 15 58 1990 depth_BROAD/BROAD_BnB.c Page 4

```

** Func on : This function performs a depth first branch & bound search
** Prior to calling BB, a call to CPMinS or CPUpper must be done
** to determine the current best solution. The call of the list
** ordered list for the BB heuristic also provides BB with an initial assignment,
** for early pruning in the search tree.
**
** The depth-first search is continued until all branching nodes
** have been visited or rejected.
**
** Arguments : IdList (I) : Index value of current task in list schedule.
**              TGraph (I) : pointer to task graph structure;
**              PGraph (I) : pointer to processors structure;
**              Sched (I) : pointer to schedule structure.
**
** Func Value : None.
**
**...../
/*
BB (IdList, TGraph, PGraph, Sched)
int IdList;
PtrPGraph PGraph;
PtrTGraph TGraph;
PtrSched Sched;
begin
  int i;
  int k = 0;
  int next;
  int delay;
  boolean Better, Consider, Reject;
  OrdList list (MAX_PROCESSORS);
  int Compare_Delays();
  int Nb_Skip;
  /*
  /* counter */
  /* next task index to consider */
  /* delay value */
  /* used for [i-p] parameter in argv */
  /* vector containing ordered list */
  /* function used for sorting */
  /* number of empty processors to */
  /* consider if necessary */
  /* increment counter of nodes */
  /* if all tasks assigned */
  /* update search statistics */
  BState Nb_Parused++;
  if (Sched->SList[IdList] == TGraph->Exit_Vertex)
  begin
    BState Nb_Solutions++;
  end
  /*****
  ** Determine how to compare this assignment to the current best one. The
  ** comparison may depend on global options.
  **
  **...../
  Better = ((delay = Eval_Assignment (TGraph, PGraph, Sched))
    < Sched->Best_Solution.Delay);
  if (Better at Prudg)
    Better = (delay == Sched->Best_Solution.Delay) &&
      (NbMax_Proc_Used (TGraph, PGraph, Sched)
        < Sched->Best_Solution.Nb_Proc);
  /*****
  **
  ** Check if the new assignment found is better than the current best assignment.
  **
  **...../
  ** If so, make it the new current best assignment
  **
  **...../
  if (Better)
  begin
    Sched->Best_Solution.Delay = delay;
    Sched->Best_Solution.Evaluated = Sched->Best_Solution.Delay / (1.0 * ApproxPrct);
    for (j = 0; j < TGraph->Nb_Tasks; j++)
      Sched->Best_Solution.Assign[j] = Sched->Best_Sol[j];
    Sched->Best_Solution.Nb_Proc = Number_Proc_Used (TGraph, PGraph, Sched);
    if (Gt_Latency)
      /* if trace on, output a lot of */
      /* diagnostic trace time */
    end
    return;
  end
  k = 0;
  /****
  **
  ** If intermediate empty processors may be considered, determine how many have
  **
  **...../
  if (NoEmptyProc)
  else
    Nb_Skip = 0;
  Nb_Skip = Unassigned_NonSucc (TGraph, Sched, IdList);
  /****
  **
  ** Create the set of active branching search nodes. The set corresponds to the
  ** current task assigned to all processors with enough to hold it and for which
  ** the estimated lower bound on delay is lower (or "lower or equal") than the
  ** current best delay (from current best assignment)
  **
  **...../
  for (j = 0; j < PGraph->Nb_Processors; j++)
  begin
    if ((Enough Room (Sched->SList[IdList], j))
      continue;
    /* ignore processor if not */
    /* enough room */
    BState.Total_Nodes++;
    Assign_Task (Sched->SList[IdList], j)
    /* assign the task */
  end
  /****
  **
  ** Compute the estimated lower bound on total delay for the partial
  ** assignment
  **
  **...../
  if (Prudg)

```

```

Jul 13 15:58 1980 depth_R0AD/ROAD_AnB.c Page 5

    Consider = ((delay = Low_Bound_Delay (TGraph, PGraph, Sched))
               <= Sched->Best_Solution.Ebound);
    else
        Consider = ((delay = Low_Bound_Delay (TGraph, PGraph, Sched))
                   < Sched->Best_Solution.Ebound);

    if (Consider)
        begin
            Lat[h].Delay = delay;
            Lat[h].Processor = j;
            h++;
        end

    Deassign (Sched->SList[idList]),
    /* remove task from assignment */

    if (j != 0) &&
        (PGraph->Processor[j].ProcTime ==
         PGraph->PConfig.ProcTime)
    begin
        if (Nb_Skip == 0)
            break;
        else
            Nb_Skip--;
    end

    end

end

/* do not attempt further assignments */
/* if all following processors are */
/* empty. This rule is only valid */
/* when intermediate empty proc */
/* may not be considered. */
/* attempt assign to this processor */

end

end

/*****
**
** Order the active set of search nodes in decreasing order of lower bound
** function value.
**
** *****/

qsort (Lat, h, sizeof (OrdList), Compare_Delays);

na = idList + 1;
/* assign the next task in list */
/* schedule */

/*****
**
** For all search nodes in the active set, pursue solution growth by depth-first
** search, then backtrack.
**
** *****/

for (j = 0; j < h; j++)
begin
    if (ProcOpt)
        Reject = (Lat[j].Delay > Sched->Best_Solution.Ebound);
    else
        Reject = (Lat[j].Delay >= Sched->Best_Solution.Ebound);
    if (Reject)
        break;

    Assign_Task (Sched->SList[idList], Lat[j].Processor); /* assign task */

    if ((OutTrace) && (idList <= TrcBdDown))
        Trace_Tree (TGraph, Sched, idList); /* Trace partial tree in trace file */

```

```

DB (next, TGraph, PGraph, Sched);
Deassign (Sched->SList[idList]);
end
return;
end
/* pursue search further down */
/* unassign task before backtracking */
/* backtrack */

```

END

1|5|0|4|9 1

FIN