DTIC FILE COPY

(1)

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: DISTRIBUTION STATEMENT A Approved for public release Distribution Unlimited
2b.	
AD-A220 135	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Brown University; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: ONR Resident Representative
6c. ADDRESS (City, State and ZIP Code): Division of Engineering, Box D, Providence, Rhode Island 02912	7b. ADDRESS (City, State and ZIP Code): Harvard Univ., Holyoke Ctr, 2nd Floor, 1350 Massachusetts Ave., Cambridge, MA 02138-4993
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Dept. of the Navy; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N00014-87-K-2023
8c. ADDRESS (City, State and ZIP Code): 4555 Overlook Avenue, SW, Washington, DC 20375	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
	51-1430-88		

11. TITLE (Include Security Classification): "A Development Test Bed and Enhanced Design Environment for AIPS-Based Systems"

12. PERSONAL AUTHOR(S)

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final Technical	FROM 06/1/87 TO 01/31/88		

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

DTIC ELECTE APR 05 1990 S D

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS ☐	

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
D. C. A. Bulterman		

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE

90 04 02 098

SECURITY CLASSIFICATION OF THIS PAGE

A Development Testbed for ALPS-based Systems

Final Technical Report

1 October 1988

Dick C. A. Bulterman, Principal Investigator
Division of Engineering
Brown University
Providence, RI 02912

Grant #N00014-87-K2023
Naval Research Laboratory
Washington, D.C.
Y. S. Wu, Contract Scientific Officer

Table of Contents

ABSTRACT

The work performed under this contract extended and expanded the principal investigator's previous work on defining a distributed architecture model that would support digital signal processing applications. Our efforts were concentrated on two aspects: the development of a reliable simulation facility that would more accurately predict the performance of our distributed system (known as the d-ALPS framework), and the exploration of models of analysis for task distribution and resource allocation within that framework.

The first portion of work work resulted in a set of computer programs that interacted within our existing behavioral model for support d-ALPS applications. The simulator itself is a flexible system that allows a variable degree of report information to be generated for specification execution sequences. It was written to be relatively portable within a UNIX-based environment.

The second portion of this work resulted in a master's-level thesis that discussed analysis methods for d-ALPS architectures in terms of four broad analytical techniques: *static analysis, state generation analysis, schedule simulation,* and *architectural simulation.* Each of these methodologies is developed in terms of a series of relevant application models that highlight the strengths and weaknesses of the d-ALPS approach.

Our original research intentions under this grant were curtailed due to the principal investigator's departure from Brown University and a resulting limiting of funding from the granting agency. It is expected that future work on this project will occur within the Naval Research Laboratory and at the Center for Mathematics and Computer Science, in Amsterdam.

1. Report Overview

This final report will summarize our activity on the NRL-supported research project entitled: *A Development Testbed for ALPS-based Systems* (Grant #N00014-87-K2023). This research was a follow-on contract to our original NRL-sponsored work in grants N00014-85-K2002 and N00014-86-K2015. The research under this grant took place between 1 June 1987 and 31 December 1987.

2. Activity Summary

The work covered under this grant spanned a period of seven months. This period represented a unilateral shortening of the contract period by the contract sponsor. As a result, only a portion of the originally-planned research was carried out under this contract. The work that was performed consisted of two projects: the development of a reliable simulator for ALPS-based systems, and the analysis of ALPS architectures with regard to task distribution and resource allocation. The activity under this grant is summarized in the following two sections. A detailed description of the work involved in presented in Appendix A.

2.1. Updated d-ALPS Simulation Model.

Of the two major thrusts of our work, the first was to prepare an updated simulation facility to conform to the d-ALPS model developed under contract N00014-86-2015. This work, which resulted in a software model of the communications and processing architecture for d-ALPS was completed in December of 1987. The source code and executable versions of this updated simulator have been transferred to NRL, along with the relevant documentation.

The nature of this work precluded any public publications or presentations.

2.2. Task Distribution and Resource Allocation for d-ALPS Systems

Along with the development of a software simulation facility, a project was initiated to study the effect of task distribution and resource modelling within the d-ALPS framework. This work resulted in the development of a Master's thesis, a copy of which is attached as Appendix A. Interested readers are encouraged to consult this comprehensive document.

The work describes four methodologies that were developed to support the investigation of allocation and scheduling problems for a class of distributed processing systems that were based on the d-ALPS model. Two perspectives on scheduling were presented: that of the configurer, who supplies an underlying application and must find a suitable configuration architecture, and second, the systems architect, who is interested in investigating scheduling and allocation issues over the entire class of ALPS applications.

In order to support the work, four investigation approaches were developed: *static analysis*, *state generation*, *schedule simulation*, and *architectural simulation*. Each of these methodologies is developed in terms of a series of relevant application models that highlight the strengths and weaknesses of the d-ALPS approach.

The descriptions of the results of this work depend heavily on the description of the d-ALPS framework. This information, including an overview of the d-ALPS model, are presented in appendix.

3. Personnel

The following personnel were actively engaged in research associated with this grant:

D. C. A. Bulterman, *Principal Investigator:*

Professor Bulterman has been the technical director of the research described in this report.

D. L. Leibholz, *Graduate Student and RA:*

Mr. Leibholz served as the lead technical research staff member on this project and worked directly on the analysis of task distribution and resource allocation aspects of the d-ALPS models. He received his Master's degree in May of 1988.

R. McConnell, *Research Engineer:*

Mr. McConnel wrote the updated d-ALPS simulator and directed independent studies projects concerning a prototype implementation of the d-ALPS network. He joined the technical staff of the NRL is January 1988.

4. Index of Reports and Publications

[1] D. L. Leibholz, "Methods of Analysis of Task Distribution and Resource Allocation in a Class of Distributed Control Multiprocessor Architectures", *Masters Thesis*, Brown University, May 1988.

5. Detailed Activity Reports

5.1. Appendix A

Methods of Analysis of Task Distribution and Resource Allocation in a Class of Distributed Control Multiple Processor Architectures

by

Daniel Leibholz

Sc.B. Brown University *1986*

Thesis

Submitted in partial fulfillment of the requirements for the
Degree of Master of Science in the Division of
Engineering at Brown University

May 1988

Abstract

This thesis describes four methodologies that have been developed to support the investigation of allocation and scheduling problems for a class of distributed processing systems. The class of systems is based on the ALPS (Alternative Low-level Primitive Structures) methodology that uses a distributed and dynamic task distribution mechanism to assign elements of an application algorithm to an architecture consisting of special-purpose processing primitives.

Two perspectives on scheduling and allocation are presented. They are the perspectives of the configurer, who is supplied an application algorithm and an specific underlying ALPS architecture and must find a suitable configuration architecture, and the architect, who is interested in investigating scheduling and allocation issues over the entire class of ALPS architectures and in independence of the application domain. The configurer utilizes application-specific, *explicit* performance criteria, such as task latency and throughput, to guide a configuration. The architect is interested in application-general *implicit* performance criteria to guide the analysis of an existing ALPS architecture and the design of future underlying architectures.

In support of these perspectives, four investigation approaches have been developed. *Static analysis* employs graph expansion and analysis methods to gain insights into an application task graph. *State generation* demonstrates that, provided with gross simplifications of the application task graph and the architecture, optimal solutions can be machine-generated. The computation complexity and lack of extensibility of this method validates the final two methodologies. Schedule simulation provides large-grained comparison of task mapping methodologies. Architectural simulation, as implemented, is high level simulation of an ALPS protocol which is part of the *distributed-ALPS* (d-ALPS) architecture. It provides a configurer with performance evaluation of a configuration allocation and allows an architect to analyze the d-ALPS architecture. The d-ALPS architecture is used as a model of an existing ALPS architectural implementation. This thesis demonstrates, via this model, that the four *general* ALPS investigation methodologies can be applied to the analysis of protocol and representation pathologies of a specific ALPS architecture.

Table of Contents

CHAPTER 1

Introduction

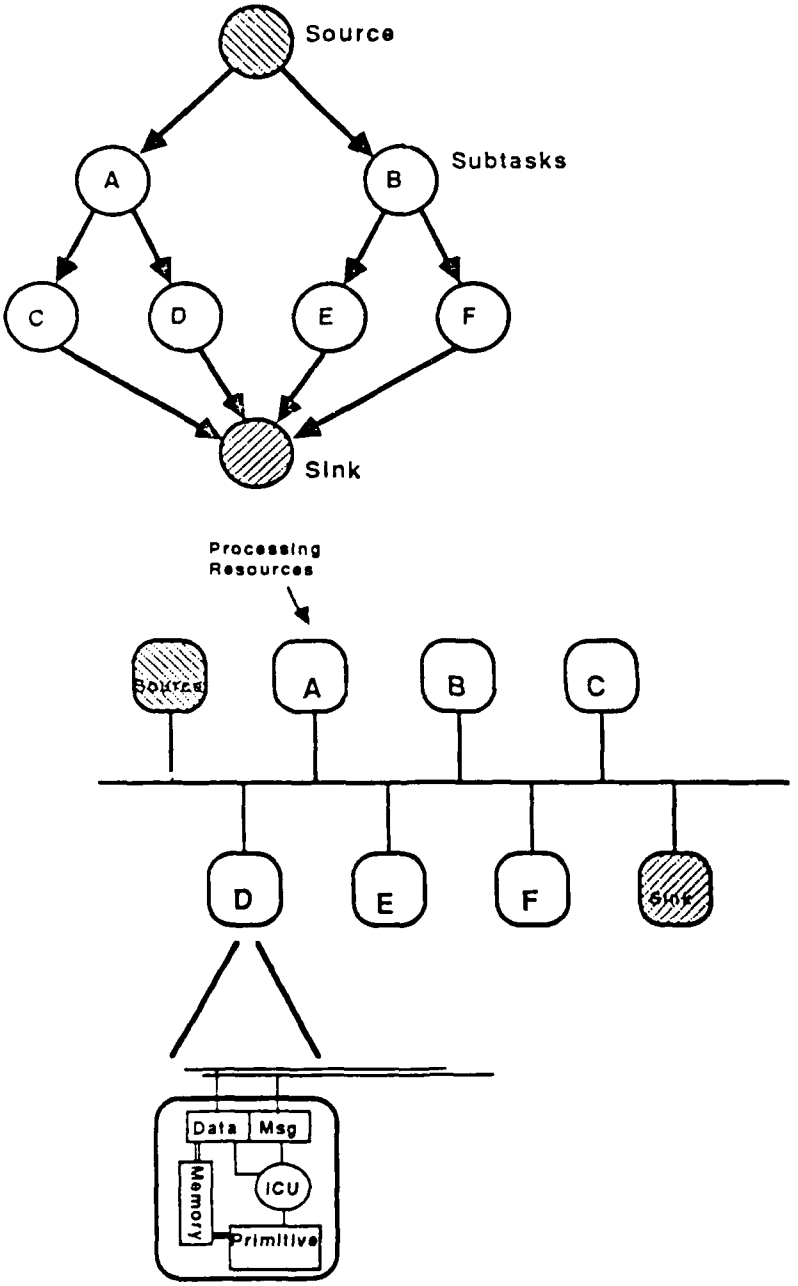
The challenge of building reliable and high speed special-purpose processing systems quickly and at lower costs has motivated a search for faster and more flexible architectures. Certain application domains, such as those which apply digital signal processing (DSP) techniques to sampled data, pose substantial real-time processing demands. These demands are growing as DSP alorithms become more complicated and as developers wish to process data at faster rates.

High speed processing capabilities are enhanced by the development of special purpose processing resources which take advantage of VLSI or VHSIC technology to provide very high speed operations. These resources can be found in the DSP realm: primitives perform DSP tasks such as Fourier transforms, matrix operations, and application-specific operations, such as adaptive beam forming. In the graphics processing realm, special purpose primitives perform transformation, clipping, and perspective operations. The technology is extending into other realms. Special purpose chips have been built to perform edit distance searches, pattern matching and neural network emulation. The prototype development of these devices indicates that there is likely to be a growing demand for high speed, dedicated processing systems in application areas that were previously served by more traditional, slower architectures.

Paralleling the desire for high speed processing is efficient system integration. Resource-based systems are currently integrated into application processors in a few ways. Special purpose configurations of high speed, special purpose primitives can be created by *hard wiring* those primitives together and scheduling tasks statically. An example of this

approach is presented in [Curt82]. The integration produces a high speed processing network, but each application must be designed from scratch. Hybrid approaches are the most common to date. In these approaches, a general purpose processor serve as the controller for a bank of special purpose resources, providing control and I/O. The EMSP Data Flow Computer [Brow84] follows this paradigm. Unfortunately, it is difficult to extend this method to large collections of heterogeneous primitives which require different types of control and interfacing: a central control bottleneck will develop. The *Alternative Low-level Primitive Structures* (ALPS) integration approach attempts to overcome this bottleneck by using a pool resource model in which tasks are dynamically mapped to available processing primitives. The resulting architecture affords reliable, robust, and changeable implementations. From an ease-of-integration perspective, this approach affords an application independence to the architecture. To create an application architecture, the application task directed graph must be provided to individual primitive controllers and a suitable configuration composition must be decided upon. The first of these steps is a straightforward encoding. The second can be decided upon via simulation and analysis of the application.

The essential model for d-ALPS [Bult86] describes application-driven configurations of processing nodes. An ALPS application configurer is presented with an application *algorithm* in the form of a directed task graph, with specified data arrival rates (fig. 1-1(a)). The algorithm can be considered a set of processing tasks that are in a precedence and data-sharing relationship. The ALPS approach dynamically maps that set of tasks onto a processing configuration architecture which consists of a collection of special (or multi-) purpose processing resources (fig. 1-1(b)). Central to integrating each individual processing element is a homogeneous *interface control unit* (ICU) which interfaces the primitive to a common interconnection structure: a set of parallel data and message busses.



Figures 1-1(a) and 1-1(b): A directed task graph and the mapping of that task graph onto a collection of processors.

One realization of the ALPS architecture approach is the *distributed-ALPS* (d-ALPS) architecture. The dynamic assignment of tasks is performed not by a central controller but by distributed controllers, called *interface control units* (ICU's), which share in scheduling responsibilities. Each element of a d-ALPS configuration architecture contains an ICU-primitive pair. The ICU has a description of the task graph, as well as information about the capability of its attached primitive to perform certain tasks in that graph. The primitive performs one or many of the tasks in the task graph and the ICU provides inter- and intra-node task management functions. Inter-node management functions include participating in a distributed and dynamic task assignment mechanism, managing queues of data blocks which are either waiting for resources or waiting on other data blocks to be produced, and handling all monitoring, setup, and status information exchange. The intra-node functions include orchestrating concurrent support facilities for memory management and data and message transfer, and providing I/O and control to the attached processing primitive. The reader is referred to the d-ALPS High Level Logical Control Specification in Appendix A for a more complete description of the implementation of these control functions.

1.1. Thesis Scope

Designing and building ALPS architectures requires research and skills on both mundane and fundamental levels. On a mundane level, a particular ALPS specification, such as the d-ALPS specification, must be translated into replicated, physical implementations. Along with this, interfaces between ICU's and each supportable primitive must be constructed. On more fundamental levels, research must proceed on many architectural challenges including: determining how to distribute tasks, managing queues of data, minimizing communications complexity and overhead, and providing robust protocols that deliver performance regardless of the application. This thesis is primarily concerned with one of these research issues: task distribution.

Task distribution and allocation is of central concern to the underlying approach for ALPS. Task distribution refers to all methodologies that are necessary for mapping directed task graphs onto a sets of processing and communication resources. The minimal objectives of research into task distribution are to provide low cost, application independent mapping mechanisms which do not deadlock. These have been realized in the *distributed-ALPS* (d-ALPS) ICU specification. The overarching objectives are to find mapping mechanisms which capitalize on general or specific task graph, architecture, and performance parameters so that the underlying architecture can be tailored to specific applications by *either* utilizing low cost, application-independent mechanisms *or* by utilizing application-specific mechanisms that are implementable in an ALPS support architecture.

Task distribution, or "scheduling," can be thought of as the analysis and manipulation of parameters which effect a particular execution order. These parameters can be classified into four categories: *task definition parameters, architectural parameters, task mapping parameters* and *performance parameters*.

1.1.1. Task Definition Parameters

An application task graph imposes a precedence relationship among the subtasks that compose it; that relationship is a *partial ordering* of all subtasks, in that subtasks that are not predecessors or successors of each other can conceivably be executed in an arbitrary order (fig. 1-2). The task definition level, then, is rooted in the underlying application precedence and provides the static ordering of some or all of the task graph. Any execution order which can be derived from the partial order is valid.

1.1.2. Architecture Parameters

The underlying architecture imposes certain constraint and cost parameters for task scheduling. As the configurer chooses the composition of a configuration, the chosen

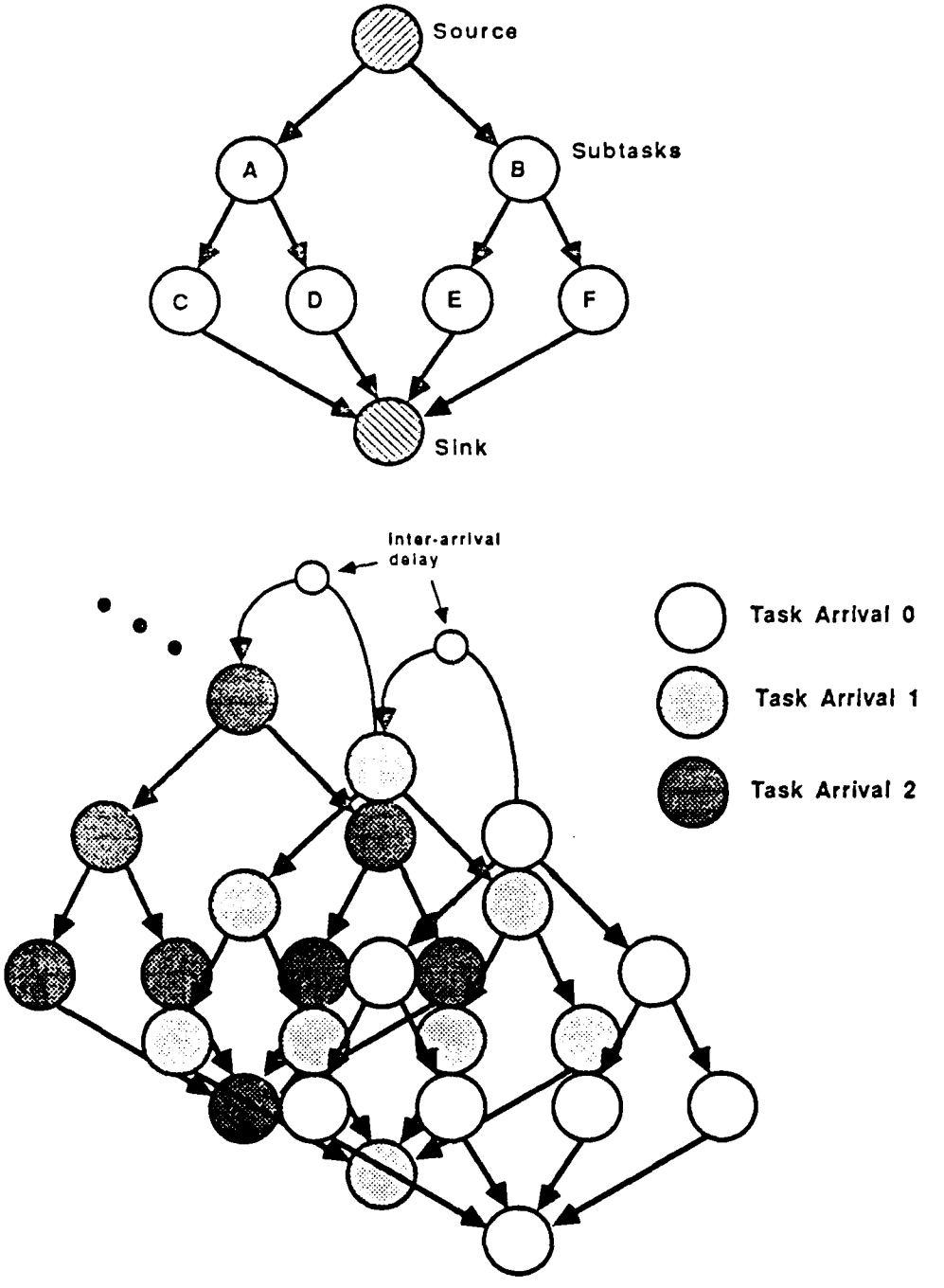


Figure 1-2: Subtasks E and F have an execution order independent of each other and of C and D. Tasks of different arrivals have execution orders that are independent of each other.

allocation places bounds on resource availability. This impacts task graphs which have peak demands for resources that are greater than the allocated supply. Other parameters are less application-specific. For example, the architectural implementation may impose static or dynamic runtime costs which must be accounted for in a task schedule. Architecture parameters, then, include the composition of the configuration allocation and the runtime task distribution and processing costs.

1.1.3. Task Mapping Parameters

The task mapping level refers to the alternative mechanisms that can be employed to choose an execution order that is based on an incomplete task schedule definition. A completely specified task definition means that at the task definition level, the execution order and resource assignment of all subtasks has been specified. If the task is not defined to this level—*the typical case--the mapping parameters* can utilize random or heuristic-guided assignment rules to dynamically assign the task graph. Task mapping parameters, then, are the description parameters to a mechanism for choosing specific execution orders from an incomplete static task definition.

1.1.4. Performance Parameters

A d-ALPS configurer is confronted with a mosaic of often conflicting performance criteria. Some of these are *explicit*, and can be encoded in the application description. These include task interarrival times, maximum latencies and maximum resource counts. Some performance criteria are *implicit* and more difficult to quantify, such as correctness, stability, reliability, recoverability, and reconfigurability. They may be expected by (or demanded of) the configurer but their fulfillment does not necessarily hinge on particular configurations: they are architectural goals as well. Performance parameters delineate both the implicit and explicit application criteria.

1.2. Thesis Goals and Organization

This thesis describes efforts that have been motivated by the general objectives of research into task distribution in the context of an ALPS architecture. As a first stage of this research, this thesis provides a presentation and representation of the scheduling and allocation problem in terms of the above parameter classification and presents methodologies for studying task scheduling and allocation. The overarching goal is to provide an investigation framework which is useful to both a designer/analyzer of an ALPS architecture (an architect) and a configurer, who will utilize an available underlying ALPS architecture to generate specific application instantiations of that architecture. The results that have been derived from initial use of these methodologies are of more than passing interest. Where appropriate and relevant, those results will be presented.

The organization of this thesis is as follows. Chapter 2 introduces the basic categories of allocation and scheduling methods— *enumerative*, *graph theoretic*, and *heuristic*— and then presents some example scheduling methodology/system pairs. These examples demonstrate how schedule generation and analysis is performed in the context of both simpler, static architectures, such as processor pipelines and more general processor networks. The goal of this chapter is to delineate the classification boundaries of the d-ALPS scheduling/system pair.

Once the general background information is presented, an introduction to the specific problems of scheduling in a d-ALPS environment is given. This takes the form of an investigation of the four classes of parameters which describe the dimensions of the scheduling problem in an ALPS-specific context.

Chapter 4, "Problem Setup," introduces four analysis methodologies that will be presented in subsequent chapters. Central to this introduction is a presentation of the basis and objectives of each methodology and a discussion of how each methodology can be used

by an architect and a configurer. The next four chapters discuss each methodology in detail and present examples.

Chapter 5 presents a set of methods for statically analyzing task parameters by themselves and in conjunction with some system and performance parameters. An architect can model the impact of designed-in node limitations such as communications capability on example applications. For a configurer, the techniques provide a delineation of minimal and performance-optimal resource allocations; a *feasible* allocation typically lies between these bounds.

Chapter 6 presents a method for generating and evaluating task schedules on a hypothetical architecture which can distribute and process tasks with no overhead. Given a complete description of an application task graph and a composition of that simplified architecture, scheduling *states* can be enumerated which demonstrate the various ways that tasks can be scheduled in order to buffer instantaneous demand for resources. The approach is modeled after schedule analysis techniques for static, pipeline processors, introduced in chapter 2. Its objective is to highlight the complexities of schedule and allocation analysis in the d-ALPS context and justify the use of simulation-based analysis to both an architect and a configurer's

Schedule simulation, presented in chapter 7, allows an architect to investigate mapping parameter alternatives in the context of a task graph and an architecture. It also allows major architectural assumptions which impact statically and dynamically on a schedule to be modeled in. A configurer can make a first-pass estimate of the viability of a given configuration: if this relatively inexpensive technique demonstrates that the task graph is executed with unacceptable performance, it will not be necessary to submit the task graph/allocation to more detailed architectural simulation. Given the specification for the d-ALPS support architecture, the configurer has no choice of mapping procedures that can be

applied in a particular configuration. At some point, though, the configurer will be provided with ALPS architectures which contain mapping *alternatives*. The schedule simulator will be useful as a first-pass analysis of those alternatives to determine which are most appropriate for the particular application.

Chapter 8 presents detailed architectural simulation as a means of generating and evaluating configuration allocations for d-ALPS. Architectural simulators that model the d-ALPS specification in Appendix A are key to evaluating the efficiency and overhead of this architecture. These simulators can provide a basis of comparison of this architecture to alternative ALPS architectures, as well as to other task processing architectural approaches.

The thesis conclusion, chapter 9, reviews how the overall goals of d-ALPS architecture research have been served by the development of analysis tools and methods. It summarizes the objectives and achievements of each methodology and presents a framework for future development.

CHAPTER 2

Background

In order to frame the scheduling problems inherent in the ALPS approach, two types of background information are necessary. First, allocation and scheduling methods which fall into the general categories of *enumerative*, *graph theoretic*, and *heuristic* will be discussed. An explanation and example of each of these types of analysis will be presented. Second, scheduling and allocation techniques that are employed in a number of example systems will be presented. These example systems fall under problem domains ranging from single-purpose, static allocation to general purpose, dynamic allocation. The analysis techniques and algorithms used (at run-time) to generate the operation schedule or allocation vary not only with the amount of up-front information about the tasks, but with the costs that would be involved in seeking and generating optimal scheduling solutions.

On the low-end, static pipelines are presented. Static pipelines use a static, centralized and predetermined control strategy which takes into account the enumeration of task arrivals and collisions. This control strategy generates optimal schedules that can be implemented efficiently. In this structure, queueing can be employed to maximize utilization of processors or allow an application-driven schedule. In this case virtually all salient information about the individual tasks is provided. An enumerative scheduling technique is employed. This example of low-level, deterministic task scheduling is complemented by an example of static resource allocation: register allocation to local variables, performed by optimizing compilers. The salience of this example is that it presents a method for state generation and alternatives enumeration for resource assignment in much the same way that the pipeline example provides methods for enumerating task initiation alternatives. A graph-based

representation is employed and a commonly used graph-theoretic algorithm is used to generate allocation alternatives.

Moving up one level of granularity are tasks that are presented with information about their requirements, but the system is not given a schedule of arrivals of these tasks. A distributed system for real-time task processing is presented. This system guarantees the scheduling of tasks so that specified time deadlines are met. This system uses distributed data about the load on individual processors to determine where to send tasks so that their execution will occur before a deadline is met. Moreover, the execution of tasks that have just arrived will not affect the previously made promises of deadlines being met by tasks introduced and scheduled earlier.

2.1. An Overview of Allocation and Scheduling Analysis Techniques

2.1.1. Graph Theoretic/Enumerative Models

Graph theoretic techniques provide a framework for the representation of task scheduling and allocation problems and the generation of solutions. They transform these problems into well-studied but computationally unattractive problems of graph partitioning, graph coloring, longest path searches, etc. The advantage of these techniques is that the space-time problems of allocation and scheduling can be visualized, and well understood combinatorical techniques can be applied. The two disadvantages are that devising an acceptable representation that includes enough salient information about the tasks and the target architecture is not easy, and that once a model is produced, it is usually unsolvable (or impractical to solve) because of the exponential growth in the number of states to be evaluated. Enumerative techniques rely on graph representations as well as more standard representations (state tables, Gantt charts, etc.) and use a variety of techniques to enumerate scheduling alternatives. These representations can be made more precise in modeling important task and

system information by expanding the representation to include detailed, information as additional states but after exhaustively enumerating of all possible solutions, rely on computationally expensive searching and scoring functions.

An example graph theoretic approach for task allocation is the network flow graph model. Here, task initiation sites are represented as *sources* and task completion sites are *sinks* [Ston77]. Each subtask in the task set is represented by a node and the interconnection of subtasks is represented by links weighted by the interprocess communication costs between the two subtasks (fig. 2-1(a)). In addition, each processor is assigned a graph node and links are drawn to each subtask. The weight of these links is determined by the cost of executing the subtask on all of the other processors. For a system with N processors, the contribution of subtask A_m executing on processor x_i to the link between x_i and that subtask is $\frac{-(N-2)E(A_m, x_i)}{(N-1)}$, where $E(A, x)$ is the cost of executing subtask A on processor x, and the contribution to the links between A_m and all other processors $x_{k \neq i}$ is $\frac{E(A_m, x_{k \neq i})}{(N-1)}$. Cuts

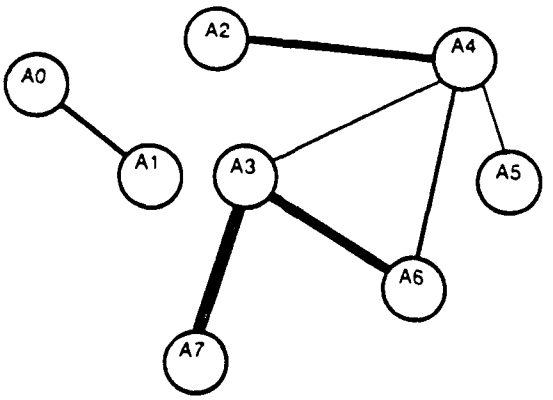


Figure 2-1(a): An example task set with interprocess communication costs denoted by link widths.

through this network must be made so that in each region there is only one processor (fig. 2-1(b)). By minimizing the sum of links that are broken by cuts, the minimum cost for task execution can be found. The proof of this is as follows. There are two types of costs taken into consideration: $E(i,x)$ and $CO(i,x,j,y)$, the cost of the communication between subtask i resident on processor x and subtask j resident on processor y. The inter-process communication costs between co-located subtasks are assumed to be zero and these subtasks will be in common regions in the cut graph. All other subtasks will be across a boundary from A_m and the IPC cost will be taken into consideration. The execution cost $E(A_m,x_i)$ will be contributed by the sum of the broken links which define the apportioning of the graph; this is why

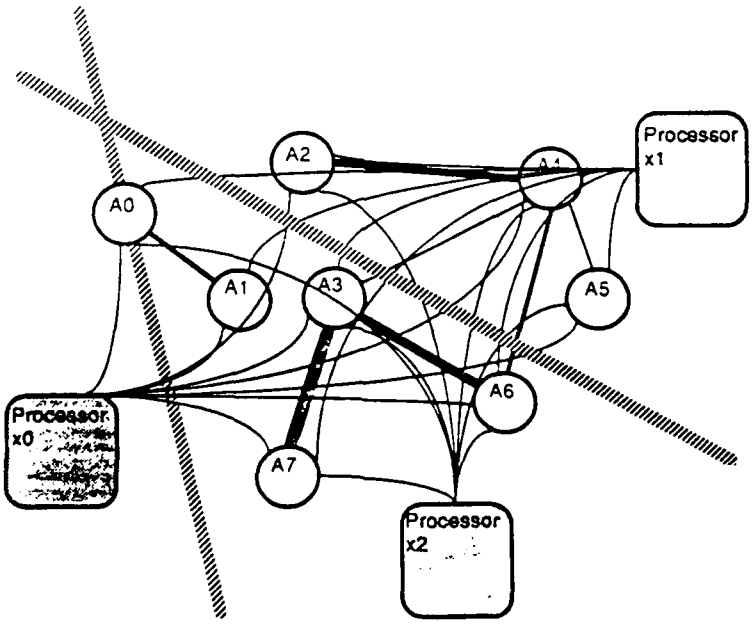


Figure 2-1(b): A task set/processor network. The cuts represent assignment of subtasks to processors.

the execution cost weights are opposite-sign.

A significant obstacle in this approach is the need to make substantial simplifying assumptions in order to coerce both a set of tasks and an operating environment into this representation. In this example precedence constraints and concurrencies are not explicitly accounted for. In addition, the model would have to be significantly augmented to represent the subtask timing. An example of a system simplification that is made is that interprocess communication is simply assigned a cost and that cost is assumed to be independent of the nature of the communication: whether it is a single transfer or a series of communications.

2.1.2. Heuristic Models

Heuristic techniques are those which apply general servicing, partitioning, or delaying rules to specific problems. Their use is motivated by the observation that optimal solutions are difficult to derive and are often impractical in real systems, especially when runtime variables such as memory and communications contention must be included and timing estimates become degraded. Heuristic algorithms have met with varying success. The distinguishing features in heuristic approaches are their complexity, or number of factors they take into account, and the performance improvement they offer over a large set of tasks; scheduling and allocation performance improvement can be measured as a ratio of the performance of a schedule derived via heuristics to the performance of an "optimal" schedule, using as a measure of performance some combination of the criteria the heuristic biases towards (task completion, utilization, etc.) and the costs incurred by the heuristic-generated schedule.

A heuristic approach for task allocation in systems with time-critical tasks has been proposed by Ma [Ma84]. His approach seeks to balance processor utilization, minimize interprocess communication by colocating tasks that share information, and meet the time-critical task requirements. Each full execution of the task is called a processing *thread* and

it is the *port to port* (*PTP*) execution time (start to end of thread) which is specified to the system. The contributing factors to the *PTP* time are the task execution time, the queueing delay time (*QDT*) due to multiprogramming of the processors, and the interprocess communication (*IPC*) The optimization function that results is simply a minimization of the contributing cost factors. A task preference matrix denotes the exclusion of certain subtasks from particular processors; a task exclusive matrix prevents large-instructioned or frequently enabled tasks from being co-located to reduce *QDT*. A coupling factor matrix biases towards co-locating tasks with large *IPC*. A task redundancy matrix allows for multiple copies of a task for re-allocation or low-cost re-invocation after faults. These problem definition matrices are fed to a task allocation model which is a *branch and bound* tree search based algorithm. An optimum solution is generally not feasible, as Ma asserts it is an *NP* hard problem, but the constraint matrices defined above limit the breadth of the search; in a static implementation, the up-front, one-time cost of finding this solution would be acceptable if the schedule could then be hard-coded in the system. For problem analysis on-the-fly, Ma proposes that this allocation algorithm be performed by *preprocessors*, resources especially dedicated to scheduling. The first preprocessor generates the constraint matrices and the second performs the search.

The heuristic Ma employs seems to ignore current resource state information which could dramatically affect the performance of implemented of allocation decisions. If this information is to somehow be included in the schedule generation heuristics, Ma does not indicate its role or how such information is gathered. It is conceivable that previous allocation decisions could be saved to approximately model the state of the system but this would limit the algorithm to a single task server; the algorithm would require distributed information gathering and analysis mechanisms to be implemented at each task initiation site.

2.2. Pipelines

A straightforward example of the application of enumerative scheduling techniques is employed in the scheduling of processor pipelines. A task is described as a sequence of visits to independent processing stages. (fig. 2-2). The task is broken into independent units of approximately equal execution time. At each stage some set of operations is performed and the next stage is invoked. In a linear pipeline, each stage is passes results to an adjacent stage; in a more general case, results could be passed to a number of stages and can be fed back to earlier stages as well. An assumption is made that each stage in the pipeline requires a standard unit execution time; longer stages can be broken into *virtual* stages, each requiring a unit execution time. In the baseline case, data is injected into the pipeline and when it exits the pipeline, new data can be presented to the system. The goal of pipelining, though, is to send data (initiate tasks) as often as possible by using earlier portions of the pipeline for new tasks while older tasks are being processed by later stages. To this end, a schedule must be conceived which allows data to transit to each processor with no danger of colliding with data from previous (or subsequent) arrivals.

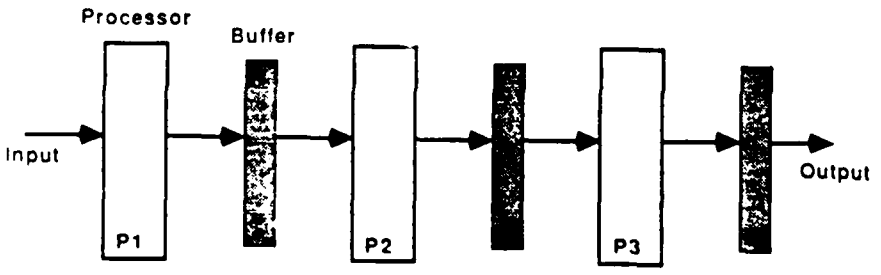


Figure 2-2: A linear pipeline.

For linear pipelines (fig. 2-2), scheduling presents little difficulty, provided the stages take equally long to execute: a new task can be presented to stage 1 as soon as that stage is finished with its current task and has passed it to stage 2. From an implementation point of view, this means that data is passed one stage at each (system) clock tick. For nonserial pipelines, and most evidently, for those in which physical pipeline stages are reused (recirculating pipelines), the scheduling is more complicated (fig. 2-3).

The job sequencing problem in pipeline scheduling, then, is to schedule tasks awaiting initiation in order to avoid collisions and to achieve a high throughput. The throughput of a pipeline system is the rate at which data is presented to the system, the implication being that data leaves the system at the same rate. It can be expressed as:

$$TH = \frac{n}{k\tau + (n-1)\tau},$$

where k is the number of stages of the pipeline, τ is the clock period of the pipeline, i.e. the basic time required for each processing element to execute (data movement time is included

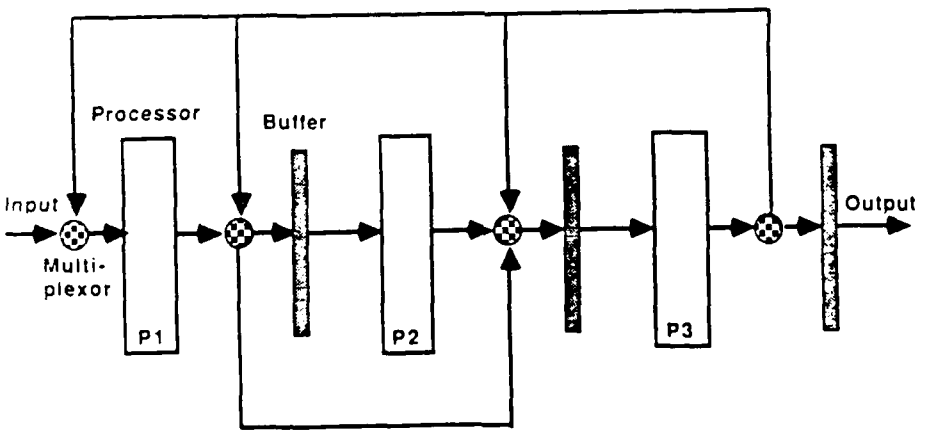


Figure 2-3: A recirculating pipeline in which results from stage 1 is fed forward and results from stages 2 and 3 are fed backwards.

in this measure), and n is the number of tasks being processed during the period $k\tau + (n - 1)\tau$; when the pipeline is full, the throughput is simply $\frac{1}{\tau}$. Two performance criteria, latency and throughput, play a role in determining how to formulate a schedule.

A representation of a pipeline scheduling problem is provided by a reservation table (table 2-1). In this space-time view, processing element P_1 is required to execute at timesteps 0, 3 and 6 for each task initiation. In the language of pipeline processors, the *latency* of a pipeline is the number of time units between two initiations.[1] For static, linear pipelines, the latency can be one time unit. For nonlinear pipelines, where processing units are re-used, or for pipelines with varying execution times, the latency may be a constant (1 or larger for static, single-functioned pipelines) or it may be a *sequence*. That is, the latency can vary between successive initiations (task arrivals) to most efficiently map pending tasks onto processors while avoiding collisions. A control strategy is a procedure to choose a latency sequence that *cycles*. The criteria in choosing a latency cycle is to choose one which maximizes throughput and avoids collisions. This is done optimally by choosing one which has the minimum average latency (the average of the inter-arrival times of the task initiations in one latency cycle). A nonoptimal strategy is to choose a cycle which minimizes the

Reservation Table								
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
P_1	X			X			X	
P_2		X						X
P_3			X		X	X		

Table 2-1: An example reservation table for the pipeline in figure 2-3.

[1] In the language of the network world, this is the interarrival time.

latency between the most previous arrival and the current arrival; this is intuitively equivalent to a *greedy* strategy in memory allocation in which at each step a locally optimal or beneficial decision is made, regardless of the downstream consequences.

A procedure for finding and enumerating latency cycles and detecting disallowed cycles is presented in [Hwan84]. A row on the reservation table represents the allocation of a task on a physical processor. For each row of the reservation table, the distance between marked (reserved times) represents a possible collision. Multiple entries on the same row indicates that the task reuses that physical resource (processing element) and once the task has been initiated it will require that resource at the time stages indicated on the reservation table. A collision will result if a subsequent task is initiated so that it requires the resource at one of the times that the current task requires the resource. If the spacing between two reservations on a row of the reservation table is s then a collision will result if two tasks are initiated with a latency of s; those two tasks to not have to be adjacent arrivals to collide. The method used to determine latency cycles is to create a vector (the *collision* vector) which indicates the feasible latencies that may be chosen:

$$C = (C_n, \ldots, C_2 C_1),$$

where $C_i = 1$ if there is a reservation table row distance of i. The initial collision vector then corresponds to the state of the system where the first arrival has been presented, and subsequent arrival are limited to the zeros in the collision vector. The collision vector is then shifted to the right, a zero shifted into the left-hand-side, until the rightmost zero is shifted out (a zero shifted out indicates that an arrival is allowed at that time, i.e. if at the third shift, a zero was shifted out, then a latency of three would be allowed—there would be no collisions. The new collision vector (the old one shifted until the first zero) is bit-wise *ORed* with the previous vector to represent subsequent collisions that are possible between the next task and the previous tasks. The new collision vector represents a new *state* in the

system; a state transition diagram enumerating each collision vector linked with an arrow indicating the latencies between two arrivals which bring the system to that state is then drawn (fig. 2-4). From each collision vector all subsequent vectors (states) new ones can be derived and these states can be linked into the transition diagram. Eventually a trail of states will connect to a previous state or with itself. A cycle in this state diagram indicates a control strategy, a latency schedule that will not induce collisions among different tasks. The transitions, labelled with the latency required to move between states, indicate this schedule. The optimal schedule (emboldened in fig. 2-4) is the cycle which has the minimum average latency, that is, the sum of the latencies at each transition divided by the number of stages in the cycle. Since any closed path is a feasible schedule, a state can be visited any number of times. A simple schedule is one in which each state is visited only once: since there can be many optimal schedules, it may be advantageous from a control implementation point-of-

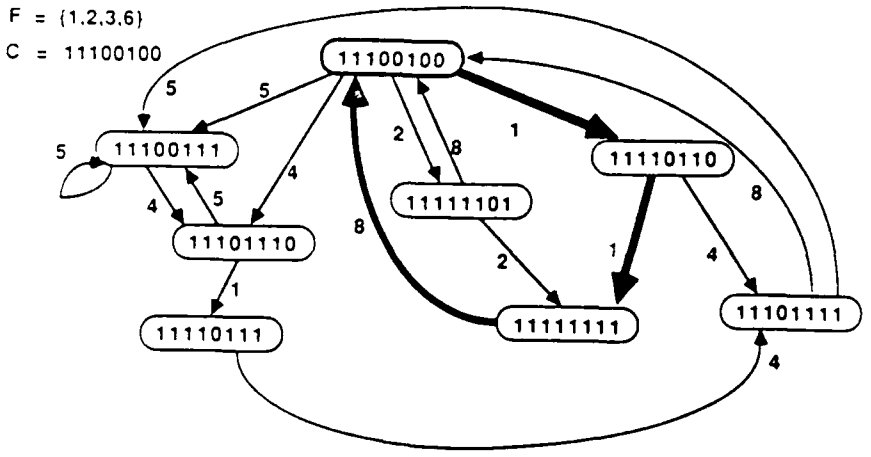


Figure 2-4: A collision vector state diagram; the arrows indicate the control strategy with the minimum average latency.

view to choose simple schedules over complex ones.

The above method is a straightforward state *enumerative* technique which allows not only the production of collision-free arrival schedules, but allows a maximum throughtput schedule to be generated. It is applicable to static pipelines which are nonlinear. An extension to this method can be applied to multifunction pipelines, pipelines which can operate on different functions by interconnecting different subsets of stages for ecah function in the pipeline. The scheduling problems of a pipeline processor which can perform many functions (execute p different tasks) can be represented by p overlaid reservation tables. A static, multifunction pipeline is designed to perform a single-type task for a while and then perform a task of a different type after a minimum waiting period. An optimal latency schedule for each of the task types is determined and a control structure determines the wait time necessary between two different typed tasks. Each task to be executed is tagged with the particular function that is required to distinguish it from arrivals of the same task-type and arrivals of different task types. The extension to the single-function algorithm is that an arrival can collide with a task of the same type or a different type. Collision vectors which describe the collisions possible between any two tasks can be generated from the overlaid reservation tables: there are p^2 collision vectors each with length n for a system of n stages to execute p different tasks. The process of generating individual latency cycles is similar to the static-pipeline method and is described in detail in [Hwan84].

The significance of this extension is that there is an automated method of generating schedules for the mapping of multiple tasks onto fixed hardware pipelines which implement the specific functions by transferring data across dedicated interconnections. The subtask times are completely specified; in fact, the processors operate at ratios of a basic pipeline cycle time. All communication, whether between adjacent pipeline stages or spanning multiple or parallel elements, requires dedicated resources and equal time per transfer. Finally,

the control structure is static and centralized. Once tasks are introduced to the system, their travails are predetermined. The above control strategy determines an optimal latency structure of tasks of different types so that throughput of each task type is maximized.

An example of a static, reconfigurable pipeline is the TI-Advanced Scientific Computer (ASC). This pipelined arithmetic processor allows instructions which require different stages to take different data paths through the pipe. The control for this processor is contained in a ROM accessable by execution logic and provides the route through the pipe for each instruction type. The base address for this information is provided by instruction decode logic. When several instructions of the same type are present, for example when a vector operation is specified, the instructions can be pipelined through the hardware. The latency cycle for these streams is specified in ROM. When instructions can be overlapped, average speed increases from 0.5-1.5 MFlops to 3-10 MFlops.

2.2.1. Reservation Table Optimization Through Delay Insertion

For single functioned pipelines, the optimal latency cycle is optimal with respect to the given reservation table. As an improvement in throughput, delay elements, or buffers, can be inserted in each stage. This has the effect of modifying the reservation table to allow tasks to interact with less collisions. The maximum achievable throughtput is attained when all elements of the pipeline (all processors) are fully utilized. The method of modifying the table as described by [Davi78] is a bit involved but seems to work. First, a set G_C is generated from the chosen (optimal with no inserted delays) latency cycle C, consisting of the inter-arrival differences, i.e. the time between any two (not necessarily consecutive) initiations. Second, a set $G_C \ mod \ p$ is computed by modding each element of G_C with the period of the cycle C and removing duplicate members. Finally, the reservation table can be modified by inserting delays to eliminate inter-row distances which are members of the set

$G_c \bmod p$. A maximum utilization latency cycle can be generated by choosing a constant latency cycle equal to the maximum number of entries in any row of the reservation table. The reservation table is then modified by adding delays to make that cycle allowable. Alternatively, some application-specific cycle (a pre-determined schedule unrelated to the collision problems in the pipeline) can be implemented, i.e. made allowable, by adding delays to remove any conflicts between the forbidden latencies in the reservation table and the inter-arrival times in the specified cycle. It should be noted that adding delays contributes directly to the task end-to-end time/throughput tradeoff: while processor utilization and throughput are maximized, the end-to-end task time is increased via queueing.

2.2.2. Extension to Reconfigurable, Dynamic Pipelines

In a static pipeline, all initiations are subject to the constraints listed in a single reservation table or one that is comprised of overlaid reservation tables, one for each function the pipeline can support. Transitions between initiations of different types are costly: the pipe may have to empty before a task of a different type can be initiated. In a dynamic pipeline, initiations of different typed tasks can occur simultaneously, and complex scheduling strategies allow multiple numbers of initiations of different functions in the same pipeline. A more complicated model of pipelines (one with bypass structures to enable certain tasks to jump over unused stages as well as interconnection control, routing and queue management functions) would have to be incorporated to implement optimally scheduled, reconfigurable pipelines.

2.3. Graph Coloring and Application to Resource Allocation

A technique for defining resource contention and minimal allocation is utilized in optimizing compilers to allocate high-speed registers to program variables. The essential problem is laid out as follows. A subroutine has a number of local variables each having a

lifetime defined as the time at which the first *write* is performed on that variable to the time when the last *read* is performed on the variable. There may be variables present which are accessed frequently and others which are defined but not accesed at all. Given a processor with a number of high speed local registers, it is desirable to make efficient use of these registers by allocating as many of these registers to local variables as possible while allowing the allocation of a register to one variable to not interfere with the later allocation of the same register to a new variable. That is, if a variable is first written to at instruction A and is last accessed at instruction $A + \delta$ then a new variable which is first written to at an instruction later than $A + \delta$ can be assigned to the same local, high speed register. A table

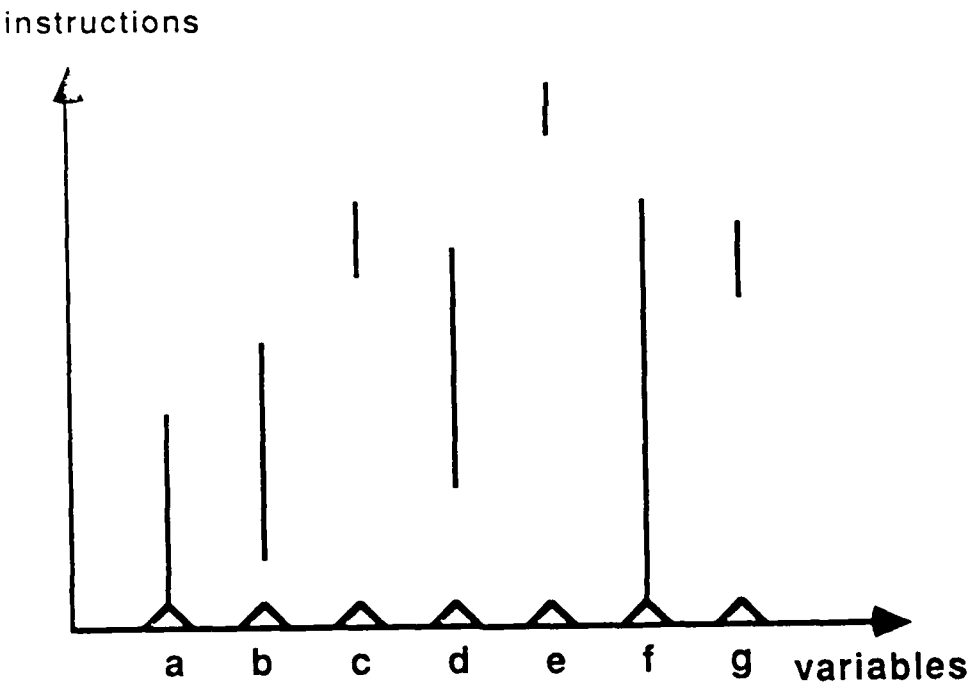


Figure 2-5: A lifetime table for variables a through g. The y-axis represents subroutine progress along an instruction stream.

can be constructed which is not unlike the reservation table for pipeline processors, charting the lifetime of each local variable (fig. 2-5). Given a fixed number of high speed registers and an ample amount of stack storage for the remaining variables, the goal then is to determine how few storage locations are required to provide a noninterfering memory allocation: the more efficient the memory allocation, the more variables can be mapped to high speed registers.

A graph representation is chosen to model the local variable timing relationships. A graph is a collection of edges and vertices; a unique vertex will be used to represent each local variable, and edges will be used to indicate the timing relationship among the local variables. First, a vertex is drawn for each local variable. Second, observing from the local variable lifetime table an edge is drawn connecting vertices representing variables which have overlapping lifetimes. That is, if a local variable *foo* is born (is first written to) at instruction A and dies (is last read from) at instruction $A + \delta$, it is connected to any variable *bar* which is born before instruction $A + \delta$ and dies after instruction A (fig. 2-6). The final

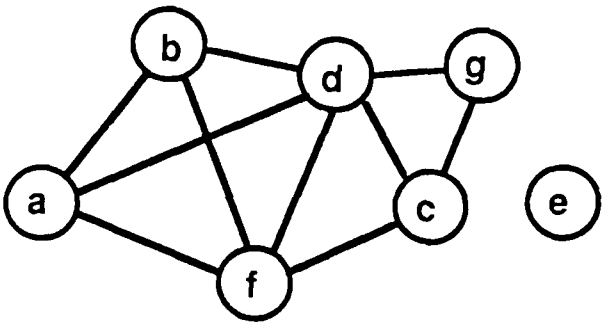


Figure 2-6: An edge-vertice lifetime relationship between variables. Connected vertices indicate overlapping lifetimes.

step in this approach relies on the concept of graph coloring to determine the minimal allocation of memory areas to yield a noninterfering mapping to local variables.

At this stage, the concept of graph coloring is used to determine minimum allocation. The graph coloring problem, simply stated, is to find the minimal number of colors required to color the vertices of a graph so that no two adjacent vertices have the same color (fig. 2-7). The graphs are limited to simple graphs, i.e. there can be no edge between vertex *foo* and itself, because no coloring scheme is possible and it is not necessary to represent the fact that the lifetime of variable *foo* spans itself. There are a number of proofs and algorithms related to graph coloring; an introduction can be found in [Mott83]. An algorithm by Welsh and Powell is presented to color a graph. Heuristics to find the minimum coloring are (1) to note that triangles always require three colors, and (2) that the degree of a particular vertex is d then at most d colors are required to color the vertices adjacent to that vertex. Assuming that the number of local variables is relatively small, a plausible coloring can be found: if it is not minimum, the allocation approach will not be optimal but will still work.

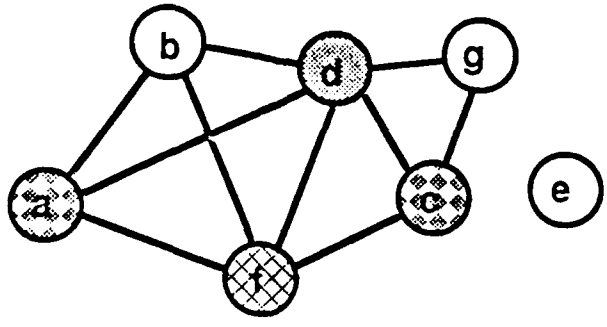


Figure 2-7: A coloring of the variable graph. Variables with similar colors (shading) can be assigned to the same register.

After the graph is colored, the number of colors represents the required memory locations to ensure isolation of variables.[2] If that number is less or equal to the number of local registers, then all variables can be placed in local registers provided that the allocation is controlled by the knowledge of when that register is used for which variable. If that number is greater than the number of local registers, then a subset of the color-groups can be mapped onto local registers. As a further optimization, local variable groups which have higher expected access rate could be mapped to high speed registers. Alternatively, the coloring algorithm may bias grouping according to access rate so that highly accessed but independent local variables are placed in the same group; that group could then be mapped to a high-speed register.

The register allocation problem introduces a graph representation for demand (vertices), interrelationship of demand (edges) and supply (colors). The difficulty in establishing a minimal allocation can now be seen as directly related to the complexity issues of a well known graph problem. From a process-scheduling point of view, the register allocation problem is akin to the problem of statically assigning tasks to (homogeneous) processors so that the minimal number of processors is used. While an optimal (minimal) allocation can be found, the assignment method can not be extended to periodically initiated tasks unless the representation was changed and the underlying graph coloring problem redefined.

2.4. Bid-based Task Scheduling

A fundamentally different approach to task scheduling and sequencing is based on two underlying assumptions. First, task arrivals are not known in advance: the system is a general purpose task processor in which requests for processing are made at random times. A task is characterized, then by its start time, computation time, deadline (when it is required

[2] Unfortunately, it is difficult to prove that a given coloring scheme for a graph is the minimal one.

to complete), and possible its reinitiation time. In addition, it may have enumerated resource requirements and internal precedence constraints. When the task is created and compiled, the precedence and compute time can be encoded; when the task is invoked or requested, the deadline and possibly the task's periodicity (reinitiation time) can be specified. From the individual processing node's point of view, tasks may arrive at the node by some process invocation service which initiates tasks which are scheduled periodically. Alternatively tasks may arrive as a result of being shipped to the node by some other node.

2.4.1. Structure of the Schedulers

Each node in the system has a scheduler that is identical to schedulers on other nodes (except for some node-specific information, such as processing capability). When a task arrives the node attempts to schedule it locally, i.e. it checks to see if it has enough *surplus processing time—processing capacity* above the time alloted to jobs already guaranteed—to execute the task task so that the task's deadline is met. If it is able to meet the deadline then the task is scheduled locally and is considered to be guaranteed to meet its deadline. If the local node can not meet the task's deadline then requests are made to other processors to accept and guarantee the deadline of the task. Tasks that are accepted and guaranteed locally are fed to a dispatcher service. The dispatcher monitors the run queue of the local processor and invokes the task with the earliest deadline. Tasks that can not be accepted and guaranteed locally are given to a bidding service which queues up tasks that must be farmed out. Though these tasks may eventually miss their deadlines, the tasks that have already been guaranteed will not run late due to the arrival of new tasks.

The heart of the distributed scheduling process is the node interaction which permits a node which has tasks that need to be scheduled but has no excess capacity from which to draw. The scheduler on the node with tasks to farm out broadcasts a request for task servic-

ing, receives bids for that task by nodes which have excess capacity and makes a decision on where to send that node based upon the bids received. The request for bids includes the task's computation time, deadline, size (for estimating communication and memory allocation costs), the time at which the bid request is being sent, and the deadline for bid requests. This deadline is set so that nodes which are processing a number of bid requests can decide whether to process this request; factors in this decision include time for nodes to process the bid, time for the requester to evaluate these bids, and time remaining for the task to be sent, scheduled and executed. Some of these factors are estimated bid processing times, which include communications costs which are factors of the load on the network: the point here is that this deadline is composed of estimates. The nodes that receive bid requests must decide whether they can accept the task and guarantee that it will meet its deadline. This decision rests upon several factors: the load on the node, the estimated computation cost of the bid-for task, and the estimated time at which the task will arrive. Once a decision is made it is returned to the bid requester for bid processing. The requester receives a list of bids from various serving nodes which give an estimate of their ability to process the task. This estimate is the surplus compute time at the node between the expected arrival time of the task and its deadline. The bid requester then chooses among the bidders based on these surplus estimates.

One extension to the bidding scheme is to attempt to guarantee tasks that have precedence constraints. Suppose three tasks A, B, and C have the precedence relation $A < B < C$ (A precedes B which precedes C), and A and B have been guaranteed on two different nodes, 1 and 2. The site at which C arrives, 3, attempts to guarantee it. This is done by assuming a start time of the task of D_B, the (already guaranteed) deadline of B. If C can not be guaranteed locally at node 3, the node will send out requests for bids for C but will ask bidders to return the bids to the node which owns and has guaranteed B, node 2.

Node 3 will then ship C to node 2, presenting it with the task of resolving bids for C. Node 2 will attempt to guarantee C locally; if it can not, it will attempt to modify the deadline it has imposed on B so that the start time of C will be pushed earlier. If D_B can be reduced so that B is still guaranteed and C can now be guaranteed locally, C is scheduled locally. If C can not be guaranteed locally even after modifying B, node 2 waits on incoming bids for C and chooses the best bidder. When that bidder is informed that it is to receive and guarantee task C it is also given the new start time of C.

2.5. Conclusions

The above example systems demonstrate the use of graph theoretic, enumerative, and heuristic approaches in modeling allocation and scheduling problems. Heuristic approaches are more popular in dynamic scheduling applications where information about tasks and task arrivals is limited, information about the state of the system must be estimated, and the time to make the scheduling decisions bears directly on the processing load on the system. Enumerative and graph theoretic techniques are popular when system interconnections and task schedules are known (or are to be determined statically). The time to generate schedules or allocations grows dramatically with the size of the problem, and optimal schedules may degrade or, in fact be forbidden, if the system or task description does not match that with which the schedule was generated; these techniques are less popular with dynamically scheduled systems. The analysis methodologies presented in this thesis will make use of scheduling techniques similar to control strategy generation for pipeline processors and will consider allocation decisions similar to those seen in the graph theoretic-based register allocation example. The heuristic-based task assignment system reviewed in this chapter is a useful introduction to the heuristic-based task servicing disciplines that will be presented in chapter 7.

CHAPTER 3

Problem Specification

Scheduling and processor allocation can be considered most generally as the mapping of a set of tasks onto a system of processors. The central scheduling and allocation problem, regardless of the particular implementation of an ALPS architecture, is the efficient and effective distribution of an application specific set of tasks onto a pool of resources. Scheduling acts on an application task graph that has been described by *task definition parameters*. The task graph is mapped onto a processing environment, characterized by *architectural parameters* which include the composition of resources and the specification of the underlying support architecture. The mapping is facilitated by *mapping parameters* which, while conserving the basic task definition, generate execution orders and assignments under the guidance of task servicing heuristics. The goal of this mapping is to facilitate the execution of a task graph with some defined level of performance. The performance criteria can be described by *performance parameters*. The above process is illustrated in figure 3-1. Scheduling and allocation issues, as applied to ALPS, is a problem of operating within broad boundaries imposed by the application task to find a combination of task definition, architectural, and mapping parameters which yield an execution order which satisfies performance criteria.

The goal of this chapter is to transform the discussion of the general problems of scheduling and allocation into a discussion of the mechanisms and parameters behind scheduling and allocation in ALPS. Once the reader is equipped with this background, subsequent chapters will present methodologies for investigating scheduling via manipulating these parameters. This chapter will first present two perspectives on the investigation of

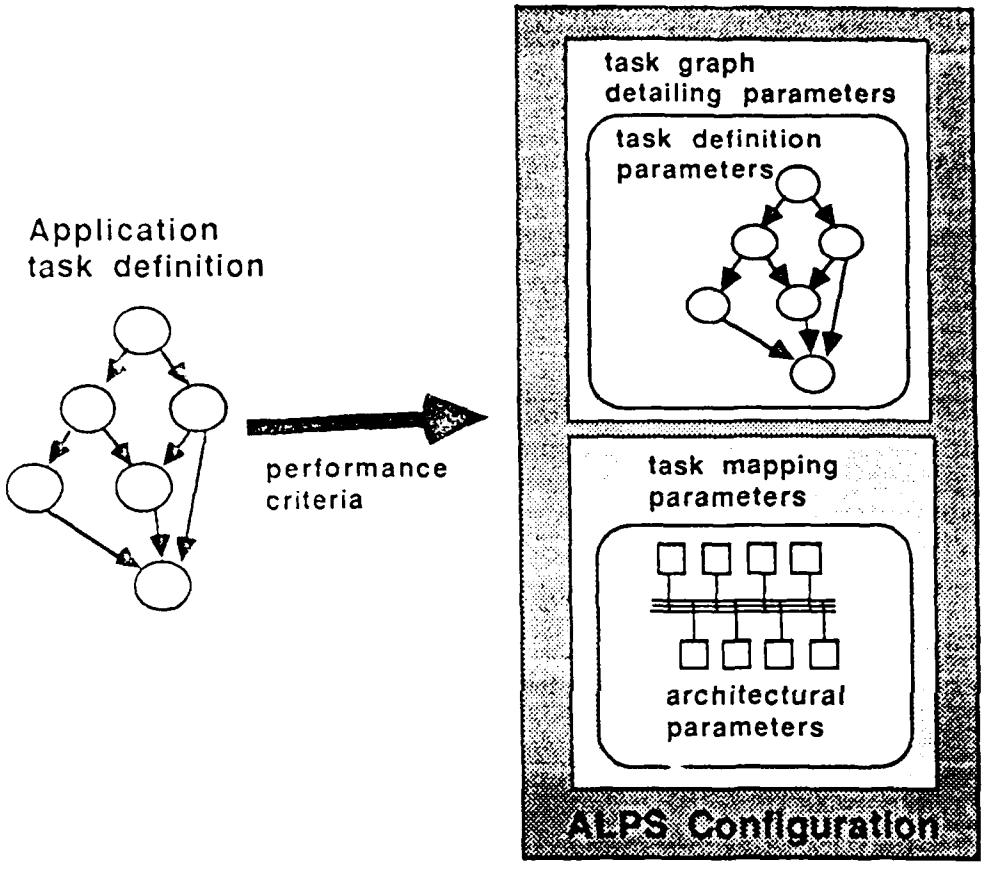


Figure 3-1: The ALPS configuration is composed of task definition, architectural and task mapping parameters. The choice of these parameters is guided by performance criteria.

these parameters: the configurer's and the architect's. Following will be a discussion of performance parameters which highlight the contrasting objectives of the architect and configurer. The task definition, architectural and mapping parameters will be presented in subsequent sections. The architect and configurer have different motivations in investigating these parameters; these motivations will be discussed in each section.

3.1. Parameter Investigation Perspectives

Performance, as broadly defined, encompasses straightforward, specific demands on the application architecture designated to fill a particular application. These demands can include requirements on latency (how long it takes for tasks to complete), throughput (the rate at which tasks must be processed), and system size (how many resources can be alloted to the application because of size or power constraints). Given an underlying support ALPS architecture such as the d-ALPS architecture, a "configurer" should be able to either determine a configuration architecture that meets these *explicit* performance criteria or determine that the requirements are too imposing for any d-ALPS configuration. From this perspective, the configurer would investigate the task definition, system and mapping parameters that can be manipulated by the configurer. The configurer can not change the underlying architecture, but can constructively modify the task definition, can vary some parameters of the configuration architecture, such as resource pool composition, and may be able to choose from a limited number of ways in which the task mapping is effected. A *constrained investigation* of the parameters which affect scheduling and allocation is desired by a configurer so that the configurer can either decide on a configuration that meets explicit performance criteria or can decide that such a configuration is not possible.

The "architect" desires to investigate the underlying scheduling and allocation problems as they are addressed by a range of architectural approaches. That is, the architect will not presume that the only manipulable parameters are those imposed by single ALPS architecture such as the d-ALPS architecture. Furthermore, the architect is interested not only in application-specific *explicit* performance criteria, but general performance expectations of the ALPS architectural approach. These expectations can be broadly labeled *implicit* performance parameters, and may include efficiency (how many resources are required over some ideal, minimal allocation), fault tolerance (how the architecture responds to component

failures), and stability (how the architecture responds to slight deviations in presupposed timings and requirements). The architect's perspective should be supported by an *unconstrained investigation* of the parameters of a particular architecture in addition to an investigation of alternatives that may not be encoded in a particular architecture.

3.2. System Performance Specification

There are two views of system performance when examining scheduling and allocation issues. The first view is that the goals of scheduling and allocation are to create a system which correctly executes the tasks within the bounds of a given set of performance criteria. From this view, the performance criteria are specified in advance and allocation and control alternatives are tried until a system which meets the criteria is found. This can be thought of as the *configurer's* view. The second view is that regardless of the static performance criteria (which in this view can be regarded as part of the algorithm specification), there are meta-performance expectations that transcend any particular application. The goals of scheduling and allocation then are to find approaches that have *generally* beneficial results and can be applied across an application domain. This is the *architect's* view. The above distinctions are made because depending upon one's view of the scheduling and allocation problem, the relative value and even the definition of individual performance measures will vary. Each performance criteria discussed below presents challenges to both the architect and the configurer. The way in which each performance criterion motivates investigation of the task definition, architecture and mapping parameters will be presented.

3.2.1. Latency

Latency is a measure of the end-to-end processing time of tasks. It is the time that a frame of data entered the network at a source subtracted from the time the resulting data

block reaches a network data sink.[1] There may be more than one latency measure if there are multiple sources and sinks, especially if the task has disjoint parts (fig. 3-2). From the configurer's view, the latency may be a critical parameter to the original application; an allocation/schedule must be found that meets this criterion. From the architect's point of view, latency has avoidable and unavoidable components. There is a minimal latency that can be found by summing the computation and communications costs along the longest path in a task graph and adding the overhead time for associated transactions. Above this value, latency results from intra- and inter-task contention for resources. The architect investigates efficient dispatching mechanisims which impact configurations characterized by both lightly and heavily resource contention. This would involve, as a first step, analysis and design of the components of the underlying ALPS architectural implementation which cause transaction overhead. In addition, the architect wishes to find ways of causing latency to increase smoothly or predictably with this contention; this is most likely afforded by investigation of

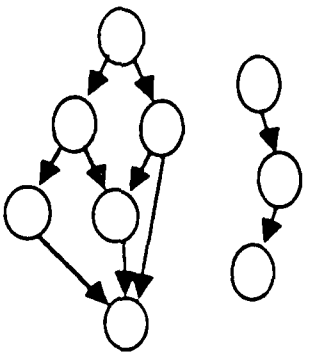


Figure 3-2: A task graph composed of two disjoint parts.

[1] This is a network definition of latency, and also one which is commonly accepted; it is not to be confused with the definition of latency used in the pipeline world, which is what the network world calls the inter-arrival time.

mapping parameters.

3.2.2. Throughput

The inter-arrival time of data (initiation schedule), which is likely to be part of the task definition, implicitly determines the system throughput, in that the entry rate of tasks should, in the long run, equal the task exit rate. In the ideal case, tasks will enter the system at some proscribed rate, and after the "pipe" is filled up, i.e. after the first task is fully processed, tasks will exit the system at the exact rate that they entered. Complicating this ideal case is that task latency may not be constant when contention is present, and the first task latency may be lower, due to less contention, than that of subsequent tasks.

From a configurer's point of view, throughput can typically be categorized as a specification parameter instead of a performance measure, though a configurer may be interested in measuring the additional capacity that a particular allocation can handle. This measure will give the configurer some indication of the robustness of the system to temporary interruptions in processing. From an architect's point of view, a general scheduling goal is to accomodate a significant amount of inter-task contention; a scheduling method can be tested by specifying a task and then seeing how a system which incorporates this method responds to larger throughput demands.

3.2.3. Allocation

The categorizing of allocation as a performance measure as well as a system parameter requires a review of the coupling between allocation and scheduling problems. Two related couplings can be posed:

- Given a task set, a minimum latency, a maximum throughput and a scheduling methodology, find a minimal allocation that will accomodate these criteria.

- Given a task set, a minimum latency and a maximum throughput, and an allocation, find a scheduling methodology which allows the allocation to accomodate the requirements.

Though a configurer supplies an allocation via architectural parameters, the allocation can be viewed as a performance criteria in that the configurer tries to find the most efficient allocation or one which accomodates high *utilization* of resources. The architect views allocation as a performance metric in that allocation it can be the basis of comparison for different scheduling methodologies operating under the same task requirements.

3.2.4. Reliability

In general, any ALPS-based architectural configuration should be resilient to single point failures in that performance degradation (along many criteria dimensions) should be measurable and containable with system degradation. The central advantage of scheduling mechanisms that adhere to a dynamic assignment approach is that at the time of binding a processor, any number of system faults may have occurred, but as long as there are sufficient resources, the task execution will proceed.[2]

From the configurer's point of view, reliability is a requirement on the relationship between a fault in the allocation or scheduling (processor deaths or task deaths/delays) and the degradation of a performance criteria. Reliability, then, can be codified in terms of the impact of a particular fault or eventuality on an explicit performance criteria. An example of this relationship that a configurer could impose is as follows: "If one FFT processor dies, the system should be able to produce results at the same rate and the latency can increase by no more than 10%." The difficulty in deciding upon these relationships and finding an application architecture that meets them can not be understated.

[2] The system may crash under a variety of circumstances, some tied to the aggregate demand for resources and some tied to the peculiarities of demand synchronization.

The architect is concerned with finding scheduling approaches which reduce drastic repurcussions due to *site* eventualities. Consider an architecture which has N processors and requires $N-1$ of those processors to marginally process a given task. Reducing site eventualities means ensuring that the system works with a failure of one processor, regardless of the *particular* processor that failed. Investigation of these concerns requires a study of the architectural mechanisms that implement a scheduling method, as well as an investigation of those methods.

Each of the above performance criteria impacts the investigation of task definition, architecture and task mapping parameters. Task latency criteria are met by the configurer by static analysis of the task definition parameters and the allocated architecture. The architect studies the details of an architectural specification through architectural simulation to determine how to systematically reduce latency. Testing for throughput requirements by the configurer is facilitated by architectural simulation; the architect considers task mapping methods—via schedule simulation—which ensure an orderly execution of task arrivals. Allocation is the key architectural parameter to a d-ALPS configuration. The configurer chooses an allocation which allows a system to meet other performance criteria and which affords a high resource utilization. The architect compares necessary allocations for given tasks in one ALPS architecture with other architecture classes or other ALPS architectures to determine the processing efficiency of the ALPS architecture. Reliability and other implicit performance criteria are directly impacted by architectural and task mapping parameters. The architect and the configurer use architectural simulation to study the effects of site eventualities. Following is a description of the task definition, architecture and task mapping parameters. The descriptions include an indication of which parameters are interesting to an architect and a configuerer.

3.3. Task Definition Parameters

The task definition parameter group describes an application task graph. This section provides a detailed description of the components of that graph and the ways in which the task definition can be altered to affect the eventual execution order.

3.3.1. Directed Graph Notation

The specification of a task in the ALPS framework makes use of *directed graphs*. Each node in the graph represents an independent *subtask* which requires a single processing node to execute. These subtasks typically represent computationally intensive processes. Subtasks are connected via directed links which not only specify a precedence relationship among tasks but a communication of data as well. A subtask is initiated once the assigned processor has received all of the data for the task; implicit in receiving this data is a signal to execute.[3] The representation is similar in structure to *data flow* representation but slightly different in interpretation. Nodes in data flow graphs represent processing stages: their execution is initiated by the receipt of data, whose flow is implied by links between nodes in the data flow graph. The data flow world uses directed graphs to represent the flow of data and control, commonly referred to as "activity," to different processing resources. Directed task graphs in the ALPS domain represent the dependence of tasks and a corresponding passage of data. Control flow in the corresponding particular ALPS underlying architecture is not implied in these graphs.

We will assume that the configurer can not change the fundamental breakdown of tasks into subtasks. That is, the functional decomposition of the application task into separately computable primitive operations can not be altered. This assumption is not made because it is unforseen that the configurer will make a contribution to this decomposition. It is made

[3] A data block that is sent to more than one subtask initiates multiple subtasks.

because the scope of this thesis is to consider how functionally and topologically unalterable task graphs are mapped onto a described architecture.

3.3.2. Graph Weights

Directed links represent communications that must take place between two tasks. From a timing cost point of view, the *weight* of the link is representative of the amount of data that must be transferred; likewise, the weight of nodes is representative of the expected execution time of the task. This weighting of both links and nodes poses some representational difficulties. The exact weighting is not necessarily known at the problem definition stage and may be dependent on the particulars of the target system configuration. For example, the execution cost of a task may depend on the operating characteristics of the particular resource on which it is assigned at runtime. A second representation problem is that most graph analysis algorithms treat *either graphs with weighted links or graphs with weighed nodes*. In addition, communication links in this representation combine a system cost (communication time) with a task interconnection and precedence relationship. Modifying the representation so that the actual communication actions are represented by additional nodes resolves these problems but introduces some additional problems when relating the task graph to a representation of its mapping onto an architecture.[4] The architect is interested in using graph analysis tools, such as those presented in chapter 5. The architect can not change the graph weights, as they are derived from the intersection of the task definition and the configuration resource pool composition. The configurer can only change the weights by choice of resources composition, i.e. by changing architectural parameters.

[4] This will be discussed in chapter 7, when the concept of binding is introduced.

3.3.3. Sources and Sinks

In the ALPS framework, graphs contain *sources* and *sinks* which represent interfaces between the ALPS system and the "outside world." Sources operate as task initiators: they are invoked, typically by the receipt of data, and initiate a new task arrival. In d-ALPS, that arrival, represented by a collection of data frames with a unique arrival number, is then introduced to the system with the implicit demand that it be operated on by processors assuming roles of the directed task graph. In the d-ALPS specification, the sources are assumed to operate at fixed rates; the inter-arrival time between task demands is constant.[5]

3.3.4. Graph Detailing Parameters

The above sections described the essential components of the task definition provided to the configurer. Within a task definition, there are a number of elements that can be changed. The task graph representation provides a set of overarching relationships among tasks that must be conserved, but does not completely specify an execution schedule. In particular, the representation does not provide scheduling information about inter-task-arrival relationships. Both an architect and a configurer would want to investigate these changes because they can affect an eventual execution order. These changes are called *graph detailing parameters* because they impart static scheduling details to the task graph on top of the static precedence information already encoded in the task graph. The parameters that can be changed depend to a great deal on the underlying ALPS architecture. The changes to the task definition which are legitimate and have an affect on performance of graphs executed by a d-ALPS configuration will be investigated. The reader is referred to the d-ALPS specification in Appendix A for a more complete description of the task distribution mechanism it employs. In addition, those basic changes which could be applied to other underlying

[5] This assumption can be modified to more general task arrival notions without changing the underlying representation.

ALPS architectures will be presented.

3.3.4.1. Link Ordering

In d-ALPS, a node transfers information corresponding to successor subtasks via DMA-style block transfers. Nodes in d-ALPS can not engage in block transfers of different information at the same time, but may transfer the same information to more than one place simultaneously. As a node can engage in only one physical data transfer at a time and as it may have to transfer a number of blocks of data, there must be some (not necessarily static) ordering to these transfers. For N transfers of the *same* information, there can be 0 to N simultaneous transfers[6] of the data to different destinations, and the order of those transfers could be specified via detailing the the task graph. In d-ALPS, this ordering can not yet be effected, but could be implemented with minor changes to the graph encoding and servicing mechanism.

Receiving nodes are subject to a similar serialization. Data transfers represented by communication links into a node can not occur simultaneously in the d-ALPS architecture because the node that is the target of multiple receive requests can physically receive only one data transfer at a time. The order of the received data blocks could be specified in a manner similar to that used for outgoing communication links.

The task graph represents data transfers and broadcasts of different information as possibly concurrent operations. To reflect the mapping of that behavioral description of tasks to an architecture-specific, implementation ordering, the graph should reflect concurrency only in subtasks that are plausibly concurrent. From a representation point of view, a set of graphs can be generated, each of which provides a different ordering of transfers into a

[6] It may not be possible or desirable to send to all destinations of the same information at the same time, i.e. a logically complete broadcast, though the cost is additional data transfers to those receivers which have not been addressed.

node. Each of these graphs would then represent the original task graph which has been *detailed* with additional ordering information. A detailed procedure for creating these representations is presented in chapter 5.

3.3.4.2. Link Priorities

In d-ALPS, a simple mechanism ensures that a single physical node is bound to a subtask. This mechanism is described in detail in chapter 7 and again in Appendix A. One of possibly several links that merge to a subtask is statically labeled a *priority* link; when bids are processed for this link, the receiving node is bound to the subtask. Only the node which has received the priority link can bid for the remaining links. This mechanism guarantees that a single physical node is bound and requires no control communication between the receiver of the priority link and holders of other links until those links are bid for. There are two major considerations with this approach. First, owners of nonpriority links that are finished before the priority link will engage in futile bidding requests. Second, the node which has received the priority link before all nonpriority links are finished will be in a bound-but-not-processing state until all remaining links (and subtasks leading to their eventual transmission) are finished. The choice of priority links is encoded in the task definition.

3.3.4.3. Delay and Precedence Insertion

A task graph can be modified to include signals and delays that serve to break up synchronizations of demand and reduce polling. For example, delays can be inserted to equalize paths of execution that eventually merge. In the d-ALPS architecture, the absence of these delays does not imply that collisions will occur, as in the case of pipeline processor collisions, but will result in unnecessary polling or processor binding: additional overhead. While delays can be added anywhere in the task graph without disturbing the functional relationships among tasks, their insertion at a task behavior specification level, where execution

costs are not known, is problematic.

Delay insertion at the task graph definition level can be thought of as a timing-static technique for path equalizing or subtask ordering; it is a problematic technique when specified in an environment where timing relationships are not fixed or not determinable. As an alternative, an asynchronous method of performing the same functions is signaling. A nonoperational graph node can be added which gathers results or signals from one set of nodes and causes the initiation of another nonintersecting set of nodes. While this representation gives no clue as to how the signal is implemented—there is currently no mechanism in the d-ALPS architecture to support these signals—it provides a timing-independent method of synchronizing subtask paths.

A technique similar to signal node insertion is precedence insertion. A precedence link is one which has no data block (communication subtask) associated with it but is treated like other links. In d-ALPS, as a priority precedence link, it would cause the binding of a processor; as a nonpriority precedence link it causes a bound processor to delay execution until this signal is received.

3.4. Architectural Parameters

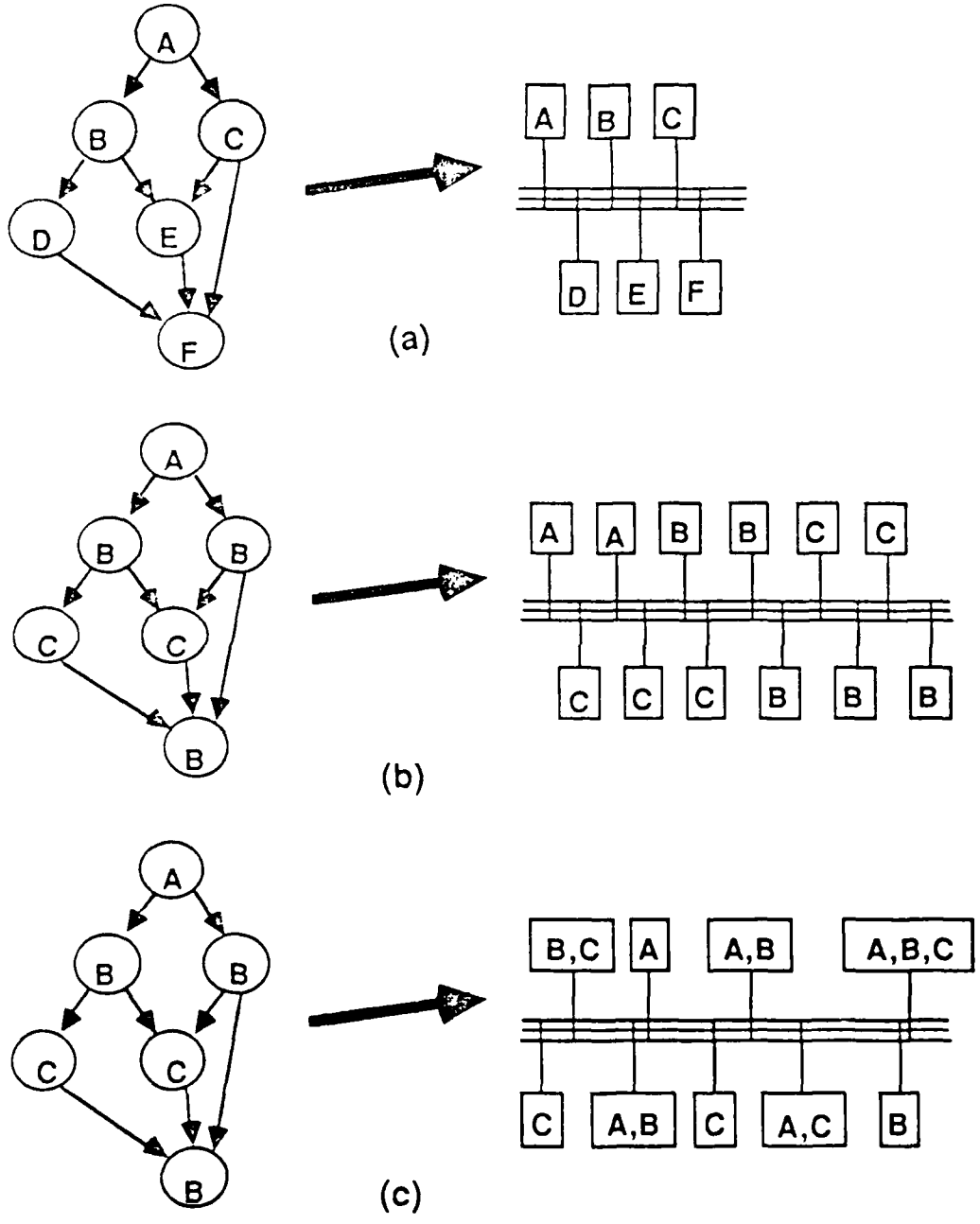
A detailed presentation of the d-ALPS architectural specification and a discussion of the high level logical control is provided in appendix A. This specification includes an overview of the functional requirements of an ALPS system and provides a detailed presentation of the distributed internal and network control and communications functions, currently implemented by interface control units. In order to specify and study ALPS architectures, it is important to know what the parameters to the system are. Some of these parameters are obvious. A system configuration is based on pools of similar resources, where the resource count is a key parameter to that configuration. Some of these parameters are less obvious.

A resource interface control unit (ICU) contains memory used for queueing output results: the amount of memory available for output queueing is, within limits, a parameter specifiable by the configurer. Following is a description of the parameters that can be supplied to a d-ALPS configuration architecture. Other architectures may allow more or less variability in the architecture; while it is not the scope of this thesis to discuss the architecture descriptions that are possible with architectures that have not yet been specified, where appropriate some proposed descriptions that an eventual architecture should (or may) allow will be presented.

3.4.1. Resource Pool Size and Composition

The size and composition of the resource pool in an ALPS system gives an upper bound on the processing capacity of the particular configuration. At minimum, there must be enough processing capacity to handle the processing demands set by the task graph and arrival rate of tasks. This architecture-*independent*, baseline allocation can be determined statically via procedures described in chapter 5. The desirability of allocations which are near this minimum lies in the small node count: processor utilization will be maximized at the expense of task latency and response to unanticipated delays and innefficiencies. It is difficult to derive minimum *feasible* allocations—allocations which meet execution performance and correctness criteria *and* account for scheduling and control overhead.

In the most simple mapping case, each subtask in the directed task graph will map to a distinct resource, and individual resources in the architecture will be single-purpose processors (fig. 3-3(a)). This allows a simple mapping scheme because each resource can only assume one "role" in the task. If we generalize slightly, there may be many subtasks in the algorithm that require the same processor type; single-typed processing nodes in the architecture now may execute any of those subtasks that require that single type (fig. 3-3(b)). If we generalize some more, some processing resources may be able to execute as many of the



Figures 3-3(a) through 3-3(c): A direct mapping can be made between a uniquely typed subtask and a uniquely identified resource; or, subtasks of the same type can be mapped to single-typed processing nodes; or, subtask of different types can be mapped to multiple-typed processing nodes.

processing types required by the task. A multiple typed node can map to any subtask in the graph which requires one of the resource types that this node can function as (fig. 3-3(c)).

The allocation of resources refers to the type and composition of the processing (and communication) resources assigned to a particular application architecture. The *performance* of those resources refers to the individual processor execution time, in the case of processors, and block communication transfer time (bit rate), in the case of busses.

Subtasks requiring the same processor type may place different demands on that processor and incur different execution costs. In the signal processing application realm, the amount of data that a subtask requires may be the indicator of the execution cost difference between two subtasks that require the same processor. For example, the cost of a vector add operation may be considered linearly dependent on the size of the operand vectors. These relationships get more complicated with the type of operation and the implementation of the processing primitive.

There may be processing primitives in a configuration that are capable of performing an identical operation but have different operating characteristics due to different physical implementations. For example, a SIMD processor array implementation of a two dimensional FFT operation has a processing complexity of order $\log_2 M$ for an M-point FFT whereas a serial implementation (single processor) has a complexity of order $M^2 \log_2 M$. An ALPS architecture which contains special purpose FFT primitives as well as general purpose serial computing nodes (e.g. DSP boards with software libraries) will be able to handle FFT tasks with varying execution costs; a particular FFT subtask will incur a cost determined by the actual processor which receives that subtask. In this case the task definition can not encode the execution costs for methodologies which model d-ALPS because the cost will depend on (and vary with) the binding of a subtask to a resource, decided upon at runtime.

3.4.2. Communications Capacity

Communications in the d-ALPS architecture is supported by multiple, parallel data busses. Access to these busses is controlled by a *token-passing* scheme in which a node which currently "owns" the token is able to reserve space on data channels to set up and execute a small number of data block transfers. Aggregate communications capacity can be simplistically viewed as the number of data busses times the width of each bus times the clock rate. Ignoring all overhead and inefficiencies, the aggregate capacity can not be less than the aggregate demand that the task graph presents to the system. This aggregate demand can be computed in a manner similar to that used to compute processing demand; this will be reviewed in chapter 5. Unfortunately, demand for communications is bursty, meaning that aggregate demand will give only a lower bound on demand. In addition, demand will typically be buffered due to the restricted number of data channels; even though data channels may be fast, their limited number forces serialization points in the execution of an algorithm. Determining the number of communications channels should, under the most benign conditions, depend on the aggregate demand for data transfers; however, the degree of parallel demand and the performance degradation of the system when this demand is serialized might influence a configurer to change this allocation.

3.4.3. Memory

In d-ALPS, physical memory limits on processing resources bound the number of output data blocks that a node may queue up. Under this physical limitation, bounds may be placed on the number and composition of queued data blocks. By imposing *dynamic* limitations on the size and composition of queues so as to minimize the disparities in queue sizes, servicing rates across distributed queues can be equalized. Limitations that are placed for this reason may or may not be blind to the *particular* data blocks that are located on each

queue, as the overall service discipline is equally unbiased. The actual queue ceiling should be considered an architectural parameter but the mechanism employed to service these queues is a task mapping parameter.

3.5. Task Mapping Specification

The setting of graph detailing parameters, described under task definition parameters, represents methods of detailing the original directed task graph so that while it is a functionally equivalent graph, its mapping is more precisely, statically specified. The *task mapping parameters* describe how the architecture executes the nonstatically ordered portions of the task. These parameters define relationships among concurrently executing tasks (different arrivals) as well as relationships between concurrently executing subtasks of the same arrival which are competing for processing and communication resources.

Some of the above graph detailing parameters, as inputs to a configuration architecture, would impose an ordering or partially static control above the baseline precedence relationships of the original signal flow graph. For example, static link ordering on all links in the graph would determine the scheduling order within each task although it would not necessarily dictate inter-task-arrival scheduling. If some, but not all, of the task scheduling is not predetermined, then the system will dynamically determine the remaining options. This is one way to view the current scheduling decision hierarchy. There is a general relationship among tasks and a nominal ordering of task execution (priority and nonpriority links and possible graph detailing) above that. The rest of the scheduling decisions are made by the system and as long as the system doesn't deadlock by excluding critical ordering options, the tasks will be processed "correctly."

The function of subtask servicing then is to fill in the gaps between (1) the order (static partial schedule) imposed by the directed flow graph and detailing parameters applied on top

of this graph and (2) a random, unbiased choosing of when and where subtasks that are not related by any restrictions in (1) are assigned.

3.5.1. Data Queueing

In the general ALPS approach, subtasks waiting for assignment to a physical resource will be queued until a suitable resource is found. This queueing can be done at a number of logical locations. Subtasks can be queued at the site of the previously executed task: data blocks sit on *output queues*. Alternatively, data blocks can be transferred to a node that will eventually execute the pending subtask. Subtasks are then held on *input queues*. Finally, data blocks can be transferred to an intermediate holding buffer which then seeks to dispatch these pending tasks. This is logically similar to output queues but it has different ramifications on queue servicing and overhead. The choice of where and how to store data blocks (subtasks) pending execution depends on the architectural support for queueing—in d-ALPS there are output queues and no input queues, and currently no specification for intermediate queues—and the queue servicing mechanisms which are the kernal functions of task mapping.

3.5.2. Queue Servicing

In a correctly functioning d-ALPS system, data blocks will queue up at the sites that they were processed until the associated interface control unit can present a requests for bids for that data block and a bid is received. The assumption is made that a directed task graph has been provided which does not provide a complete set of intra- and inter-task relationships. That is, a node which has several data blocks to service does not have an accompanying servicing list. Servicing decision types can be classified as either graph dependent or graph independent. The specification of a heuristic involves deciding how servicing decisions are to be made and then encoding local weighting or servicing rules.

Graph independent servicing disciplines make no connection between specific data blocks and their association with particular subtasks and task arrivals. Servicing mechanisms in this category can base decisions on general *fairness* principles; examples are equal servicing rates, age, LIFO, or FIFO.

Graph dependent servicing disciplines can make use of static information about locally resident subtasks to assign priorities or preferred orderings. Mechanisms which fall into this category might assign some relative preference based on the global age of a subtask (e.g. its arrival frame number) or on the number or type of subtasks that must wait a subtask to complete. This latter mechanism is called a *downstream cost* servicing policy. The d-ALPS specification has a single mapping mechanism and is based on a graph independent FIFO discipline, though it is not strictly FIFO. The configurer operating with the d-ALPS architecture underlying the configuration would not be able to choose alternate servicing mechanisms. The architect is interested in investigating these mechanisms; d-ALPS mechanism is by no means the final word on task distribution for ALPS.

3.5.3. Priority Bidding

It may be desirable to assign subtasks for which a gradient of processor performances exists to faster processors. This is particularly true if the configuration architecture contains special-purpose primitives for some but not all subtasks processing types. A general purpose resource (such as a DSP primitive which can perform a multitude of functions) may be comparatively fast in executing some subtasks and comparatively slow in executing others for which special-purposes devices are allocated. The specification of priority bidding involves encoding a partial mapping preference of specific subtasks to particular processor performance classes. The d-ALPS architecture provides a subtask-independent mechanism for prioritizing utilization of processors of higher performance.

3.6. Conclusion

The above parameters require isolated and joint investigation. Isolated investigation of task definition parameters is afforded by static analysis methods, which assist the architect or configurer in determining basic task requirements and choosing graph detailing parameters. Simulation approaches integrate the task definition into investigation of architecture and mapping parameters. Architectural parameters are studied in isolation by a careful reading and analysis of the particular architecture specification. Individual components can be prototyped or modeled; they can be supplied with random requirements so as to avoid biasing the investigation by a particular set of task definition and mapping parameters. The underlying architecture can be simulated or *emulated*. These are joint investigation methodologies that require task definition and mapping parameters to function. Task mapping parameters can be studied by schedule simulation, a high-level tool for investigating task distribution heuristics independent of an underlying architecture. This simulation requires task definition parameters and *configuration architecture* parameters, such as resource pool composition. Particular implementable task mapping mechanisms can be studied by architectural simulation. This methodology incorporates mapping, task definition and architectural parameters that are salient to the level of detail of the simulation. The remaining chapters introduce and describe the static analysis, schedule simulation and architectural simulation methodologies.

CHAPTER 4

Problem Setup

This chapter will introduce four methodologies for studying allocation and scheduling problems for d-ALPS architectures. The development of these methodologies was motivated by the limited usefulness of detailed architectural simulation to address a range of study objectives. These objectives included gathering basic information about the underlying task set, characterizing the complexity of scheduling and allocation problems, examining some performance tradeoffs of particular scheduling approaches, and verifying in detail a particular approach. In the sections below, a discussion will be given of how the alternative methodologies evolved from perceived and identified shortcomings of the detailed architectural simulation, the fourth methodology. Next, a review of each of these methodologies will be presented. These reviews will serve as a preface to the remaining portions of this thesis. The scope and structure of the remaining chapters is to examine and evaluate alternative problem representations, explain why and how those representations or models are useful, describe the analysis and conclusions drawn from these methods and discuss further applicability of the approach.

The four methodologies are *static problem analysis*, *state/schedule generation*, *scheduling simulation*, and *detailed architectural simulation*. Static problem analysis refers to analysis techniques which extract basic information about the demands that a specified task and arrival rate poses on an application system. The objective of this group of techniques is to characterize the task set so that basic resource requirements can be discerned and alternative orderings of tasks can be enumerated and compared. State generation encompasses methods of representing a simplified, deterministic version of the task and the architecture.

The task-architecture conjunction is viewed with a finite state machine representation. Finding and enumerating schedules can be viewed as a problem similar to state generation and cycle detection. There are techniques of deriving the states of this FSM; however, they are computationally expensive. By demonstrating these techniques the difficulty of generating even simple, static schedules can be established. Schedule simulation is a method of comparing task mapping and allocation heuristics. Its objective is to provide an inexpensive way of analyzing the underlying task set in a highly instrumentable environment and determining what types of allocation and scheduling strategies may work better than others. Detailed architectural simulation provides a performance view of the actual processing of a task on a configuration architecture. The objectives are to evaluate the performance of a chosen allocation and scheduling strategy on a target architecture and to decide whether that configuration meets performance criteria. Simulation can be used to determine allocation and tradeoffs by iteratively simulating with different initial parameters; this is a fairly costly approach but may be well worth it when considering an application configuration.

A straightforward approach to studying allocation and scheduling problems within the context of an ALPS-like architectural approach is to choose a set of example problems, simulate the behavior of a given architecture that implements each problem, and derive conclusions or observations about the system, task mapping and task detailing parameters. Consider the following example. Figure 4-1 shows an application graph and table 4-1 gives corresponding information about data transfers between subtasks. Table 4-2 gives information about the execution architecture. By submitting this algorithm and architectural description to an architectural timing simulator, various performance results can be generated. These results reflect the capability of the allocation and the underlying scheduling mechanisms of the configuration architecture. It is important to keep in mind that the configuration architecture is comprised of individual nodes and busses that are described an architectural

Algorithm (task) Information"		
Subtask	Processor Type	Transmit Block Size (k bytes)
d1	Source	2
d2	Source	2
mos	Magnitude	2
o	Vector(add)	2
iso2	Vector(subtract)	2
done	Sink	NA

Table 4-1: Task information for simulation example.

specification. One performance metric of this architecture is the utilization of processing resources, the percentage of time that processors are executing. A simulation of the architecture can provide this utilization information. Figure 4-2 shows processor utilization for the architecture as it executes the algorithm in figure 4-1. Note that processors spent most of their (aggregate) time sitting idle. Two other metrics are the throughput and later of the system. The throughput is the rate at which tasks are processed by the system, and the

Architecture Information		
Resource Type	Execution Time (μsec)	Quantity
Source	2000 (interarrival time)	2
Vector	[add] 800 [subtract] 1500	4
Magnitude	1200	3
Sink	0	1
Bus	[16 bits@10 MHz] 50 per 1K block	2

Table 4-2: Architecture information for simulation example.

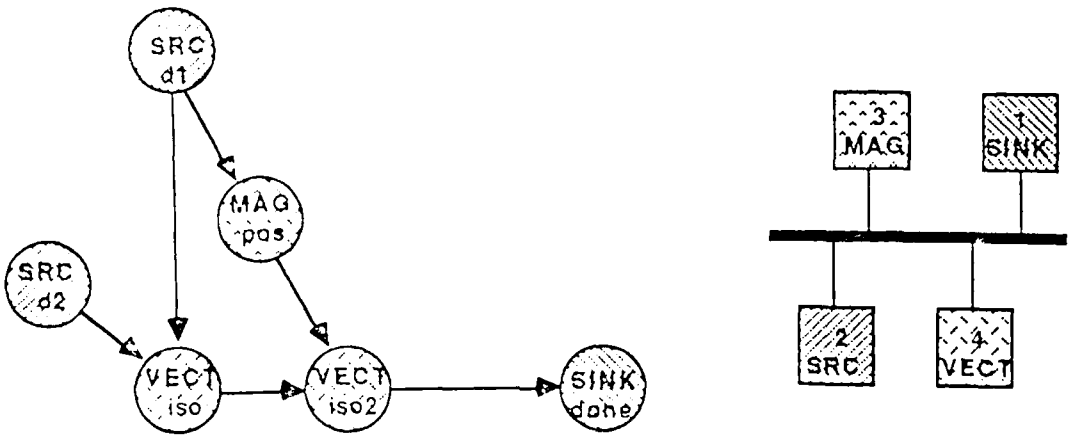
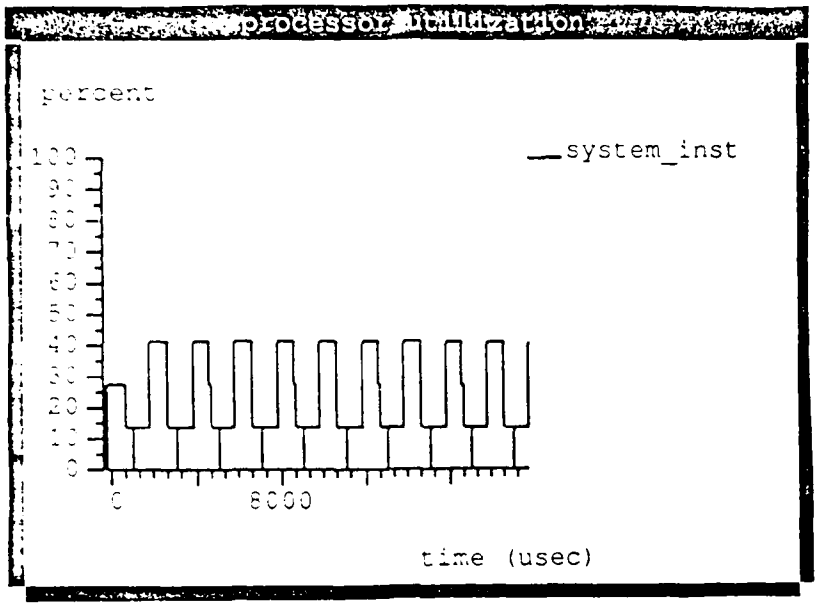
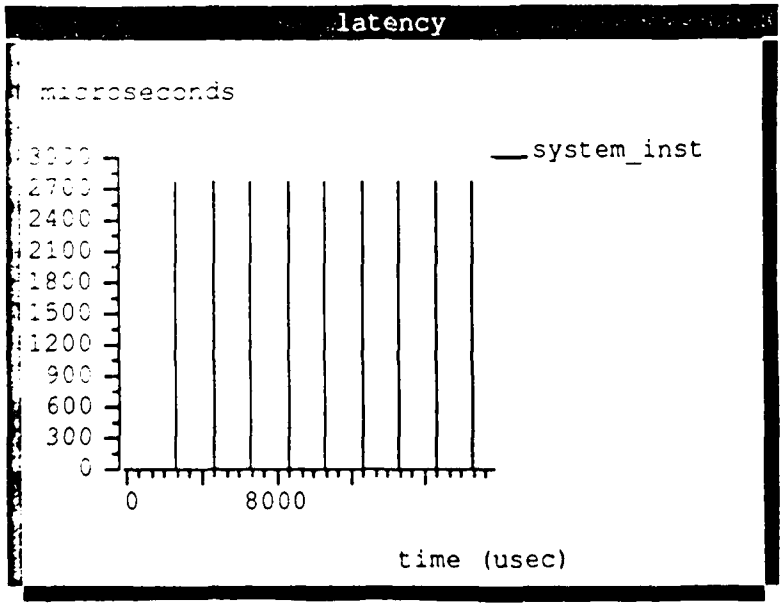


Figure 4-1: An example task graph and architecture.

latency is the time it takes to process those tasks. Figure 4-3 demonstrates that, after the "pipe" fills up, tasks are being processed at a rate of about 1 every 2000 microseconds. This rate is comparable to the interarrival time of tasks: the system is keeping up with the specified arrival rate. The latency of the system is 2800 microseconds and is fairly constant. As a result of the above simulation, we might want to reduce the pool size of various resource types and investigate the results. This may yield a more efficient (and inexpensive) implementation that maintains a desired level of performance. A new, reduced allocation is shown in table 4-3. Performance metrics for this allocation are shown in figures 4-4 and 4-5. Notice that the processor utilization is much higher (there are less processors for the





Figures 4-2 and 4-3: Processor utilization and system latency for the example task graph and architecture in figure 4-1.

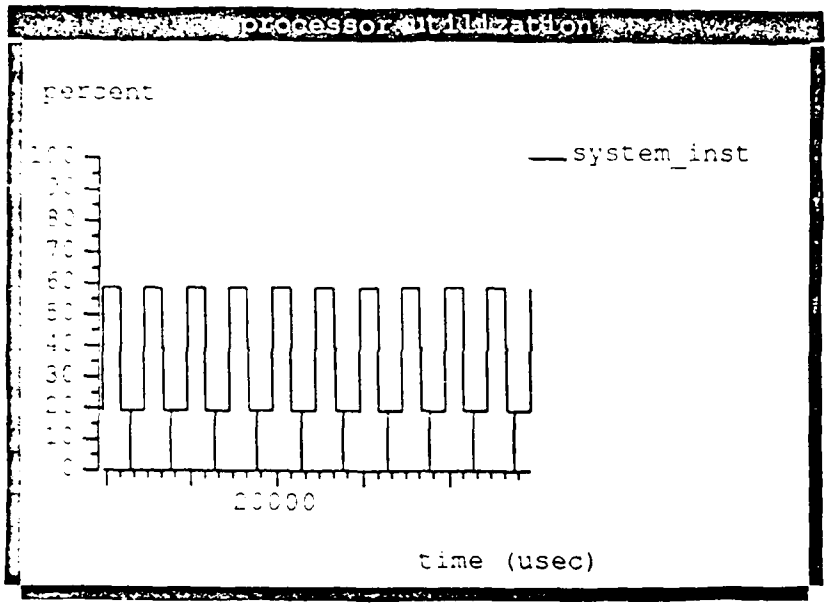
Architecture Information		
Resource Type	Execution Time (μsec)	Quantity
Source	2000 *	2
Vector	[add] 800 [subtract] 1500	2
Magnitude	1200	3
Sink	0	1
Bus	[16 bits@10 MHz] 50 per 1K block	1

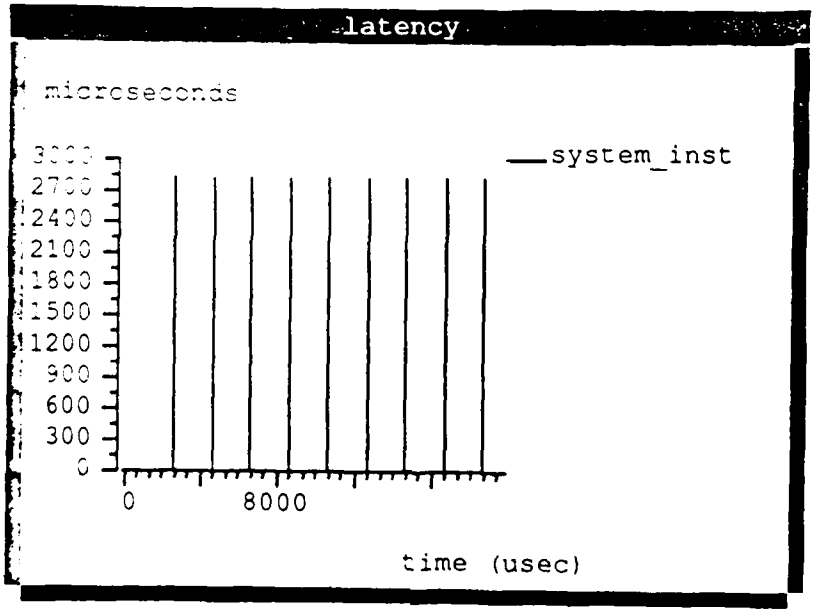
Table 4-3: Modified architecture information for simulation example.

percent calculation). The latency is slightly larger (2871 μsec), but the system is still able to accomodate the specified throughtput.

Many of the parameters described in the previous chapter could be varied to develop relationships among them. This approach, straightforward as it may sound, is problematic. Results derived from this detailed simulation and analysis on "real-world" example problems are subject to numerous extraneous considerations: What is the effect of the underlying support architecture? What are the effects or influence of memory requirements or limitations? What are the scheduling mechanisms employed by the architecture? Moreover, gleaning extensible information and observations about scheduling and allocation parameters from specific problem examples hinges on the necessity of finding useful *benchmark* problems. In the above example, the results provided are applicable only to that specific algorithm and architecture. This algorithm can not be considered to represent a *class* of applications and therefore may say little to nothing about allocation or scheduling approaches to related problems.

Following is a review the kinds of information that architectural simulation provides, and the view of the system that it assumes. From the user's point of view, an architectural





Figures 4-4 and 4-5: Processor utilization and system latency for the task graph in figure 4-1 and the architecture information in table 4-3.

simulator casts the effects of varying system parameters into performance measures for which the simulator was instrumented.[1] In the above example, the performance measures that the detailed architectural simulator supported included latency, throughput and utilization of processors, busses, and memory. These measures were computed over a limited number of domains. Adding more instrumentation to a simulator is often a difficult enterprise, but more importantly, from a methodology point of view, presupposes to some extent knowledge of the objective of the simulation. The simulator builder must be conscious of the metrics that will be interesting to both an engineer performing an ALPS configuration study— a *configurer*— and an engineer studying basic allocation and scheduling problems and designing an underlying support architecture such as d-ALPS—an *architect*. A detailed system simulation, as implemented, provides a configurer with *performance* view of the system. A simulation system which caters to an architect should provide "tools for tinkering around." What this means is that scheduling and allocation investigation can be supported by separate but integratable utilities that perform some specific analysis chores and include different assumptions of the underlying architecture.

The methodologies that will be presented attempt to fill in the knowledge gaps by providing investigation tools to the architect. The objectives of the modeling methodology then are reconciled with the varying scopes of the investigation. In exchange for excluding many of the operational details of a particular architectural specification, alternative representations and system models provide some additional insight into underlying scheduling and allocation problems. Once a representation proves fruitful, it can be refined and retrenched so that it more closely characterizes the underlying architecture.

[1] It is also possible to create a simulator that provides an *execution trace* of the system that it is simulating. This type of information is problematic because it is often too difficult to filter out useless information.

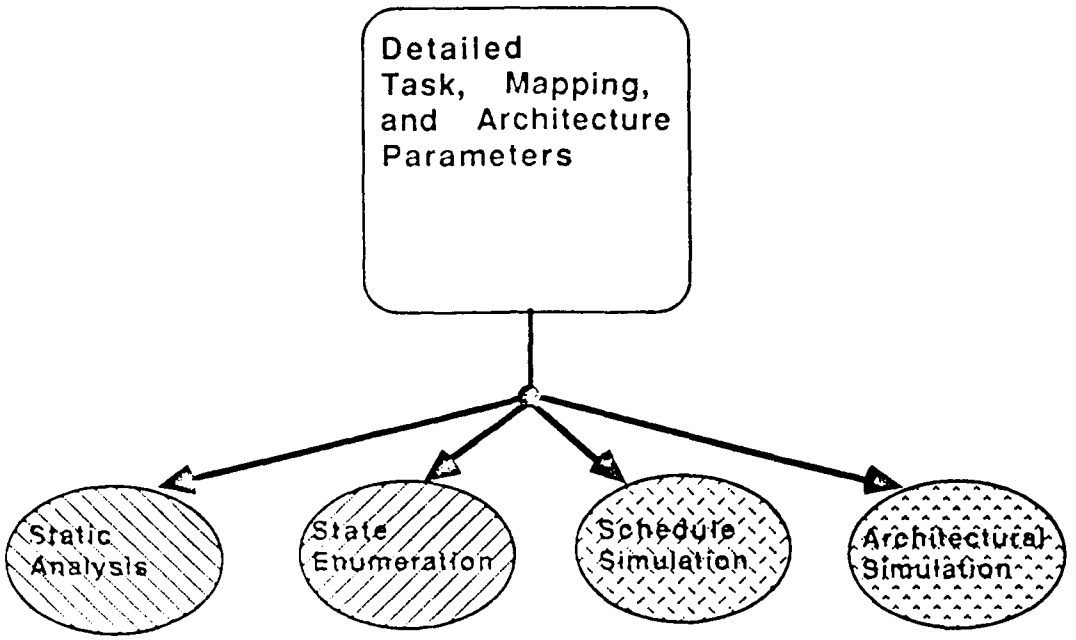


Figure 4-6: A representation of the methodology information hierarchy.

The alternate approaches chosen were developed by deciding upon the objectives of the methodology and then deciding how to fulfill these objectives. The first approach, static analysis, fills the objective of learning about the baseline demands that a task set and arrival will place on a hypothetical system and whether a "reasonable" system can be configured. This provides pre-mapping information, whereas detailed simulation provides information about a *particular* architectural mapping. The second approach, state enumeration and generation, gives a measure of how difficult it would be to investigate a particular scheduling strategy and provides an example framework for generating these strategies. This provides a complexity comparison and a comparison to a methodology suitable to a hypothetical architecture with scaled down timing considerations. The third approach, schedule and allocation

simulation provides a useful set of tools to evaluate different scheduling heuristics and to try different graph ordering and priority assignments. This allows the user to change some underlying architectural assumptions that a simulator would not provide, and provides a low-cost way of generating these alternatives where a simulator would require high-cost modification.

The four modeling methodologies do not make identical assumptions about the underlying scheduling problems and then perform different types of analysis. Instead, the four

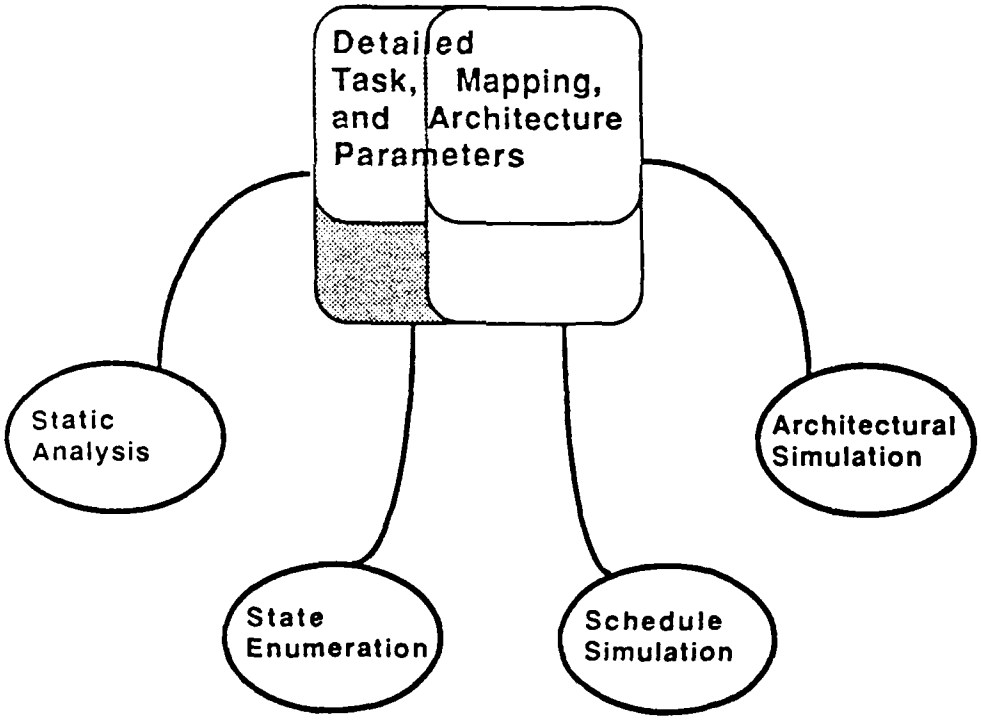


Figure 4-7: An alternative representation of the methodology information hierarchy.

methods make use of different types of information and require differing levels of detail about the underlying architecture. Two approaches to representing the information hierarchy are shown in figures 4-6 and 4-7. The former approach provides a super-set of information about the possible descriptions of the task set and architecture, but part of this information lies in alternatives that are never enumerated or never considered. A more realistic view then is shown in the representation of figure 4-7, in which overlapping subsets of information are extracted and supplied to the methodologies. Table 4-4 provides a summary of the types of information about the task/architecture that is provided to each of the modeling methods. Note that the static analysis method is architecture-independent, whereas architectural simulation requires a full description of the architecture which includes detailed implementation information that would be useless to other analysis methods.

Following is a review of each of the four modeling methodologies that have been introduced. In each section below, a methodology is summarized, its objectives are reviewed and a brief evaluation is given. These sections serve as a reader's guide to the remainder of this thesis.

Architectural Information Required by Model	
Static Analysis	none
State Generation	resource composition, queue servicing heuristic
Schedule Simulation	resource composition and addressing, queue servicing heuristic(s), graph mapping alternatives
Architectural Simulation	complete architectural specification, fully defined mapping specification and implementation

Table 4-4: Architecture information provided to each model.

4.1. Static Analysis

A directed flow graph provides an algorithmic representation of a problem in terms of computation subtasks and communications and precedences among those subtasks. The mapping of this graph onto a distributed architecture—in which each algorithmic node is mapped onto an architectural node—produces two classes of allocations. The first class is one in which there are ample resources to process this task set on a *demand* basis. That is, subtasks wait only on the completion of other subtasks with which they are involved in a precedence relationship and do not have to wait on processors or resources once they are ready to execute. The demand graph can be modified to account for all architectural and timing necessities *except* a lack of resources.[2] An example of an acceptable modification is the re-ordering of a some of the processing and communication subtasks to accomodate processing and communication by a target architecture. The incorporation of demand graph modification allows the class description to encompass a large class of static solutions. This demand processing environment can take many forms. In a static form, resources are allocated and connected in a temporally-equivalent manner, i.e. there are either private processors and data channels or reserved time/space on existing channels for each subtask. In a dynamic form, ample resources accomodate the demands of the task set and while time/space is not reserved, there is enough surplus processing/communicating power to execute the graph directly. The basic characteristic of this class, regardless of its implementation form, is that task execution follows the *demand* described by the directed flow graph and a schedule can be directly inferred from the (possibly modified) graph. An allocation which facilitates this demand execution will be referred to as a *class one* allocation.

---

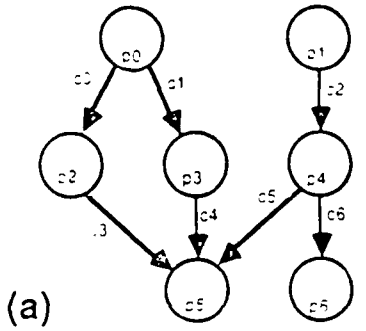
[2] These architecture allowances may not include systematic queueing to make up for lack of resources unless that queueing were explicitly built into the original application graph.

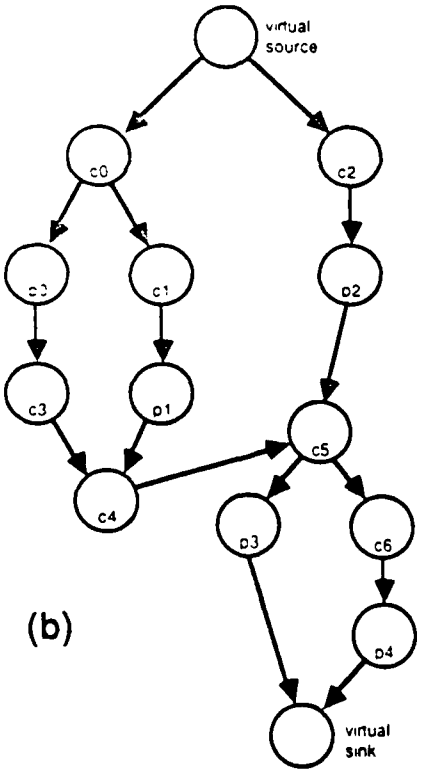
In the second execution class, the task definition poses processing requirements on the system, but an implementation schedule can not be inferred from this definition. The system processes tasks according to its own scheduling and assignment rules. This schedule may be generated statically or dynamically. General purpose computing systems fall into this broad category as they have scheduling policies and resource availabilities which are defined independently of a particular task graph.

The reason these two classes have been defined is as follows. When evaluating the directed flow graph, profiles of processor and communication resource demands can be made. Architectures of the first class require the minimal system resource allocation that accomodates peaks in the demands of the task description. Architectures of the second class require system resource allocations that must, as a minimum requirement, provide resources to accomodate average demands of the task description.[3] This allocation will be referred to as a ''class two'' allocation. The difference in the two classes, then, is whether allocation and scheduling decisions must accomodate peak or average resource demands.

The ''static analysis'' modeling methodology is used to expand a task definition graph to include some assumptions about the underlying architectural specification and protocol (provided in appendix A). This is done in order to develop rough estimates of the peak and average allocation requirements. The steps in deriving this expansion are as follows. A standard representation for processor and communication resource requirements and precedences is created, including explicit representation of communication tasks. The ordering and binding of processors is then taken into account by serializing the order in which communication tasks leave common sources and merge to common destinations. A steady demand period is derived by computing the latency of the graph and then overlaying

[3] From a statistical point of view, if a fixed servicing rate equals an average demand rate, the system is bound to bottleneck. The servicing rate must be greater than the expected demand rate.





Figures 4-8(a) and 4-8(b): An example application graph and.
and an expansion of that graph to represent an execution order.

representations by a defined interarrival time. The computations are performed by linear programming techniques.

A simple example of this technique is as follows. Consider the application graph in figure 4-8(a) and the subtask timings provided in tables 4-5 and 4-6.[4] The graph can be expanded to represent one of many possible execution order. This expansion is shown in figure 4-8(b). Notice that the precedences of figure 4-8(a) have been preserved, but

4-5: Processing Subtask Information		
Subtask	Processor Type	Execution time (generic units)
p0	Source	8
p1	Source	8
p2	A	8
p3	B	4
p4	C	5
p5	Sink	0
p6	Sink	0

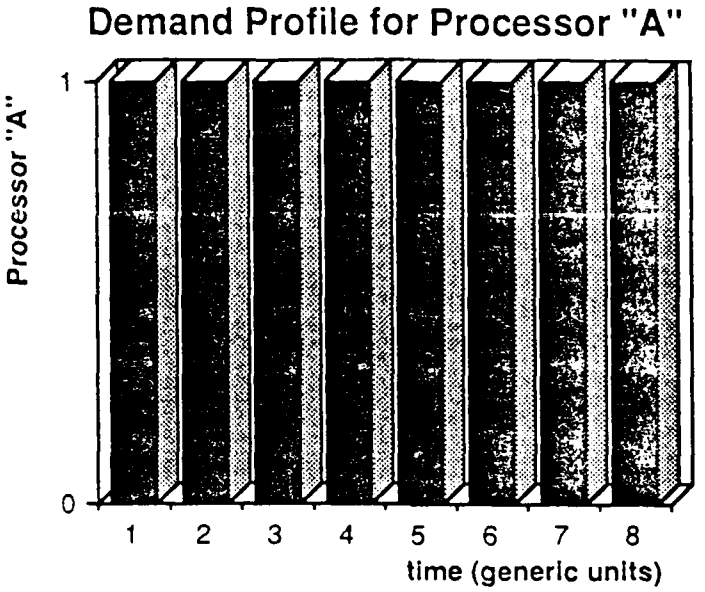
4-6: Communication Subtask Information	
Subtask	Communication time (generic units)
c0	2
c1	3
c2	5
c3	2
c4	4
c5	8
c6	6

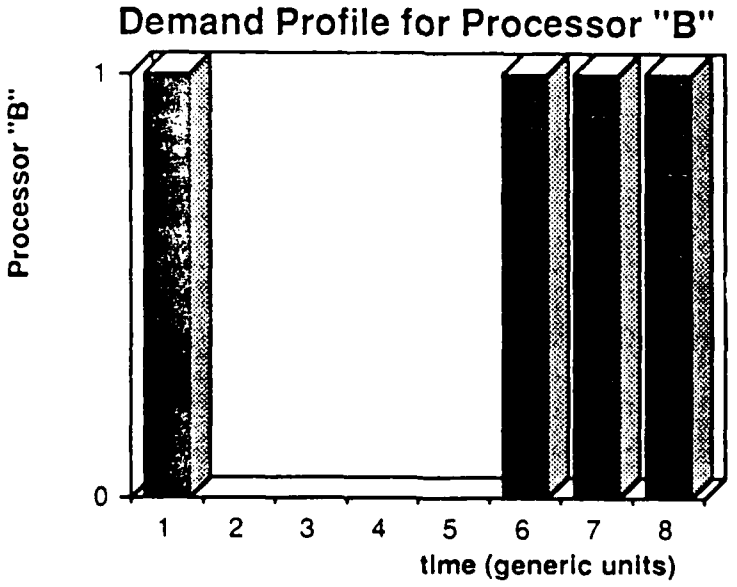
Tables 4-5 and 4-6: Processing and communication subtask information corresponding to task graph in figure 4-8(b).

[4] The timings provided for the sources represent the interarrival time of tasks.

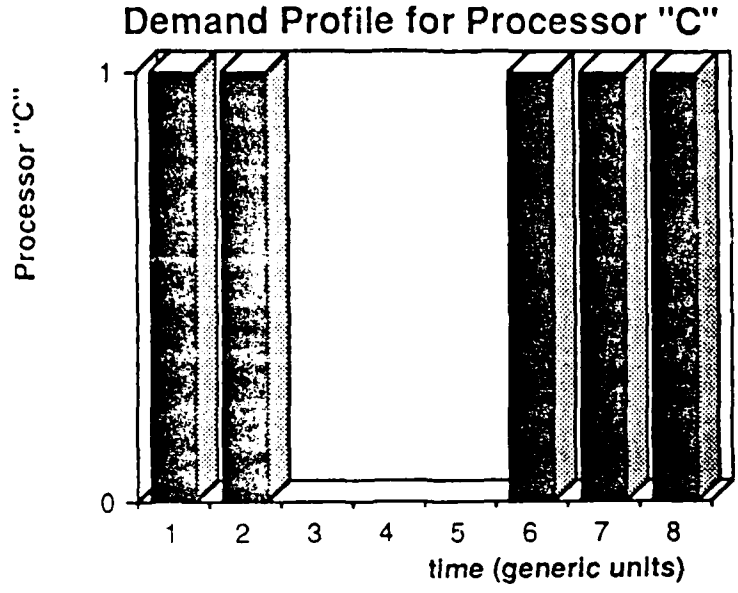
additional ones have been added to *serialize* communication broadcasts and mergings. The latency of the resulting graph is thirty time units and can be computed via linear programming methods by considering it as a maximum path problem.

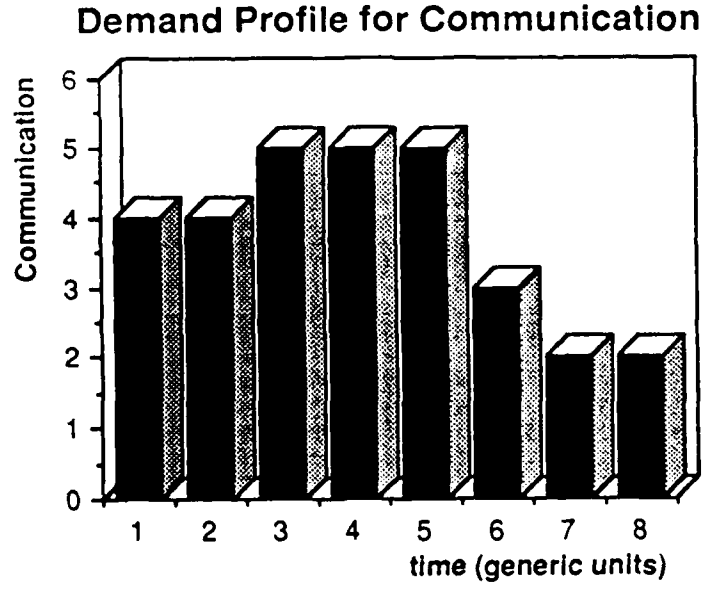
The demand profiles generated by this technique as an intermediate result represent the demands for processors and communication resources that are imposed by the underlying task description in the absence of restrictions imposed by configuration allocations or overhead costs, but in the presence of communication serialization and binding. If a configuration architecture which falls roughly into the first execution class is desired, i.e. an execution schedule is implied in the task description, allocations at or above the peaks in these demand profiles are required. All other allocations will create a system which falls into the second class. Assume that the inter-arrival time of tasks in the above example is eight units. A set of demand profiles for the steady state system is provided in figures 9(a) through 9(d). Notice that both class one and class two processor allocations can be accomodated by providing one of each resource type. The peak demand is one processor and the average is less than one. Of course, depending on the inefficiencies of the execution environment, one processor may, in fact, not be sufficient. In figure 4-9(d) the demand profile for communication resources is provided. This application is communication intensive. A raw ratio of communication to processing time is 1.75. In order to provide a class one allocation, at least five communication channels are required (assuming no inefficiencies). In order to provide a class two allocation, at least four communication channels are required. Chapter 5 will describe the application of static analysis in terms of an example and will evaluate its applicability.





Figures 4-9(a) and 4-9(b): Demand profiles for processor types A and B.





Figures 4-9(c) and 4-9(d): Demand profiles for processor type C and communications resources.

4.2. State Control Strategy Generation

A second methodology for studying scheduling alternatives involves the generation of states which represent the mapping of the task graph onto an architecture. A control strategy is a deterministic, cyclic schedule of task (or subtask) initiations and communication events. It can be used to either impose a schedule on a system or *describe* a system operating in a steady state. That is, if a system has a complete description of a nonvarying task set, a deterministic procedure for mapping those tasks, and a fixed task arrival rate, then the system will exhibit a periodic behavior.[5] Given a simplified model of the task set and the underlying system, it should be possible to generate schedules which demonstrate this periodicity in a manner similar to the control strategy generation for processor pipelines.[6] The substantial difference between pipeline scheduling and this method is that the processor pipeline case is one in which a strategy is developed to *determine* a mapping, whereas in the enumeration methodology a strategy is determined to *describe* a mapping. A control strategy for processor pipelines is implemented as a hard-wired schedule, whereas this mapping description falls out of the scheduling and control mechanisms of the underlying architecture.

The applicability of the schedule enumeration approach to the configurer is limited because of its computational complexity and the number of assumptions about the underlying architecture that must be made. The direct utility of this method to the comparison of configuration architectures is fairly limited due to the number of states that must be included in order to develop a reasonably detailed model. The applicability of the methodology to the architect lies in the comparison of the complexity of the class one and class two scheduling domains. The methodology provides a demonstration of the mapping complexities of the d-

[5] The above assumptions are too simplistic if tasks take longer to execute than scheduled or if there is any invariability in demand on the system or in resources available to the system.

[6] The control strategy discussed here is the general one, presented in chapter X, in which latency schedules and delay insertions are the degrees of freedom in determining an *optimal* schedule or one that meets application demands.

ALPS architectural approach by applying a modeling methodology that is widely used in simpler architectural domains to a domain which has additional scheduling and allocation dimensions. The size of the problem into perspective by providing a relationship between the complexity of scheduling and mapping and the complexity of graph path analysis problems. The approach is a valid academic exercise in that it demonstrates the scale of the problem and validates the investigation of heuristic-based alternatives for ALPS scheduling and allocation. Chapter 6 will provide a detailed description of this state generation technique and will discuss its relation to graph analysis techniques.

4.3. High Level Scheduling Simulation

A third approach to studying task servicing and scheduling alternatives is schedule simulation. The approach was motivated by the desire to model scheduling events at a macro-event level—as opposed to performance metric level—while automating the modeling process so that large-grain architecture parameters could be accounted for and studied. A simulator was developed which provides a method for examining scheduling and allocation alternatives. The parameters to this simulator include the task graph, the allocation of a configuration architecture (or a method of iteratively generating these architectures), and the subtask ordering methodology employed by this system.[7]

The simulator takes a demand-versus-supply view of scheduling and allocation problem. The application architecture is modeled as a task demand server; at each stage a demand is presented to the system, represented by the activation of new subtasks, the ongoing execution of subtasks, and the arrival of new tasks. Some of these demands can be put off (queued) and some must be served immediately. The simulator then acts to fill demand, assigning as many subtasks as allowed by the allocation. When the demand outweighs the

[7] No assumption is made as to the feasibility of the described ordering methodology. Some methodologies may not have feasible implementations in a distributed, dynamic assignment architecture because of the magni-

supply, the simulator turns to a heuristic-based delay server which ranks the importance of pending subtasks according to one of a suite of heuristics. This ranking makes use of either centralized or distributed information about tasks and determines which tasks must wait and which can proceed. Allocation alternatives are considered in the context of scheduling heuristics by iteratively applying the simulator to systems of different allocations; the simulator keeps track of the types of delays that are induced by nongenerous allocations. The experimenter is then provided with a view of these ordering tradeoffs as well as resulting performance measures.

From a configurer's point of view, this level of simulation can used to determine an initial allocation and scheduling structure for a particular application. Since timing information particular to an architectural specification (such as bidding time) is not incorporated, certain allocations will be useless, from a configurer's point of view, if they are based on a peculiarity of task timing synchronizations that are explicated by this simulator. On the other hand, comparisons of scheduling heuristics will be applicable (i.e. scale up to more "accurate" simulators) because the scoring and heuristic delaying functions are not precise, and will model imprecision in timing synchronizations in a final system. Furthermore, variabilities in timings, such as execution time distributions, can be added to this level of simulation.[8] From an architect's point of view, this level of simulation can provide useful comparison information about general scheduling and allocation approaches. Chapter 7 will describe this type of simulation in detail and will illustrate its effectiveness by considering several examples.

---

tude of control information transfer.

[8] A general model of a software module considers it as requiring a random number of instructions, each taking a random time. By extension to the central limit theorem, the computation time would tend to a Gaussian distribution [Dubo82].

4.4. Architectural Simulation

A fourth method of analysis is the extraction of information about the performance of particular tasks on fully specified architectures. A simulator which models the timings of a particular architecture was developed. The simulator models the appearance of new tasks on physical sources at fixed intervals and the subsequent assigning of subtasks via a distributed and dynamic bidding scheme. The simulator provides an environment in which an *algorithm*[9] and an architecture can be specified and the performance of the architecture can be viewed by choosing from a set of performance metrics and *domains*, or categories, over which those metrics are computed.[10] The simulation environment provides verification and performance comparison of a subset of the task, system and mapping parameters presented in chapter 3.

A basic comparison between architectural simulation and a scheduling simulation is that an architectural simulator trades detail and accuracy in the representation of a specific architecture for degrees of freedom in parameters to vary. Some of the mapping parameters are realized in the architectural specification. These include queue servicing disciplines and link ordering choices. In order to vary these parameters to an architectural simulator, an *implementation* approach must be designed and verified, and a simulator built to model that approach. For example, if a priority queue servicing discipline is to be investigated via detailed simulation, it would be necessary to first determine modifications to the architectural specification that would implement that discipline. A second (and costly) necessity would be the design or modification of a simulator to accomodate that new specification. Chapter 8 will describe a simulator which models the performance of an initial ALPS architecture which roughly corresponds to the architecture described in appendix A.

---

[9] An algorithm refers to a signal processing algorithm, described by a directed flow graph. The term *task set* is used in this thesis to generalize the applicability of the architecture to other domains.

[10] A description of a prototype configurer's simulation environment is presented in [Mano87].

CHAPTER 5

Static Analysis

Static analysis of a task refers to analysis of that task independent of particular allocation or scheduling disciplines to derive some basic information about the task. A directed task graph representation provides, on inspection, interconnection information about the graph. By some fairly straightforward computation, the representation can also provide timing information. This timing information can be used to derive rough estimates of processor and communication resource requirements. These estimates can then be used to determine whether it is conceivable to configure an ALPS architecture that provides an *on-demand* processing environment, as opposed to one which requires queueing to buffer the demand for resources. Static analysis includes enumerating the *graph detailing* parameters described in chapter 3. Through this enumeration and subseqent application of some simple techniques, relationships between specific graph orderings, resource requirements and latencies can be derived.

5.1. Overview of Methods and Objectives

Following is an overview of the methods and objectives of static analysis. The task representation is refined and a method of generating these refinements is given. The objective here is to impart a uniformity to representation. The expanded directed graph representation provides a view of the most concurrent mapping of a task onto an architecture. Limitations imposed by the underlying architecture are included so that concurrency represented in the graph matches concurrency that is conceivable in the architecture. Those limitations that affect this most-concurrent representation are built into the representation. Once a representation for a particular ordering is obtained, static analysis allows the manipulation of

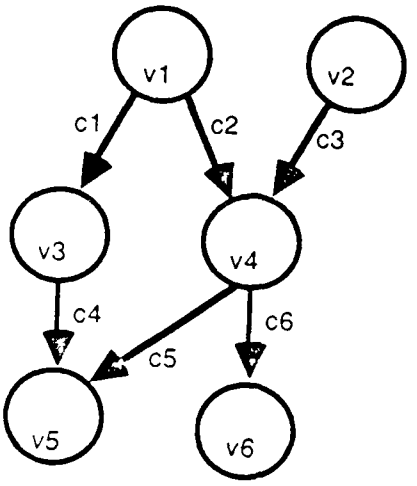


Figure 5-1: An example directed task graph.

graph detailing parameters so that lower bounds on latency can be determined. Likewise, upper bounds on the concurrency of task execution can be found by experimenting with these parameters. Finally, a profile of resource demands that a task imposes in a class one allocation environment can be found. This profile can be determined for an individual task, and can be considered a composite measure of task concurrency. The profile can also be found for a system in which tasks arrive at a constant rate. It will be shown that in an environment in which there is no queueing of subtasks, the resource demand profiles that are generated for a period equal to the task interarrival time are steady profiles. That is, the demands are periodic, and this period can be found easily. These periodic profiles represent the "resource scheduling" of an *idealized* system in which all timing is statically determined, resources are always available, and decision-making time is zero.[1]

[1] If the decision-making times were known and constant, they could be easily included in the model. This is still an idealized case.

5.2. Graph Transformations

Consider the directed task graph in figure 5-1. Nodes v_1 and v_2 represent sources, or task initiation sites. From a *task processing* point of view, those sources perform some operations and then allow their successor nodes (v_3 and v_4) to commence processing. The directed links c_1 and c_2 indicate these successors. When both v_3 and v_4 are finished, nodes v_5 and v_6 can commence. These nodes are *sinks*, and their completion represent the completion of the task. From a *processor network* point of view, nodes v_1 and v_2 represent interfaces to the network from the outside world. They are hardwired physical devices that receive data, perform some operations on some of that data, and then find successor processors to act as the successor nodes in the directed graph. The directed links c_1, c_2 and c_3 indicate the connections between subtasks and imply a communication of information required for execution. In a uniprocessor or tightly coupled environment, this information

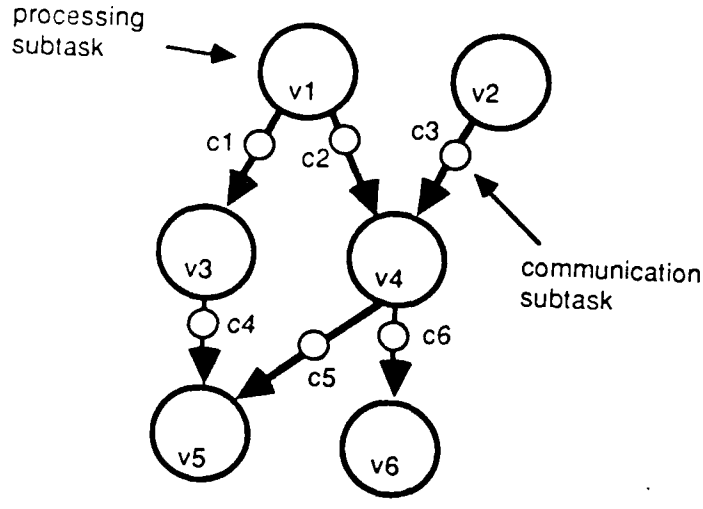


Figure 5-2: A task graph which explicitly represents communication subtasks.

may be a signal or a pointer. In a loosely coupled environment, such as the architecture supported by the specification in Appendix A, this information may include signals, code, and data. The links can be *weighted* with the cost of this communication subtask in much the same way that nodes can be weighted with the cost of processing. A simple transformation of the task graph which explicitly represents communication *subtasks* is shown in figure 5-2. The links between subtasks of this graph represent precedence relationships and have no weight assigned to them.[2]

A representation convenience is to use nodes that lack some of these attributes to aid in connecting "resource-based" nodes. These nodes, called *null* nodes lack resource type and execution cost attributes. An intermediate transformation can be made to a graph which isolates subtasks. This isolation is shown in figure 5-3: the dark shaded nodes are null nodes which precede and succeed each processor or communication subtask. This isolation will aid in representing the permutations that will be made on the graph.

Once the task graph has been transformed to explicitly represent and isolate communication and processing tasks, the graph can be modified to represent a set of execution orders that are supportable on an underlying architecture. If a network of processors could be built that could accept the problem graph in the form shown in (fig. 5-3), then this next step would not be necessary. If considerations must be made to accomodate capabilities of individual nodes to transmit and receive information, and to accomodate interconnection limitations, then this step is necessary. Herein lies a semantic problem. The architecture we are discussing is an underlying *approach* which is, in its ideal, application and configuration independent. The graph modification should, at this stage, accomodate the specification limitations and not the limitations of a particular configuration architecture. One way of viewing

[2] Thus far, the nodes in these graphs have the following attributes: weight, or execution time; resource type required; and the number of links which merge to and emanate from the node.

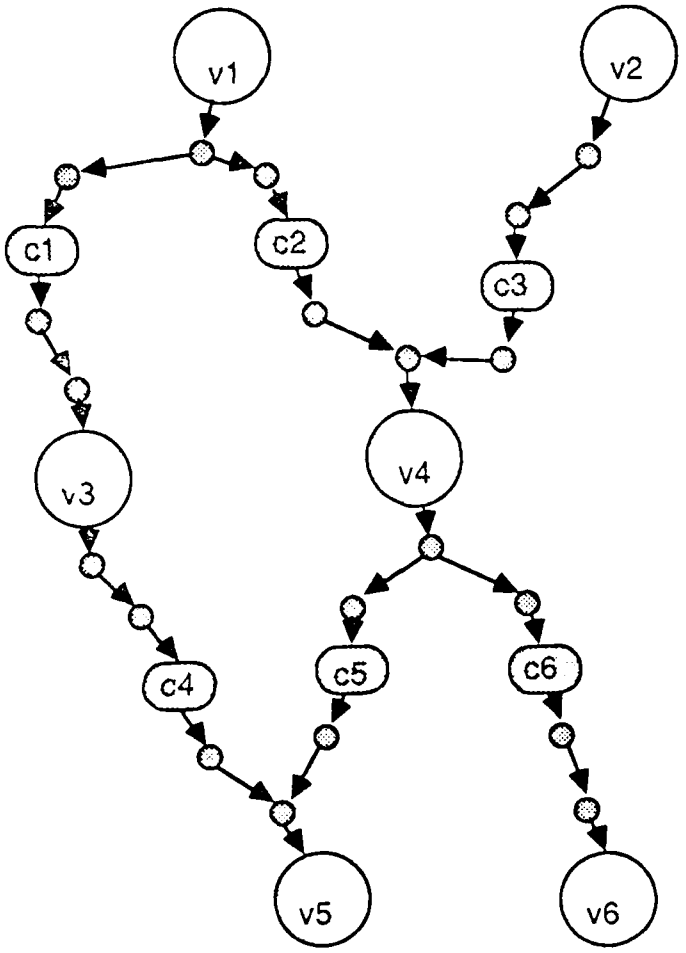


Figure 5-3: A task graph transformation with isolated subtasks.

and following this distinction is to model the limitations of individual nodes in the network first and later model in limitations that "reasonable" configurations would impose. For example, it would be acceptable to model in a limitation that nodes can only transmit one data stream at a time, but it would not be acceptable to, at this stage, model in a bound on the number of resources that can exist. These global limitations will not be ignored, but will be viewed as part of an allocation problem, not a representation problem.

As an example, we will consider a set of limitations that are imposed if the underlying support architecture described by the specification in appendix A is chosen.[3] The architectural limitations stem from the receiving and transmitting capabilities of nodes. It is assumed that all processing nodes can broadcast data and receive broadcasts. Nodes can transmit a single stream of data to multiple destinations but can only receive one stream of data at a time. Further architectural assumptions will be played out at a higher level, but at this level it is important to consider that a data transfer takes place uninterrupted; transfers are not multiplexed onto the channel. Likewise, the particular ALPS architecture described in Appendix A predicates that subtasks demand uninterrupted time on resources that are capable of acting as the attributed type. Given these restrictions, the directed graph, as it stands, is ambiguous with respect to the ordering of transmissions into and out of nodes. A further ambiguity is the representations of groups of links as ports. While the communications out of nodes appear to be distinct, some of the communications may be transfers to multiple receivers. A grouping of these receivers into a *transfer port* signifies that a single communication event is initiates multiple nodes.

A first step in resolving these ambiguities is to note the degrees of freedom. A node with many communication ports emanating from it can send those ports in any order. Some of these ports may contain a single link and some may contain multiple links; the ports with multiple links can be sent repeatedly, as long as all links are eventually sent. This means that new communications events can be spawned by removing links from ports. Each node which has N ports with one link in each port contributes to $N!$ permutations on the task set. Each port with M elements contributes, via multiple sends, a large number of possible permutations on the task set: A group with 2 links contributes 3 possible sending orders; A

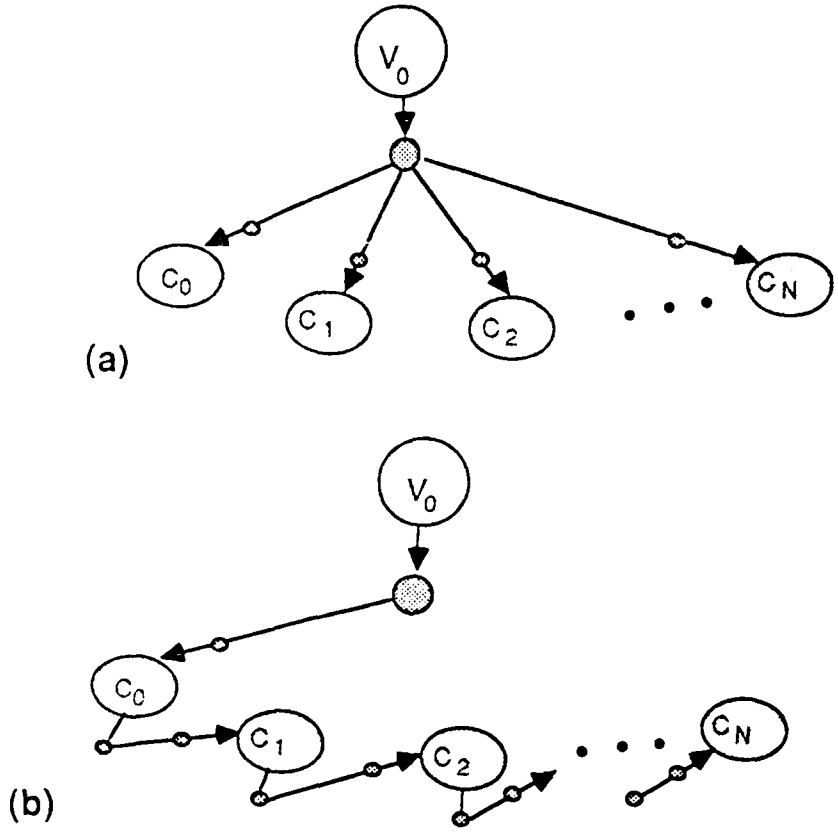
[3] The point to be made here is that this stage in modeling is not a convenient quick fix to support the example underlying architectural specification. Rather, the underlying architecture is a convenient example to demonstrate the graph transformations.

group with 3 links contributes 13 possible sending orders; A group with 4 links contributes 55 possible sending orders.

The degrees of freedom on the receiving side are as follows. A node with many *input ports* merging to it can receive these links in any order. There is always one link per input port. Each port with N merging elements contributes $N!$ permutations on the task set. Keeping in mind that each of these contributing permutations *multiplies* the total ordering possiblities, it may be impractical to enumerate all sending and receiving orders for large task sets. Complicating matters is that, as will be shown later, some of the orderings are not feasible because they introduce deadlocks in the execution order. That is, they require A before B and B before A. If one of those static orderings was applied to a system, i.e. if any underlying architecture is required to process links in that fixed order then it would block on servicing subtasks in the graph. These deadlocks, fortunately, are easy to detect.

5.2.1. Resolving Send and Receive Orders

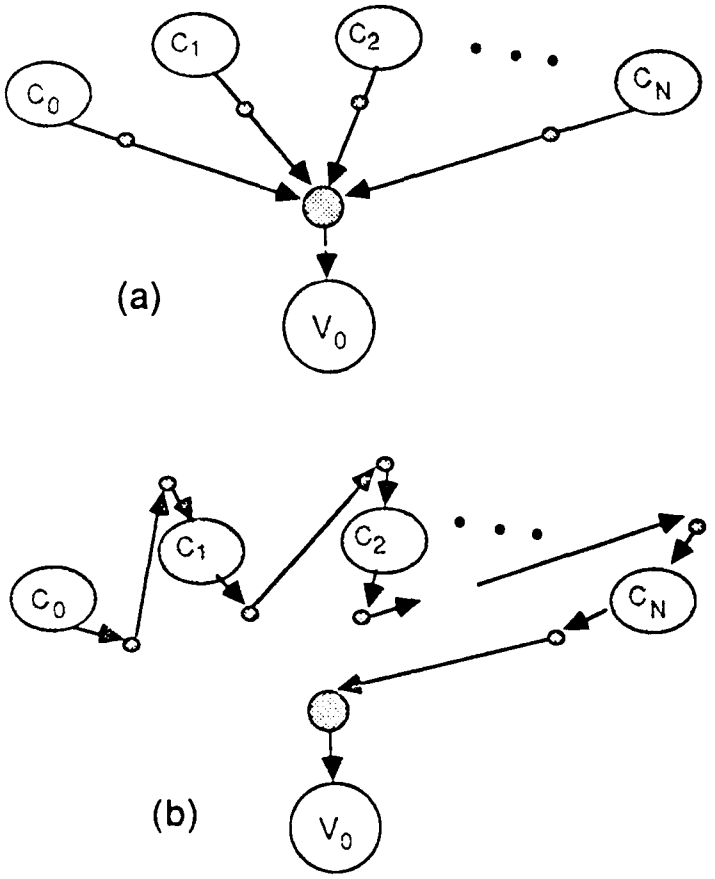
Following is a procedure for incorporating the sending and receiving link orderings into the directed graph representation. A task graph can initially be labeled arbitrarily. Transfer ports that encompass several links are shown by encircling the tails of those links. Representations which include the architectural limitation of sending data a single block at a time can be derived by serializing multiple links leaving from a single node. Assume that the communications subtasks are ordered $c_0, \ldots, c_N$ and communication c_0 will occur first (fig. 5-4(a)). The tail of the link leading out of c_0 is removed reattached to the null node that precedes the communication event c_1. The tail of link leading out of c_1 is then removed and reattached to the null node that precedes the communication event c_2. And so on (fig. 5-4(b)). Representations which include the architectural limitation of receiving data a single block at a time can be derived via a similar serialization. Assume that the merging com-



Figures 5-4(a) and 5-4(b): A subtask with four emanating communications subtasks and the serialization of those subtasks.

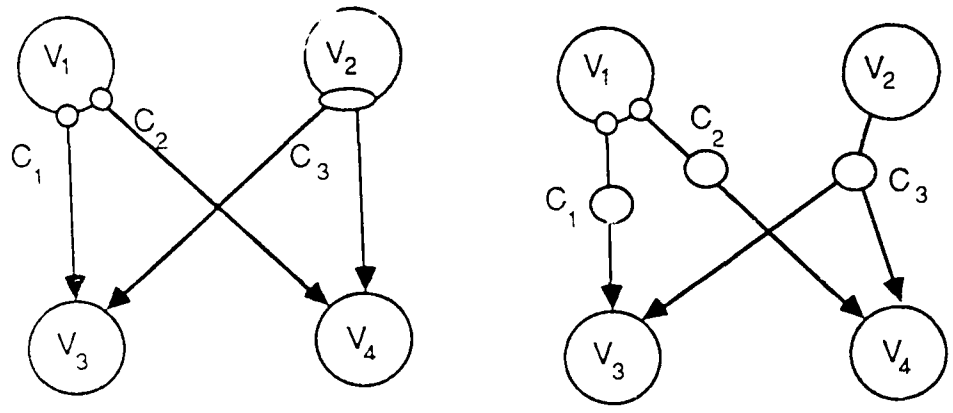
munication subtasks are ordered $c_0, \ldots, c_N$ and communication c_0 will be received first (fig. 5-5(a)). The head of the link leaving c_0 is removed from the null node that precedes the destination (v_0) and is attached to the null node that precedes c_1. Likewise, the head of the link leaving c_1 is attached to the null node preceding c_2. This procedure is repeated for nodes $c_3, \ldots, c_{N-1}$ Link c_N is not disturbed (fig. 5-5(b)).

The following example will make use of these serializing methods to illustrate some link enumeration and ordering concepts. Consider the task subset in figure 5-6(a). This



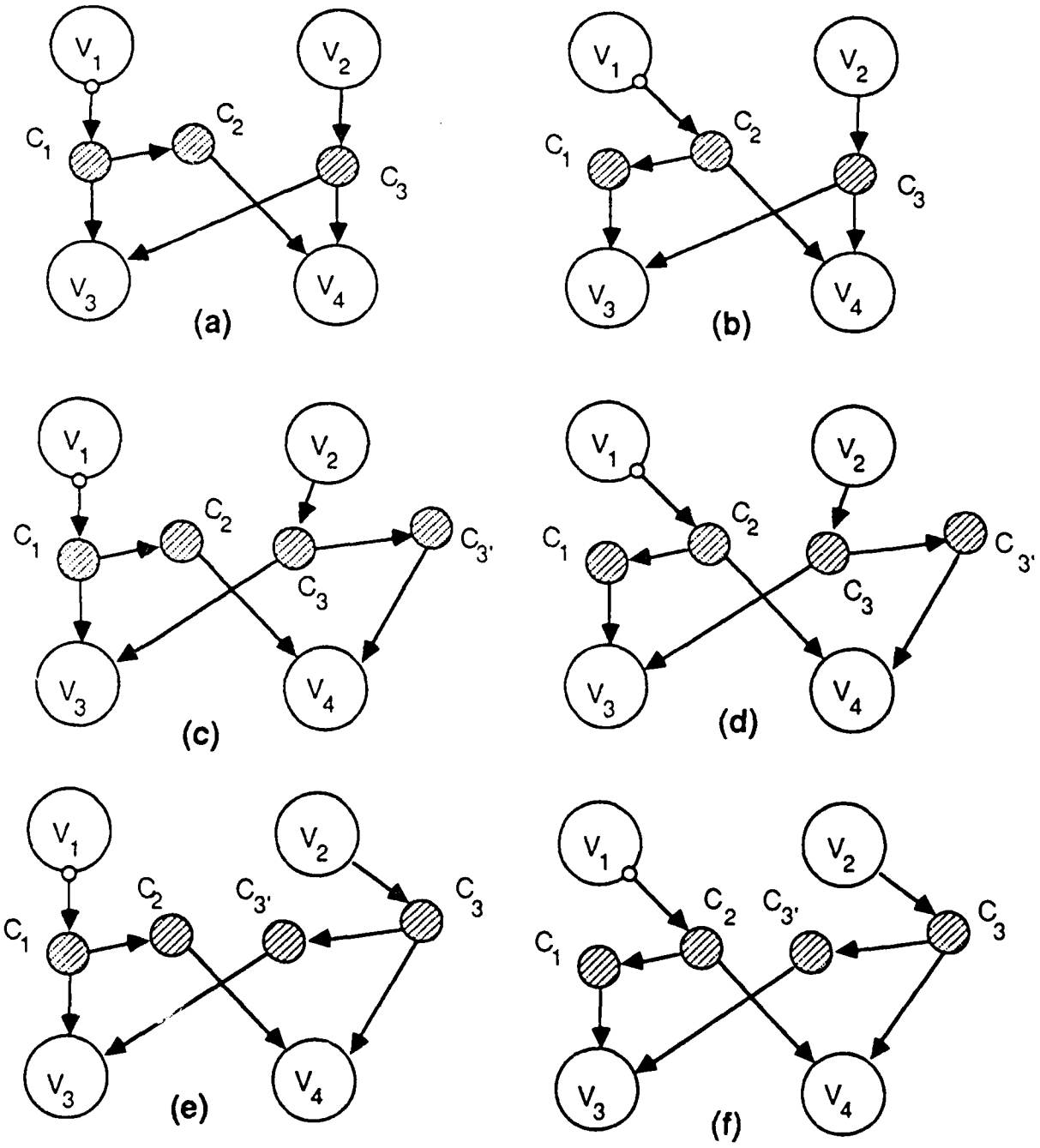
Figures 5-5(a) and 5-5(b): A subtask that is merged to by four communications subtasks and the serialization of those subtasks.

subset will be called a *subgraph* because it represents a subset of a task graph. In this example, there may be subtasks that precede v_1 and v_2 and there may be subtasks that succeed v_3 and v_4. Figure 5-6(b) shows an expansion of the task subgraph which clearly identifies the communication subtasks. Note that v_1 initiates two distinct communication subtasks each with a single destination but v_2 initiates the single communication subtask c_1 which has two destinations. This latter communication subtask represents a *broadcast* to multiple nodes, and as mentioned earlier, the broadcast can occur once to service all reci-



Figures 5-6(a) and 5-6(b): A task subgraph and corresponding expansion.

pients, or multiple times to service groups of recipients. Figures 5-7(a) and 5-7(b) illustrate the transformation of the task subset to accomodate the limitation in the underlying specification that a node can only engage in one communication at a time. This transformation is done by enumerating the possible orderings of communication subtasks and then adding precedence links to reflect each order. The ordering choices "c_1 before c_2" or "c_2 before c_1" represent two possible permutations on the task subgraph. In figures 5-7(a) and 5-7(b) the communication subtask c_1 is shown as representing a simultaneous broadcast to v_3 and v_4. This subtask can be replicated to represent the optional servicing orders of the recipients of the broadcast. Figures 5-7(a) through 5-7(f) enumerate all of the possible sending orders in the subgraph. Notice that figures 5-7(a), 5-7(c) and 5-7(e) represent the same ordering of c_1 and c_2 but enumerate the alternative orderings of c_3; since there are three possible orderings of c_3 there are six *sending* permutations on the subgraph. Notice that in figures 5-7(c) through 5-7(f) the communication c_3 is sent twice. Since the subtask is repeated, an additional node c'_3 is added to the task graph.



Figures 5-7(a) through 5-7(f): A subgraph with different communication subtask orderings.

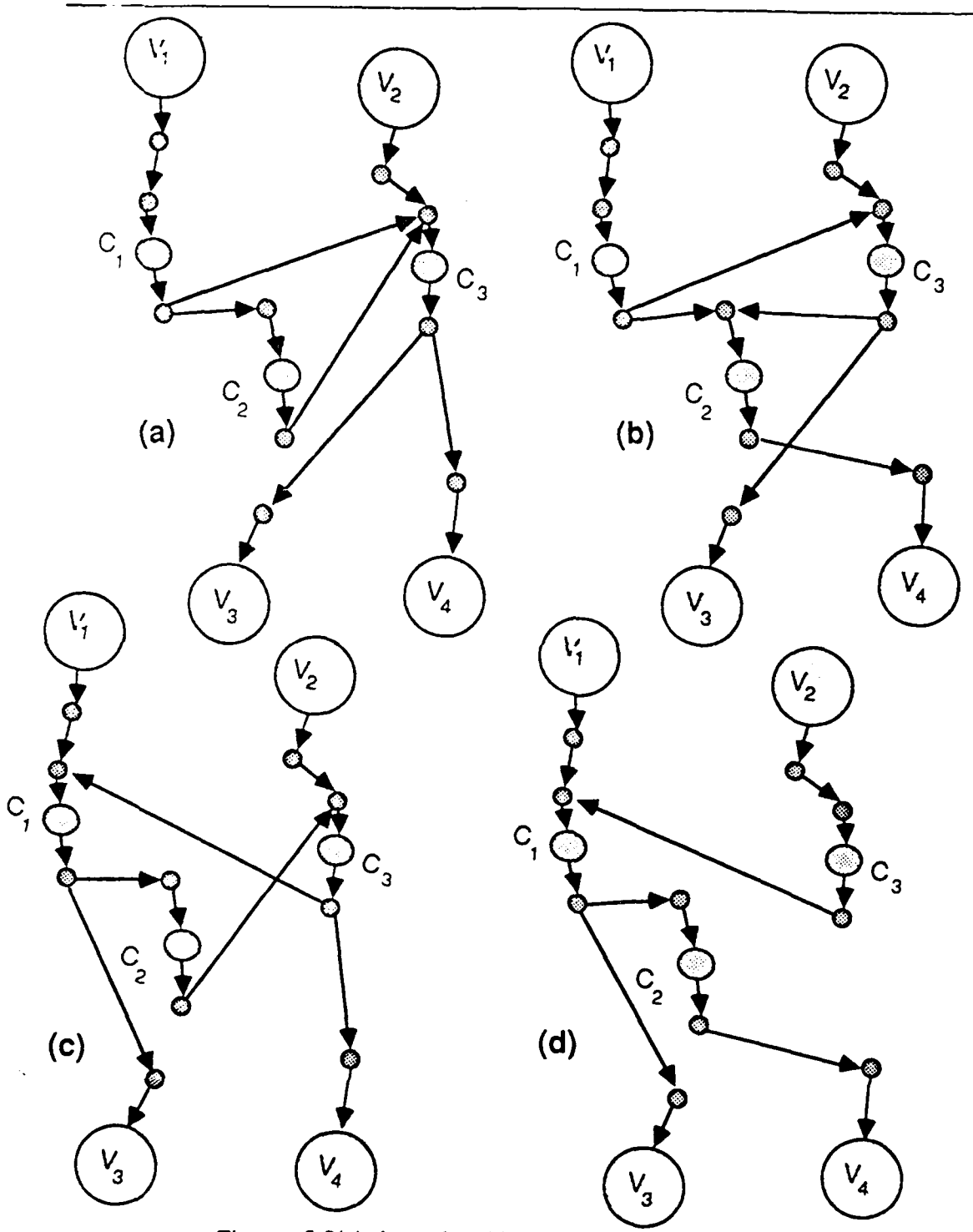
Once the sending permutations have been established, the receiving permutations can be enumerated. For the task subgraph in figure 5-6(b), there are four receiving permutations. These can each interact with the six sending permutations so that for this small subgraph, there are 24 possible orderings. Before the receiving orderings are derived, the subgraph is redrawn to isolate each of the subtasks by a null node. This allows precedences to be changed more easily and facilitates the comparison of subgraphs with different precedence orderings. The receiving orderings impose a precedence on the communication subtasks that merge to a particular node. Figures 5-8(a) through 5-8(d) show the four receiving permutations applied to the sending orderings of figure 5-7(a).[4] Notice that in all of these figures, c_1 precedes c_2, as it does in figure 5-7(a), but the communications into v_3 are ordered so that communications subtask $c_1 < c_3$.[5] Likewise, in figure 5-8(a) the communications into v_4 are ordered so that $c_2 < c_3$. The representation is logical, with respect to the ordering of communication tasks, but the destination of communication subtasks are no longer derivable by inspection. This is because the notation from which this graph originated used directed links to represent the communication subtask and the precedence of that task, with the implicit understanding that the successor subtask was the destination. In figure 5-8(a), for example, the subtask of c_1 has multiple immediate destinations, none of which are v_3, the processing subtask to which it communicates data. The destinations must be given in the communication subtask attribute lists.

5.2.2. Deadlock Detection

Once the receiving orderings have been enumerated, the transformed subgraphs can be used to extract information about the combination of the sending and receiving orderings. In particular, the presence of deadlocks can be detected by graph cycle detection methods.

[4] The remaining 20 permutations are not shown.

[5] The operator "<" indicates a precedence relationship: $a < b$ indicates that a precedes b.



Figures 5-8(a) through 5-8(d): Four receiving permutations applied to the sending ordering of 5-7(a).

Consider figure 5-8(a). The precedence list can be created: $v_1 = v_2 < c_1 < c_2 < c_3 < v_3 = v_4$, where the symbol = indicates an equivalent precedence. In figure 5-8(c) there is a deadlock: c_3 precedes c_3. This can be seen by tracing the graph from c_1 to c_2 and then to c_3 which then loops back to c_1. An explanation of the above deadlock is as follows. The broadcast c_3 is used as the first communication to v_3, but is also used as the second communication to c_4. A precedence list would show that $c_3 < c_1$ in order to initiate v_3, and $c_2 < c_3$ in order to initiate v_4. This transitively implies that $c_2 < c_1$, and that can not be the case because the original sending order imposed the restriction that $c_1 < c_2$.

An inspection of the deadlocked subgraph in figure 5-8(c) shows that a cycle has been created. Deadlocks caused by a particular choice of sending and receiving orderings can be detected by looking for cycles in the transformed subgraph. This fulfills one of the objectives of static analysis: enumerate the link orderings and, along the way, eliminate those which are not plausible. The methodology is fairly straightforward, as graph cycle detection algorithms are well known and can be applied directly to a transformed graph.[6] The graphs themselves will not be large; graph ordering transformations increase the number of nodes in the task graph by about a factor of four, depending on the number of communication links and the grouping of those links.

There may be a large number of ordering permutations for each graph. Inspecting for cycles in each graph is easy and generating each graph is simple and quick. Generating all of the graphs may take a long time, but this up-front deadlock analysis can be set up and run overnight. In the context of a larger objective, it is important to remember that the link ordering information can be critical to making scheduling scheduling decisions. Task graphs which have only a small number of feasible orderings or which require that broadcasts to multiple receivers be broken up to prevent deadlocks may impose unintuitive resource

[6] An example is a topological sorting algorithm presented in [Knut68].

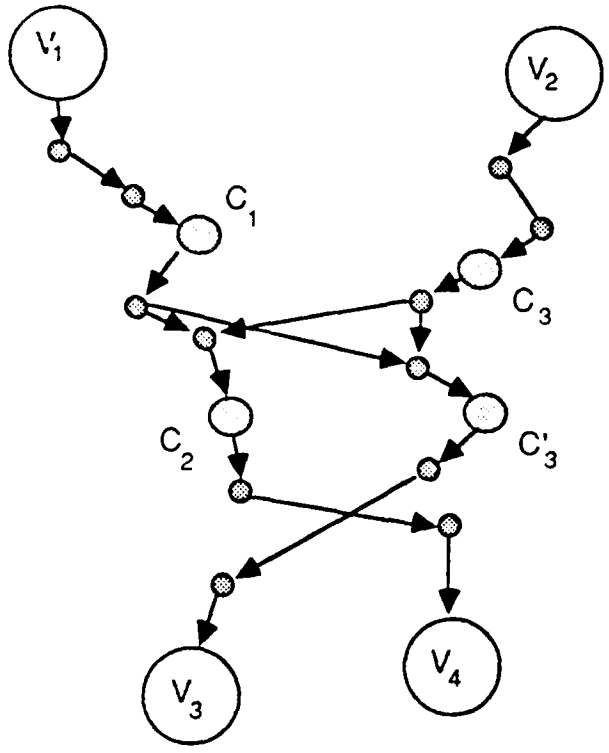


Figure 5-9: A nonconflicting subtask order which reduces communication time to 2 units.

requirements. Moreover they may fail on certain static priority lists or do poorly when the deadlock problems conflict with subtask ordering heuristics. For example, in the above task subgraph, it may be desirable maximize the concurrency of communication by ordering subtasks so that c_1 and/or c_2 can occur at the same time as c_3. If the communication time of the three communication subtasks is equal, say 1 time unit for each communication, there may be a particular sending order which allows all three tasks to proceed in two time units. Because of architectural considerations, it can be shown that this is only possible if c_3 is sent twice; if the bandwidth is available, this may be desirable. By enumerating all of the subtask orders, some nonconflicting orderings may be found that reduce the communication time

to 2 units. Figure 5-9 illustrates such an ordering. This ordering uses the sending priorities of figure 5-8(c) combined with the the receiving priorities of figure 5-8(b). This result is unituitive—sending something twice which need only be sent once actually reduces the overall latency—and demonstrates the utility of static analysis.

5.2.3. Latency Determination

Given a graph that has been fully ordered, the latency can determined. The latency of an ordered graph is a measure of the longest execution path through the graph. Finding the latency of a particular graph is an end in itself and a means to other ends. Graphs with different subtask orderings can be compared to find ones which minimize the latency. The above goal may be modified by subjecting extraneous constraints such as allocation constraints. As a means to other ends, the latency is needed to compute the resource utilization of a graph that is executing in a class one allocation environment. An intuitive explanation of this is as follows. It is desirable to study the behavior of a pipeline once the pipeline "fills up." The longest execution path latency indicates, in part, when the pipe has filled up.

Extracting the latency of a directed task graph is a straightforward procedure that relies on the definition of a starting and ending point from which to compute path lengths through the graph. In the general case there will be many sources and sinks in the graph. This makes it a bit difficult to define latency: where is a task initiated and where does it complete execution? The assumption has been made that all sources will produce initiations at the same rate. With a single source graph, that source can be thought of as initiating the entire task set at a fixed rate. This results in the highest-precedence subtasks being executed. With a multiple sources graph, each source initiates a subset of the highest-precedence subtasks. As a representation convenience, multiple sources can be considered to stem from a single *virtual source* which distributes the initiation responsibility to these multiple sources.

The representation works if there is a logical one-to-one relationship between initiations on each of the sources. A similar technique can be applied to sinks. Multiple sinks which share a one-to-one firing can be joined to a *virtual sink* which represents the completion of all execution streams in the task. The streams do not have to complete at the same time for this representation to work; the virtual sink will be "activated" when the last sink is finished.

The latency calculation relies on this virtual-source-to-virtual-sink definition. For a given graph there may be a number of paths from the virtual source to the virtual sink. Given a class one execution allocation in which this graph directly defines an execution schedule, and assuming that all of the processing and communication times are known, the graph latency is simply the time it takes for the longest path to complete and will correspond directly to the execution latency. If the allocation is class two, this latency will be the minimum execution latency. That is, a class two execution environment must still obey the execution order of the task graph and can not decrease the latency of of the longest path.

A simple method of calculating the latency is to enumerate all of the paths from virtual source to sink and then add up the execution costs along each path. This method may or may not be impractical, depending on the rigorousness of the analysis. If the architect is going to go through the trouble of enumerating each possible graph permutation, then computing distances along each path in each specific graph is not a lot of additional work. However, since the number of paths in a particular subgraph pales in comparison to the number of graph permutations that are possible given the number of communications subtask orderings in the original graph, it is useful to limit the amount of computation that must be performed on each graph.

A linear programming technique was developed to efficiently compute graph latency. It treats the directed graph as a maximum cost critical path problem. Each directed edge is

described by its source, destination and weight. The simplex method is used by setting up a simplex tableau in a edge-oriented approach. For each edge, a "+1" is placed in the constraint row corresponding to the node from which the edge emanates and a "–1" is placed in the constraint row corresponding to the destination node. Since each column corresponds to a unique branch, it will contain exactly one "+1" (source) and one "–1" (destination). The sum of all of the constraints will be zero, implying that the rows are linearly dependent. This means that one constraint row can be eliminated: the virtual sink node is neglected. An initial basic variable for each constraint can be chosen on inspection as the column in which the first "+1" occurs. Each constraint row will have a "+1" since every node has an emanating edge (the virtual sink is eliminated). Each column has only one "+1" so the basic variables are distinct. A description of the simplex method can be found in [Hsia82]. The simplex method can then be applied over this tableau and will yield a solution which gives the cost along the critical path.

This method is useful not only for finding critical source-to-sink paths but for finding intermediate node distances. A simple technique of adding large value links can be used to trick the simplex method into finding distances between an arbitrary origin-destination pair. All other path options are eliminated by inserting large value links between the origin and the virtual sink and then the destination and the virtual sink. The path that will be chosen as maximum will be one which encompasses these two nodes. The path length can then be extracted. This technique will also detect if there is no path between the origin and destination.

An alternative method of finding critical paths is a straightforward, recursive, depth-first search technique to enumerate the paths through the graph and the subsequent length computation on each path. This technique is useful for small graphs but may become unwieldy for larger graphs, and may be difficult to apply to arbitrary paths within the graph.

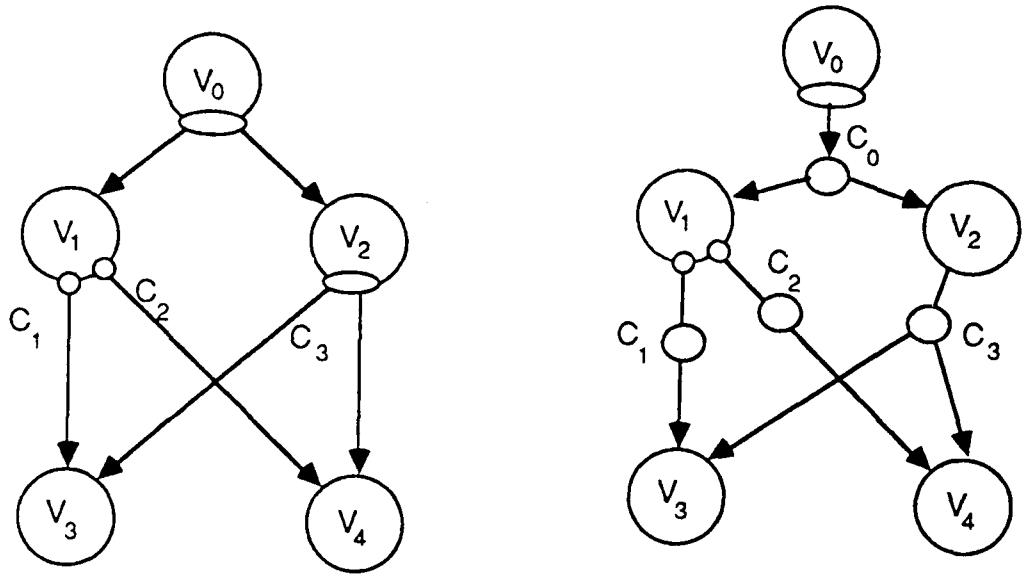


Figure 5-10: The task subgraph of 5-6(a) with attached source and sink and its expansion to explicitly represent communications subtasks.

An advantage is that the paths, as well as the lengths, can be extracted.

5.3. Example Analysis

Consider the task graph in figure 5-10. Notice that it is comprised of the task subgraph in figure 5-6(a) with a source attached to it. Subtask execution times can be assigned to each processing and communications subtask (table 5-1). Using the linear programming technique described above, the latency for the task graph (assuming a class one execution environment) can be found. This is done by first choosing the sending and receiving link orders and then converting that expanded graph into a simplex tableau. Note that a source has been attached to these subgraphs and that nodes v_3 and v_4 are sinks. Table 5-2 gives the task latencies that correspond to the link orderings of figures 5-8(a) to 5-8(d) and figure 5-9, which is a most-concurrent ordering. The latency for the most-concurrent case is not the

minimal. This is a result of the particulars of the timing information supplied to this example. If all communication and execution subtasks took equally long to process then this would have been the minimal latency ordering.

The latency of a graph can be a useful calculation when considering a particular set of orderings or a single initiation of the graph. For a system with periodic task arrivals, though, latency is only one characteristic of the task set. The task arrivals combine to yield a demand for resources that is based on the demand imposed by a single task. This demand can be found from an ordered task graph whose latency has been determined. Determining the baseline resource demands imposed by a particular task is a reasonable objective of static analysis. This objective can be met by refining the task description to incorporates the periodic arrival of tasks and then using this description and the task latency to derive the demands for resources. The peaks and averages in these (processor and communication) resource demand *profiles* can then be used to generate minimal allocations needed for class one or class two execution environments.

Algorithm (task) Information		
Subtask	Subtask Type	Execution Time (μsec)
v_0	Source	0
v_1	A	1100
v_2	B	1000
v_3	Sink	0
v_4	Sink	0
c_0	Comm	100
c_1	Comm	50
c_2	Comm	30
c_3	Comm	25

Table 5-1: Task information for latency calculation example

A task with a periodic arrival rate can be represented by chaining individual task graphs (figure 5-11), where a delay inserted between each graph gives the inter-arrival time. This representation is made easier if the graph has been transformed into one which has a virtual source.

A profile of demands for each resource can be generated by first creating that profile for a single task set and then overlaying the profile onto itself. If the profile is examined after the first latency, a periodic demand can be detected, the period being the interarrival time. The proof of this is fairly simple. The demand profile of a single task can be considered as a discrete function $d[n]$, which is valued in the range $n = 0$ to $n = N$ (the latency is N). Assume that the interarrival time of the task is A, and A is a multiple of N, so that $m * A = N$. If the function is replicated onto itself at $n = A$, $n = 2A$,... then a new function $g[n]$ can be described:

$$g[n] = d[n] + d[n-A] + d[n-2a] + ,...$$

At (or after) $n = N$, this function is periodic in A:

(1) $g[N+r] = d[N+r] + d[N+r-A] + d[N+r-2a] + , . . . , + d[N+r-mA]$
(2) $g[N+r+A] = d[N+r+A] + d[N+r] + d[N+r-A] + , . . . , + d[N+r-mA]$

Latency Information	
Link Ordering (*figure*)	Latency (μsec)
5-8(a)	1305
5-8(b)	1305
5-8(c)	Deadlock
5-8(d)	1205
5-9	1280

Table 5-2: Latency information for latency calculation example

(3) $g[N+r+2A] = d[N+r+2A] + d[N+r+A] + d[N+r] + d[N+r-A] + \ldots\ldots + d[N+r-mA]$

These expressions are equivalent for any r because $d[n]$ is zero for $n > N$. Pictorially this periodicity can be shown by overlapping random sequences. In figure 5-12(a), a random sequence is shown which is valued from $n = 0$ to $n = 20$. This sequence, $d[n]$ is added to the sequence $d[n-7]$, $d[n-14]$,..., $d[n-49]$. The resulting sequence is plotted in figure 5-12(b). Note that the sequence is periodic after an initial pipe filling. Since the partial sequences $d[n-56]$, $d[n-m*7]$ are missing, the sequence tails off. This observation has been

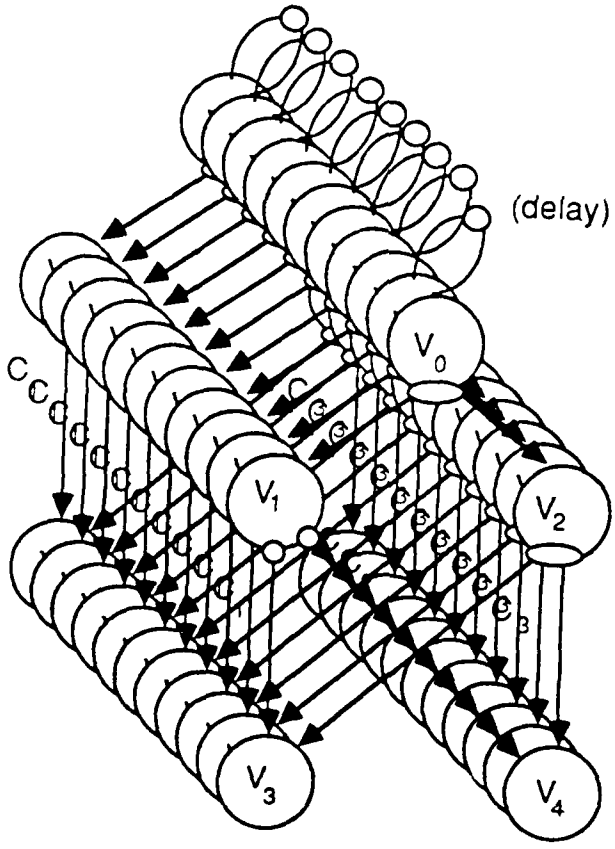
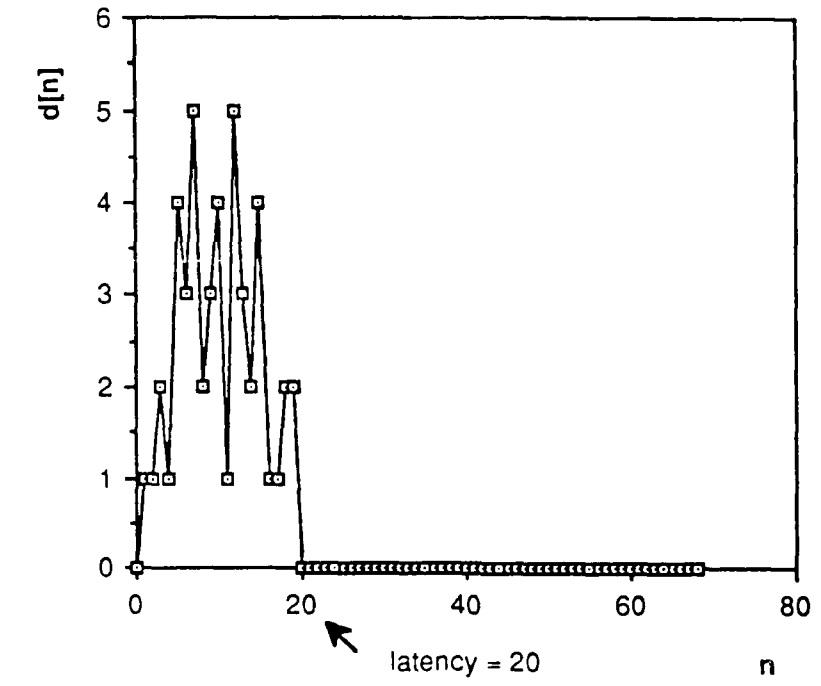


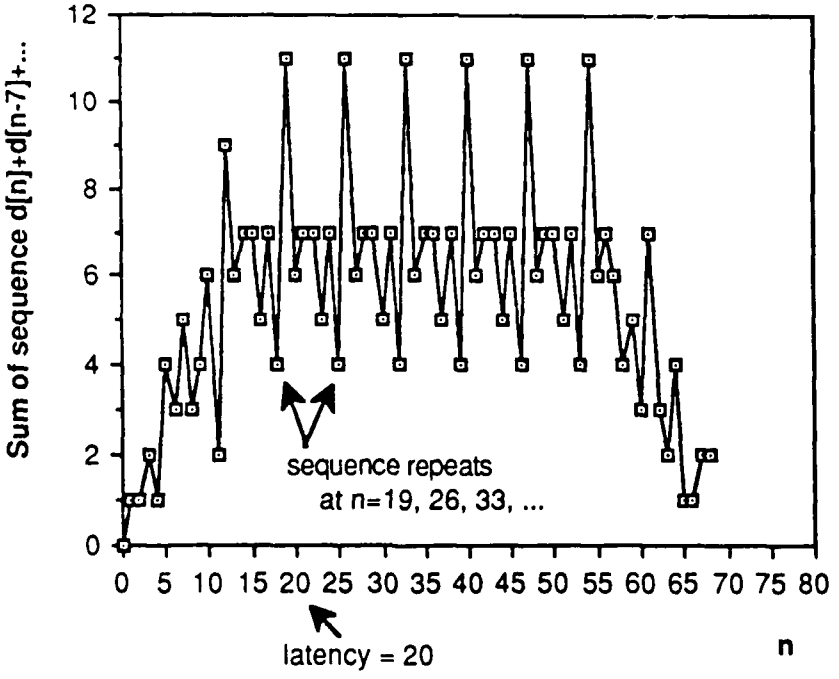
Figure 5-11: The task graph of 5-10 with a delay inserted between each arrival of that task.

integrated into a program which derives the latency of the task and, given an interarrival time of tasks, determines the demand for communication and processing resources. As an example, consider the example task graph shown in figure 5-13(a) and the associated task information provided in table 5-3. The generation of the steady state period will be demonstrated for two different link orderings. In the first ordering, the broadcast communication c_0 will occur once and will have 12 simultaneous recipients. The subsequent communications c_1 through c_{12} will occur serially to accomodate reception by the sink. In the second ordering, the broadcast c_0 will be repeated for each recipient: it will occur 12 times. The merging communications will be serialized as above. Figure 5-13(b) shows the task graph expanded to show the serialized merging. A demand graph for processors and communication resources is shown in figures 5-14(a) and 5-14(b). This represents resource utilization for a single firing of the task. Demand profiles which corresponds to this task graph and an

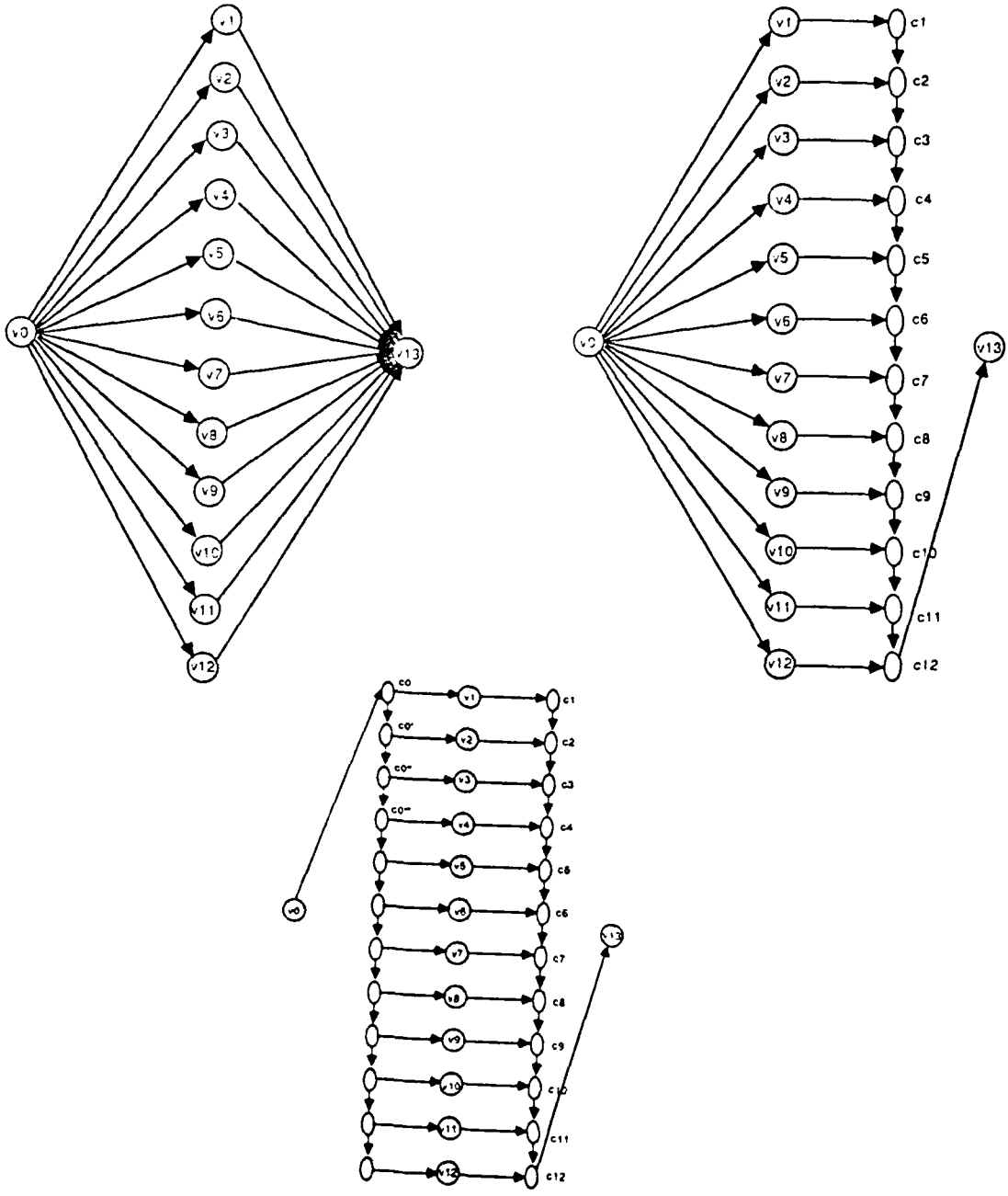
Algorithm (task) Information					
Subtask	Subtask Type	Execution Time	Subtask	Subtask Type	Execution Time
v_0	Source	0	c_0	Comm	100
v_1	A	400	c_1	Comm	100
v_2	A	400	c_2	Comm	100
v_3	A	400	c_3	Comm	100
v_4	A	400	c_4	Comm	100
v_5	A	400	c_5	Comm	100
v_6	A	400	c_6	Comm	100
v_7	A	400	c_7	Comm	100
v_8	A	400	c_8	Comm	100
v_9	A	400	c_9	Comm	100
v_{10}	A	400	c_{10}	Comm	100
v_{11}	A	400	c_{11}	Comm	100
v_{12}	A	400	c_{12}	Comm	100
v_{13}	Sink	0	c_{13}	Comm	100

Table 5-3: Subtask information for demand profile generation example.

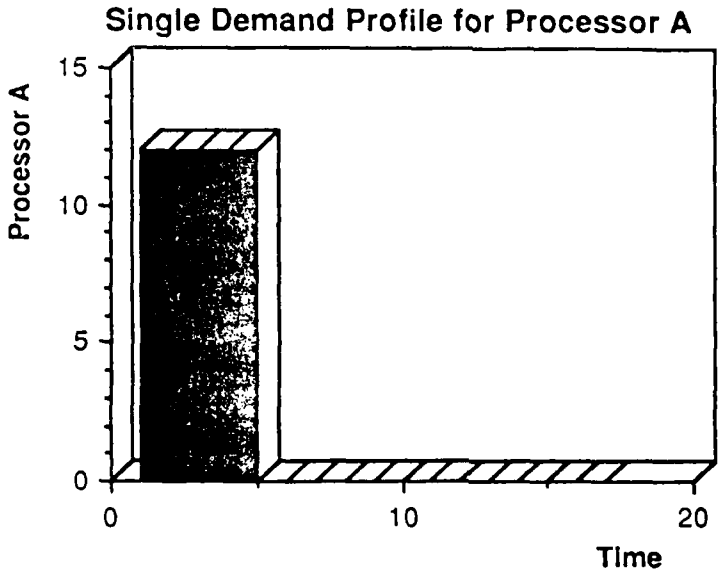


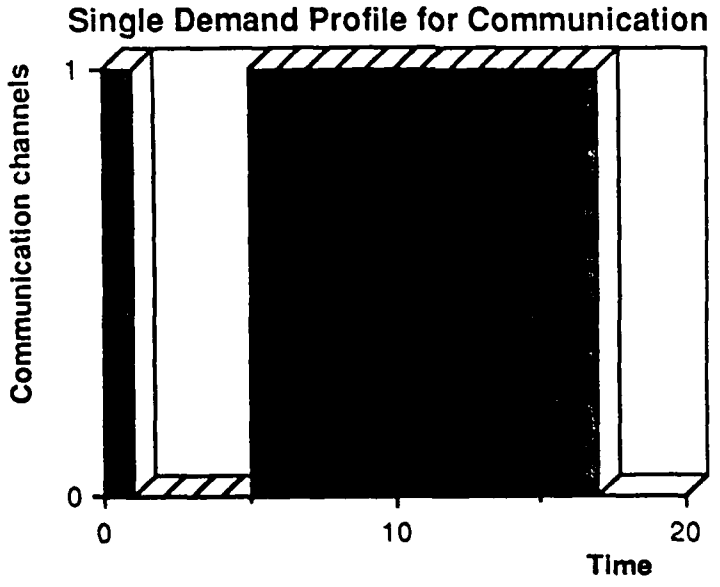


Figures 5-12(a) and 5-12(b): A random sequence valued up to n = 20 and its overlapping at n = 7, n = 14, ...



Figures 5-13(a) through 5-13(c): An example task graph with three link orderings. In fig. 5-13(c), the demand graph was transformed to serialize the initial broadcast.





Figures 5-14(a) and 5-14(b): Demand graphs for processor and communication resources for a single firing of the task graph in 5-13(b).

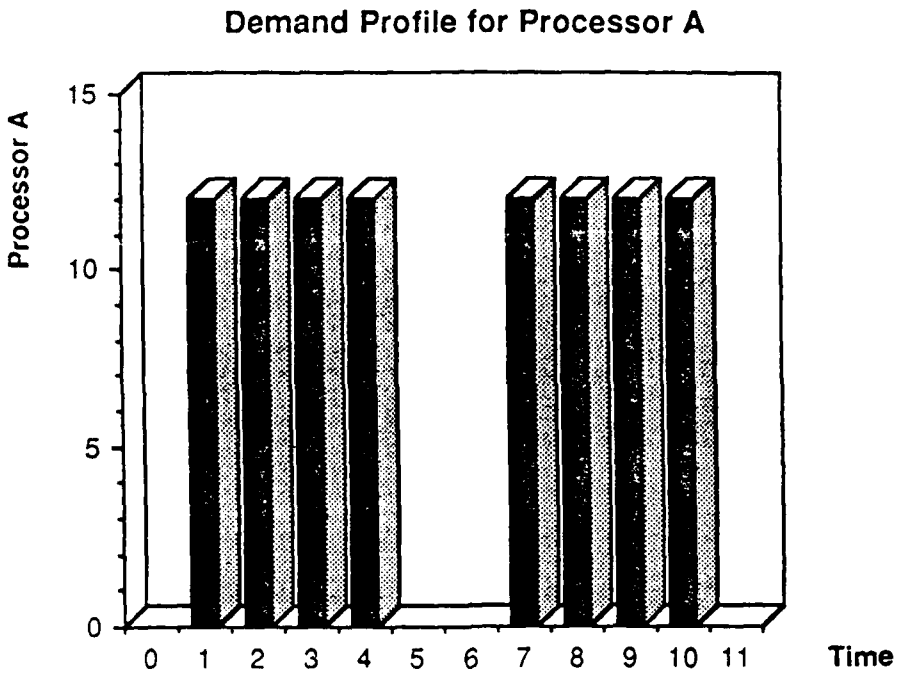
interarrival time of 600 units are shown in figures 5-14(c) and 5-14(d). Note that the single broadcast to 12 processors of type *A* means that there is an instantaneous demand for these 12 processors: the peak demand is 12 while the average demand is about 8. The peak communications demand, though, is fairly low and is near the average demand. How can this graph be modified to reduce these peak demands? One option is to stagger the demand for processors by initiating only one processor execution at a time. Figure 5-13(c) shows a transformation of the demand graph in which the broadcast is repeated for each processing node. The single-firing demand profiles for this arrangement are shown in figures 5-15(a) and 5-15(b) and the steady state demand, once the pipe has filled up, is shown in figures 5-15(c) and 5-15(d). The peak demand for processors is now equal to the average demand (8) but the consequence is that additional communications resources must be utilized to repeat the broadcasts.[7]

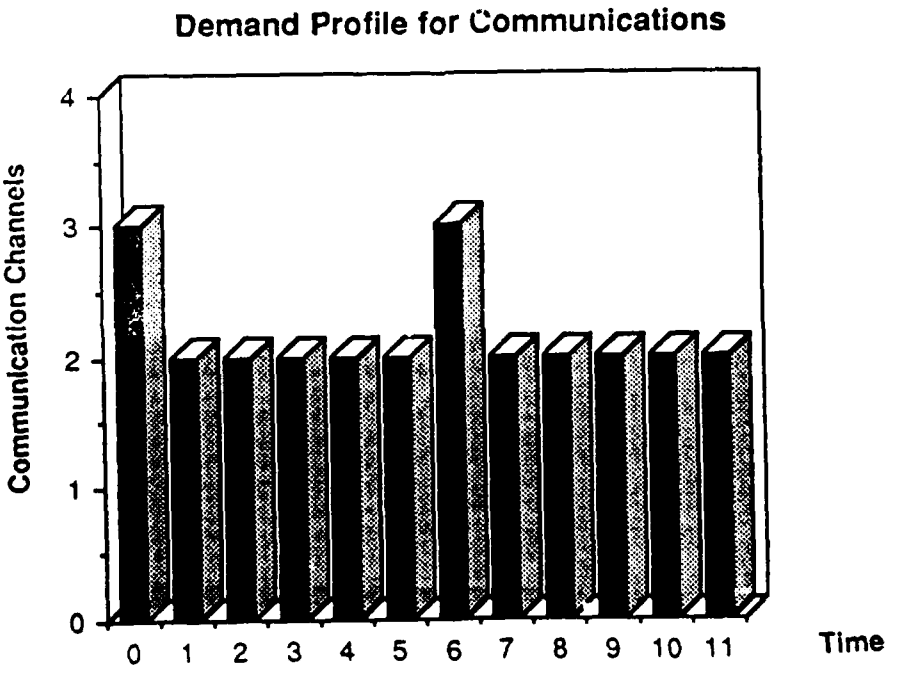
5.4. Conclusion

This example provided several insights into the utility of these techniques, as well as the limitations. The demand profiles will yield the minimal allocation of resources under any execution environment. This can be used as a baseline for allocations. The comparison of peak versus average demand give an indication of the extent of buffering that will be required in a class two execution environment. Finally, the effects of graph enumeration parameters can be explored in terms of the change in the synchronization of resource demands. If a task graph with large variations in resource demands can be converted into one with fairly stable demands, a class one execution environment may be feasible. Furthermore, systematic queueing is reduced.

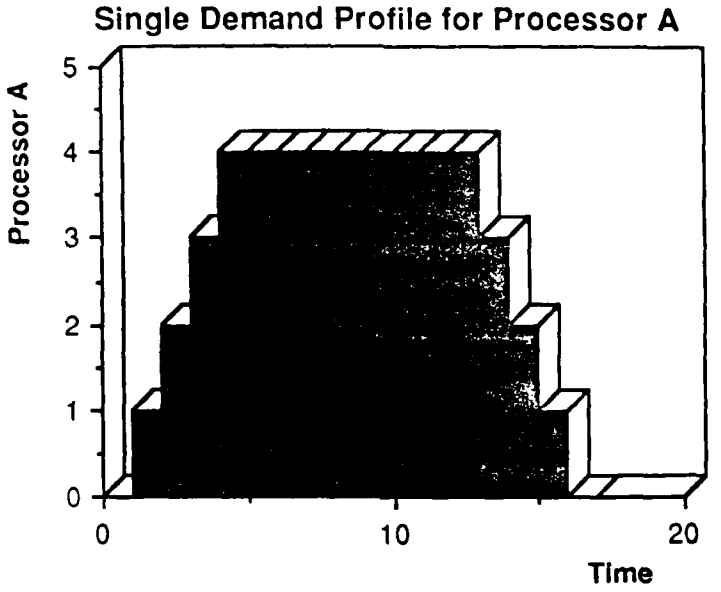
---

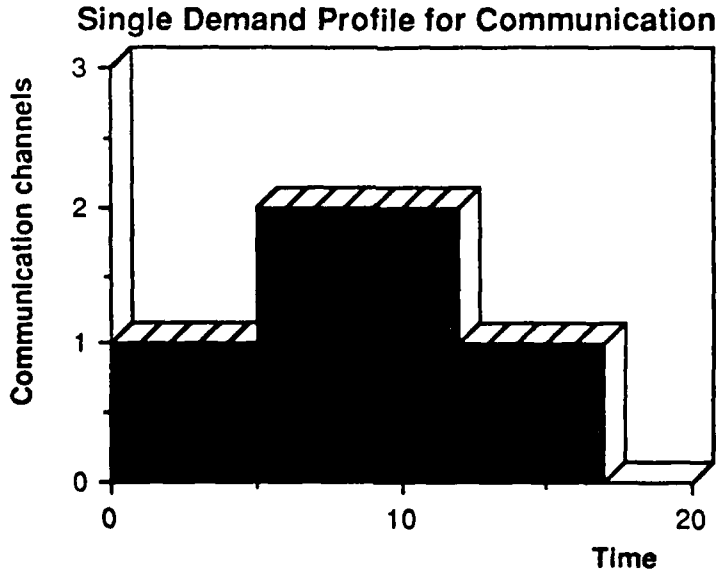
[7] An alternate solution is to insert delays so that the processor demand profile remains the same but the communications demand is reduced. This technique, akin to the delay insertion method used for pipeline processor scheduling, will be explored later.



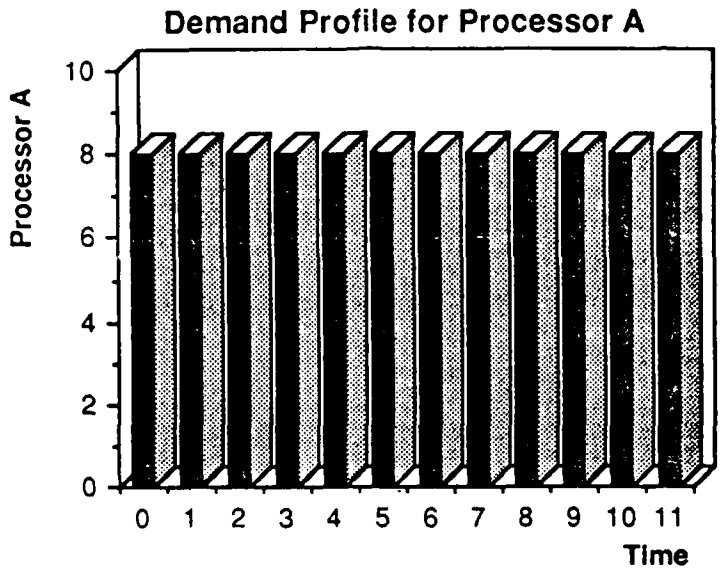


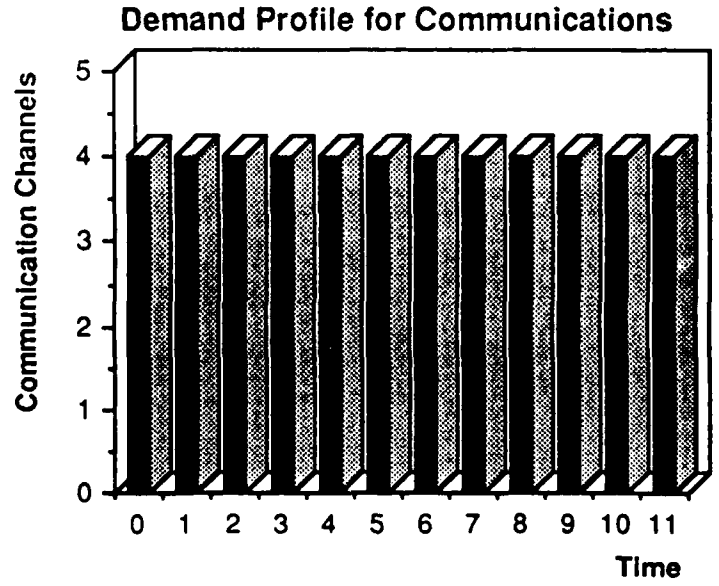
Figures 5-14(c) and 5-14(d): Demand profiles for the task graph in 5-13(b) with an inter-arrival time of 600 units.





Figures 5-15(a) and 5-15(b): The processor and communications demand graphs for a single firing of the task in 5-13(c).





Figures 5-15(c) and 5-15(d): The processor and communications demand graphs for the task in 5-13(c) with an inter-arrival time of 600 units.

The main limitation of this analysis technique is that it is very difficult to incorporate allocation and detailed architectural information into the model. The techniques that have been described become difficult to implement when the number of graph variables grow. In addition, the essential technique is based on manipulations of the task, not the architecture. A "supply-side" view is hard to obtain and integrate; coercing the task graph to fit a specified allocation is possible but the resulting transformation is not periodic. If the problem is not periodic, or if its period can not be easily determined, simple task replication techniques can not apply.

This method, then, is limited in its ability to take architectural assumptions into account because of the growing dimensions of the problem. Furthermore, static analysis of the task graph is limited by the underlying representation and periodicity constraints in its extensibility to general allocation problems.

CHAPTER 6

State Generation

Suppose that a given task has all of its timing requirements specified. How can that task be mapped onto a set of resources? As we saw in the previous chapter, some estimation of the number of resources required—both processors and communication channels—can be made by looking at the peaks and average of the the resource demand profiles. If an allocation was made which met the peaks in the demand profiles then the task graph could be considered an execution schedule for that allocation and a direct mapping could be made. There would be no buffering of subtasks and the latency of the graph would equal the latency of the implemented system. Nothing was said about the ability of that system to execute the task if an allocation was made that was at or above the *average* resource demands in the graph. It was implied that the system would have to queue up tasks to flatten the peaks of the demand graph, and some systematic methods of effecting that flattening were investigated. These involved changing the demand graph to find a subtask ordering which had flatter profiles. If none of those ordering attempts reached a suitable solution then queuing techniques which buffer tasks to accomodate contention for resources would have to be employed. The allocation would then create a class two execution environment in which subtask queueing occurs.

This chapter contains an investigation of the process of mapping subtasks in a class two allocation environment. An example is given which shows a state representation of the execution of a task (or series of task arrivals) onto a set of processors. That example is expanded to show the complexities of the mapping representation when delays are introduced to flatten the demand profile. The example will illustrate state diagrams that were

derived manually. Once the reader has been introduced to the difficulties of generating state diagrams that represent the states of the mapping process in a class-two environment, the process of finding and evaluating delay lists will be introduced.

A state diagram representation is chosen to describe subtask mapping in a class two environment. State generation provides a means of representing the mapping in a class two environment and gives a sense of how subtask delaying approaches work. Furthermore, it illustrates the complexity of both the problem representation and the subsequent search for delay approaches. The number of states which are required to describe the assigning of subtasks in a class one environment is very large, but state representation of mapping in this environment is not necessary because by definition, the directed graph accurately describes the scheduling of those tasks; the state representation is trivial. The number of states which are required to describe the assigning of subtasks in a class two environment is even larger. The significance of this is that the size of the representation reflects the difficulty of manipulating and extracting information from the state diagrams. Consequently, it is difficult to find a particular subtask delaying approach which meets a specified set of allocation, performance, and mapping criteria.

By choosing a state diagram representation, individual subtask mappings can be not only enumerated, but alternate mappings can be generated. A procedure which automatically generates subtask mapping state diagrams will be described. This procedure can enumerate all of the delaying alternatives and generate a field of schedules. That field can then be explored by looking for cycles in the state diagram; these indicate execution schedules in much the same way that cycles in pipeline latency sequences indicate plausible control strategies. Theoretically, an schedule which meets performance criteria (latency and stability) could be chosen from this field. The problem is that the approach fails on extensibility. This automated technique does not scale to problems with large task graphs, large

allocations, and fine timing granularities. And even if it could, it would not be extensible to problems which are less completely defined and which include more assumptions about the underlying architecture. It will be shown that the number of states that need to be evaluated is explosive and grows along many dimensions. Some of these dimensions are critical, such as the number and composition of resources that are being allocated, and some of the dimensions are arbitrary, such as the execution time of subtasks. Because of this state explosion, the technique is more demonstrative than it is applicable. In that sense, this chapter sets up a paper tiger and then slays it. The purpose in doing this is that explaining the infeasibility of solution-by-enumeration techniques justifies heuristic-based solutions and simulation-based analysis methodologies.

6.1. State Diagram Representation

The process of state diagram representation and generation will be explained via a simple example. Figure 6-1 shows a directed task graph. The "source," graph node v_1 is also the first subtask; we'll call this subtask A. It executes and then initiates communications subtask c_1, which is a broadcast to subtasks v_2 and v_3. They are initiated and eventually

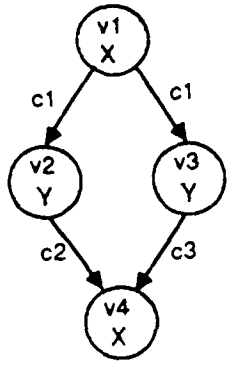


Figure 6-1: A directed task graph.

send information, via subtasks c_2 and c_3, to subtask v_4; we'll call v_4 subtask B. The task graph can be broken into separate graphs for each processor type (fig. 6-2). This decoupling will aid in reducing the number of extraneous variables in the representaiton. Unfortunately, as will be demonstrated later, the different-typed subtasks are related via graph precedences so that changes to one graph affect all other graphs. Consider the task graph for processor type X (fig. 6-2(a)). There is an arrival of tasks every two time units and for each arrival a demand for processors for three units followed by five units of delay in which subtasks of other types are executing and subsequently a demand for three more units of processing. This demand graph can be redrawn in a directed task graph form (fig. 6-3(a)) or in a gantt chart form (fig. 6-3(b)). The gantt chart shows the demand for processors in terms of lines proportional in length to the demand timing that they represent. Notice that the demand graph has a repetitive segment whose length is proportional to the interarrival time. Each column represents a new arrival and the arrivals are staggered by two time units (each row represents a time unit) to show the interarrival time of tasks. The peak and average demands for processors can be viewed by scanning across rows of the task graph and counting entries in each row. In this example, the peak demand is for 4 processors but the average demand is 3. This becomes more evident in a demand profile for this resource (fig. 6-4). If we wanted to map this demand graph onto an architecture with four processors of type X—a class one allocation—we could do so by assigning awaiting subtasks onto awaiting processors in a round-robin fashion. A state diagram which represents this assignment is shown in figure 6-5. The individual processors have seven states, described in table 6-1. Since there are four processors and seven states, there are 4^7 possible states of the system.[1] However, since this allocation is class one, the system will reach a periodicity at or after the latency of the task—the 11th time unit—and the steady initiation period will be the

[1] For this simple example, if there are p processors and s states per processor, the number of global states is s^p

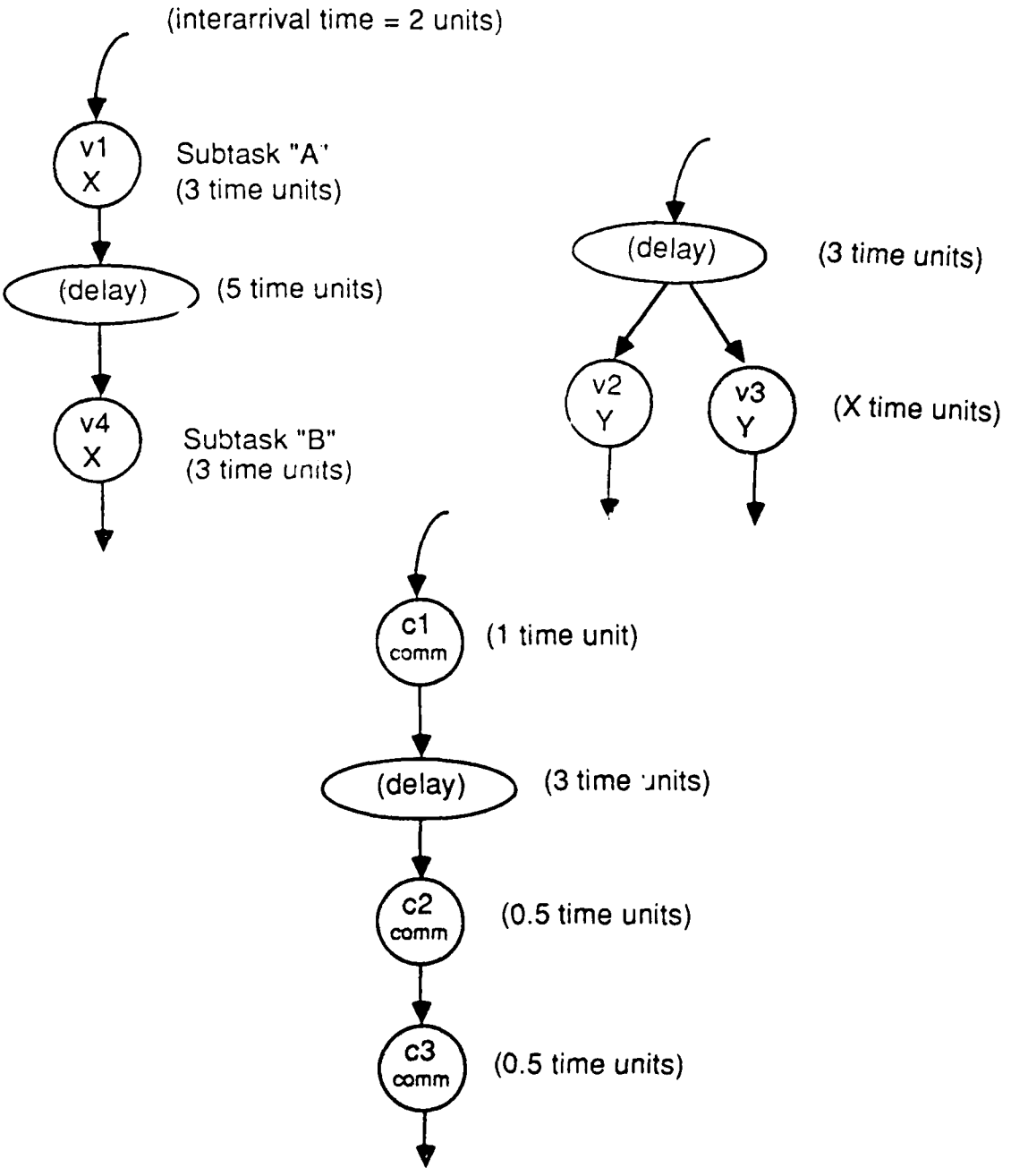
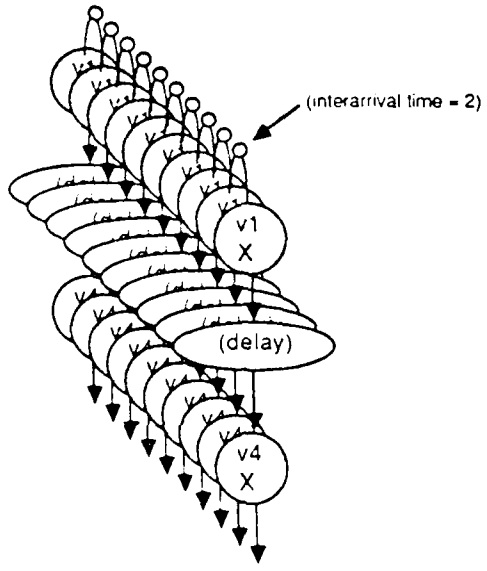
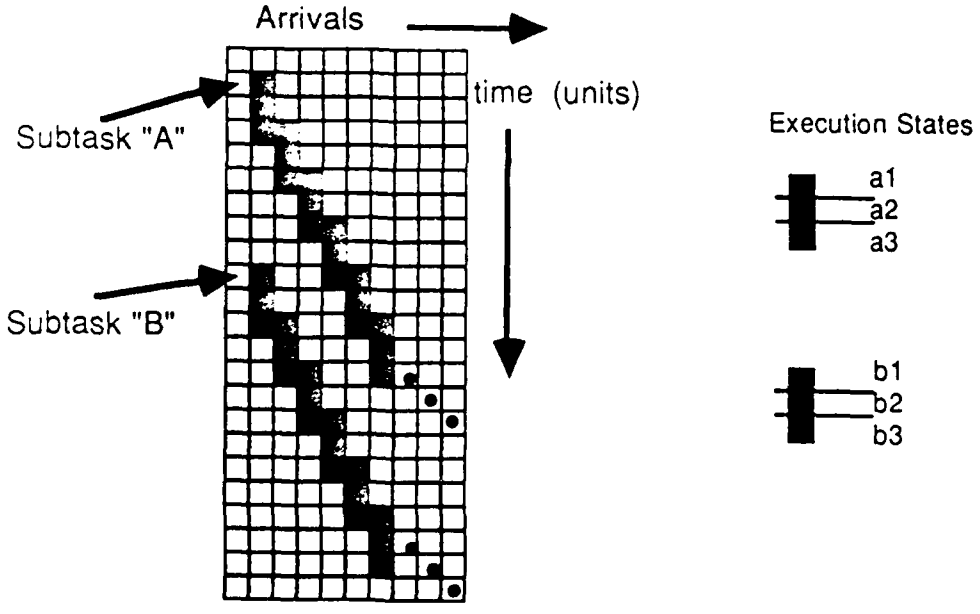


Figure 6-2: Partial task graphs for subtasks processor types X and Y and communication subtasks.





Figures 6-3(a) and 6-3(b): Directed task graph and gantt chart notation for the partial directed task graph of 6-2(a).

State	Description
0	Not processing
a_1	Processing first third of subtask *A*
a_2	Processing second third of subtask *A*
a_3	Processing last third of subtask *A*
b_1	Processing first third of subtask *B*
b_2	Processing second third of subtask *B*
b_3	Processing last third of subtask *B*

Table 6-1: Description of states for example task graph.

interarrival time. Not all 4^7 states will be visited. In figure 6-5, the first state (state 0) represents all processors idle. State 1 represents the assignment of subtask *A* onto processor 1. State 11 represents processor 1 completing subtask *B*, processor 2 starting subtask *B*, processor 3 completing subtask *A* and processor 4 starting subtask *A*. This is the last state in

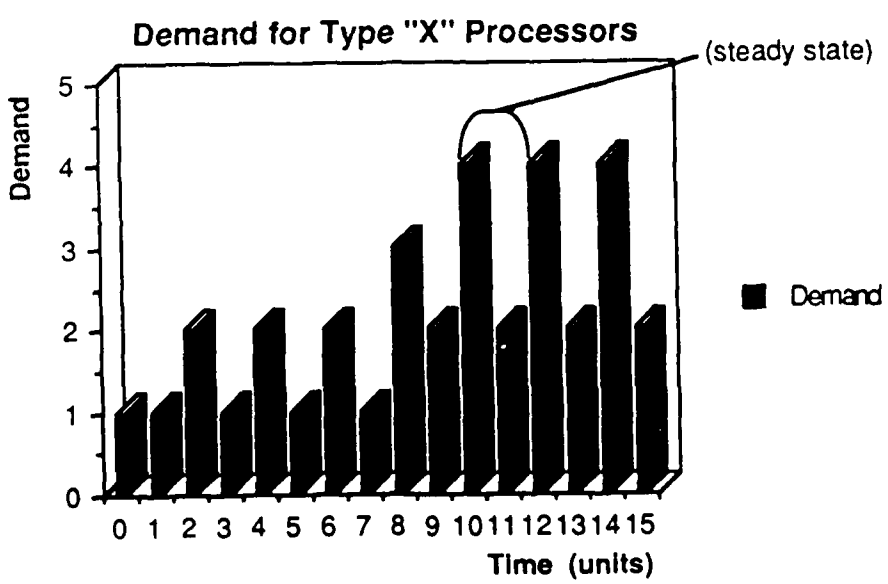


Figure 6-4: A demand profile for the partial task graph in 6-2(a).

the first initiation of the task. Every two states subsequent show an identical initiation of subtasks, i.e. state 11 contains the same subtask segments as state 13 and state 12 contains the same segments as state 14. While the *location* of these segments—which processor (represented by the column position) they happen to be initiated on—varies, the initiations are the same. A cycle which repetitively generates these segments will be called an "initiation cycle." The salient system state is then the list of subtask segments, as opposed to an ordered list. A "mapping cycle" represents the same segments being operated on by the same processors. For this cycle, the the column-location of the subtask segments must be preserved—the system state is then represented by an ordered list of subtask segments. State 14, is the beginning of such a cycle: the segment list is: a_2, 0, b_2, 0. The cycle runs to

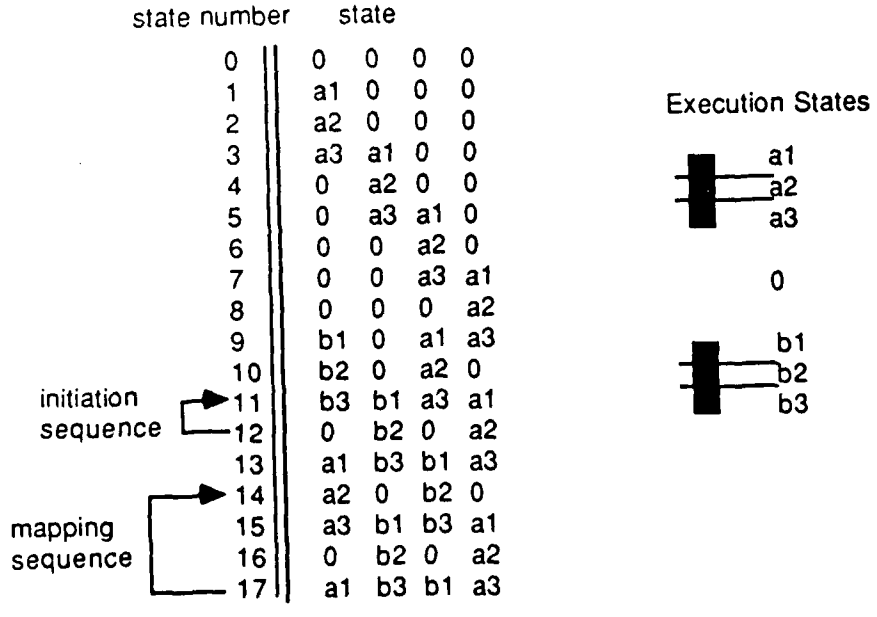


Figure 6-5: A state diagram representation of the round-robin assignment of the demand graph of 6-2(a) onto a collection of four type X processors.

state 17; state 18 will be identical to state 14. This type of cycle represents the subtask mapping from a processor point of view in that the states represent what each processors happens to be doing at each stage as well as what subtasks are being processed. The period of the mapping cycle is longer than the initiation cycle period because the interarrival time is shorter than the longest execution time of one of the subtasks. It would not be possible to represent the continuous execution of subtasks which require 3 time units on a unique processor in the space of a period of 2 time units. The period of the mapping cycle for a class-one allocation, then, is at least as long as the longest subtask time and is a multiple of the interarrival time.

This task graph could be mapped onto a system of three processors if we can find a way of buffering the peak in demand in such a way that the peak is reduced to the allocated number of processors. Since, in this example, the average demand is three, the peak demand will be reduced to three. Since we are not reducing overall demand, the valley in demand must be increased to three. The buffering is performed in the following manner. At each time unit, subtasks that have already been assigned continue to execute. The number of new subtasks that need assigning is compared to the number of available resources. If there is a conflict then the buffering procedure that is being uniformly applied is used to choose the subtasks to buffer. It turns out that for this example there are several plausible procedures to buffer new tasks; two of them will be illustrated. The first is shown in figure 6-6(a). In this procedure, a conflict arises at the 11th timestep. Subtask B of arrival 1 and subtask A of arrival 5 are awaiting initiation but there are two subtasks already being executed. The conflict is resolved by delaying the later arriving subtask. This rule is continually applied whenever there is a conflict. Note that when a subtask is delayed, its successors are also delayed. In this example, this means that if subtask A is delayed by one time unit then subtask B must also be delayed. Note that while the resulting gantt chart has a periodicity to it,

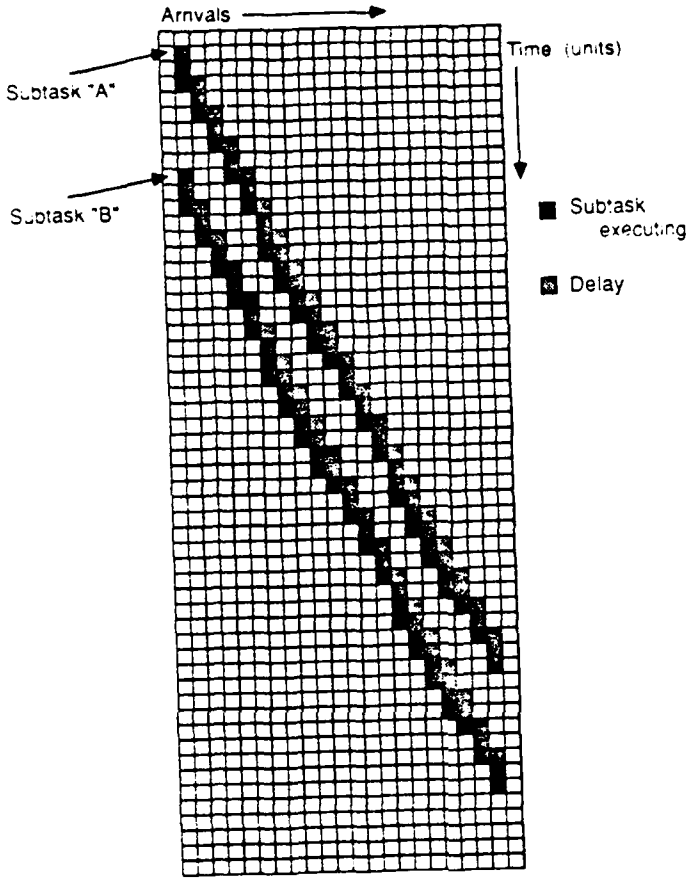


Figure 6-6(a): A gantt chart representation of the subtask scheduling under the first buffering procedure.

the initiation cycle period is no longer equal to the interarrival time. After the pipe filled, there was a conflict for resources and new arrivals were queued. The queueing was no longer needed after four arrivals, but demand *resynchronized* four arrivals hence. The interaction between new and old tasks both competing for a common pool of resources. If we examine a state diagram of this mapping we can see that additional states are needed to model the queueing of a subtask at a particular processor. The simplest representation would require 3^6 states. Each processor can be processing a subtask (4 possible states) or

could have a subtask awaiting execution. The description of the queueing can be limited to two states—either a subtask is waiting in a queue or it isn't. In general, though, a complete description of the queueing would require more specificity: what subtasks are waiting and how long have they been waiting?

An alternate way of effecting the delaying is to delay subtask *B* whenever a conflict arises (figure 6-6(b)). The gantt chart for this technique shows that its periodicity is much smaller, though the tasks are always delayed by one unit. What is interesting about this delaying is that it neatly solves the problem of clipping the peak demand while creating no downstream conflicts. An inspection of the gantt chart reveals that the subtask executions have a steady pattern with a periodicity equal to the interarrival time. This pattern is the *initiation cycle* (fig. 6-6(c)). Notice that this pattern can be shifted by 1 arrival every two time units and the entire gantt chart in figure 6-6(b) can be generated. The interpretation of this can be seen if each of the six components in the sequence are explained. The task can be divided into 6 execution segments—one for each time unit of execution of the task which has had the subtask *B* delayed by one unit. The initiation sequence is comprised of these 6 segments spread over 6 task arrivals. That is, at each step, three of these segments for three of six arrivals is "processed." Because of the judicious use of delays, half of these segments are non-executing segments so the subtasks can be cleanly mapped onto three processors. The actual mapping state diagram takes longer than two time units because the assigning of subtasks to processors is performed in such a way as to prevent subtasks from having to "jump around" from processor to processor. As with the class one allocation case (four processors allocated), the state diagram cycles at some multiple of the length of the longest subtask and task interarrival time.

This example has demonstrated a few important points. With a gantt chart representation, we can identify a cycle of task initiators which corresponds to the portions of subtasks

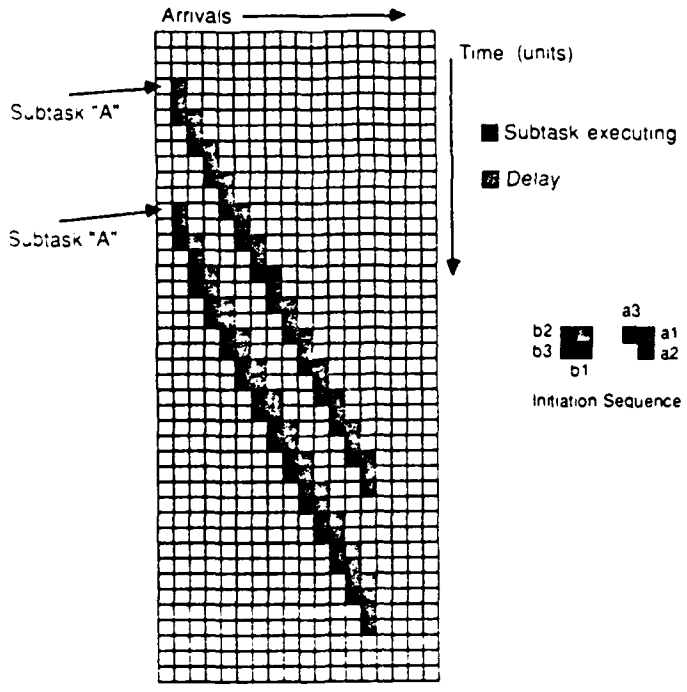


Figure 6-6(b): A gantt chart representation of the subtask scheduling under an alternate buffering procedure.

which require execution at each time unit. These initiators have a periodicity which is equal to the interarrival time for class one allocation environments but their periodicity is distorted in the presence of delays. In addition, their periodicity is a fraction of the periodicity of the mapping state diagram because this diagram encompasses the *assignment* of those subtask portions. The mapping period is longer because it must demonstrate that subtasks are fully executed on single processors. If the underlying architecture provided cost-free subtask preemption and migration (something not very commonplace), then some subtask A could begin execution on processor p_1, move over to processor p_2, and finish up on processor p_3. At the same time, a second subtask B could begin execution on processor p_2, move over to processor p_3, and finish up on processor p_1. In this case, an assigment could be performed

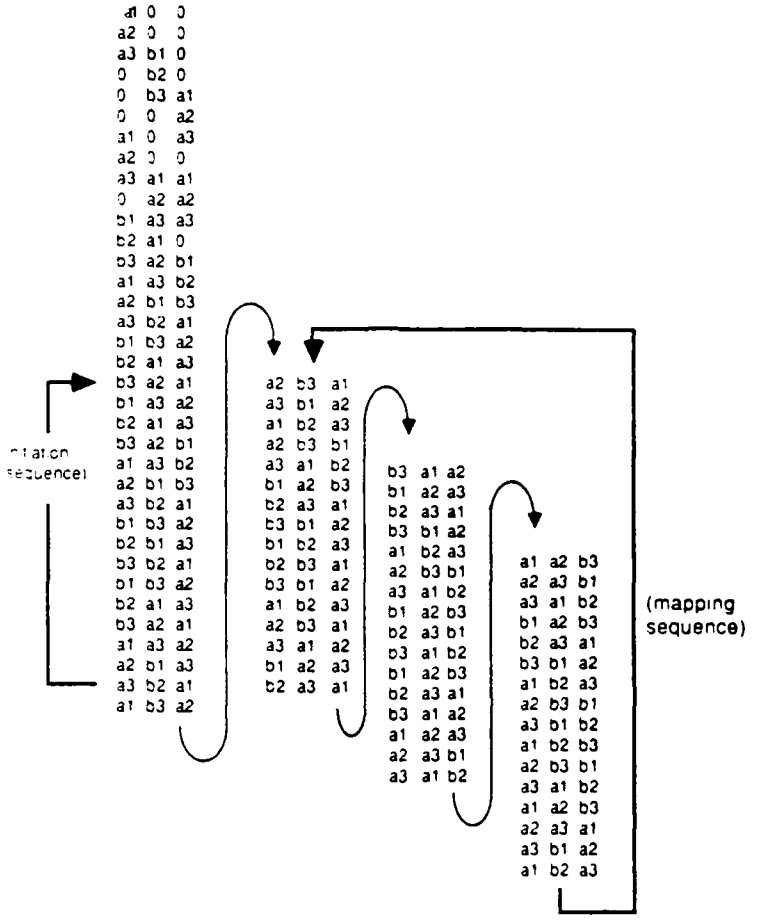


Figure 6-6(c): A state diagram of the mapping of subtasks to particular processors, demonstrating an initiation and a mapping cycle.

in the space of the initiation sequence and the mapping and initiation periods would be identical. Finally, while subtask delaying is necessary to flatten peaks in demand, it is not always possible to find delaying techniques which do so without imposing a cost on system performance. In this example, two approaches were shown. The first increased latency by one time unit for half of the tasks and the second always increased latency by one time unit. The first had a more complex initiation sequence associated with it and therefore required a

larger number of states to minimally represent the mapping of subtasks onto processors. The first delaying approach resulted in tasks being processed with a lower average latency but was less stable and possibly harder to implement because of the nonconstant application of delays. The second delaying approach—always delay subtask *B*—might be easier to implement but resulted in a higher average latency.

6.2. State Generation

The previous example demonstrated how a state-diagram notation could be used to describe the initiation and mapping cycles. The initiation cycle was a repetitive sequence of subtask initiations which described when particular subtasks were initiated and, implicitly, when they were delayed. The mapping cycle encompassed the initiation cycle but also showed *where* those subtasks were processed, i.e. it encompassed both task and processor states. It was shown that the period of the mapping cycle was dependent on the period of the initiation cycle, the interarrival time, and the time of the longest subtask.

The state diagram notation is useful not only for describing these cycles, but generating them as well. Consider the task graph in figure 6-2(a) and its gantt chart representation. In the previous example, two different delaying rules were used to generate the resulting mappings of the task set onto a class two allocation. The first rule could be summarized as: "delay new arrivals whenever there is a conflict." The second rule could be summarized as: "delay subtask *B* whenever there is a conflict." From a procedural point of view, in both of them, an allocation was decided upon and a delaying rule was chosen. As explained in the example, there are an infinity of delaying procedures that can be applied, some are based on regularly applied heuristics and some based on specific delay lists. Delay rules based on heuristics could read: "delay all new tasks whenever there is a conflict" or, more generally, "whenever there is a conflict, delay the *n*th subtask belonging to arrival which is *m* arrivals

old." These procedures generate repetitive delay lists. Specific delay lists are lists of subtasks to delay at particular times or situations and may not follow a particular pattern. One of these lists could read: "Delay subtask A/arrival l at time x; delay subtask B/arrival m at time y; delay subtask C/arrival n at time z." These arbitrary delay lists are as viable as heuristic-based delay rules in that they address the same problem of holding over peaks in demand to allow a class two allocation but they are harder to investigate and may be harder to implement. Despite the forseeable implementation difficulties, the existence of these specific delay lists is noted because the domain of delay sequence candidates must be broadened to include any arbitrary, workable sequence.

The problem of finding delay sequences is, then, expanded. We must now consider any arbitrary sequence that fulfills the objective of buffering demand. A procedure for suboptimally choosing a delay sequence would be to consider a couple of heuristic-based sequences (because they are easy to generate and evaluate) and then choose the one which best meets certain performance objectives. This procedure yields suboptimal solutions because (a) the choices are being made from a subset of the possibilties and (b) there is no way to show that the optimal solution always resides in that subset. A procedure for *optimally* finding a suitable sequence could be: enumerate all of the delay-ordering possibilities and then select the sequence that best fits the performance objectives. "Optimal," in this sense, implies making the best choice from the entire domain of possibilities. A few questions then arise. For the general problem of mapping a predetermined task set of known arrivals, is it possible to optimally select a delaying scheme? If so, what is the usefulness of the optimal sequences? And how do we relate optimal solutions formed over one resource type to a possibly conflicting optimal solution formed over another resource type? The remainder of this chapter investigates these questions by outlining and then discussing a procedure for enumerating the possible delay lists.

Assume that, as in the example task graph in figure 6-2(a), the task timing and allocation have been fully specified. The mapping of tasks onto processors can be performed in a step-by-step manner. When a mapping conflict occurs—when the instantaneous demand for resources is greater than the instantaneous supply—instead of applying a *particular* delaying rule, all alternatives are explored. This creates of branches in the state diagram. Each of these branches is independently investigated. Along each branch, when a conflict arises, it is handled by generating all possible subtask delay alternatives which push back the conflict (fig. 6-7). When some state X is generated, the paths leading back from X to the root of the state diagram are investigated to see if the state has already been generated. If it does, the path leading to X is terminated by connecting the states prior to the newly generated state X to the previously generated state X. This forms a loop in the state diagram. This represents a cycle in the state diagram from which a repeating sequence of delays can be extracted.

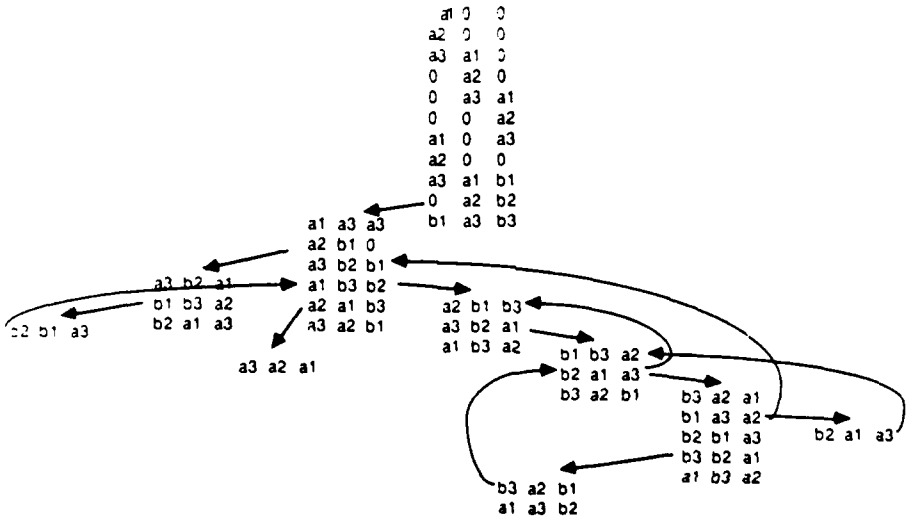


Figure 6-7: A schedule generation by state enumeration along each branch.

Each path in the state diagram will eventually terminate by looping backward. It will no longer be necessary to generate states after a loop because the subsequent states will be equivalent to those up the tree. Once the state diagram has been generated, a delay sequence can be chosen. In a manner akin to control strategy generation, delay sequences can be simple or complicated. Simple sequences involve paths through the graph that, in one cycle, visit each state in the cycle only once. There are a finite number of these simple sequences. Complex sequences involve paths that revisit states; there are an infinite number of these schedules. That is, if there is no limit on the length of the period of the sequence, then the period can be infinite. If there is an imposed limit on the periodicity of the sequence then the sequences can all be enumerated.

The problem of enumerating delay sequences has been demonstrasted in terms of finding all possible cycles in a mapping state diagram. Applying this technique is implausible because of the complexity of the underlying state description and the size of the resulting cycle detection and evaluation problem.

The complexity of the state diagrams rests on the following parameters The number of states of each subtask is equal to the execution time of the subtask divided by whatever the smallest time increment is. An additional state for a subtasks is "not processing." If there are M subtasks with a serialized execution time of N, then there will be $N + M$ states of execution for each processor. The allocation will yield the number of processors that can occupy one of these states. If there are P processors then there are P^{N+M} processing states Delays introduce extra states in two ways. First, if a subtask is delayed by U units, then there are U additional states to that particular subtask. If there is no ceiling on the amount that a subtask can be delayed, then there will be no ceiling on the number of states to describe the status of the subtask. One would think that a reasonable ceiling would be the execution time of that paticular subtask. That is, if a subtask takes twice as long to execute

as its stated execution time, then the delaying approach responsible for this situation is flawed. Unfortunately, this overlooks some of the key synchronization issues of the task graph mapping. If a subtask is to be fired after a merging of links from short and long paths, the subtask must be *systematically* delayed during the difference in these paths. This delay is a property of the task specification and not the allocation. Additional delays can be attributed to the delaying approach. To create a delay ceiling that is meaningful, then, two approaches can be used. The first is to set all ceilings to a large number, such as the anticipated latency of the entire task. In this case, if a delaying approach results in a particular subtask being delayed for a rediculously long period of time, that delay approach will be rejected. The problem with this ceiling is that it may not reject all approaches that should be rejected. The second ceiling could be created by factoring out the path differences leading to subtasks and then assigning subtask delay ceilings that are based on some standard delay allowance plus some additional allowance to subtasks which do not lie along the critical execution path of the graph. The delay ceiling imposed on the subtask adds to the number of possible states of that subtask. A final complication is that each processor may be currently executing a subtask, so it may be in one of $N + M$ execution states but the queued subtasks must somehow be accounted for. The actual location of these subtasks is dependent on the underlying architecture. If we make a simple assumption that all of the queued subtasks are held in some bin, then the composition of that bin must be described. Complicating matters is that any number of task arrivals may be in the system. If the number of arrivals in the system is A and the sum of all of the allowed delays is D, then $A * D$ states are required to represent all of the delayed subtasks. To more completely describe the subtasks that processors are executing, each processor now can be in one of $A * (N + M)$ states, so the number of system states is now $A * D * P^{(A * (N + M))}$.

Assuming that the state diagram for utilization and contention of each individual resource could be generated, the enumeration of all simple delay sequences (cycles) is costly, even if a lower bound on the frequency of the sequence is imposed. The enumeration of all complex cycles is, obviously, impossible. If all allowed simple cycles are enumerated, the optimal one could be found by inspecting the impact of each sequence on a variety of performance metrics. If sequence *A* is used, what will be the resulting latency of the task? If sequence *B* is used, will the demand for resources be uniform or will it be very low except for peaks right at the allocation level? Choosing the *optimal* sequence rests on the application of an evaluation scoring to the candidate sequences.

The state generation technique arose from a method for determining latency schedules for static and dynamic processor pipelines. The mapping of heterogeneous, dependent subtasks onto a system of heterogeneous processors is a fundamentally more complicated problem than the pipeline control strategy generation problem. In the pipeline problem, it is assumed that all of the states can be enumerated. This assumption is partly based on the simplification that subtasks (in this case subtasks are pipeline stages) all execute in one time unit. Furthermore, for processor pipelines, each subtask must be mapped to a unique physical resource. Finally, the state generation procedure fails when timing information is either not fully provided or if timing is not static; the techniques do not seem generalizable to dynamic timing systems.

The goal of this chapter was to provide some justification for simulation and heuristic-based analysis by presenting the unsuitabilty of a methodology that is successfully applied to problems in simpler domains. The enumeration of mapping sequences turns out to be as problematic as the enumeration of subtask orderings that was demonstrated in the chapter on static analysis. The next two chapters will present analysis methods that provide information that is more suitable to both the configurer and the architect.

CHAPTER 7

Schedule Simulation

An intermediate level of simulation models the assigning of subtasks onto a class two allocation of resources under the direction of delaying heuristics. The technique is motivated by a desire to experiment with and evaluate a number of heuristics for clipping peaks in demand for resources without having to determine an architecture and implementation strategy for each of those heuristics. Some of the heuristics may prove to be useful for some task graphs but impossible to implement. Others may be easy to implement given an underlying architecture but unworkable for some task graphs. Both an architect and a configurer would be interested in this level of simulation: an architect would want to evaluate an ever-expanding number of heuristics and refinements under classes of tasks and allocations. A configurer, working from an architecture with a predefined set of supportable heuristics, would want to investigate how these alternatives applied to a specified task and set of allocation constraints.

The chapter is organized as follows. Notation is introduced which provides a way of describing the resource requirements of subtasks and of relating their initiation times and delays. This notation involves the use of discrete variables represent resource utilization. Subtask initiation and precedence-based timings are incorporated by index-shifting. The simulator itself is then described as a time-step-based evaluation of resource demand and supply and a subsequent application of a specified delaying heuristic. After the simulator is introduced, some delaying heuristics are presented, along with the motivation for studying them. As an example of the extensibility of this simulator into more implementation-specific areas, two concerns of the architecture specified in Appendix A are introduced. These

scheduling concerns are integrated into the scheduling simulator. Finally an example simulation is presented to give the reader a feel for the usefulness of this methodology and some results and conclusions are presented.

7.1. Representation by Sequences

A compact representation of the processing of a subtask is by discrete sequences. At each stage of execution a subtask requires some resources. The evaluation of a subtask discrete variable will yield a value of 1 when that particular subtask is utilizing a resource (processing) and 0 when it is not. Consider the example graph in figure 7-1(a). A suitable discrete variable representation for subtask v_1, which executes for 3 time units is

$$v_1[n]: \quad v_1[0] = 1, v_1[1] = 1, v_1[2] = 1, v_1[3] = 0, v_1[4] = 0, \cdots$$

A precise explanation is that the value of the variable represents the processor-space required by that subtask. Since we are dealing with subtasks which confine themselves to execution on one type of processor and only one processor at a time, this notation is suitable.

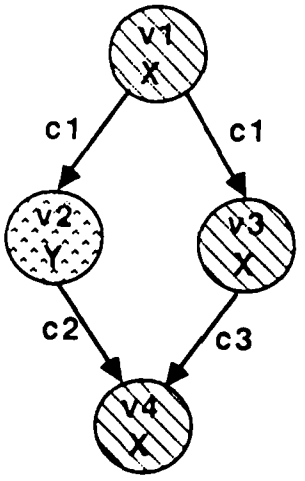


Figure 7-1(a): An example task graph.

Precedence relationships among the subtasks can be described by a precedence matrix which aids in noting that successor subtasks are initiated only after their parent subtasks are executed. In this notation, a subtask is considered to fire at some basic time plus the time required by all of its predecessors to fire. For example, if the task in figure 7-1(a) is initiated at time 0, subtask v_1 would fire immediately, but subtask v_4 would fire after v_1, v_2, and v_3, as well as after the communications subtasks c_1, c_2, and c_3. Precedence information can be used to find the longest execution path to the subtask via the linear programming method described in chapter 5. As explained in chapter 5, an alternate method is to calculate and compare distances along all the paths from the subtask back to the (virtual) source and then choose the current longest path as that which contributes to the time-shifting of a subtask initiation. Subtask v_4 would, in effect, be *delayed* by its parent subtasks that lie on the instantaneous longest path. If we note the latency of each of the subtasks on this longest path by $L[v_i]$ and $L[c_i]$ then given the longest path shown emboldened in figure 7-1(b), v_4 will be executed at time $L[v_2]+L[v_3]+L[c_1]+L[c_3]$. Another way of expressing this is that for a single initiation, the utilization of processors by v_4 at any time n is given by: $v_4\left[n - (L[v_1]+L[v_3]+L[c_1]+L[c_2]+L[c_3])\right]$. More generally, the resource utilization of any task v_i is given by:

$$UTILIZATION = v_i\left[n - \max_{p\in P}\left[\sum_{j=0, v_j \in p}^{j=T} L[v_j]\right] - \max_{p\in P}\left[\sum_{j=0, c_j \in p}^{j=T} L[c_j]\right]\right],$$

where p is a set of subtasks on a path from the subtask v_i to the virtual source and P encompasses all of the sets p. The notation provides an easy way of representing resource utilization when a task graph is executed. If a subtask is initiated at time 0 and requires resource type X, a measure of the resources required by that subtask at any time subsequent to initiation can be retrieved by the discrete representation:

$$\text{X-Required}[n] = v_1[n].$$

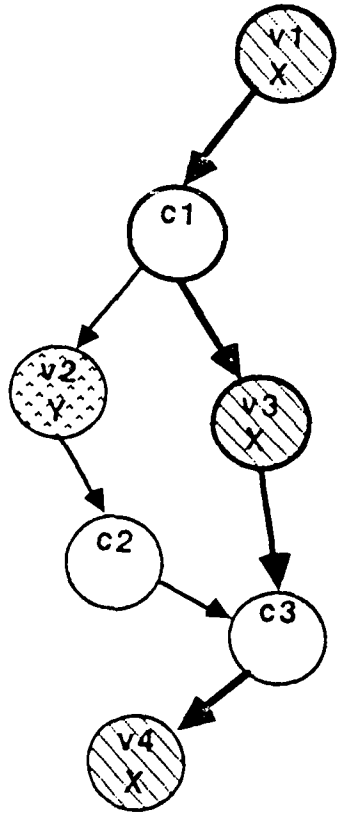


Figure 7-1(b): A longest-path analysis of the task graph in 7-1(a).

Since, in the example, both v_1 and v_4 utilize type X processors, a more complete description would be:

$$\text{X-Required}[n] = v_1[n] + v_4\left[n - \max_{p \in P}\left[\sum_{j=0, v_j \in p}^{j=T} L[v_j]\right] - \max_{p \in P}\left[\sum_{j=0, c_j \in p}^{j=T} L[c_j]\right]\right],$$

where p is a set of subtasks on a path from the subtask v_4 to the virtual source and P encompasses all of the sets p.

An execution of a task graph with an interarrival time of IAT is really the instantiation of all subtasks with start times of $0, IAT, 2IAT, \cdots$. Each task arrival then has a start time

which is an indexed multiple of the interarrival time. A representation of the utilization of processors by subtask v_1 across all initiations of the task would be:

$$\text{X-Required[n]} = v_1[n-0] + v_1[n-IAT] + v_1[n-2IAT] + \cdots$$

Generalizing one more step, since both v_1 and v_4 utilize type X processors, a complete representation would be:

$$\begin{aligned}\text{X-Required[n]} = \quad & v_1[n] + v_4\left[n - \max_{p\in P}\left[\sum_{j=0,v_j\in p}^{j=T} L[v_j]\right] - \max_{p\in P}\left[\sum_{j=0,c_j\in p}^{j=T} L[c_j]\right]\right] \\ + \quad & v_1[n-IAT] + v_4\left[n-IAT - \max_{p\in P}\left[\sum_{j=0,v_j\in p}^{j=T} L[v_j]\right] - \max_{p\in P}\left[\sum_{j=0,c_j\in p}^{j=T} L[c_j]\right]\right] \\ + \quad & v_1[n-2IAT] + v_4\left[n-2IAT - \max_{p\in P}\left[\sum_{j=0,v_j\in p}^{j=T} L[v_j]\right] - \max_{p\in P}\left[\sum_{j=0,c_j\in p}^{j=T} L[c_j]\right]\right] \\ + \quad & \cdots\end{aligned}$$

Once the framework for representing subtasks is in place, it is easy to add delays to each subtask that are incurred because of competition for resources. Remember that this competition is what chacterizes a class two allocation environment. The initiation of subtasks can be delayed as a result of buffering to reduce instantaneous demand for resources. The above framework is useful because the delay of a subtask is the sum of any delay applied to it directly plus the delays applied to its predecessors. This delay-precedence relationship can be cleanly added into the representation. The delay incurred by subtask v_i which is initiated as part of task arrival m is denoted $D_{v_i}[m]$. A subtask v_4 which is part of initiation m executes at time $\hat{n}$:

$$\hat{n} = n - D_{v_4}[m] - \max_{p\in P}\left[\sum_{j=0,v_j\in p}^{j=T} L[v_j] + D_{v_j}[m]\right] - \max_{p\in P}\left[\sum_{j=0,c_j\in p}^{j=T} L[c_j] + D_{c_j}[m]\right].$$

7.2. Schedule Simulation

The above notation provides a convenient framework for manipulating and representing start times and delays of individual subtasks. The schedule simulator, structured around the

repeated evaluation of the resource-utilization equations, can track allocation supply and demand. It is called a schedule simulator because its objective is to evaluate scheduling alternatives—heuristics that buffer demand and underlying initiation mechanisms—in the presence of supply and demand parameters such as the task description and allocation.

The schedule simulator takes as input a description of the directed task graph similar to that given to static analysis techniques. A specific ordering to subtasks must be specified and some method of determining the execution time for each subtask must be fully specified. The simulator also receives a description of the resources that are available. This description, for now, is limited to specifying single typed resources. The reason for this is that it simplifies describing two subtasks which present different requirements to the same resource type (modes). In addition, it allows the simulator to more easily describe changes in the processing behavior of individual subtasks. The front end of the simulator performs the graph expansion and transformations necessary to represent subtasks in a unified manner and to explicitly represent the particular link orderings that were chosen. A representation of the utilization of resources by subtasks as discrete variables is then created. A structure which holds for each subtask the possible paths through the task graph back to the virutal source is created from the subtask precedence matrix. This structure is created by a recursive depth-first search procedure and is used to evaluate the execution state of subtasks while the simulator is running. Generating this structure reduces the computation that must be done at simulation runtime by facilitating the use of efficient indexing schemes. This representation, as explained earlier, directly allows delays to be included in the description of when a particular subtask executes.

The scheduling simulator is time-based: the simulator does some processing for the "current time" and then increments its notion of time by one unit. Since the simulator deals on the level of subtasks, its granularity is significantly larger than an architectural

simulator. The processing (or communication) time of subtasks must all be specified in some common, smallest increment; the choice of increment is application specific but reasonable choices are tens, hundreds, or thousands of microseconds.

At each time increment, the simulator browses through subtasks that are part of the task that has most recently arrived and subtasks that are part of tasks that have arrived in the recent past. Each of those subtasks is described by a discrete variable which gives its own resource requirements time-shifted by the time of subtasks which precede it and the delays which it and its predecessor subtasks incur. Once all of the time offsets have been computed, the *status* of each subtask—waiting to be initiated, waiting for other subtasks to finish, already initated, just finishing, and completely finished—can be immediately inferred. If a subtask is already initiated, it continues to require resources. If a subtask is waiting to be initiated, it represents a new demand for resources.

The simulator first assigns resources to subtasks that are in the midst of processing; this satisfies an underlying assumption about preemption. The resource assignment is done by matching the resource required by each subtask to a description of the resource pool that was provided to the simulator. If there is a surplus of resources, the new demands can be met. New demand may, however, be greater than available, unassigned supply. This is what characterizes a class two execution environment.

Delay heuristics come into play at this point. A subtask servicing mechanism is the arbitrator between demand and supply. The simulator pares down demand by delaying subtasks that find themselves at the head of subtask delay lists. The lists are constructed by ordering subtasks according to one of several scoring algorithms. The scoring algorithms represent the application of queue servicing rules or delaying heuristics and are based on task, architecture, and mapping parameters. The simulator mechanism for this paring is fairly simple: the delay coefficient, D_{v_i} which corresponds to that particular subtask/arrival is

incremented.

Demand for all resource types must be examined so that it along all resource dimensions, instantaneous demand is less than or equal to instantaneous supply. If any delaying is made, the timestep is resimulated to see if there were any unanticipated effects of that delaying. For simple graphs, this resimulation is not important. Resimulation is necessary to catch second-order effects caused by graphs which contain complicated signalling constructs or which contain inter-arrival initiations.

The simulator continues step-by-step subtask assignment and, along the way, monitors results of the simulation. The choice of what to monitor is a critical element in this simulator. Unlike the tools which support the state generation and enumeration methodology, this simulator does not explicitly keep track of, or search for, initiation or mapping cycles. In addition, the state of the system is not explicitly enumerated. The only information that is saved from step to step are the delay coefficients because they are needed to compute the status of subtasks. The simulator can be instrumented to generate performance metrics such as latency and throughput. These metrics can not be directly used to evaluate a configuration architecture. Instead, they can help to provide a rejection test: if the combination of the task graph, allocation, and delay heuristics does not meet an application's performance standards under this simulation, then it will not perform better in an architectural simulator (or a real system). Subsequent detailed simulation will yield better performance if there is some peculiarity about the synchronicity of tasks that is revealed in the scheduling simulator and is masked in the architectural simulator. In this case, it is still better to address these synchronizations in the choice of allocation and ordering parameters than to assume that they will never occur under detailed simulation. If the allocation and mapping parameters do not adequately serve task and arrival demands, this simulator should help determine whether there are fundamental problems in the allocation (too few resources) or if

the assigning and delaying heuristics are poorly matched to the task. Aside from using performance metrics to provide rejection tests, the simulator can provide runtime information about the low-level sequences of initiations and delays. For example, bounds can be placed on the delaying of critical subtasks and the simulator can detect whether those bounds are exceeded. This is useful in determining delaying approaches which favor initiation of subtasks that lie along critical paths of the task.

7.2.1. Complexity

Some bounds must be placed on subtask delays in order to coerce some level of system performance. If a delaying heuristic causes some subtasks to delay indefinitely, the heuristic should probably be modified. A more practical concern is that the simulator should be required to evaluate a stable number of subtasks: tasks that arrive and never leave require evaluation at each stage of simulation. If tasks could take infinitely long to execute, then the number of subtasks for the simulator to consider would grow linearly with the timestep of the simulation. Note that the restrictiveness of the maximum-delay bounds is not at issue: since they are upper bounds, they can be made generously large and still serve both above purposes. If some latency bound may have been placed on the task graph as part of a set of configuration performance metrics then the choice of bounds is made a bit easier: if the task must complete in B time units, then no subtask can be delayed by more than B units. Moreover, subtasks along the longest path (which has a latency of, say, L) can be delayed by an aggregate amount of $B - L$ units.

The complexity of this simulation lies in the size of various parameters. Each simulation cycle will require the evaluation of all subtasks, S, in each task arrival that must be considered for the current timestep. The number of arrivals, A, that must be considered is related to the maximum latency of the task set, B and the task interarrival time, IAT. This

relationship is as follows:

$$A = B \ div \ IAT + \begin{array}{l} 1 \text{ if } (B \ mod \ IAT) \neq 0 \\ 0 \text{ if } (B \ mod \ IAT) = 0 \end{array}$$

There are $A*S$ subtasks to evaluate per timestep. Each subtask requires a computation of its current initiation state, which involves a summation over all of its predecessor subtask paths. At each arrival, a new task must be considered but the oldest arrival that was being evaluated can be removed, as can all delay coefficients saved for subtasks for that arrival.

7.2.2. Delay Heuristics

An arrival-based servicing policy biases access to contended resources towards subtasks that have certain age characteristics. In the architectural specification in Appendix A, tasks are tagged with an arrival number. This arrival number allows all of the subtasks in the system to be ranked according to relative age. One arrival-based heuristic orders all of the subtasks according to age and then initiates tasks which are oldest. An intuitive explanation is that tasks that have been waiting the longest are, in some way, the most deserving of resources. This policy is similar to age-based service policies in operating systems. The longer a process sits on a run queue, the higher its priority gets, until it is eventually run. An implementation structure here could be some sort of FIFO. An alternative heuristic would order all the tasks and then bias servicing towards tasks that are the youngest. In this case, resources are allotted to tasks that haven't been aging, with the assumption that new tasks which are being processed efficiently should get whisked through without being burdened delayed by old tasks that have already been tainted by a delay. A youngest-first policy is similar in implementation structure, though not exactly in intent, to least-recently-used memory management policies which allow memory pages which are most active to be preserved in fast, primary memory, while pages which are not as recently active are sent back to secondary storage.

The aging heuristic which biases towards older tasks is a flow-control heuristic. In a task servicing environment, a good general observation is that old subtasks are old because either they are not ready to be serviced, i.e. not all of the parent subtasks have completed, or adequate resources are not available to allot to them.[1] Schedule simulation has shown that over a broad range of task graphs, the application of this policy causes all tasks to complete—the flow of tasks into and out of the system is conserved—though not necessarily at a minimal latency.

The second policy, bias towards younger tasks, achieves higher latency performance for some subtasks at the expense of others. When properly tuned, newly arrived tasks will get preference for resources and will execute at a minimum latency. The task latency will not necessarily be as low as that in a class one execution environment because there may be competition for resources within a task and fully tuning the aging computation may not be possible. When newly arrived tasks get preference for resources, something must get crowded out. Subtasks that are downstream in the graph will tend to be unfairly biased against subtasks of newer arrivals which are near the top of the task graph. What this means is that older subtasks will tend to get crowded out of servicing by newer subtasks. Tasks which have relatively high peaks in resource demand caused by competition among different arrivals will be most susceptible to this crowding. The subtasks which belong to older arrivals will, when this demand peak occurs, become delayed and must wait for all younger-arrival tasks to receive service. If new tasks arrive and begin to compete for these resources, the subtasks that have been waiting around will get crowded even more: the delaying gets larger and larger until the subtask is hopelessly aged. This heuristic is interesting because although it fails in resource-poor environments, in some time-critical applications, it is less important that all results get processed as it is that some results get processed

[1] For the purpose of this simulation, the costs involved in attempting to service tasks which are not ready are ignored.

in a timely manner. A more realistic application of an aging heuristic would take both arrival and subtask age into account. The two age heuristic domains, arrival and subtask hierarchy, can be combined in the obvious ways.

7.2.3. Cost Based Heuristics

Another class of heuristics is based on the evaluation of the task graph and the assignment of costs to the delaying of certain tasks. Several factors come into play when considering the cost of delaying a particular subtask. First, a subtask may be part of the critical (longest) path of the task graph. If this is true, then delaying it will directly affect the latency of the graph. Subtasks that impose larger latency costs could be prioritized over non-critical-path subtasks. A second factor is the hidden cost of delaying. If a particular subtask with many successor subtask chains (fig. 7-2) is delayed, then all of those chains will be delayed. This naturally poses a bias towards initiating tasks which are towards the top of the task graph over those near the bottom. For example, in figure 7-2, subtask v_2 has 7 successor tasks, whereas subtask v_3 has only 3 successors. In this example, v_2 would have a higher cost-of-delaying than v_3. As mentioned earlier, this cost method should not necessarily be extended into inter-frame contention because biasing towards both earlier subtasks and earlier arrivals can, due to inter-arrival competition, result in earlier tasks never completing.

7.3. Architectural Implementation Considerations

Depending on the implementation architecture, it is likely that many of the above heuristics can not be applied globally. That is, many of them require keeping track of the status of other subtasks and resources. For distributed implementations, it is never cost-free to exchange state and control information about resources and subtasks; furthermore, the processing and transfer of this information can add to the load that is being monitored: the

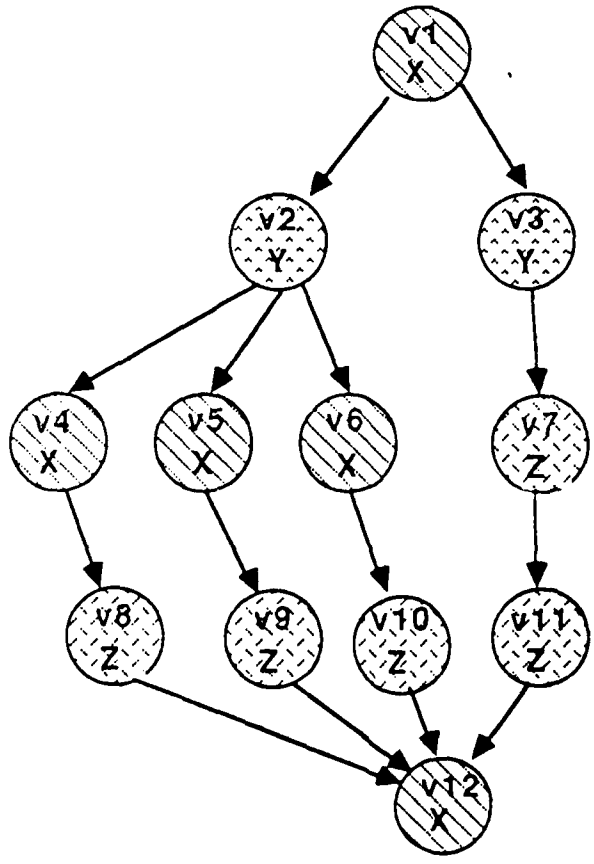


Figure 7-2: Note that processing subtask 2 has 7 successors, whereas subtask 3 has 3 successors.

uncertainty principle is applicable to network instrumentation. In addition, underlying architectures may have low-cost primary mechanisms for distributing subtasks and higher cost secondary mechanisms. From an architect's point of view, the design of these low-cost mechanisms warrants extensive experimentation and simulation. From a configurer's point of view, the low-cost mechanism may or may not be usable given a particular application. The evaluation of higher cost and lower cost mechanisms needs to be supported. Scheduling simulation provides a route to the cheap investigation of these approaches. Two examples of this investigation will be given. The first is the evaluation of an underlying low-cost task

servicing mechanism provided by the d-ALPS architecture in Appendix A.[2] Given this mechanism, schedule simulation can be used to compare higher cost but possibly more suitable servicing mechanisms. The second is the implications of a limited input queueing structure which in turn requires that resources be committed to serving a subtask some time before actual initiation. This results in additional utilization of resources. In a class one environment, the timing of the committing periods can be modelled statically, and the additional load encoded in an expanded demand graph. In a class two environment, this is not possible; the burden changes dynamically. Schedule simulation can present the impact of the requirement to bind resources in either environment and can evaluate alternative committing strategies.

The low-cost scheduling in the distributed implementation described in Appendix A is based on a local and global round-robin servicing of tasks. Each node in the architecture has an output queue that is served in a round-robin fashion. The local queues are arranged in a logically circular queue, each receiving a "control token," allowing it to attempt to initiate all of the children of the subtask that is on top of the queue. As a basis for comparison, facilities to simulate this low-cost scheduling mechanism were built into the scheduling simulator. These facilities involved creating structures to represent a circular list of subtask queues. The simulator maps a subtask to a particular resource (assigns the subtask) when it initiates that subtask, and provides this information to a scoring function which orders subtasks based on location. It should be mentioned that, in the general (ideal) case, the location is an arbitrary subtask attribute. However, in configurations with small node counts, location will be related to the particular timing of graphs with concurrent elements. That is, if subtask assignment is based on location, then timing-adjacent subtasks will be mapped to location-adjacent resources.

[2] This appendix gives a specification for a distributed ALPS architecture (d-ALPS) which is comprised of a token-bus based connection of primitive-control unit pairs.

A reasonable implementation of some of the above heuristics is to layer them under the low-cost control token passing mechanism. The heuristic is then applied to each subtask in a local output queue but servicing is still passed according to location. Altering the *local* servicing policy—the servicing mechanism that each node employs in isolation—to implement one of the above heuristics would not be terribly difficult. Given the architecture described in Appendix A, sorting and costing functions required for implementation of one of these heuristics could be executed when the node would be otherwise idle. Changing the global servicing policy—the node-to-node round robin control token passing—could be difficult and expensive. Since the above heuristics require several stages of transferring or exchanging information, a complete implementation implies that the low-cost control-passing mechanism would have to be bypassed.

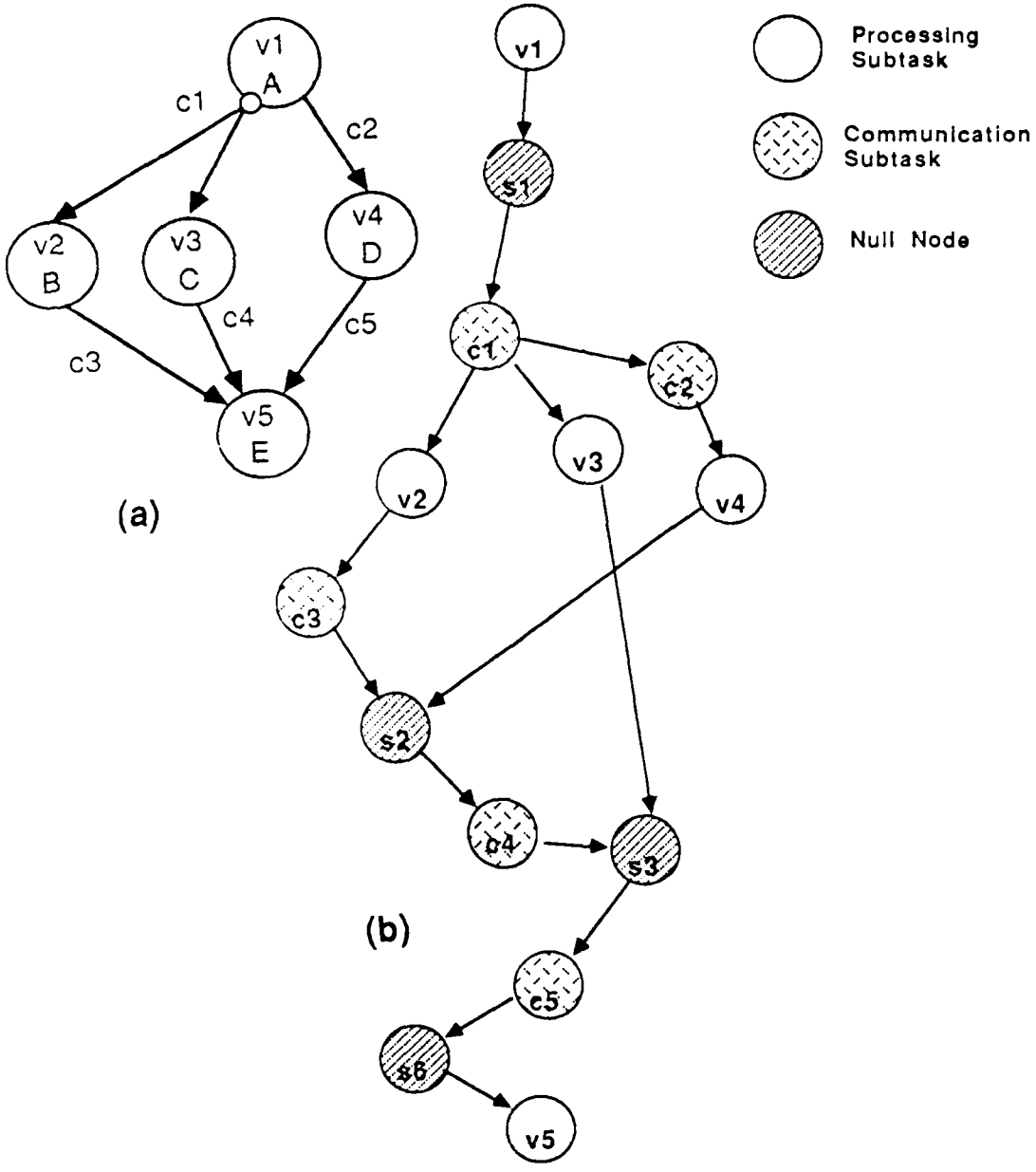
7.3.1. Processor Binding

A particular architectural problem that warrants investigation via schedule simulation is processor binding. The architectural specification in Appendix A describes a system in which output queues are used to buffer data for subtasks that have not yet been initiated on processors. The data for a particular subtask may logically reside with several predecessor subtasks and physically may sit in several different output queues. As explained in chapter 4 and in the d-ALPS specification, a priority link is used to establish logical subtask to physical primitive mappping. This process is implemented in an architecture where there is no input queueing of subtasks except for the subtasks that is about to execute. That is, in figure 7-2, a processor of type X can not accept data for subtasks v_4, v_5 and v_6 at the same time. This lack of input queueing implies that all data for a subtask (all communications subtasks preceding that processing subtask) must sit in output queues until a physical processor has been *bound*, or instantiated to that particular subtask. Furthermore, if a processor is bound to a particular subtask, it must stay bound until the remaining communication subtasks

execute. This identifies a fundamental utilization problem. If the binding (priority) communication subtask is ready to be initiated before the remaining communication subtasks are ready, the processor will sit committed but idle during the interim. Given this implementation strategy the bound-but-not-executing state represents a systematic utilization inefficiency.

7.3.2. Commit Groups

There is a fundamental representation extension which must be made to describe processor committing. Unfortunately, the purity of directed flow graphs composed of nodes with independent costs is lost. Consider the task graph in figure 7-3(a). Subtask v_1 initiates communication subtasks c_1 (a broadcast) and c_2 which in turn cause processing subtasks v_2, v_3, and v_4 to execute. Both c_1 and c_2 are the *binding* subtasks in that they cause resources to be committed to processing v_2, v_3, and v_4. A graph expansion can be performed which causes the communication subtasks to be explicitly represented and ordered according to a predefined sending priority: $c_1 < c_2$ (fig. 7-3(b)) The graph now looks similar to those in chapter 4. Processors for v_1 and v_2 would be found before the processor on which v_1 resides initiates c_1. In a class one allocation environment, these processors are assigned instantaneously. In a class two allocation environment, finding these processors could involve delays due to contention for these resource types. Regardless of the allocation environment, after processors are found, c_1 can begin. Again, in a class two environment, c_1 may not begin immediately. The allocated processors remain allocated-but-not-processing until c_1 completes. Then their state changes instantaneously and they are now executing v_2 and v_3. As soon as c_1 completes, the same committing and then communicating process is undertaken for v_4. Figure 7-3(c) shows this process graphically. Nodes v_2' and v_3' represent commit "subtasks" for v_2 and v_3. The time they require is dependent on the time it takes for c_1 to be initiated and then executed. In a class one allocation environment, this is simply



Figures 7-3(a) and 7-3(b): An example task graph and the expansion of 7-3(a) explicitly representing communication subtasks and a single subtask ordering.

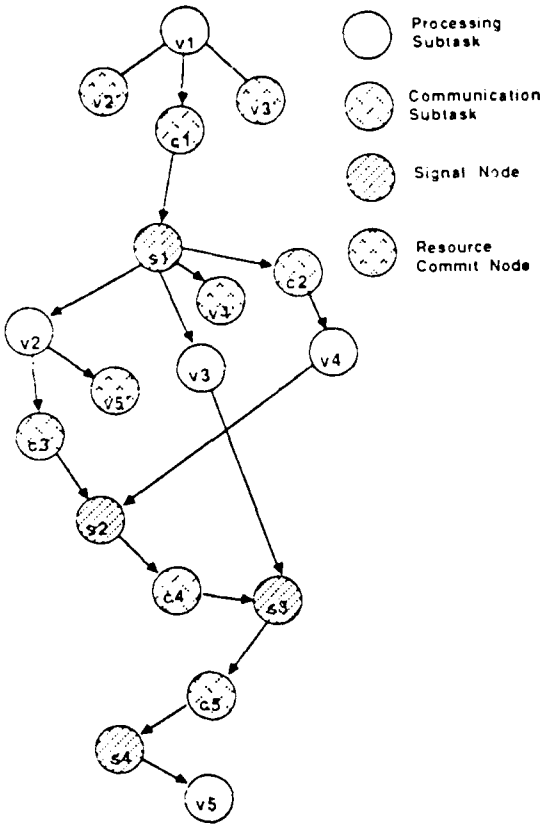


Figure 7-3(c): An expansion of 7-3(a) showing resource committing.

$L[c_i]$. In a class two allocation environment, this is a *variable* time.

A reasonable way to denote this variability is to define the collection of nodes which participate in the committing. A "commit group" is then composed of three elements. The first is set of subtasks that must be created to act as commiting nodes; in this example, v_2' and v_3' comprise this set. The second element is a (single) communication subtask which can be initiated after the commiting nodes are assigned; in this example, c_1 can be sent after v_2' and v_3' are initiated. The final element is a *signal* node whose initiation represents the

end of the communication subtask and causes the processors which were "executing" v_2' and v_3' to begin execution of subtasks v_2 and v_3. This signal node is so named because, in effect, it signals that v_2' and v_3' are finished and v_2 and v_3 can begin.

Commit groups can be used equally well to represent the merging of several communication subtasks to a single processing subtask. Consider the task graph in figure 7-3(a). The merging can be expanded to represent predefined priorities: $c_3 < c_4 < c_5$. In this example, subtask v_5 must be committed before c_3 can be sent, and can be initiated after c_5 is sent; for the merging case, the signal node follows the last merging communication. Figure 7-3(c) depicts this committing.

7.3.3. Integration

The commit groups can be integrated into scheduling simulation cleanly by observing a few basic principles. First, from a time-step simulation point of view, the simulator can reference some structure holding the status of various commit groups to investigate the state of commit subtasks. The communication subtask is given an infinite initial delay until the commit subtasks are assigned; the commit group then causes that initial delay to be set to whatever delay was incurred in assigning commit subtasks. This prevents the communication subtask from being initiated until at least the commit subtasks are initiated. The execution time of the commit subtasks is set as infinite until the signal node is reached; it is then set to zero so that in the next timestep the commit subtasks will not require processors and the processor will be available for the processing subtasks. The above technique has been fully implemented as part of the scheduling simulator. A commit group manager detects and creating commit groups for various task graphs as required. During runtime, it monitors the progress of signals and manipulates the delay coefficient lists and execution times to accurately model the scheduling implications of processor commits.

7.4. An Example

An example simulation is as follows. Figure 7-4(a) shows a reasonably complicated task graph. This task graph demonstrates parallel demands for both communications and processing resources. Note that communications subtask c_1 is a broadcast to subtasks v_1 and v_3, whereas c_2 initiates c_2. The graph is expanded to represent both the subtask orderings and resource bindings (fig. 7-4(b)). For this example, resources must be committed to subtasks before the subtasks' preceding communications. For example, in figure 7-4(b) processors for subtasks v_1 and v_2 must be assigned before c_1 can initiate. A more complicated example is that a processor for subtask v_4 must be assigned before communication c_4 can initiate, but the subtask can not execute until after c_5 has occurred.

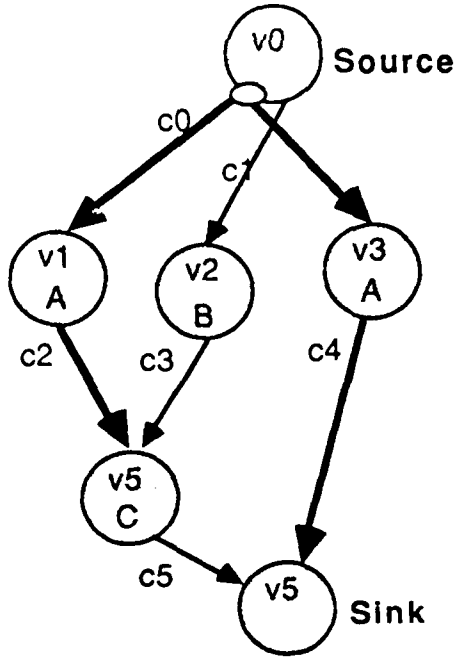


Figure 7-4(a): An example task graph.

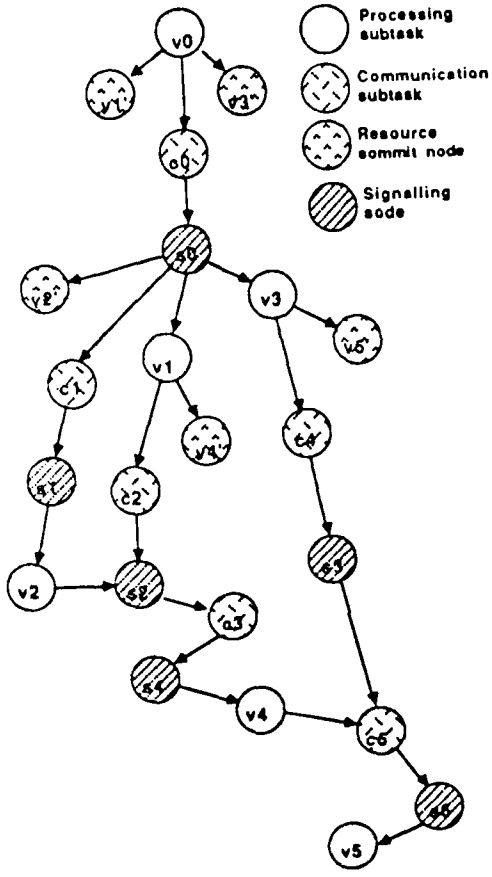


Figure 7-4(b): An expansion of the task graph in 7-4(a) to show subtask orderings and resource bindings.

Tables 7-1(a) through 7-1(c) provides subtask allocation information. Note that a class two execution environment has been created by severely restricting the numbers of processors and communications channels and by choosing a relatively low interarrival time. The ratio of the interarrival time to the latency, should this graph be executed in a class one environment is about five to one. That is, there are about five new task arrivals during the execution of one task. This ratio is incresed in a class two environment since the latency

7-1(a): Processor Subtask Information			
Subtask	Graph Level	Processor Type	Execution Time (usec)
v0	0	Source	2000 (IAT)
v1	3	A	2500
v2	5	B	3500
v3	5	A	2000
v4	5	C	4200
v5	8	Sink	0

7-1(b): Communication Subtask Information		
Subtask	Graph Level	Communication Time (usec)
c0	1	300
c1	3	200
c2	4	900
c3	6	400
c4	4	300
c5	6	700

7-1(c): Allocation Information	
Resource	Allocation
Source	1
A	10
B	10
C	5
Sink	1
Comm	1

Tables 7-1(a) through 7-1(c): Subtask and Allocation Information for Schedule Simulation Example

will increase, but the arrivals will remain fixed.

Simulations were performed choosing the six different delaying heuristics listed in table 7-2. In this table abbreviations of each of these heuristics is also provided. Task latency

Delay Heuristics for Example Simulation	
Abbreviation	Description
FF	Delay younger arrivals
FF_FA	Delay younger arrivals/higher level subtasks
FF_BA	Delay younger arrivals/lower level subtasksIder subtasks
BF_FA	Delay older arrivals/higher level subtasks
BF_BA	Delay older arrivals/lower level subtasks
LO	Use location-based delay mechanism of d-ALPS (Appendix A)

Table 7-2: Delay Heuristics for Example Simulation

was recorded to give an example performance comparison. Figure 7-5 shows the latencies resulting from the application of the six heuristics. Using three of the six heuristics, the simulator demonstrated that the combination of the allocation and delaying heuristic could adequately process the task graph although with slight differences in average latency. These three heuristics all involved the biasing of delays so that older tasks and/or subtasks were given preference when demand outweighed supply. The two heuristics which biased towards newer tasks caused the system to initially deliver tasks at low latency, but once the "pipe" filled up, the older tasks which had not yet finished were crowded out. The final heuristic is representative of the queue servicing mechanism used by the architecture described in Appendix A. For this graph, it can be considered equivalent to a random delay servicing heuristic since the task graph timings are not highly synchronized. It also failed to provide flow control for the graph executed in this allocation environment.

7.5. Results and Conclusions

The scheduling simulator provides a framework for investigating subtask mapping in a class two allocation environment. The principal parameters to any class two system are the

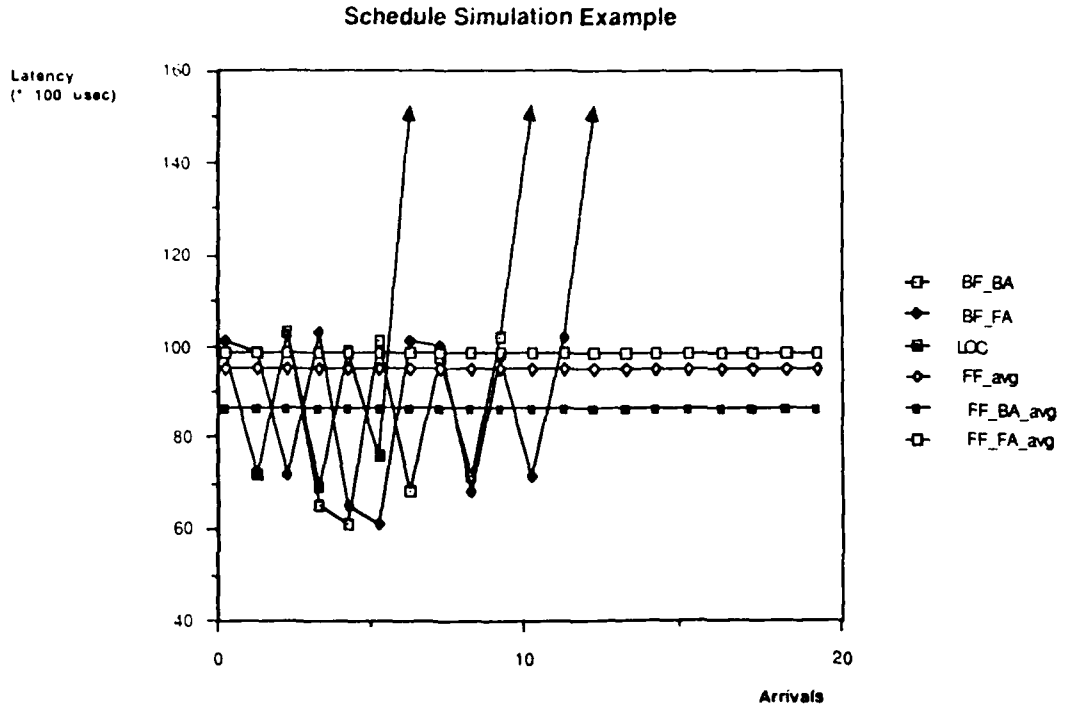


Figure 7-5: Latencies resulting from the application of six delaying heuristics on the task graph of figure 7-4(b).

task graph, the specific graph detailing parameters, the allocation, and the mechanism(s) that the underlying architecture will use to assign subtasks to resources. The simulator provides a step-by-step statement of the assignment without requiring all of the principal parameters to be unified into a single mathematical or state relationship. As the scope of this thesis is to present a range of analysis methodologies which support the investigation of mapping parameters such as delay heuristics, evaluation of these heuristics beyond a broad and preliminary level falls beyond its scope. Some preliminary investigations with this simulator have shown that the low-cost location-based assigning mechanism utilized by the architecture in Appendix A does not regulate the latency allocations as well as age-based heuristics in

highly constrained allocations. Furthermore, heuristics which bias towards newer jobs tend to yield a high variation in latency. It it anticipated that this simulator will be useful in conducting a more thorough study over a broader range of problem graphs and allocations.

CHAPTER 8

Architectural Simulation

This chapter will introduce architectural simulation as methodology for studying allocation and scheduling problems. A simulation framework will be introduced. This framework has been developed to support the d-ALPS architectural specification described in Appendix A. It includes a task and configuration design capture facility; a simulation interface and interaction environment; and instrumented, event-based architecture simulators. The chapter will provide some example simulation studies which demonstrate how simulation can be used by a configurer to decide upon a particular application configuration and by an architect to gain insight into the architecture itself.

An architectural simulation provides detailed timing information about the progress of a task graph when it is executed on a particular system. The system includes the mapping parameters and servicing heuristics that the architecture employs, as well as some model of the *implementation* of those parameters so their costs can be assessed. To narrow the field of this discussion, the d-ALPS architectures in which we are interested can be broadly characterized as a networked or bus-based multiple processor system. The simulation can represent system functions at different levels of detail, and in doing so can vary in accuracy in its representation of the architecture it is modeling. For example, the representation can be at the gate level, the functional block level (processor-memory-switch), or the network node level.

There are a variety of support environments for modeling concurrent systems at the varying representation levels. On the gate level, a set of simulation tools under the Mentor Graphics IDEA System [Ment84] provide support for the design and analysis of digital

circuits. The event driven simulator, SIM, verifies the functionality of a design that has been specified via design editors. The simulator expects that systems be defined to the chip level and provides detailed timing information, taking into account technologies, propagation delays, and logical behavior. While a SIM simulation would most accurately reflect the logical functionings of individual elements of a concurrent system, it would be inappropriate for modeling the behavior of collections of these elements because there are no abstractions for higher level behavior modeling. On the functional block level, a PMS-level simulator, Network II.5 [Garr87], provides a simulation of computer systems that are described by processing elements, data transfer devices, data storage devices, (software control) modules, and files. The event-based simulator activates control modules according to time, message, and semaphore conditions. The control modules then initiate activity among the hardware elements. The simulator is intended to model concurrent systems and reports on higher level events such as utilization and status of network resources and run-time reports of significant events. Simulation at this level appears to be more appropriate to model the behavior of concurrent elements. However, this particular tool is not suitable for distributed and dynamic, message-dependent control systems. The simulators that were developed to support the d-ALPS architecture described in Appendix A fall between these levels. They consider the architecture to be composed of a number of nodes which run an identical protocol. Each of these nodes, in addition, has separate resource attributes and processing capabilities. The simulators model the execution of the common protocol each node participates in to distribute and process subtasks.[1]

Architectural simulation is an intuitively simple and computationally manageable approach to studying the allocation and mapping problems associated with the scheduling of a task graph onto a set of processors. The methods described in previous chapters introduce

[1] Detailed information about these simulators can be found in [Leib86], [McCo87], and [Mano87].

ways in which the task graph and the mapping parameters can provide allocation and mapping information and how basic architectural assumptions can be inserted into the methodology representation. Simulation allows parameters of the underlying architecture to coexist with the task graph and mapping parameters to provide a view of how a workable system would perform under that combination of parameters. These architectural factors include details of task distribution methods that can not be abstracted into higher level representations, timing information, and protocol implementation costs. As mentioned in previous chapters, some of this information can be represented in usable hierarchies, such as state diagrams or expanded directed graphs. But these approaches can not be expanded indefinitely. The modifications and special-cases that must be made to include this information will eventually obscure the representation and reduce the feasibility of analysis techniques which rely on the representation. Furthermore, the complexity of various analysis techniques multiplies with the number of states or nodes that are added to expand the representation.

The type of information that architectural simulation provides depends to a large extent on the level of simulation that is performed, the limitations of the simulation implementation, and the limitations of the simulation environment. The level of simulation, as explained earlier, can be structural or behavioral, and can provide, as its highest level of granularity, information on a gate level to information on a network transaction level. Problem definition plays a key role in deciding upon the level of simulation. A gate level simulator would be useful to a *builder* of a high level architectural specification. To study allocation and mapping problems in the context of a d-ALPS specification, a protocol-level simulation is desirable, provided that reasonable timing costs of those protocol transactions are assessed. The simulator implementation refers to the method by which the simulator is constructed, the capabilities of the underlying simulator support mechanisms, and the costs of implementing

additional instrumentation. The simulation environment can delimit the interaction boundries of the simulator. It includes the types of information that are provided to the simulator, the ways in which the simulator can be controlled, and the extent to which interesting parameters can be defined or specified. It may be difficult or impossible to extract certain information or perform unanticipated monitoring functions if the interaction environment is not generalizable.[2]

Architectural simulation can be employed by both the architect and the configurer to study a particular underlying architecture as it applies to example applications. A configurer takes the underlying architecture as a given. The performance of a configuration on an application is investigated in a specify-investigate-respecify cycle. The configurer specifies the task graph and associated resource demands, creates an initial configuration, simulates that configuration as it operates on the task graph, evaluates its performance, and modifies the configuration according to simulation results. This configuration cycle is facilitated by simulators that provide monitoring functions can be corroborated with the general performance criteria by which the configurer is guided as well as additional information which may ellucidate why a configuration doesn't meet those criteria. Additional information may be resource-specific, such as load on, or contention for, particular processing or communication resources, or it may be resource general, such as queue utilization.

The simulators and simulation environment that have been built to support the d-ALPS architecture project provide the following information transfer. The user enters a task graph and configuration architecture through a design capture and editing facility [Gold86]. The task graph is comprised of a directed graph where each node contains a processing resource. A processing primitive library which maintains a description of the types of primitives avail-

[2] Though it seems unintuitive, it may be more difficult to design methods of specifying measures and communicating that information to a simulator than actually computing those measures.

able to the configurer can be accessed and updated at this point. The architecture is comprised of pools of resources which correspond in type (but not necessarily in number) to the primitives in the task graph. When a simulation session is initiated, the applicaiton task graph, the configuration architecture and the primitive library are downloaded to the simulator. The user then chooses the types of measurements that the simulator should monitor and report. These measures include task latency, resource utilization, and memory utilization. For each of these measures, a *domain* can be chosen. The domain is the category over which the measure is monitored. Categories that are presently supported are system-level measures—inclusive of all system resources—or resource type-level measures; additional domains can include subtask- or arrival-level. The simulation is commenced and the user receives a view of the d-ALPS architecture in terms of the monitoring measures that have been specified.

8.1. Example Configuration Study

The following example demonstrates how a configurer might use simulation to decide upon an application architecture. The example is taken from [Hart86] in which it is used to demonstrate the functioning of a data flow-based signal processor, DFSP. Their architectural approach makes use of a centralized task dispatching mechanism which controls the execution of a bank of execution-independent processing elements. The processing elements perform block processing operations, fetching data from a shared data storage. They are controlled via operation packets and they return result packets to a central activity store when they finish execution. The activity store contains a representation of the directed task graph and performs operand matching, memory management, and activity detection operations.

The example task graph is provided in figure 8-1. This task is called a three sensor problem, as its function is to take inputs from three geographically distributed sensors that

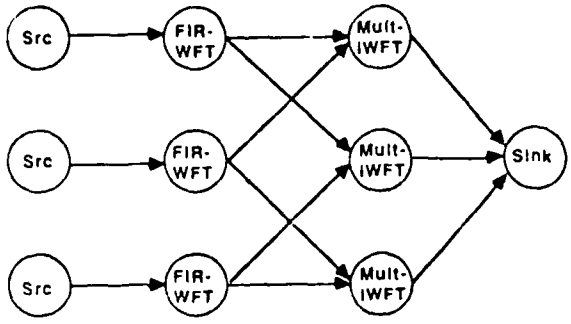


Figure 8-1: A task graph for the three sensor problem.

listen to a signal emanating from a single source. The signals at the sensors should be identical except for additive noise, signal strength, Doppler shift, and a delay proportional to the distance between the signal source and the sensor. A simplification of the problem considers a single transmitted frequency band and looks for the distance differences in the signals to triangulate the location of the source. More details of this application can be found in [Mint81]. The simplified signal processing algorithm is as follows. The signal is initially demodulated and an FIR filter is applied. The Winograd Fourier Transform (WFT) of each signal is then taken. The signals are then cross correlated by multiplying the transformed signals together point-by-point, performing an inverse Fourier transform (iWFT), and searching for the maximum in the lower half of the resulting sequence (peak select). The first task set, the FIR filtering and WFT are performed by a single processing resource. This resource will be denoted: FIR-WFT. The second task set, the multplication, iWFT and peak select, are all performed by a second resource. This resource will be denoted: Mult-iWFT. The resulting value (and index) for each correlated signal is sent to a display processor. Processing requirements, in terms of sampling rates and block sizes, were provided with this exam-

ple. They are listed in table 8-1.

There are several ways of considering a configuration problem. Hartimo et al describe the processing times they used when simulating this example task graph on their architecture. The processing times were based on the number of instructions each of the four macro-operations (filter, WFT, vector multiply, peak select) required times the cycle time of the signal processing primitive (IBM *RSP* research signal processor). The processor bank in the DFSP architecture was comprised of these general purpose processors.

One perspective of the configuration question is to assume that high-speed *special purpose primitives* can be used for each of the above macro-operations. Execution times for these high speed primitives can be determined by consulting the literature. Once execution times have been chosen, the configurer can be provided with a primitive *library* which contains candidate primitives. A configurer would then want to find a d-ALPS configuration architecture which meets the throughput and latency requirements of the task at hand. For the purpose of this exercise, we will consider two configuration scenarios. The first is a resource-constrained scenario and the second is a task requirement-based scenario.

Processing Requirements	
Resource	Requirement
Source1	1K words per 10 ms
Source2	1K words per 10 ms
Source3	0.5K words per 10 ms
FIR-WFT	1K words per 10 ms
Mult-iWFT	1K words 10 ms

Table 8-1: *Processing* Requirements for Example Simulation Study.

The resource-constrained scenario is as follows. Suppose the configurer can build a system with a fixed number and composition of resources. What is the maximum sampling rate (interarrival time) that the system can support? In this case we will assume that two types of fast primitives exist; the first one can perform the demodulation, FIR filter, and WFT, and the second one can perform the complex multiplication, iWFT, and peak select; this breakdown is shown in figure 8-1. For this example, we choose an allocation presented in table 8-2. Note that the execution times of the primitives are relatively fast. Simulation can provide us with the minimum interarrival time (IAT) by starting with a generously large IAT and iteratively increasing it until the system reaches capacity. A configurer uses this simulation by specifying the task graph, choosing these two types of processing primitives for the two subtask-types, configuring a fixed-allocation architecture as described in table 8-2, and providing an initial interarrival time. By measuring system latency and processor utilization, the capacity of the system can be monitored. Table 8-3 shows the effect of decreasing the interarrival time.

Allocation Information		
Type	Processing Time	Allocation
Source	1K every 10 ms	3
FIR-WFT	5 ms	7
iWFT-MULT	6 ms	7
Sink	0	1
Bus	10 Mwords/second	1

Table 8-2: Allocation Information for Example Simulation Study.

Performance Information			
Interarrival Time (μsec)	Latency (avg.) (μsec)	Peak FIR-WFT Use (out of 7)	Peak Mult-iWFT Use (out of 7)
10000	12760	3	3
7000	12500	3	3
5000	12800	6	5
4000	13000	6	6
3500	13700	6	6
3000	16800	6	7
2500	unstable	7	7

Table 8-3: Simulation Results for Example Simulation Study.

Task throughput, the rate at which tasks are processed by the system, can be derived from latency information by noting that latency is recorded as a latency *event*. The time differences between these events indicates the throughput, and should, on average, be equal to the task interarrival time. The system met this throughput when operating at all arrival rates except at the 2500 microsecond interarrival time. From table 8-3 it is evident that as the demand for processing and communication resources increases, the resulting contention adds to the task latency. At an IAT of 2500 microseconds, the peak demand for processing resources reaches 100% for both types of primitives and the throughput of tasks no longer matches the interarrival time. Among these simulations, an IAT of 3000 microseconds is minimum.

A second configuration scenario is to consider as a task requirement a specific interarrival time and to find an allocation that is suitable. For this example, we will assume that the configurer must find an allocation which executes the task graph in figure 5-1 with an interarrival time of 2000 microseconds. The configurer can specify the task graph in the

same manner as above, but will generate an architecture that is resource-rich. Such an architecture can be found by using the static analysis techniques of chapter 5 to find (approximately) the number of resources needed to create a class one execution environment. Alternatively, the configurer can rely on past simulation experience, such as the previous simulation, to decide on a generous allocation. The allocation in table 8-4 will suffice as an initial guess.

Simulation to find the minimum feasible allocation can proceed by examining the load on various system resources as the quantity of those resources is iteratively lowered. Table 8-5 shows the effect of decreasing processor resources. The latency increases slightly until the allocation is insufficient; at that point the system no longer delivers tasks at a steady rate—the throughput does not match the interarrival time—and the latency of those tasks varies eratically.

While the demand for processors remains fixed, the supply decreases; the resulting contention causes the overall latency to rise. When the processor allocation is reduced to 9 of each primitive type, the system can not process tasks to meet throughput requirements.

Allocation Information		
Type	Processing Time	Allocation
Source	1K words every 10 ms	3
FIR-WFT	5 ms	20
iWFT-MULT	6 ms	20
Sink	0	1
Bus	10 Mwords/second	1

Table 8-4: Allocation Information for Example Simulation Study.

The system latency grows slowly as tasks are buffered for longer time periods and eventually the buffering grows out of control. Given these simulations, the minimal resource allocation requirement is 10 of each primitive type.

8.2. Example Architecture Comparison Study

An architect might be interested in simulating tasks which have interesting loads or concurrencies to measure the effectiveness of task servicing facilities of the *target* architecture.[3] Alternatively, an architect may wish to extract overhead and efficiency information by determining the cost of task distribution and management facilities. This overhead can be revealed via simulation comparisons with other architectural approaches or "ideal" approaches. That is, the task overhead can be viewed as the difference in the latency of a task which is implemented on the target architecture and the latency of that task as calculated as the longest path through the task graph. Simulation comparisons can be made by finding tasks which have been used to measure the efficiency or overhead of competing architectures. By simulating the target architecture under the same load and demand

Performance Information				
Allocation of FIR-WFT	Allocation of Mult-iWFT	Latency (avg.) (μsec	FIR-WFT Use (peak)	Mult-iWFT Use (peak)
20	20	11120	9	12
15	15	11170	9	12
10	10	11320	9	10
9	9	unstable	9	9

Table 8-5: Simulation Results for Example Simulation Study.

[3] The target architecture is the underlying architectural specification that the architect is studying.

conditions as a competing architecture, the two architectures can be evaluated. This type of comparison is typically considered *benchmark* comparison. An exhaustive performance comparison is not possible given a few benchmark tasks, as those benchmarks can not cover all application domains and load conditions.

An example benchmark comparison can be made between the DFSP architecture and the d-ALPS architecture. Both archtectures take the same fundamental approach to executing task graphs and incur costs that are within an order of magnitude of eachother. In addition, both architectures assume a microprocessor implementation of the task distribution, memory management, and processor management control sections. The DFSP architecture has a single, centralized control whereas the d-ALPS has a distributed control. Consider the task graph in figure 5-1. Hartimo et al simulate their DFSP architecture on this task graph. They use the execution times and processing rates in table 8-6. These execution times were derived from cycle counts of a single processor implementations of the filtering and transform operations presented in [Mint81]. For the DFSP architecture, there is a single, common data store so there is no need to pass data blocks between processing primitives. The longest execution path is 87.2 ms (milliseconds). The DFSP architecture can service

Allocation Information		
Type	Processing Time	Allocation
Source	1K words every 10 ms	3
FIR-WFT	58.2 ms	20
iWFT-MULT	29 ms	20

Table 8-6: Allocation Information for Example Simulation Study.

tasks at an input rate of one 1K block per 10 ms (100 kHz) with a latency of 95.5 ms; the delay over an ideal implementation is 8.3 ms and is due to system overhead, such as data flow control, resource assignment, control packet transfer, etc.

The ALPS architecture can be simulated under the same load and task graph using a simulator that models the specification in Appendix A. This simulator is called the d-ALPS simulator, so named for the specification it models. This simulator, described in [McCo87], is an event-based simulator that shares the VISA interface and support environment with a PMS-level simulator. The PMS-level simulator was used to develop the d-ALPS specification and to provide a prototype simulator to develop the simulation framework. A description of this simulator can be found in [Leib86]. The two simulators have been cross-verified by applying the same task graphs to both and evaluating the resulting simulations for coherency. The key feature of the d-ALPS simulator is that it estimates the timing required for each stage of the protocol and provides a realistic view of the overhead incurred by a system that is built to this specification. Its monitoring capabilities are, at this point, equivalent to the PMS simulator but can be enhanced because of the finer granularity of the implementation.

The three sensor problem with the primitive timings listed in table 8-6 can be executed in 89.6 ms by a system conforming to the d-ALPS specification. The configuration architecture is shown in figure 5-2. This figure also shows that peak utilization of FIR-WFT primitives was 18 and peak utilization of Mult-iWFT primitives was 10. The overhead due to data transfers, task distribution and resource assignment is about 2.5 ms. For the task graph in figure 8-2, both architectures had to assign six processing primitives every 10 ms. The DFSP architecture did so with an average overhead of 1 ms per subtask. The d-ALPS architecture of Appendix A had an average overhead of 0.5 ms per subtask.

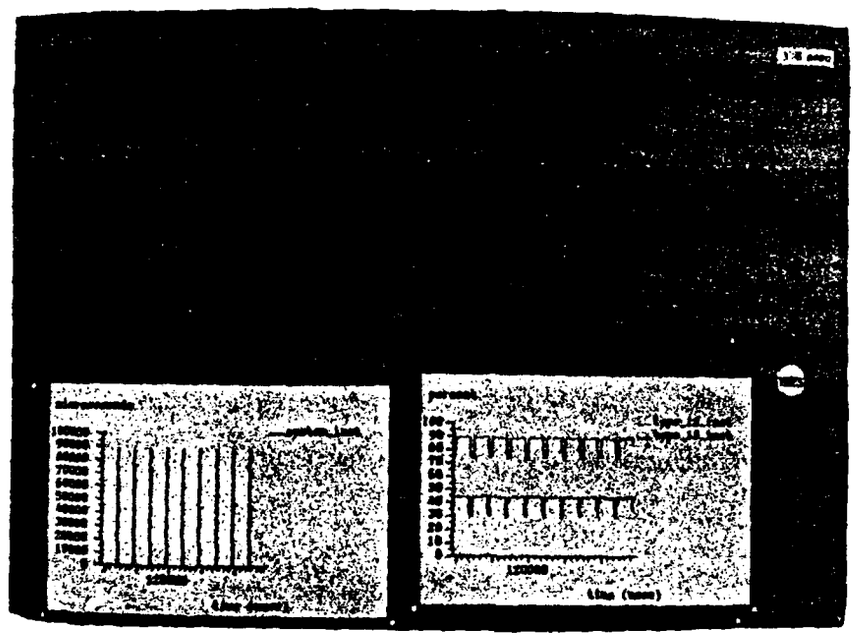


Figure 8-2: An algorithm, architecture and performance results for the three sensor problem.

8.3. Conclusion

The usefulness of architectural simulation is that it verifies the performance of a particular architecture under a specified load. Architectural simulation requires that servicing heuristics be specified and an implementation structure for those heuristics to be created. Determining that implementation structure may not precede a desire to gain information about the particular heuristic. In this case, architectural simulation would be premature.

Developing simulators is a challenging and time-consuming enterprise. Furthermore, instrumenting these simulators, verifying them and integrating them with design transfer sys-

tems adds to the development time. Utilizing existing interface and interaction envriron-ments reduces this time, but the implementation process is inherently lengthy and costly. Above all, the simulation methodology assumes that the investigator has something to simulate, e.g. a specification. The architect, who is interested in *generating* these specifications, can investigate existing specifications via architectural simulation and can prototype newer approaches with less costly, more flexible methodologies.

CHAPTER 9

Conclusion

This thesis described efforts that were directed at supporting the investigation of scheduling and allocation procedures and implementable task distribution and mapping mechanisms. The perspectives of both an *architect* and a *configurer* were considered, as they are the developers and employers of ALPS architectures.

9.1. Research Goals

An architect is interested in the analysis and design of ALPS-like sytsems. This involves assessing the performance of existing ALPS architectures or architecture specifications, such as the d-ALPS specification in Appendix A. New ALPS architectures will most likely require refinements in the fundamental methods they use for distributing tasks, as well as in the implementation mechanisms that support those distribution methods. Analysis methodologies must support the investigation of existing and proposed task distribution methods and mechanisms.

A configurer has a more practical view of scheduling and allocation problems. The mapping mechanisms are assumed fixed or limited for an existing underlying support architecture. For the d-ALPS architecture, there is a single mapping mechanism; for future architectures, there may be alternative mechanisms. The configurer implements an application task graph by amassing a collection of ICU-primitive pairs and plugging them into a rack. In doing so, a set of explicit and implicit performance criteria—definitions of how well the assemblage must work—are observed. The configurer should be able to iteratively select, analyze and verify a configuration; analysis methodologies should support this system integration paradigm.

9.2. Achievements

As an interface between the definition of the scheduling issues and the methodologies that support the investigation of those issues, the parameters of the scheduling problems that are specific to ALPS architectures and ALPS problem specifications were developed. *Task specification* parameters describe how a configurer can represent and manipulate an application task graph. Manipulations which conserve the task graph's trueness to the application (i.e. those which maintain subtask connectivity and precedence) provide alternative static partial schedules. *Static analysis* methodologies provide a route to observing the impact of these alternative partial schedules on resource demand. The methodologies have been implemented as a collection of graph manipulation, expansion and analysis tools.

Architectural parameters describe the ALPS architecture on which a task graph is to be applied. These parameters have configuration-general and configuration-specific portions. The underlying architectural specification, such as the d-ALPS specification describes the basic limitations of the support architecture, such as data transmission rates, task distribution overhead, and memory bounds. The configurer adds to these restrictions by supplying an application-specific allocation of resource pools; this allocation provides the processing potential of the configuration architecture. Static analysis provides the configurer with allocation bounds. Lower bounds describe the minimal processing power needed to service the application, excluding overhead and task distribution inefficiences. Upper bounds provide the configurer with the maximum *effective* processing power (exclusive of system inefficiencies) to handle peak demands for resources—a generous (class one) allocation which would require no buffering of demand. Implications of both general architecture parameters and configuration-specific allocation decisions can be viewed via iterative *architectural simulation*. This provides a performance perspective of the architecture under varying load conditions. Architectural simulators which model both a prototype ALPS

architecture on a *PMS* level and the d-ALPS architecture on a protocol level have been built, verified, and integrated into a design capture and simulation environment. The simulators have facilitated example configuration experiments and have served as platforms for architecture comparison and evaluation studies.

While simulators have been influential in the design and analysis of ALPS architectures, simulator development is expensive and time-consuming. In addition, the underlying simulator model is often difficult to change. Investigating large variations on the architecture, especially those on the *task mapping level* has been facilitated by *schedule simulation* techniques. Task mapping parameters describes how a task graph which encodes a partially static ordering of subtasks is mapped (dynamically or statically) onto processors via heuristic-guided demand buffering procedures. Schedule simulation has provided a route to the investigation and comparison of these mapping parameters by allowing a course-grained model of the architecture and task graph to coexist with a framework for simulating the cost-free dispatching and buffering of subtasks under the guidance of user-supplied heuristics. An approximated representation of basic architectural mapping mechanisms can be added to this framework. For example, the effect of *committing links* on resource binding has been added to the model. As this first pass approximation can be difficult to refine, architectural simulation seems to be more appropriate for analysis of the details of the implementation. The preliminary results of the application of schedule simulation are twofold. Its design and development provided insight and impetus to consider additional alternative task mapping procedures. In addition, analysis has suggested that task servicing policies which consider graph depth and arrival of subtasks tend to cause tasks to execute with more stable—though higher—latencies.

9.3. Current Work and Future Directions

A prototype d-ALPS implementation which is based on the d-ALPS specification is being constructed at Brown University under the direction of Professor Dick Bulterman. In addition, ALPS hardware and software *emulators* are being constructed. The hardware emulator is an implementation of the d-ALPS logical control protocol on general purpose processor boards. It is being constructed at the Naval Research Lab under the direction of Y. S. Wu. The software emulator "nodes" are UNIX[1] processes which emulate the d-ALPS logical control protocol and use network facilities (TCP/IP) to communicate to each other. It is based on a single node model written by the author and is being implemented on a collection of workstations at Brown University. The emulators and prototype architecture can, along with the architectural simulator and simulation environment, provide reasonably instrumentable testbeds for analysis of and modifications to the high level control protocol.

Research and development of essential task distribution functions can be facilitated by an expansion and integration of the support tools described in this thesis. Two of these methodologies, static analysis and schedule simulation should be refined to facilitate iterative investigations of the interaction between specific task definition parameters and mapping parameters. Future work on application independent heuristics that have been developed should focus on analyzing the feasibility (and cost) of distributed implementations.

Two directions for expansions of the analysis methodologies are proposed. The first is a clarification and codification of the implicit performance criteria, such as reliability and stability. These criteria are central justifications for the ALPS dynamic assignment approach over more static approaches. Task distribution mechanisms that is not essentially random have the potential of addressing some of the recoverability and reconfigurability issues that are fundamental design criteria; those mechanisms will also be more sensitive to system

[1] UNIX is a trademark of AT&T/Bell Laboratories

state. The second direction is the inclusion of more stochastic models of resource availability and resource requirements. As a placeholder to this dimension, resource requirements and essential system timings can be modeled as distributions around estimations. This simplification should be viewed as a prelude to more statistical modeling approaches that can be more representative of the operation of the underlying architecture.

APPENDIX A

d-ALPS High Level Logical Control Overview

Following are the first three chapters of the *distributed-ALPS Initial Design Specification* [Netw87]. These chapters comprise an introduction to the distributed ALPS approach and the specification for the high level logical control functions of the interface control unit (ICU). The first two sections of this appendix provide an overview of both the d-ALPS protocol model and the ICU structure. These sections were written by Prof. Dick C. A. Bulterman. The last section provides a description of the high level ICU functions and structure. This section was written by D. Leibholz. The *Initial Design Specification* contains additional chapters which describe in detail the ICU logical control memory elements, the detailed inter-ICU protocol transactions and functions and structures that are grouped under "ICU Low Level Control." Low level control functions and structures support data transmission, physical memory management and format conversion. These chapters were not supplied in this appendix because they do not contribute to an understanding of the scheduling and allocation issues presented in this thesis. The reader is referred to the d-ALPS High Level Logical Control specification in [Leib87] and to the entire d-ALPS ICU Specification in [Netw87].

10.1. The distributed-ALPS Protocol Model

10.1.1. Architectural Characteristics

The deterministic nature of most DSP applications has made them suited to implementation on several classes of conventional architectures: these include fast uniprocessor architectures, in which each DSP function is encoded as a software or firmware routine, with the processor's operating system providing a fast switching service [1]; a dedicated hardware architecture, in which each DSP function is implemented as a hardware circuit, with physical communication paths providing a system switching service [2]; and a network of (relatively) general-purpose processors and global memories, in which DSP functions are encoded as software and firmware routines, bound to one of several processors based on the needs of an application, and in which data transferred between processors is buffered in high-speed global queues [3].

d-ALPS uses a fundamentally different design approach. Instead of preallocating processing components of an algorithm to a particular processor, processing is supported by pools of special-purpose processors that are interconnected over one or more high-speed (40MByte) interconnection networks. When a particular task needs to be accomplished, a hardware bidding scheme is used to select a candidate processor from the pool to service the needs of a particular logical (algorithmic) process. d-ALPS, which is based on the Alternative Low-level Primitive Structures (ALPS) framework [4,5], is a fully distributed system that provides the advantages of a special-purpose hardware structure while providing improved reliability and reusability over dedicated hardware and prescheduled networks. It has the following characteristics:

- Interactions between processing components are not prescheduled;
- The structure of the interconnection network, while biased toward a class of applica-

tions, is not based on a single application's structure;

- There is some inherent reliability advantage over non-distributed approaches;
- The implementation is able to gracefully accommodate changing needs, albeit with some modification to the particular configuration of the implementation; and
- The performance of the resulting implementation is comparable to that of a special-purpose implementation (that is, the performance may not be significantly better, but it should not be significantly worse), assuming the same technology is used in both cases.

d-ALPS is intended to provide the designer of DSP applications with some of the potential advantages of distributed architectures. First, by having component interactions that are not prescheduled, changing configurations of a system can be accommodated more easily than in a fixed interaction configuration. Second, by having a generalized implementation structure, the system becomes reusable, both for a changing version of a single application, and across applications of the same class. Third, by providing a recovery scheme for failed components or busses, the designer is given an improvement in inherent reliability, allowing the implementation to be more robust (for the same designer effort) than special-purpose applications, although at the cost of a sub-optimal implementation.[1] Fourth, graceful upgrades make a single system extensible, so that changes in an algorithm may be integrated easily (although perhaps not trivially) into a particular implementation.

d-ALPS uses a hardware-controlled distributed bidding mechanism, where processing elements compete for the ability to serve a particular processing request. While a software-controlled approach was considered, the reality of many software bidding schemes is that their implementation often provides an impediment to good performance [6]; since

[1] Although a distributed implementation is sub-optimal because of scheduling overhead, we have found that performance need not suffer significantly. Discounting the advantage of primitive-based technological improvements, we have found that the simulated performance of d-ALPS compares quite well with other implementation techniques. This is considered in section 7.

performance is crucial in our application domain, we decided to develop a hardware control mechanism that was efficient (to allow the resolution of a particular bid in approximately 450 nanoseconds) while keeping the system structure flexible enough to accommodate a wide range of DSP applications.

10.1.2. The ALPS System Model

The Alternative Low-level Primitive Structures (ALPS) model represents one way of taking a graph-based specification of a DSP algorithm and describing a system-level interconnection structure than can be used as the basis for an implementation. The ALPS model was developed to strike a compromise between the dedicated hardware and (reusable) software network implementation approaches for DSP systems. In the ALPS model, a shared hardware primitive structure is used to approach the performance of the dedicated hardware model by capitalizing on the processing node and communication-path idle time inherent in most DSP applications. Unlike the software network model, which also attempts to provide a shared approach to system implementation, ALPS uses a self-scheduling primitive assignment scheme that eliminates the need for a (software) scheduling algorithm. This implies that expense of programming (and verifying) the scheduling the interactions of processing components can be avoided, and that "systems design" more closely models a hardware building block assembly process.

A generic ALPS network model is given in Figure 10-1. The network is composed of hardware primitives that are connected to each other via three "circuses":[2] a *data circus*, which is used for high-speed transfer of data from one source primitive to one or more destination primitives; a *control circus*, which is used to arbitrate the primitive's access to the data circus—providing a convenient focus for scheduling activity in the network; and a

[2] The term *circus* is used in the sense of an English roadway roundabout (traffic circle), and is not necessarily meant to describe either a physical interconnection structure or a (chaotic) logical access protocol.

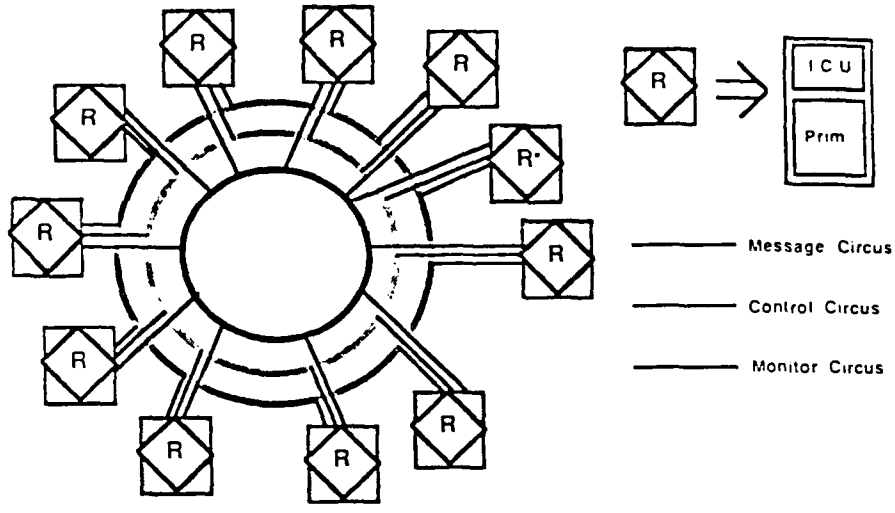


Figure 10-1: An ALPS Network Model

monitoring circus, which is used to monitor the status of the system (including reliability checks). Each hardware primitive is connected to the ALPS network by a standardized Interface Control Unit (ICU). The combination of the ICU and the primitive is called a *resource*. The ICU manages the interaction with the signals that travel through the circuses, including directing the flow of information into and out of a either the ICU's locoal memory, or local memory of the primitive (if it exists). The hardware primitive portion of the resource may consist of the implementation of a single logical primitive (such as a VHSIC FFT processor), a collection of related primitives (such as a "vector" primitive that performs vector addition, subtraction, multiplication and division), or a programmable processor (such as a high-speed digital signal processing microprocessor in the class of the TI TMS-320).

The interconnection graph of Figure 10-1 represents a single ALPS *cluster*. The model allows for an expansion to multiple clusters when a single cluster of circuses and resources

can not support the bandwidth required by the application. Each such cluster may contain all or some of the resources types defined in the application graphs. Clusters communicate with each other via *inter-cluster controllers*. Multiple clusters may also be used to provide additional reliability in the system. Beyond noting their existence, however, the ALPS model does not define any partitioning or allocation rules for separate clusters. For example, each cluster may have one or more instances of a particular resource type, with the number of resources used being a function of the original flow graph and the characteristics of the implementation environment and requirements.

In the ALPS model, resource access is spread across a set of special-purpose primitives. This allows single copies of a resource to be shared by various logical primitives of the same type in the initial system graph. The process of developing an ALPS-based implementation model for a particular algorithm consists of balancing the utilization of a shared communication facility with the utilization of individual system resources. If the ratio of communication-time to processing-time is low, then a relatively low-bandwidth communication facility can be used to interconnect relatively many resources. If the ratio is high, then a higher-bandwidth communication facility is required, or several disjoint clusters may need to be defined.

10.1.3. ALPS Implementation Concerns

The basic structure . the ALPS model allows it to be implemented as a (static) network or a (dynamic) distributed computing system. For a given DSP application, a static implementation would pre-allocate the required number of processing nodes of each primitive type based on an analysis of circus contention. This approach would be similar to that reported in [7,8] and structurally similar to that of the processor network, with the exception that scheduling control does not come from a control program but from the interactions of

the resources. A distributed implementation would function in a similar way, with one important exception: the distributed implementation, with dynamic rather than static resource associations, could permit spare resources to be placed on the network that could be used to transparently take the place of some failed nodes during system execution. Unlike the static model, this substitution would not need to be preplanned, but could be done on a demand basis without altering the function of the bidding scheme. Note that not all failure modes of the distributed system could be accounted for in this manner—only the isolated failure of one or more resources. Any further fault-tolerance must be explicitly encoded in a high-level protocol.

d-ALPS uses a dynamic scheduling protocol, which can be summarized as follows: just before a node finishes processing the data it had received earlier from its input queue(s), it sends a message on the control circus inviting all resources of an appropriate type to bid for being the designated recipient of the source node's data. The first resource responding to the bid (i.e., the first resource of the appropriate type that responds to the allocation request) becomes the target instance for the corresponding graph node. If no resource responds, the data is queued (either locally or globally, depending on the implementation) until a target resource is ready.

Although the bidding scheme allows any node of an appropriate type to act as the instance that carries out a logical graph function, this can create two potential problems: First, since a DSP application graph has nodes that are able to accept multiple input arcs (i.e., connections from multiple other nodes), the d-ALPS implementation must be able to synchronize requests initiated in various nodes to insure that they are all serviced by a single target resource. For example, if two resources (representing different sending nodes) each separately send out bids for a common target node, then the allocation of that target node needs to insure that a single receiving node will process both requests. Second, there is no

inherent frame synchronization mechanism in the ALPS structure: an output data frame is sent from one resource to the next as soon as a target resource can be found (and as soon as the data circus is available). Any delayed frames are queued for later transmission, but there is no guarantee on the maximum delay that can be incurred. This means that there is no inherent mechanism to insure that all data is sent in order, since a subsequent frame may be processed and ready before the current frame has a chance to be processed.

In order to address the first problem, the design environment that is used to encode the graph-based representations of the algorithm requires the designer to define a *priority link*; this link, which can be any one of the input arcs into a node, determines the actual committal of a physical node. If non-priority links make a request for a resource's services, they are masked until the priority link makes its request. As will be described, each scheduling message carries a desired-node type, the requester's logical ID in the graph, and a frame identifier; this allows queued requests to be resolved to their proper target nodes. The second problem is addressed by providing an integrated system simulation environment that allows the user to determine the amount of queueing of requests in the network, and to determine if enough computational and communication resources exist to accommodate the demand. The design and simulation environment is described in the collections of articles placed under the heading "Design Environment" in the References and Bibliography section of this report.

10.1.4. A Distributed ALPS Interconnection Network

The d-ALPS implementation model consists of a collection of one or more clusters, where each cluster is responsible for executing either all or part of the DSP application graph. Clusters may be interconnected in a number of ways to allow them to be used as components of the system functional pipeline, or the may be connected so that they execute

functions in parallel. The partitioning of an algorithm among clusters is highly algorithm- and application-requirements dependent. There are a number of general partitioning considerations, however, that apply to all applications. In practice, the connections between clusters require special purpose *gateway* resources. In addition, the achievable data rates between clusters is typically an order of magnitude slower than the data rates between resources within a cluster. Therefore, it is desirable to minimize the amount of inter-cluster communication in order to prevent communication bottlenecks and to minimize the overhead costs of having multiple clusters. (The overhead results from the required gateways and the interconnections between them.) In general, maximum system utilization is achieved by minimizing the number of clusters. Note that the amount of processing that is possible within a cluster is limited by the resources allocated to the cluster (although this is configurable) and the bandwidth of the intra-cluster network. Inter-cluster design is also affected by the reliability requirements of the system. Multiple clusters and/or multiple communication paths between clusters can be defined in order to provide redundancy in case a cluster becomes either disconnected or fails internally.

In this presentation, we will consider only single-cluster implementation network models. Figure 10-2 shows the components of a single cluster. Since a cluster may need to support a large number of resources (on the order of 200), the cluster structure is segmented into a collection of subbusses. Each subbus is connected to a subbus backbone via a repeater stage. Again, this segmentation is solely required for electrical constraints and may be omitted if the number of resources is small. That is, resources or gateways may either attach to a subbus or directly to the subbus backbone. Attached to each subbus is a subgroup of the resources used in the cluster. There is no significance (other than for reliability) to where a resource should be placed within the cluster or within a subbus.

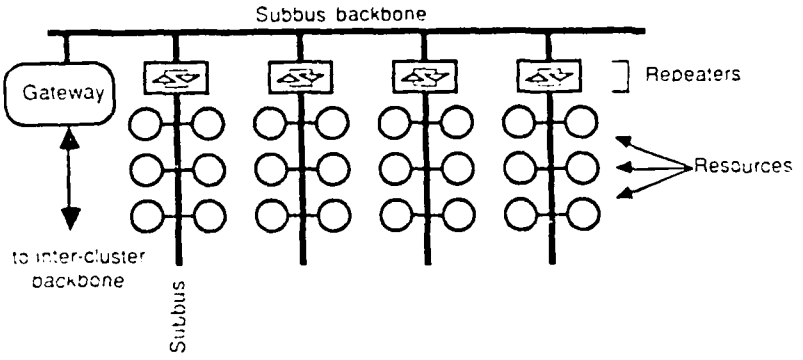


Figure 10-2: A cluster and its components.

10.1.4.1. *Intra-Cluster Network Design*

The fundamental limit in the performance of ALPS-based implementation models is the time used to schedule and implement the transferring of information among resources. This means than the intra-cluster interconnection network must efficiently support two tasks: the transfer of control among resources and the transfer of data among instanced resources. Control information is used to support the dynamic assignment of logical primitives to resources; this assignment is made when one instanced resource needs to send data to another resource in the cluster. Control delay is a combination of both the ICU decision time as well as the time to transfer the required messages between resources in the network. It is important to be able to minimize the control overhead since that time represents resource idle time and therefore lower resource utilizations.

The data transfer task of the network is used to implement the movement of data across the logical algorithmic links between logical primitives, thus defining logical channels between resources. As is typical with loosely coupled systems, the data transfer between processors can easily become a system bottleneck. A real-time task for a particular algorithm requires that a specific amount of processing, and therefore a specific amount of data

transfers, must be accomplished within each specified time period. Since more processing power can be added by simply adding resources, the ability of the network to transfer the data in the required time becomes the limiting factor in the model.

The topology that defines both the control and data paths within a cluster impacts the effective communication bandwidth, the number of resources (and therefore processing power), and the reliability of the cluster. Unfortunately, each of these aspects is best served by a different topological structure. Our distributed implementation uses a single control bus architecture and a multiple data bus architecture. All data transfers are performed using DMA-style burst transfers between the queues of a source resource and one or more destination resources. The transactions that occur on the control bus establish virtual circuits that last for one data frame (one block transfer). Once the virtual circuits are established the data busses can be used as simple DMA channels.

10.1.4.2. The Interface Control Unit (ICU)

Figure 10-3 shows the general block structure of an ICU and its connections to the network and its primitive.[3] The ICU is based around the *message control unit* and the *data control unit*. The message control unit provides the central control for all of the ICU operations. The data control unit is a slave to the ICU, although it does execute its functions in parallel.

Media access is implemented by allowing a single ICU to be in control of the message bus at a time. The arbitration of control is accomplished by using a (fast) token passing scheme. The ICU is primarily responsible for passing control from one ICU to another and coordinating the bidding for receivers for the data on the output queues. The decisions of

[3] Rectangles with rounded corners represent control units and rectangles with square corners represent memory units. (Note that connections between blocks primarily represent data paths and control connections are generally implied.)

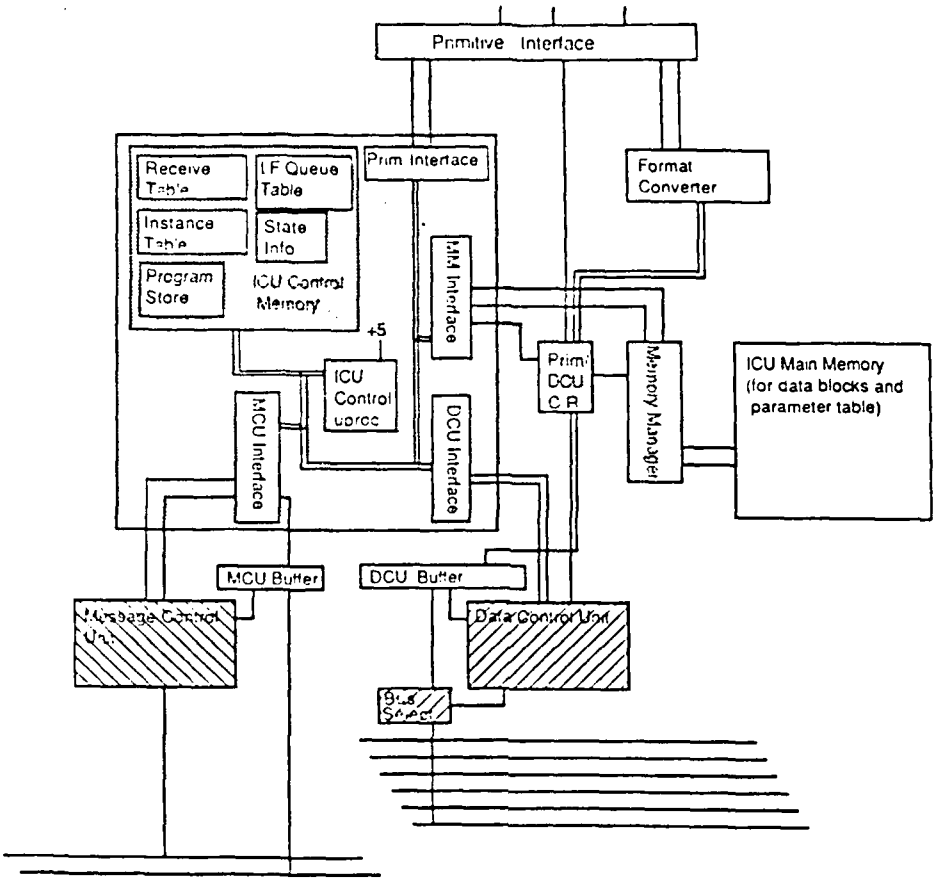


Figure 10-3: The ICU block diagram.

how to create requests for links and whether to accept requests is based on information contained in the *transmit* and *receive* tables, respectively. In general terms, these tables contain a complete description of the connectivity of the algorithm which is used to determine the dynamic connectivity of the implementation. The *parameter table* provides information specific to a particular instance (algorithmic node) for the execution of the primitive. The three tables are generally down-loaded at system startup time. It is possible, however, to send update messages to any of these tables during run-time—allowing, for example for an operator to control application parameters that may need to change during the operation of the system. The use of lookup-tables makes it possible to quickly perform the dynamic

mapping of instances onto resources. For example, using the receive table, an ICU can decide whether to accept a link in a single control cycle. Similarly, an ICU can generate a message to request bids on a link in a single cycle.

The physical primitive acts as a slave to the ICU. It begins execution after the ICU logical control has determined that all of the required input data has been acquired and is stored in the local queues. When the primitive has completed its computation and has placed the output data back in the queues, it then signals the message control to transmit the data to the next logical node in the algorithm. The primitive interface to the ICU is through the shared memory of the parameter table and the queue storage unit. The shared memory interface makes it relatively simple to incorporate new primitives so that resources may follow technology advances. The shared memory interface also provides a means to load instance-specific parameters from the parameter table. Further, the primitive can use the queue storage as if it were local memory since there will be no contention from the ICU during primitive execution. Alternatively, the primitive may copy and work with the data within memory that is more tightly coupled to the primitive. The primitive would then need to write its resultant data back into the queue storage after completing its processing.

The data control unit is responsible for both transmitting and receiving data from the network at the request of the ICU logical control. The data control unit performs block transfers of a particular block in the queue storage to other DCUs. In versions of the ICU that use multiple data busses, the ICU uses a bus number that is transferred at the end of a message transaction to determine which bus to listen to for the input data that it has agreed to process. A transmitting ICU can choose a data bus by examining the data bus busy signals and selecting the first free bus it finds. (Note that single-bus systems do not need to spend the extra control time to select and decode busses in use, at the cost of reduced system bandwidth.)

As data passes between the primitive (or parameter table in some cases) and the queue storage, its format may optionally be converted to match the format of the primitive. These conversions include both precision and format. The format converter serves two purposes: enabling heterogeneous primitives to work together in the distributed system, and compacting data to minimize the amount of data transferred. In general, the format converter will be used to transmit data in the most compact form without loosing precision. For example, an IIR (infinite impulse response) filter may internally perform its computations in double precision floating point format, and then transmit its output link using a single precision fixed point format to a display. By placing the format converter between the primitive and the queue rather than between the queue and the network, the data compaction can also be used to minimize the required queue size. Another reason not to place the format convert between the queue and the network is that the period of some conversions will be greater than the period of the word transfers. Placing the converter on the primitive side isolates the network from the conversion time fluctuations.

Whenever the queue storage is addressed (by the data control or the primitive), all addresses must be mapped using the *address manager*. This address mapping is used to reorder data within a block as well as to locate individual queues within the bulk of queue storage. One example of the use of address mapping could be to reorder the input sequence to an FFT operation. Arbitrary address mappings can be performed by using an indirection mapping table that is particular for each primitive type.

The ICU must support three basic functions: it must receive data from other resources, it must control the processing of its attached primitive, and it must transmit the results of that processing to other resources within the system. The initial state and the idle state for the ICU is the receive state. The ICU logical control unit is used to implement most flow of the finite state machine.

For standard resource types, the receive state machine constantly reads the message bus and decides whether it should accept the current link and frame number pair[4] that was most recently transmitted. When the ICU logical control unit decides to accept a link it uses a fast polling protocol described below to bid to accept the link. Once all of the required input links for the firing of an instance have been read, the receive state machine signals the primitive to start execution according to the parameters that are currently being addressed in the parameter table and using the data found in the queue storage unit.

When the primitive has completed its execution and the output data is left in the data queues, it signals the message control unit; this unit first attempts to acquire control of the message bus using the round-robin control passing protocol described in the section below. Upon obtaining control, the message control transmits messages for each output link for the current logical output port and listens for acknowledgements to the request. If a link is not accepted by any receiver, then that link is added to the list of locally queued links pending transmission. The use of a local queue means that global access to a single system-wide queue is avoided, saving transmission cycles on an already busy bus. Once all links for the current output port have been attempted, a free data bus is chosen. The selected bus number is then transmitted on the message bus and then the data control unit is signaled to carry out the block transfer. Once the transfer has been started, control may be passed to another ICU.

[4] An ICU presents a request for bids to accept a link by transmitting the link number and a frame number. The link number uniquely identifies with which link in the algorithm the data is associated. A receiving ICU uses the link number to index its receive table and use the output information to in part determine whether the link should be accepted. The frame number is used to match epochs of data. An instance that requires multiple inputs will only accept links whose corresponding frame numbers match the frame number of the first link accepted for that instance. In general, frame numbers are assigned at sensor nodes when frames enter the system.

10.1.5. Detailed Descriptions: An Overview

Detailed descriptions of the various functions of each component of the ICU and a general ICU input format is presented in chapters 4-11 of this report. The descriptions are divided into three logical groups of ICU functions: High-Level control, describing the logical ICU protocol and its support implementation (these are considered in chapters 4-6); Low-Level control, describing the bus structures and the message and data control sections that directly use these structures (these are considered in chapters 7-9); and Auxiliary control, describing the memory manager and format converter functions (these are considered in chapters 10 and 11).

10.2. Interface Control Unit Structure

10.2.1. Overview

In this section we provide an overview of the structure of the ICU. This structure will serve as a "road map" through the remainder of this specification.

Each of the headings below represent an ICU logical or physical block. Some of these blocks will be implemented as concurrent hardware processor blocks, while others will be (initially) implemented as software modules in a centralized control processor. An indication of the proposed implementation strategy is given with each heading; justifications for each decision is given in the complete description, below.

10.2.2. Inter-ICU Bus Structures

This specification restricts itself to a single cluster of ICUs, each of which is attached to a single processing primitive. The ICUs are connected to each other by two classes of interconnection busses: the *message bus* and the *data bus*.

10.2.2.1. *Data Busses*

There may be up to four data busses specified in the d-ALPS architecture. Each data bus has a 16-bit parallel data path, plus control lines. The control lines are used for clocking (although the bus can also be used in a non-clocked mode) and error control. The data rate of the bus is specified as sending one 16-bit word between any ICU pair within a cluster in 125 nanoseconds.

10.2.2.2. Message Bus

The message bus is used to support two type of activity in the d-ALPS architecture: logical control synchronization among ICU (through a hardware bidding protocol) and status/initialization processing. The message bus supports asynchronous transfers and has an 8-bit wide data path.

10.2.3. Low-Level Transfer Control

Access to both the data and message busses is controlled by the *low-level transfer control* portion of the ICU. This control is divided in to two separate, concurrent units: the *data control unit* and the *message control unit.* The current protocol makes extensive use of the notion that these units are implemented as hardware-based blocks. The bidding protocol used to assign control among ICUs, for example, will only be successful if the classical delay associated with microprocessor-based implementations can be avoided.

10.2.3.1. Data Control

The data control unit implements the logic to control data transfers between ICU pairs. Once the source and destination of a transfer are established, the data control unit provides for the error-free transfer of data in a DMA-style block transfer mode. While single-word blocks can be sent, the data control unit is structured to provide efficient services for larger-

sized blocks (typically in the range of 1K-10K words). The data control unit also maintains status information that is used by the testbed to instrument the d-ALPS implementation.

10.2.3.2. Message Control

The message control unit transfers four types of messages between ICUs within a cluster: initialization messages, status messages, network control requests, and algorithmic node bindings. The description of the uses of these is beyond the scope of this chapter; it is addressed more fully in chapters 7 and 9.

10.2.4. High-Level Transfer Control

The ICU's high-level transfer control is a logical control service that manages the response of the ICU to messages and data transfers on the network. It is at this layer that the essence of the d-ALPS protocol is based. Note that this control *does not* include the bid resolution process, which is a low-level control function.

10.2.4.1. ICU Logical Control

The logical control block manages all data transfers within a particular ICU. It controls access to the memory server on the ICU, notifies the format converter of any pending conversions, and notifies the primitive when all of the input queues for that primitive are already. The logical control portion of the ICU is implemented using a conventional microprocessor architecture; it is described in chapters 4, 5, and 6.

10.2.5. Data Queue Management

In order to provide a common block and word buffering facility for both the data control portion of the ICU and the primitive itself, the ICU also contains a memory management unit. The purpose of this unit is to manage the allocation of data queues, and to manage the access to those queues by the primitive and the data control portion. The memory

management unit is implemented as a block of 100 nanosecond memory, and a control logic block that manages contention for the memory. The memory manager is discussed in chapter 10.

10.2.6. Format Converter

The final block of the ICU is the format converter. The format converter is an optional component that can facilitate the conversion of several data types, such as integer to floating point, or two's complement integer data to unsigned data. The purpose of the format converter is to both offload the processing that a particular primitive may have to perform, and to make a test-bed system more general, by allowing primitives of several different types to be supported. The format converter is described in chapter 11.

10.3. ICU High-Level Control Architecture

10.3.1. Basic Structure and Architectural Decisions

The ICU Logical Control (hereafter called simply ICU Control) is responsible for monitoring and controlling the support hardware facilities so that the ALPS communications protocol is implemented and the internals of the ICU operate and interact in an organized manner. To these ends, the ICU Control is considered an autonomous, single point of control which has available to it special purpose facilities which, once given commands, operate concurrently and share resources in an agreed upon and controlled manner. Each of the support facilities, the message control unit (MCU), data control unit (DCU), format conversion unit (FCU) and memory manager (MM) provide control interfaces to ICU Control which are essentially command-oriented. The ICU Control issues one of a set of commands to the units and some simple handshaking and status lines indicate the completion and status of the tasks. The data and message control units have buffer interfaces for the exchange of infor-

mation with other ICU Controllers on the network. The ICU Control, as a central controller, operates sequentially and has primary access to node status and state information. The justification for a sequential controller of concurrent operating components follows by examining the basic implementation alternatives in terms of the sometimes competing interests of instrumentability, scalability and accuracy of the model.

The structure of the ICU Control unit is illustrated in Figure 10-4.

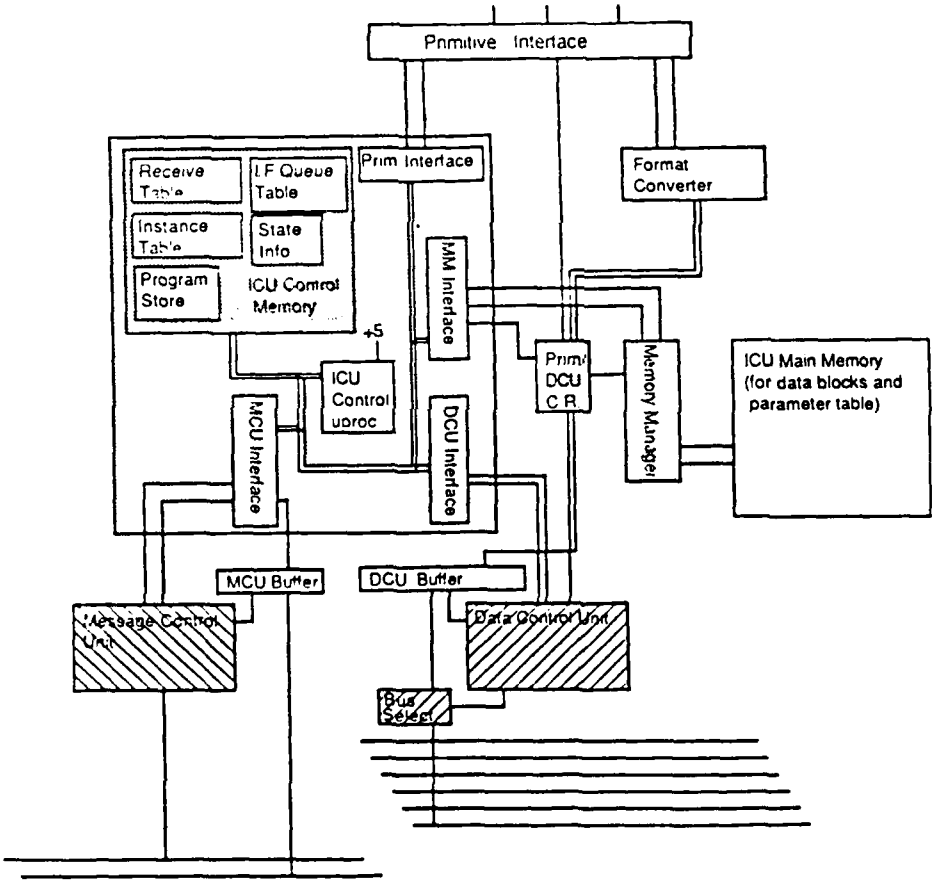


Figure 10-4: ICU Logical Control Structure

While the implementation of a centrally-based control unit presents obvious bottlenecks in speed and complexity, many of the decisions that are made by this unit are inherently sequential in nature. The information that is manipulated by the ICU Control is gathered from locally distributed sources; delays in obtaining current status information is minimal. The principal advantage of using a centralized controller is that it facilitates the development of a d-ALPS testbed, where experimentation of different high-level protocols and instrumentation of ICU activities are highly desirable.

Note that although justification exists for having a central ICU Control processor, this does not extend to the entire ICU. In other words, while the ICU Control will operate sequentially, the support functions of the ICU and the primitive must operate concurrently. A single, serial ICU controller implementation is insufficient to accurately model the current d-ALPS protocol, since it would incur significant operational delays that would not occur in a generally-concurrent ICU. Furthermore, a fully serial implementation (i.e., one that is based upon a microprocessor which fully controls activity on message and data busses and which implements low-level memory management) would not be a scalable one, in that it would not resemble the physical structure of the actual ICU operations. The design decisions involved in a microprocessor-based ICU would not reflect similar decisions that must be made for an operational system based upon concurrent functioning ICU components. Designing this type of system would then be in isolation of the problems of contention and control faced when designing the *next* system.

Our current plans for the control portion of the ICU call for a microprocessor with a small amount of local memory. The processor may access and monitor a set of support facilities via memory-mapped I/O interfaces to finite-state machines (FSMs) which directly monitor and control the above mentioned support facilities. With this approach, changes to the essential protocol and the status and initialization messages can occur without affecting

the interfaces to the underlying support facilities. Such changes would consist of a reprogrammed Logic Control microprocessor. As the protocol evolves and the instrumentation requirements recede, the central control microprocessor can itself become a simple finite state machine, issuing simple but powerful commands to interface FSMs. The state information is then distributed and copied to the local FSMs and integration of this information is minimized to operational necessity. The advantage of this approach is that the parallelism in the protocol decision-making points can be enhanced and concurrency of operations, once it is identified in a final protocol, can be facilitated.

10.3.2. A Description of Essential ICU Control Functions

The implementation of a d-ALPS-based protocol relies on the passing of data blocks between heterogeneous special-purpose primitives in such a way that a signal processing algorithm is executed. To this end, the current protocol uses a dynamic scheduling and assignment of these primitives in that a primitive is scheduled to operate on a particular epoch of data as the data becomes available, and the physical primitive that is chosen is not predetermined. The central justification for the dynamic scheduling approach is that it avoids the need for a central controller and avoids the synchronization problems that occur when unexpected delays or dead nodes occur. Dynamic assignment offers fault resilience by allowing any capable node to accept data instead of relying on a predetermined and possibly dead node.

The requirements that these concepts place on an interface control unit in a distributed-control environment are support for the ablility to assume the role of any algorithmic instance that requires the node's attached primitive and the ability to stage a bid for the next receiver of data that has been processed. These are the essential protocol-related functions of the ICU. Unfortunately, while these functions are conceptually clean, their

implementation must be based around a system in which lower-level concerns such as memory management, higher-level support for data queues, and general initialization and monitoring functions must also be supported. The support functions can be divided into two basic groups: *protocol-based support functions* and *extra-protocol support functions.* Protocol-based support functions of the ICU Control provide the basic implementation of the protocol's functionality. Extra-protocol functions are support operations that facilitate the initialization, modification, and monitoring of a functioning system.

10.3.2.1. A Overview of Protocol-Based Support Functions

The principal functions of the ICU are the sending and receiving of messages and data to assign nodes as algorithmic instances. The operation of an ALPS system will typically involve passing as much data through the system as can be handled. It is expected that in the normal case, data will be pipelined -- data blocks generated at different times will be processed by the system. The interarrival time of data into the system is guided by the application domain, and the performance requirements of the actual application architecture may be specified in terms of *system latency* and *system throughput.* System latency is the time a frame of non-processed data enters the network at a source subtracted from the time the processed data block reaches a network data sink. System throughput is the number of frames per second that enter and exit the network. In general, the latency will be significantly larger than the interarrival time, but the throughput will will generally match the interarrival time. The impact of these operational characteristics on ICU function is that any particular ICU/Primitive pair, or *node*, must be classified not only by which instance it is currently assuming but which time-slice, or frame, it is currently processing. A node then operates on an instance/frame, a time and algorithm based description of a subtask performed by an individual node. The ICUs must negotiate the fate of multiple frames of data, seeing to it that the signal processing algorithm is performed over pipelined data.

A second operational characteristic of this protocol is that data that has been partially processed must be queued while resources are unavailable. Since an allocation of processors may not and, in general should not be overly generous in that processors are always available to bind to instances, data should be stored in an organized manner and actually sent over the network only when a receiver is found. Furthermore, the limited data channel bandwidth poses a communication bottleneck which results in the temporary buffering of data while other transfers are in progress. The effect of the queueing and buffering of data is that both communications channels and processing sites should be considered system resources. The are two cases of data buffering: *transient buffering* and *recurrent buffering*. Transient buffering results from processing errors, data transfer errors or node losses (loss of synchronization); recurrent buffering results from a particularly lean allocation of resources that still meets system throughput and latency requirements, or high contention for data channels due to algorithm demands and the synchronization demands that can occur when data frames are pipelined. As long as there are sufficient resources to overcome either transient or recurrent buffering, the system should tend towards a steady state and the protocol should gracefully handle local queueing of data and dynamic changes in the load on those queues.

The remaining essential functions of the ICU Control are to facilitate the processing primitive's access to data. When an ICU accepts a block of data it must allocate physical memory for the data block and logically integrate that block with other blocks that refer to the same instance/frame and with other instance/frames of data. The memory allocation task also involves providing the primitive with a straightforward view of data blocks with which it is currently concerned.

The allocation issues become a bit more complex depending upon the source of that data. In the standard paradigm, data is transferred from one single purpose primitive to

another. In a more general model, and one which is supported by this protocol, primitives are multifunctioned and may in fact be able to receive as input the data they have just produced. The ICU should allow the co-location of output and subsequent input data via simple memory allocation techniques. The advantages of this are a possibly reduced set-up time for the subtask (provided that the co-locating requires little time in comparison to allocating new blocks of memory for data coming from an external source), and reduced contention for the data channels. For the case of a single link between the destination and source, a savings of the time for that block transfer is realized as potential access time on the data channel by a different pending transfer. This savings will not always be realized, as it depends on the application algorithm, the processing and communication resource allocation and the pipeline synchronization, and the load on the system. The disadvantage is that this does add complexity to the underlying protocol and the actual implementation may preclude some modifications of the protocol. It is believed that this additional complexity is worthwhile given the current view of the central control structure of the ICU and the potential data channel savings, which are highly valued.

The data itself may be stored in one of several standard data formats. The data formats may differ not only in concept, i.e. floating point vs. integer, but in size: four bits for some formats vs. sixteen bits for others. Thus a format conversion must be performed if the network is to support primitives which operate under different formats or if the network will support signal processing algorithms which by necessity require different data formats at different points of computation. The additional support for data compaction or expansion would allow savings in data channel usage under certain conditions. If a data word for a particular application has only four bits of precision, it could still be stored and transmitted within the network and node standard sixteen bit word, but could be stored and transmitted more efficiently if four of these words are compacted into one sixteen-bit word. The com-

plexity of then managing the primitive's view and access of these compacted words would have to be incurred but the savings of data channel time might be worth it. The ICU Control causes a format converter to perform format and expansion or compaction conversions on data is read or written into memory by the processing primitive.

As processing primitives may be multipurpose and in fact may be able to assume a variety of types, the ICU must have knowledge of the capabilities of the primitive, choose the correct current primitive function and provide the primitive with knowledge about its current configuration. The information pertaining to a primitive's current function is collected in a table called the *parameter table*. The ICU must provide the primitive with a reference to this table so that it may collect and follow its operating guidelines for the instance it is currently assuming. After parameter information is provided, the ICU Control causes the primitive to begin execution.

10.3.2.2. A Summary of Extra-Protocol Functions

The ICU Control must undertake operations which are necessary to initialize an ALPS system, monitor the performance of individual nodes and communications resources, and maintain its running state. In addition, it must coordinate the operation of support hardware and provide specific directives and handshaking so that concurrent operations can be managed.

Initialization and Monitoring Functions

There are two classes of initialization functions. The first is the system and task information uploading class. The second is the intra-ICU setup function class. Each of these is considered in the following paragraphs.

Individual nodes must be informed of the overall task, the signal processing algorithm, by providing a logical interconnection table which includes information about how each

particular primitive type can participate in the overall task. In addition, a references which give details about the production of data by each node must be provided so that the ICU can match the correct primitive output blocks with the outgoing links. Providing the primitive with system information also includes the assignment of logical addresses so that each node knows how it will be addressed by other nodes in the system. Each node will not have a list of node addresses and primitive type associations because the communications among nodes will take place first across entire classes of nodes; specific nodes will then be selected within that class by querying over broad categories. The rationale here is that all protocol related functions which cause one physical node to address another are based directly on *a priori* additional information about the node. The dynamic assignment and scheduling heuristics predicate that specific addresses of nodes operating on certain data will not be known in advance and querying for information over all eligible operators will be necessary before focusing on an intended receiver. In addition, the concept of pools of similar resources dictates that the pool be treated similarly. The concept of querying for the node which is currently performing (or has previously performed) functions of particular interest, and then finding the address of that node, is consistent with the dynamic assignment view. Unique logical addresses are therefore assigned to each node but the nodes are not given a direct view of the addresses of other nodes currently operational on the network.

The second class of initialization functions are intra-ICU set-up functions. Assuming the general signal flow algorithm is known, the ICU Control must initialize the state of data queues and support hardware so that their view of the state of the node is consistent. This involves sending functional reset commands and initializing various state and bookkeeping registers. A final initialization task is to upload and maintain a parameter table which gives the processing primitive detailed information about its function during each mode of operation.

System maintenance functions performed by the ICU Control allow individual nodes to set operational parameters and houseclean. A system monitor node may query ICUs for status information either for statistics gathering reasons or to ascertain the health and status of nodes and data. The status queries may be followed by commands to throw out old data or change the priority of tasks. The importance of these functions is twofold. First, monitoring the performance of individual nodes without greatly perturbing the system provides a usable level of instrumentation. Status queries could be staged at times in which the performance of the system is not affected at all, provided the monitoring node is given sufficient information about the system and protocol. The statistics gathering capability is one of the central justifications of building a test-bed system.

The second function of status queries is to aid in runtime parameter modification and garbage collection. The intention here is to provide either a balancing of access or synchronization by modifying the ceilings on data queueing and by selectively pausing node operations. Some of these facilities are not part of the current protocol, as the purpose and tasks of an invasive runtime optimizer will depend on situations that are discovered once the system is operational. Providing these facilities at this development stage will allow for future integration of this monitor as well as future implementation of distributed monitoring and optimizing heuristics.

Internal Interface and Mapping Functions

The ICU Control must engage in interface functions so that the operational sections of the ICU can be initialized and told what to do and when to do it. Mapping functions performed by the ICU Control allow these sections to share a common view of current data and algorithm information. In particular, the ICU Control must provide references to the memory manager so that it can in context translate data block-relative memory access requests by support hardware into physical memory references. It must also provide

contention resolution among devices sharing memory or state information so that the protocol algorithms are accurately implemented and contention is resolved in favor of tasks which have definable and justifiable priority. The details of the interfaces between the ICU Control and support hardware will be explored later but what follows is a brief functional description of these interfaces in terms of initialization and operational information and commands that must be presented to these units.

Message passing across the network is undertaken by the ICU Control and facilitated by the Message Control Unit (MCU). Commands to the MCU to send and receive addressing information followed by the actual message are presented by the ICU Control and simple handshaking indicates the success or failure of a receiving node to accept the information. The MCU also performs token and link bid operations, with the ICU Control supplying the essential commands for the auctioner and the decisions to the bidders. Sample commands to the MCU are: write message, read message, ignore incoming messages, engage in a bidding process, initiate a bidding process.

The interface to the Data Control Unit is similar in structure to that with the MCU. The DCU is responsible for monitoring the data busses for activity and engaging the downloading and uploading of blocks of data to and from the memory module. It is command-driven but the read/write buffer which connects to the data busses is accessible only by the DCU and the memory manager: the ICU Control can not read or write information to or from the data busses. Sample commands to the DCU are: begin DMA transfer from data bus X to memory, begin DMA transfer from memory to data bus X, find next free data bus.

The memory manager receives commands to allocate fixed sized pages of memory and attach pointers to these pages to data queues. It then receives sequential or random references to these queues and presents all data in a queue as a continuous block of memory. In addition, it maintains a mapping of queue numbers to logical port numbers for the primitive

so that the primitive specifies the logical port number and the memory manager accesses the corresponding date queue. Additional mapping allows the ICU Control to specify the queue from which the DCU will interact. Finally, the parameter tables will be stored on separate data queues, one for each mode the primitive can assume (internally, each primitive is indexed by mode) and the MM will contain a mapping of queue number to primitive type. The ICU Control is responsible for commanding the MM to allocate pages and then attach them to a specific queue. It also commands the MM to destroy a queue, deallocating all associated pages. The ICU Control can not actually access the memory pages; its function is to set them up and then control competing access to them between the primitive and the DCU. To accomplish this, a hardware lockout structure prevents the MM from honoring a read or write request while the DCU is operating is controlled by the ICU Control, since it is responsible for causing the DCU to operate.

The format converter translates data from its current format to a format used by the primitive while it is operating in a particular type/mode. The ICU Control will instruct the FCU to perform a read-conversion as words are read from memory by the primitive. The FCU will also deal with compaction/expansion of low-resolution formats, i.e. four bit words can be packed four to a sixteen bit word. In this case, since accessing of data through the memory manager must be in sixteen bit chunks, the FCU will fetch and buffer once for every four sequential accesses. The interface to the FCU is very simply a command to specify the read-conversion and a separate one for write-conversion. The FCU then acts as an intermediary for memory access by the primitive.

10.3.3. ICU Logical Control Operational Summary

Following is a summary of the activities initiated by key inter- and intra-ICU events. These events are: message reception, primitive finishing and deciding to seek a next con-

troller of the message bus.

10.3.3.1. Activities Initiated By Message Reception

Message reception initiates activities to first decide whether the message pertains to the node, then determine the essential type of message: whether it is a bid message, an intra-bid message, a pass control message, an initialization message or a status query. The node then reads supplementary information in the data portion of the message and processes the request or query.

10.3.3.1.1. Requests to Send a Link

Requests to send a link require the receiving node to determine whether it can accept that link. This determination is based on the current state of the primitive, the availability of memory, and the current instantiation of the node. The activities that occur are the internal status checking and then decision to take part in a bidding process for the link. The bidding process is undertaken by the Message Control Unit. If a bid was successful, the node will be either be instantiated to the receptor site of the link it has accepted or it will have already been instantiated and will remain so as additional links for that algorithmic instance are received. Once all data for an instance has been received, the ICU will instruct the primitive to execute.

10.3.3.1.2. Requests to Pass Control

A node will want to control the message bus if it has data to send to other nodes. The current controller, after utilizing the message bus for a proscribed number of transactions, will seek to pass control by sending a pass control bid notice and subsequently controlling a bidding process to find the next controller. Receiver nodes, upon receiving this bid notice, examine the contents of data queues to determine if it has information to send. If it does,

the ICU Control will cause its message control unit to engage in the bid for the control token. If the node is chosen as the next controller, it will assume clocking and sequencing responsibilities for the message bus and will subsequently use the bus to send link bid request messages.

10.3.3.1.3. Initialization Messages and Status Queries

A monitor node will gain control of the message bus in order to send initialization and set-up commands to processor nodes. These messages will be received by processor nodes and appropriate internal queries and value changes will be initiated. A node can not refuse to answer or accept an initialization command unless the command places unreasonable demands on the node, such as a command to allocate an excessive amount of memory. The activities initiated by initialization messages can include changing the state of the associated primitive, clearing and initializing data queues, and receiving new signal flow algorithm maps. The activities initiated by status query messages include requiring the ICU Control to make appropriate references to locally distribute sources: ICU local memory, support hardware or the primitive, and resolving the query. A message is then returned to the monitor node by one of two ways. Either the monitor will send the control token directly to the receiving node or it will allow the receiving node to write response data onto the bus.

10.3.3.2. Activities Initiated By a Primitive Finishing Execution

A primitive that has finished execution of a data block will be in a mode where it will seek control of the message bus when it is offered and attempt to send off data blocks as long as it has data to send. It may receive new data and generate additional blocks of data, in which case it will retain a control-hungry character until its output queue is empty. In typical operation, nodes will generally have something on their output queue to be sent, and will in general seek control when it is offered.

10.3.3.2.1. Seeking Control

A node seeks control by deciding to participate in a bidding process for control passing that is initiated by the current message bus controller. The decision to participate is made by the ICU Control after inspection of its output queue. It subseqently instructs the attached MCU to engage in the bid for control or to ignore the ensuing bid process.

10.3.3.2.2. Seeking Receivers for Data

Receivers are sought for data by posting messages that a bid will take place for a specific block of data. The data block is identified by a link identification number which references the receive table, which in turn indicates the algorithmic instance destination of the block, specifications of the type and function of the next-needed primitive, and some protocol related information. It is also identified by a frame number, which gives temporal reference to data. The controlling node posts a bid request message and proceeds to initiate a bidding process. The bidder that will be granted the block is the one which has an assigned address which places it logically to the right of the current master; the assigned address is essentially a geographical positioning of the node in a logical ring of all nodes. When a bidder has been found, a data transfer is set up by finding an available data channel, informing the receiver where to listen, and then beginning a transfer of data onto the bus. When there are no more receivers for a block of data, the memory for that block is freed.

10.3.3.3. Seeking Next Controller

A node may not hog data or message channels. To ensure this, a restriction has been set on the number of transfers that an ICU may attempt. In one instance of control of the message bus, a node may attempt to send all links that are encompassed in one algorithmic instance/frame. After that, even if no links could be sent, the node must pass control of the message bus. This is done by sending a message that a bid for control is about to occur.

The controller then initiates a bidding process and looks for the node which is logically to the right that wants to gain control. It then passes control to that node.

10.3.4. Summary

This chapter has presented an overview of ICU Logical Control functional characteristics, and some of the justifications for its structural design. Detailed operational information and structural information is presented in subsequent chapters of the *d-ALPS Initial Design Specification.*

References for Appendix A

(1) R. R. Shively, "A Digital Processor to Generate Spectra in Real-Time," *IEEE Trans. Computers*, C-17, N5, 1968.

(2) T.E. Curtis, J.T. Wickendon and A.G. Constantinides, "Control Ordered Sonar Hardware (COSH)- A Hardware Based Signal Processing Graph Implementation," *Proc. IEEE, vol 131, Part F*, pp 584-592, October, 1982

(3) N. H. Brown, "The EMSP Data Flow Computer," *Proc. Int. Conf. Systems Science*, Honolulu, 1984.

(4) Y. S. Wu and L. J. Wu, "An Architectural Framework for Signal Flow," *Proc. International Conference on Digital Signal Processing*, Florence, Italy, 1984

(5) D. C. A. Bulterman, K. L. Robbins, and D. L. Leibholz, "Self-Scheduling Distributed Processing Architecture for Real-Time Signal Processing Applications," *Brown University* August, 1986

(6) J. A. Stankovic, "Issues in Distributed Processing," *IEEE Trans. Computers*, C-33, N12, December, 1984.

(7) P. Markenscoff, "Deterministic Model for Evaluating the Performance of a Multiprocessor System with a Shared Bus" *IEEE Trans. Computers*, March, 1984.

(8) E. D. Jensen, "The Honeywell Experimental Distributed Processor—An Overview," *IEEE Computer*, V11, N1, January, 1978.

(9) D.C.A. Bulterman, "CASE: An Integrated Design Environment for Algorithm-Driven Architectures," *24th IEEE/ACM Design Automation Conference*, (Submitted for presentation), June, 1987

(10) D. C. A. Bulterman and H. Gardillo, "A Distributed Architecture for an Image Processing Application," *Brown University*, (Submitted for Presentation), July, 1986.

(11) Y. S. Wu, "Digital Signal Processor Architecture: A Historical Perspective," *IEEE-Academia Sinica Workshop on Acoustics, Speech, and Signal Proc.*, Beijing, 1986.

(12) D. C. A. Bulterman and E. Manolis, "Application-level Performance Monitoring for Special Purpose Networks," *IEEE Networks*, V1, N4, 1987.

Bibliography

[Afsh83] P.V. AFSHARI AND S.C. BRUELL, "On the Load Balancing Bus Access Scheme," *Transactions on Computers* C-32, 8, IEEE (Aug 1983), 626-636.

[Bara85] AMNON BARAK, "Distributed Load Balancing for a Multicomputer," *Software Practice and Experience* 15, 9 (Sept 1985), 901-913.

[Beiz78] BORIS BEIZER, *Microanalysis of Computer System Performance*, Van Nostrand Reinhold Company, New York, 1978.

[Brig78] FAYE BRIGGS, "Performance of Memory Configurations for Parallel-Pipelined Computers," in *Proc. 5th Ann. Symp. Comp. Arch.*, April 1978, pp. 202-209.

[Bult86] DICK C.A. BULTERMAN AND KENNETH L. ROBBINS, "The Design and Implementation of a Real-Time and Fault Tolerant Network for ALPS-based Digital Signal Processing," in *LEMS Technical Report*, Laboratory For Engineering Man/Machine Systems, Brown University, June 1986.

[Chou82] TIMOTHY CHOU AND JACOB ABRAHAM, "Load Balancing in Distributed Systems," *Transactions on Software Engineering* SE-8, 4, IEEE (July 1982), 401-412.

[Chow79] YUAN-CHIEH CHOW AND WALTER KOHLER, "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System," *Transactions on Computers* C-28, IEEE (May 1979), 354-361.

[Demp82] M. A. DEMPSTER, ED., *Deterministic Sequencing and Scheduling*, Reidel Publishing, Dordrecht, Holland, 1982.

[Dubo82] MICHEL DUBOIS AND FAYE BRIGGS, "Performance of Synchronized Iterative Processes in Multiprocessor Systems," *Transactions on Software Engineering* SE-8, 4, IEEE (July 1982), 419-431.

[Efe82] KEMAL EFE, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *Computer*, IEEE (June 1982), 50-56.

[Fine84] MICHAEL FINE AND FOUAD TOBAGI, "Demand Assignment Multiple Access Schemes in Broadcast Bus Local Area Networks," *Transactions on Computers* C-33, 12, IEEE (December 1984), 1130-1158.

[Fren82] SIMON FRENCH, *Scheduling and Sequencing: An Introduction to the Mathematics of the Job-Shop*, Ellis Horwood Limited, W. Sussex, England, 1982.

[Fuch85] K. FUCHS, "Memory Constrained Task Scheduling on a Network of Dual Processors," *Journal of the ACM* (Jan 1985), 102-129.

[Garr87] WILLIAM J. GARRISON, *Network II.5 User's Manual, Version 3* , CACI, Inc., Los Angeles (June 1987).

[Gaud85] JEAN-LUC GAUDIOT, REX W. VEDDER, GEORGE K. TUCKER, DENNIS FINN, AND MICHAEL L. CAMPBELL, "A Distributed VLSI Architecture for Efficient Signal and Data Processing," *Transactions on Computers* C-34, 12, IEEE (December 1985), 1072-1087.

[Gold86] DEBRA B. GOLDBERG, *User's Guide for the CASE Design Capture System*, LEMS/NDSG Technical Memo, Brown University, October 1986.

[Grap84] MENTOR GRAPHICS, INC., *Mentor Graphics IDEA System Manuals*,September 1984.

[Grou87] NETWORKED AND DISTRIBUTED SYSTEMS GROUP, "distributed-ALPS Initial Design Specification," in *LEMS/NDSG Technical Memo/Report to Naval Research Laboratory under contracts N00014-85-K2002 and N00014-86-K2015*, Laboratory For Engineering Man/Machine Systems, Brown University, June 1987.

[Hart86] IIRO HARTIMO, KLAUS KRONLOF, OLLI SIMULA, AND JORMA SKYTTA, "DFSP: A Data Flow Signal Processor," *Transactions on Computers* C-35, 1, IEEE (January 1986), 23-33.

[Hsia82] J. C. HSIAO AND DAVID S. CLEAVER, *Management Science*, Houghton Mifflin Company, Boston, MA, 1982.

[Hwan84] KAI HWANG AND FAYE BRIGGS, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Company, New York, 1984.

[Kinn78] L.L. KINNEY AND R.G. ARNOLD, "Analysis of a Multiprocessor System with a Shared Bus," in *Proc. 5th Ann. Symp. Comp. Arch.*, April 1978, pp. 89-95.

[Knut68] DONALD E. KNUTH, *Fundamental Algorithms* , Addison-Wesley, Reading, MA, 1968.

[Leib87] DANIEL LEIBHOLZ AND DICK C. A. BULTERMAN, "distributed-ALPS High Level Logical Control Architecturre: Initial Design Specification," in *LEMS/NDSG Technical Report*, Laboratory For Engineering Man/Machine Systems, Brown University, June 1987.

[Leib86] DANIEL LEIBHOLZ, *A PMS-level Simulator for d-ALPS Systems*, LEMS/NDSG Technical Memo, Brown University, November 1986.

[Liu78] JANE W. S. LIU AND C. L. LIU, "Performance Analysis of Multiprocessor Systems Containing Functionally Dedicated Processors," *Acta Informatica* 10, 1 (1978), 95-104.

[Liu74] JANE W. S. LIU AND C. L. LIU, *Bounds on Scheduling Algorithms for Heterogeneous Computing Systems*, Department of Computer Science, University of Illinois, Urbana, Illinois, June 1974.

[Ma84] RICHARD PERNG-YI MA, "A Model to Solve Timing-Critical Application Problems in Distributed Computing Systems," *Computer* C-33, IEEE (Jan 1984).

[Mann84] REINHARD MANNER, "Hardware Task/Processor Scheduling in a Polyprocessor Environment," *Transactions on Computers* C-33, IEEE (July 1984), 626-636.

[Mano] EVA MANOLIS, "Simulation System and Model for a Class of Distributed-Control Multiprocessor Architectures," *Masters Thesis* , Brown University Division of Engineering , Providence, RI (May 1987).

[Mark84] PAULINE MARKENSCOFF, "A Deterministic Model for Evaluating the Performance of a Multiprocessor System with a Shared Bus," *Transactions on Computers* C-33, 3, IEEE (March 1984).

[Mazz83] J. MAZZOLA, "Heuristic Procedure for Allocating Tasks in Fault Tolerant Distributed Computing Systems," *Naval Research Logistics Quarterly* (Sept 83), 493-504.

[McCo87] RODERICK MCCONNELL, *A Discrete Event Simulator for the Junel d-ALPS Architecture*, LEMS/NDSG Technical Memo, Brown University, December 1987.

[Mint81] FRED MINTZER, "Parallel and Cascade Microprocessor Implementations for Digital Signal Processing," *Transactions on Acoustics, Speech, and Signal Processing* ASSP-29, IEEE (Oct 1981), 1018-1027.

[Mold86] DAN I. MOLDOVAN AND JOSE A. B. FORTES, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," *Transactions on Computers* C-35, IEEE (January 1986), 1-12.

[Mona84] O. G. MONAKHOV, "Distributed Dynamic Resource Allocation In Computer Systems With a Programmable Structure," *Avtomatika i Vychislitel'naya Tekhnika* **18**, 3 (1984), 9-17.

[Mott83] JOE L. MOTT, ABRAHAM KANDEL, AND THEODORE P. BAKER, *Discrete Mathematics for Computer Scientists*, Prentice-Hall, Reston, VA (1983).

[Mull85] SAPE J. MULLENDER, *Principles of Distributed Operating System Design*, Mathematisch Centrum, Amsterdam (1985).

[Pate78] J. H. PATEL, "Pipelines with Internal Buffers," *Proceedings of the 5th Annual Symposium on Computer Architecture* (April 1978), 249-254.

[Rama84] KRITHIVASAN RAMAMRITHAM AND JOHN [illegible]OVIC, "Dynamic Task Scheduling in Hard Real-Time Distributed Systems," *Software*, IEEE (July 1984), 65-75.

[Stan84] JOHN STANKOVIC, "A Perspective on Distributed Computer Systems," *Transactions on Computers* C-33, 12, IEEE (December 1984), 1102-1114.

[Ston78] H.S. STONE AND SHAHID BOKHARI, "Control of Distributed Processes," *Computer*, IEEE (July 1978), 97-106.

[Ston77] H.S. STONE, "Multiprocessor Scheduling With the Aid of Network Flow Graphs," *Transactions on Software Engineering* SE-3, 1, IEEE (Jan 1977).

[Taka83] A. TAKAGI, S. YAMADA, AND S. SUGAWARA, "CSMA/CD with Deterministic Contention Resolution," *Journal Select. Areas Commun.* SAC-1, 5, IEE (Nov 1983).

[Thom86] ALEXANDER THOMASIAN, "Analytic Queueing Network Models For Parallel Processing of Task Systems," *Transactions on Computers* C-35, IEEE (December 1986), 1045-1054.

[Vrsa84] DANIEL VRSALOVICJ, "Performance Prediction for a Multiprocessor System," in *International Conference on Parallel Processing*, 1984.

[Wu84] Y.S. WU AND L.J. WU, "An Architectural Framework for Signal Flow," in *Proc. Int'l Conf. on Signal Processing*, September 1984.