THE LOWER BOUND ALGORITHM FOR THE ENHANCED MODULAR SIGNAL PROCESSOR

By

ARLEN NORMAN LONG

Bachelor of Science
Moravian College
Bethlehem, Pennsylvania
1968

Master of Science
Iowa State University
of Science and Technology
Ames, Iowa
1979

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
July, 1988

THE LOWER BOUND ALGORITHM FOR

THE ENHANCED MODULAR

SIGNAL PROCESSOR

Thesis Approved:

Sharilyn A. Thoreson
Thesis Advisor

J. E. Hodel

Don Gorge

Marilyn G. Kletke

Norman N. Durham
Dean of the Graduate College

TABLE OF CONTENTS

LIST OF FIGURES

NOMENCLATURE

ALPS	Alternative Low-level Primitive Structures
AP	Arithmetic Processor
ASP	Advanced Signal Processor
BM	Basic Memory
CASE	Computer Assisted System Engineering
CC	Circus Controller
CM	Count Memory
CP	Command Program
CPP	Command Program Processor
Cbus	Control Bus
DDSP	Data Driven Signal Processor
DFC	Dataflow Computer
DFO	Dataflow Operation
DFSP	Data Flow Signal Processor
DM	Data Memory
DTN	Data Transfer Network
ECOS	EMSP Common Operational Software
EMSP	Enhanced Modular Signal Processor
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
GIP	Graph Instantiation Parameter
GM	Global Memory
GMU	Global Memory Unit

HOL	High Order Language
ICU	Interface Control Unit
IOC	Input-output Controller
IOP	Input Output Processor
LM	Local Memory
LMS	Least Mean Squares
MAP	Mapping Table
NEP	Node Execution Parameter
ONR	Office of Naval Research
PE	Processing Element
PID	Primitive Interface Definition
PIP	Primitive Interface Procedure
PM	Processing Module
POPS	Principles of Operations
PRIM	Primitive
PRIM_IN	Primitive Input
PRIM_OUT	Primitive Output
RNS	Residue Number System
SCH	Scheduler
SI	Sensor Interface
SPGN	Signal Processing Graph Notation
SPU	Signal-processing Unit
TS	Task Supervisor

CHAPTER I

INTRODUCTION

The Enhanced Modular Signal Processor (EMSP) is the next generation signal processor for the U.S. Navy. The EMSP is an embedded real-time signal processor. It must have enough resources to meet the time requirements, in terms of signals per time, of the signal processing application, and it must be physically small enough to fit in the space available. The EMSP is a hybrid dataflow computer which executes signal processing graphs according to the dataflow methodology and executes command programs according to the standard control flow methodology.

The EMSP Common Operational Software (ECOS) is the software programming methodology for which the EMSP is designed to operate. The ECOS programming methodology is a graph-based methodology which uses dataflow graphs to describe the signal processing algorithms and traditional high order language command programs to control and configure the graphs. When the EMSP application programmer produces an EMSP application, the programmer specifies the functionality of the application, but not the numbers of functional units needed to carry out that functionality.

Because of the embedded real-time nature of the Enhanced Modular Signal Processor, it is critical that the EMSP be as small as possible. The minimal EMSP configuration for a particular signal processing application is the smallest configuration which will execute the application. The dynamic scheduling of the operations of the EMSP makes finding the minimal EMSP configuration a time-consuming and costly trial-and-error simulation process.

Given the universe of possible EMSP configurations, the lower bound configuration for a particular signal processing application is the configuration for which there can be no smaller configuration which will execute the application. The lower bound configuration does not guarantee that the application will execute with the lower bound numbers of units; it does guarantee that the application cannot execute with fewer units.

The research problem of this thesis is to identify the lower bound configuration. The outcome of the thesis is a lower bound algorithm which analytically identifies the lower bound configuration as specified by the numbers of functional units. The EMSP packager can use the lower bound configuration as a beginning toward configuring an EMSP and as a measure of minimality of the chosen configuration.

The next chapter, The Literature Review and Background Information, defines embedded real-time signal

processing requirements and reviews dataflow implementations of real-time signal processors. Chapters three and four, The Enhanced Modular Signal Processor and The EMSP Common Operational Software, describe the hardware and software for which this work applies. Chapter five, Configuring the EMSP, describes the problem of identifying the smallest system, tells why the present approach does not provide the solution to the problem, and introduces an alternate approach to the solution. Chapter six, The Algorithm, develops the lower bound algorithm. The thesis closes with chapter seven, Summary and Conclusions. The figures are in the appendix.

CHAPTER II

THE LITERATURE REVIEW AND BACKGROUND INFORMATION

This chapter reviews real-time systems and dataflow implementations of real-time signal processors. The information forms a base for the description in the next chapter of the Enhanced Modular Signal Processor (EMSP). The EMSP is an embedded real-time signal processing system which executes using a hybrid dataflow methodology.

Real-time Systems

Real-time systems are both driven and defined by their applications. Real-time systems are not batch systems printing end-of-day reports. Real-time systems are not interactive systems providing word processing facilities. Real-time systems are not transaction systems answering database queries. Real-time systems are systems in which failure to satisfy their critical timing requirements may result in an external catastrophe.

Definitions

Allworth [1] defines a real-time system as a system which contains application software that controls a set of

devices in a timely manner. This definition differentiates real-time systems from other systems by focusing on the criticality of their deadlines. If a deadline is missed by a multiprocessing system, performance of the system may be degraded; if a deadline is missed by a real-time aircraft control system, the aircraft may crash.

Glass [2] defines a real-time system as one which provides services to or control to an on-going physical process. This definition differentiates real-time systems from other systems by focusing on responsiveness and efficiency. The automobile fuel injection system must react as we depress the accelerator pedal; the anti-lock braking system must react as we depress the brake pedal.

Kowal [3] defines a real-time system as one which contains processes that operate concurrently with independent real world events and have a regular or predictable time relationship with those real world events. This definition differentiates real-time systems from other systems by flow of control. In most cases, a user sitting at a terminal waiting for a response does not constitute an independent external event; the user activity is dependent on the actions of the system. A real-time system produces output responses which must be synchronized with independent external events.

Ward [4] defines a real-time system as a system for which the elapsed time between an external stimulus and

the corresponding response by the system constitutes an important part of the performance of that system. In other words, each potential stimulus to the system has an associated quantitative deadline and system performance is acceptable if and only if the system responds to each stimulus within its deadline. This definition differentiates real-time systems from other systems in which the system performance is qualitative (the response time is acceptable to the user) or averaged (the backlog does not grow indefinitely).

Characteristics

The participants at a workshop sponsored by the Office of Naval Research (ONR) could not agree on a definition for real-time systems. However, they did agree on three characteristics of real-time systems: time is the most precious and most critical resource, reliability is crucial, and the environment in which the real-time system operates is an active part of the system [5].

The first two characteristics, time and reliability, can be thought of as components of real-time systems. The third characteristic, synergism, can be thought of as the interplay between the systems and their environments.

Time is the most precious and most critical resource which a real-time system manages. Not satisfying time constraints leads to failure. The result of a failure, as in flight control for an aircraft, or in sensor based

environment acquisition for a submarine, may be a catastrophe. Reliability of the component parts of a real-time system is crucial to meet the time constraints. Parts failure will result in failure to satisfy the time constraints.

The environment is an active part of a real-time system. The system and its environment are a synergistic pair. The aircraft may not fly without the flight control system; the submarine may not survive without its environment acquisition system.

Embedded Real-time Systems

Embedded real-time systems are not only critically interconnected with their environments, their processing power must fit in a tightly constrained space. Embedded real-time systems operate in environments where physical space is costly. The embedded real-time system for an aircraft must fit inside that aircraft along with all the other space consuming requirements for that aircraft; the embedded real-time system for a submarine must fit inside that submarine along with all the other space consuming requirements for that submarine. Space is at a premium; a too large real-time system means less space is available for other critical needs. Embedded real-time systems must satisfy the requirements for real-time systems and also must be small in size.

In addition to satisfying requirements of time, reliability, and size, most military embedded real-time systems and many non-military embedded real-time systems must be fully shielded. A fully shielded system affects only its environmental subset, is unaffected by the environment outside its subset, and is undetectable by the environment outside its subset. Fully shielding an embedded real-time computer system further constrains the space available for the processing power.

Signal Processing Systems

Signal processing systems [6], [7] are systems which respond to signals and produce other signals. A real-time signal processing system continually accepts streams of input and produces results at a rate no less than that needed to keep up with the input streams.

There are four typical reasons to do signal processing. One reason to do signal processing is to estimate the characteristic parameters for a signal: estimating the rise time of return signals from a matrix radar for distance determination might be done using waveform analysis, perhaps implementing mathematical curvefitting techniques. Another typical reason is to eliminate or reduce unwanted interference: reducing background noise during aircraft communications might be done using filtering techniques, perhaps using a finite impulse response (FIR) filter. Another typical reason is

to transform signals into another more informative form: changing time-domain sensor signals into frequency-domain power information for underwater submarine identification might be done using Fourier techniques, perhaps using a fast Fourier transform (FFT). The fourth typical reason to do signal processing is to modify the characteristics of a system: controlling an inherently unstable high performance aircraft might be done using feedback techniques, perhaps using a least mean squares (LMS) algorithm.

The four typical reasons to do signal processing exist for many application areas. Signal processing applications occur in fields such as acoustics, sonar, radar, geophysics, communications, and medicine. A number of dataflow signal processors have been designed to execute signal processing applications.

Dataflow Signal Processors

Many researchers are designing dataflow machines for signal processing applications. Researchers from Canada, England, Finland, and the United States are represented by the following dataflow signal processor designs.

The Dataflow Computer

In 1982, Wong and Ito [8], from the University of British Columbia in Canada, published a paper proposing a data-driven parallel computing machine for signal

processing applications, applications in which program code is executed repeatedly. The objectives of the machine design of their dataflow computer (DFC) were to allow concurrent computations while avoiding unnecessary replication of code. Figure 1 in the appendix shows a block diagram of the architecture of the dataflow computer.

The design of the DFC differs in some ways from traditional dataflow computers. They replaced the low level processing elements of traditional dataflow computers by small general purpose processors, which they continued to name processing elements (PEs). Using general purpose processors results in fewer types of components needed, higher resource utilization through interchangeability, and increased fault tolerance capabilities. It also increases the service rate, as the dataflow operations (DFOs) can be assigned to the first free processing element, without the DFO first being decoded.

Rather than being devoid of local storage, as is the case in traditional dataflow computers, each PE in the DFC has associated with it a substantial local memory (LM); each local memory holds colored multiple concurrent activations of a single signal processing procedure. A group of processing elements with their local memories is called a processing module (PM). Separate data memories (DMs) are backing stores for code, and separate count

memories (CMs) contain the operand counts and numbers of operands for the dataflow operations. A single task supervisor (TS) monitors the states of the PMs and updates individual mapping tables (MAPs) to indicate which LMs contain what procedures. A timeshared bus carries all execution related traffic.

Simulation of the design of the DFC provided five major results. First, timeshared busses are inadequate for their traffic. Second, if switching networks are used to solve the traffic problem of the timeshared busses, then the number of PEs should be approximately equal to the number of paths provided. Third, global count memories worked as well as independent count memories. Fourth, more capable processing elements increased the service rate. Fifth, colored multiple concurrent activations were faster than either sequential or pipelined operations.

The Data Driven Signal Processor

In 1982, Hogenauer, Newbold, and Inn [9], working at ESL Corporation, a subsidiary of TRW, published a description of their proposed Data Driven Signal Processor (DDSP). The objectives of their machine design were to allow easy programming and modular expandability while providing a maximum configuration execution rate of 71 mflops. Figure 2 in the appendix shows a block diagram of the architecture of the Data Driven Signal Processor.

The design of the DDSP is similar to the machine of Watson and Gurd [10] from the University of Manchester. Each processor contains a queue which holds input, a matching store which groups labeled tokens, and a 2.22 mflops floating point processing element. Two networks connect the processors: a circular packet switch network connects the processors for nearest-neighbor communication, and a three level tree network connects the processors for long-distance communication.

Simulation of the design of the DDSP provided three major results. First, the dataflow nature of the DDSP allows programming flexibility and effectiveness not possible with array processors. Second, although the processor efficiency (the percentage of time during which the processor is doing useful work) decreases for larger numbers of processors if the number of parallel operations is kept constant, processor efficiency increases for larger processor configurations if the number of parallel operations is also increased. Third, processor efficiency can be very high, with percentages above 90 percent for large systems.

The Data Driven Signal Processor was an unsuccessful competitor in the design race for the next generation standard signal processor for the U.S. Navy.

The Data Flow Signal Processor

From 1982 through 1983, Kronlof, et. al. [11]-[15], from the Helsinki University of Technology in Finland, published a number of papers describing their proposed Data Flow Signal Processor (DFSP). The objectives of their machine design were to use a bus oriented architecture to implement efficiently a processor mainly intended for data intensive applications such as digital signal processing while also providing expandability and convenient programming. Figure 3 in the appendix shows a block diagram of architecture of the Data Flow Signal Processor.

The design of the DFSP uses a bank of high level, and potentially special purpose, processors called processing elements (PEs). An update unit matches colored result tokens and allocates data storage for the results; the result transfer unit controls storing the results in the data storage. A fetch unit assigns executable operations and data to appropriate free PEs; after transmission, the data transfer unit deallocates the data storage. The queue allows transmitting results from a PE, via the update unit and result transfer unit, to the fetch unit without those results being stored into the data storage.

Two busses carry the data and control traffic: one bus carries the signal processing data; the other bus carries the operation and result control packets for and from the operation execution, respectively.

Simulation of the design of the DFSP provided four major results. First, the update unit is the major bottleneck in the control section of the machine. Second, it is relatively simple to obtain uniform utilization of the processors. Third, the value of the size of the packets is linearly dependent on the bandwidth of the busses and inversely dependent on the throughput of the control section. Fourth, the fetch unit is not critical for performance.

The Dataflow Binary Tree Processor

In 1984, Jamali, et. al. [16], from the University of Windsor in Canada, published a paper proposing a dataflow binary tree digital signal processor. The objectives of their machine design were to exploit the fast computational approaches of distributed, parallel, and pipeline techniques while reducing or eliminating the communication problems and indeterminacy associated with conventional dataflow architectures. Figure 4 in the appendix shows a block diagram of the architecture of the dataflow binary tree processor.

The design of the dataflow binary tree digital signal processor implements the carry free arithmetic operations of the residue number system (RNS) by using multiple large memories. The processor prestores the arithmetic operations (multiplication, addition, or subtraction) in the memories. It performs an arithmetic operation by

forming an address using the two input numbers and then reading the result from memory. The execution time for any arithmetic operation is the time to access the memory added to the time to capture the result into the latch.

Each cell, or node, of the complete binary tree processor is a computational element. Cells located at nodes which have two children are called T-cells. T-cells perform arithmetic operations. Cells located at leaf nodes are called base cells or C-cells. C-cells store the look up tables into the T-cells prior to beginning algorithm execution, receive data and coefficients from the data busses, perform the specified arithmetic operation and create an output packet with sufficient control bits to travel up the tree through the T-cells.

Jamali, et. al., state that the design of the dataflow tree processor provides four major benefits. First, there is only a 7.5 percent overhead of bits associated with packets, compared to 200 percent overhead in other dataflow architectures. Second, the packets are transmitted in parallel, thus avoiding the overhead associated with serial communication protocol. Third, the approach is deterministic and the throughput rate can be estimated. Fourth, the computation time of any arithmetic operation is reduced to the access time of the memory.

The Roman Circus System

Also in 1984, Wu, Constantinides, Curtis, and Wu [17] published a paper describing their proposed Roman Circus System. Y. S. Wu is from the U.S. Naval Research Laboratory, Curtis is from the Admiralty Underwater Weapons Establishment in England, and Constantinides and L. J. Wu are from the Imperial College of Science and Technology in England. The objective of their machine design was to execute efficiently alternative low-level primitive structures (ALPS) for acoustic signal processing. Figure 5 in the appendix shows a block diagram of the architecture of the Roman Circus System.

The design of the Roman Circus System contains three classes of functional primitive modules: processing elements (PEs), basic memories (BMs), and sensor interfaces (SIs). A standard system interface control unit (ICU) performs input and output queue management, data buffering, activation and deactivation of the primitive, and system communication and monitoring. The interface control units and their associated modules make up units: an ICU and a PE comprise a signal-processing unit (SPU), an ICU and a BM comprise a global memory unit (GMU), and an ICU and an SI comprise an input-output controller (IOC).

Three concentric communication paths carry the traffic: a serial message circus, a parallel data circus, and a parallel monitor circus. The monitor circus doubles

as a redundant data circus. A circus controller (CC), or network manager, monitors system performance, dynamically reconfigures the system, and initially loads structures onto the paths. The CC allows only one item of message, data, or signal onto each path at any time.

Wu, et. al. anticipate that the Roman Circus System will provide two major benefits in addition to executing ALPS primitives. First, the modularity of the system can allow for extra modules to increase computational power and system redundancy. Second, multiple Roman Circus Systems can be combined easily by defining each system be a module of an even higher level Roman Circus System, thus making a hierarchical cluster system with even greater connectivity, computational power, and system redundancy.

CHAPTER III

THE ENHANCED MODULAR SIGNAL PROCESSOR

The United States Navy places many embedded real-time signal processing applications into its ships and aircraft [18]. During the late 1970's, the Navy determined that its future signal processing requirements could not be attained by its current signal processing architecture, the Advanced Signal Processor (ASP). Experience with the ASP had proved that the concept of a single software development system coupled with a limited number of hardware module types could satisfy the needs of multiple Navy signal processing applications. However, new technologies would be necessary to increase the performance per cubic inch of the embedded real-time signal processors. Further, new programming methodologies would be needed to reduce the cost of creating and maintaining the increasingly numerous and complex signal processing applications. The new signal processing architecture would be called the Enhanced Modular Signal Processor (EMSP) [19]-[21].

A real-time signal processing application is characterized by repeatedly executing a well-defined sequence of signal processing algorithms against signal

values as those signal values become available. Signal processing application programmers typically begin designing an application by drawing a picture of the flow of signal values through the appropriate signal processing transformations. The picture is a directed graph where the arcs represent the flow of data, and the nodes represent the operations done to the data.

To reduce the cost of creating and maintaining the signal processing applications, the Navy decided on a graph-based programming methodology, the EMSP Common Operational Software (ECOS) [20], [22]-[24]. The Navy then requested bids for a machine architecture which would directly execute ECOS, the EMSP. AT&T Bell Laboratories, AT&T Technologies, and Unisys jointly are developing and producing the Enhanced Modular Signal Processor.

The EMSP

The dataflow methodology of computer operation is that a computer should execute an operation as soon as the operands for that operation become available. A dataflow computer executes graphs where the arcs represent the flow of data, and the nodes represent the operations done to the data. The results of a research group at Helsinki University, Helsinki, Finland show that a dataflow signal processing computer can execute efficiently real-time signal processing applications [14].

The Enhanced Modular Signal Processor is a hybrid dataflow computer. Graphs, which describe the signal processing algorithms, execute according to the dataflow methodology. Command Programs, which control and configure graphs, execute according to the standard control flow methodology.

Certain features and implementations of the EMSP are changes from traditional dataflow architectures. Each arc of the ECOS graph allows multiple instances of data elements, as compared to only one element, and is implemented as a queue. Further, each node of the ECOS graph is arbitrarily complex, as compared to consistently simple. Example ECOS node operations include Fast-Fourier Transform, Finite Impulse Response Filter, and Frequency Domain Beamformer. The results of a research group at AT&T Bell Laboratories show that the EMSP can execute efficiently real-time signal processing applications [21].

The hardware modules which comprise an EMSP can be grouped into three basic categories: functional elements used primarily for the communications within the EMSP, functional elements used primarily for the sequential command program operation, and functional elements used primarily for the dataflow graph execution. Figure 6 in the appendix shows a block diagram of the architecture of the Enhanced Modular Signal Processor. The following sections describe the functional elements and their interrelationships.

EMSP Communications

The Control Bus

The Control Bus (Cbus) is the bus which provides paths among the various functional elements of the EMSP for transmitting control and status messages. These short messages move over the 8 byte token passing bidirectional bus asynchronously at a maximum data rate of 4.61 megabytes per second. An EMSP has one Control Bus.

The Data Transfer Network

The Data Transfer Network (DTN) is the network which provides paths among the functional elements of the EMSP for transmitting messages comprised of large blocks of data. The DTN is an N by N crossbar switch and provides parallel unidirectional asynchronous communication paths for up to N simultaneous paths. An EMSP has one or two Data Transfer Networks.

The Global Memory

The Global Memory (GM) is the intelligent storage element for the EMSP. The GM frees the application programmer from many tasks of allocating and controlling resources in software. It contains instruction streams, queues, and graph variables. When a graph is instantiated (an executing graph is created from a graph template), it creates the nodes, queues, and graph variables for the

graph instance. It allocates and deallocates memory as queues are written and consumed. It identifies when a queue is over threshold, that is: when the number of operands on an arc is equal to or greater than that needed by the node to execute (providing all other arcs also meet or exceed their thresholds). An EMSP has one or more Global Memories.

EMSP Sequential Command Operation

The Command Program Processor

The Command Program Processor (CPP) is the control element for the EMSP. It starts and stops graph execution, starts and stops input, and interacts with the operator to configure graphs. It does not participate directly in graph execution. An EMSP has one Command Program Processor.

The Input Output Processor

The Input Output Processor (IOP) is the channel controller for the EMSP. It performs the input of signal values to the graph and performs the output of results from the graph. An EMSP has one or more Input Output Processors.

EMSP Dataflow Graph Execution

The Scheduler

The Scheduler (SCH) is the node scheduler for the EMSP. It contains the graph topology information needed for graph execution. For instance, the Scheduler knows which queues are inputs to a node. When the Global Memory identifies a queue over threshold, it sends a message to the Scheduler. The Scheduler checks all the input queues for the node to which the over threshold queue is an input and if all the input queues are over threshold, the Scheduler schedules execution of the node on an Arithmetic Processor. The EMSP has one Scheduler.

The Arithmetic Processor

The Arithmetic Processor (AP) is the node processor for the EMSP. It executes the node operations. An Arithmetic Processor can be executing portions of three nodes concurrently. One node may be in its setup phase, during which all needed information is read from the Global Memories. A second node may be in its execute phase, during which the operations are performed on the data. A third node may be in its breakdown phase, during which the results of its execution are stored in the Global Memories. An EMSP has one or more Arithmetic Processors.

All the functional elements work together concurrently to execute an embedded real-time signal processing application in the Enhanced Modular Signal Processor. The following section describes the overall operation of the EMSP by tracing the path of execution of one node.

EMSP Operation

When the Scheduler recognizes that a node is ready to execute, the Scheduler assigns the node to an Arithmetic Processor and sends a message over the Control Bus to the specific Global Memory which contains the instruction stream of the node. The Global Memory then sends the instruction stream over the Data Transfer Network to the designated Arithmetic Processor.

The Arithmetic Processor executes the instruction stream by completing the setup phase, the execute phase, and the breakdown phase. During the setup phase, the Arithmetic Processor sends messages requesting the data needed to execute the node over the Control Bus to Global Memories. After the Arithmetic Processor has received all the requested data from the Data Transfer Network, the Arithmetic Processor executes the node. During the breakdown phase, the Arithmetic Processor sends messages containing the results to be stored through the Data Transfer Network to the Global Memories.

As a Global Memory stores results into a queue, it also checks if the queue has gone over threshold, indicating that there is enough data in the queue for the next node to execute. For any queue which has gone over threshold, the Global Memory sends a message over the Control Bus to the Scheduler which checks to see if all needed data is available for execution of another node. This starts another cycle of execution.

CHAPTER IV

THE EMSP COMMON OPERATIONAL SOFTWARE

A real-time signal processing application is characterized by repeatedly executing a well-defined sequence of signal processing algorithms against signal values as those signal values become available. Signal processing application programmers typically begin designing an application by drawing a picture of the flow of signal values through the appropriate signal processing transformations. The picture is a directed graph where the arcs represent the flow of data, and the nodes represent the operations done to the data.

To reduce the cost of creating and maintaining the signal processing applications, the Navy decided on a graph-based programming methodology, the EMSP Common Operational Software (ECOS) [20], [22]-[24]. AT&T Technologies is developing and producing the EMSP Common Operational Software. It is also developing and producing the initial signal processing applications using ECOS.

The ECOS

The EMSP Common Operational Software is a hybrid programming methodology which uses dataflow graphs to

describe the signal processing algorithms, and traditional high order language (HOL) command programs to control and configure the graphs.

The software features which complete an ECOS signal processing application can be grouped into two basic categories: features used primarily for producing the ECOS graphs, and features used primarily for producing the command programs. The ECOS graphs and the command programs make up the signal processing application. The application programmer describes the nodes, arcs, and various parameters by using Signal Processing Graph Notation (SPGN). The programmer describes the command program by using a high order language and Command Program SPGN.

ECOS Graph Constructs

A signal processing application programmer typically begins designing an application by drawing a graph of the flow of signal values through the appropriate signal processing transformations. Each of these transformations is a signal processing algorithm which the programmer represents as a node of the graph. Each connection between transformations is a queue which the programmer represents as an arc of the graph. The programmer also specifies to the graph various parameters which allow for changing sensor conditions.

The Node

An ECOS node represents the signal processing entity in an ECOS program, or graph. Example ECOS node operations include Fast-Fourier Transform, Finite Impulse Response Filter, and Frequency Domain Beamformer. The ECOS signal processing application programmer builds signal processing graphs using these predefined signal processing operations, called primitives (PRIMs). For each node in the graph, the programmer specifies the name of the primitive to execute, and the names of the queues and variables which are connected to the logical input and logical output ports of the node.

Associated with each node is a Primitive Interface Procedure (PIP) which provides data elements to the primitive as it executes. These data elements, called primitive inputs (PRIM_INs) and primitive outputs (PRIM_OUTs), include constants, graph controls, and data from the queues.

The Queue

An ECOS queue represents the primary data storage in an ECOS program. These expandable first-in first-out structures do not branch and are connected at both ends to nodes. Two types of queues exist: data queues and trigger queues. Most queues are data queues; they contain data elements of arbitrary complexity. Relatively few queues are trigger queues; they contain only

synchronization signals and are used to synchronize nodes which must share a timing relationship but which share no data relationship. Queues are internal or dynamic. Internal queues are declared within a graph to connect nodes. Dynamic queues are defined in a command program to connect together multiple graphs or to connect graphs to input/output procedures.

The Graph Variable

A Graph Variable (GV) represents a memory location which holds one data element of arbitrary complexity. Graph variables provide communication among graphs and command programs. Graph variables are internal or dynamic. Internal graph variables are declared within the graph definition and are local to that graph; that is, they are read and write accessible to the graph, and read-only accessible to subgraphs of that graph. Dynamic graph variables are defined in the command program and are read-only accessible to a graph using them.

The Node Execution Parameters

The Node Execution Parameters (NEPs) describe the ways in which the availability of data on queues affects the execution of the nodes. Associated with each node is a Primitive Interface Procedure (PIP) which calculates the amounts for each Node Execution Parameter at run time and which provides the data items to the primitive as it

executes. Each arc of the ECOS graph allows multiple instances of data elements; this provides flexibility and reduces data transfer overheads as compared to traditional one element arcs. The application programmer specifies the number of data elements needed for the various operations per queue. Threshold, Read, Offset, and Consume amounts are the NEPs which relate to the node input ports. Valve, Produce, and Pulse amounts are the NEPs which relate to the node output ports.

The Threshold amount is the number of data elements which must be present on the queue for the node to execute. The Read amount is the number of data elements on the queue which are used by the node when it executes; for fault-tolerance, node execution reads are non-destructive. The Offset amount is the number of data elements on the queue to skip before beginning to read the data elements; often in signal processing applications, parts of the data stream are ignored as execution proceeds. The Consume amount is the number of data elements to remove from the queue after node execution; often in signal processing applications parts of the data stream are reused in subsequent executions of the node.

The Valve amount is a switch which enables or disables the output of the node; it may be that a particular graph configuration does not need the output to a specific port of the node downstream from the node. The Produce amount is the number of data elements to be added

to an output data queue. The Pulse amount is the number of pulses to be added to an output trigger queue. The application programmer does not specify the Produce and the Pulse amounts; the PIP calculates them based on the execution rules of the primitive.

ECOS Command Program Constructs

A signal processing application programmer builds a Command Program (CP) by writing in a high order language (HOL) and including in it special calls to graph operations (Command Program SPGN).

A command program has five functions: command program control, input/output control, queue control, graph instance control, and graph variable control.

Because of the dataflow nature of the system, command programs do not schedule tasks, allocate resources, or manage memory. Command programs do start and stop graphs, create queues, read and write queues and control variables, and perform exception handling. Command programs control graph instantiation and dynamic reconfiguration; they do not do any signal processing.

The High Order Language

The High Order Language (HOL) is an arbitrary high order language into which the programmer embeds predefined Signal Processing Graph Notation procedure calls (Command Program SPGN). Although the present command programs are

written in the Navy designed high order language CMS2, future command programs are to be written in Ada.

The HOL programs provide control structures within which SPGN procedure calls are embedded to control graph execution and interaction. The HOL programs also establish communications between the command program and the outside world.

The Embedded ECOS Statement

The embedded ECOS statements are predefined Signal Processing Graph Notation procedure calls (Command Program SPGN). The procedure names are prefaced with a percent sign.

The embedded ECOS statements allow command programs to start and stop graphs, create queues, read and write queues and control variables.

ECOS Programming

The ECOS programmer begins programming by drawing a graph of the signal processing application. Circles, representing nodes, are labeled with names of signal processing primitives. Arcs, representing queues, are labeled with names of queues containing data along with the values of their associated node execution parameters. Boxes, representing data elements for the primitives, are labeled with the names of the primitive inputs and primitive outputs, and are attached to the appropriate

nodes. After drawing the graph, the programmer converts the graph to the appropriate ECOS SPGN. The programmer also writes a separate command program with its embedded Command Program SPGN to control the graph.

Figure 7 in the appendix shows a graph, Figure 8 in the appendix shows the ECOS SPGN for the graph in Figure 7, and Figure 9 in the appendix shows a command program for the graph in Figure 7.

By writing high level procedures which contain all the Command Language SPGN, the programmer can make the command program look as if it is written in only a high order language; all the command program SPGN can be buried inside procedures. Further, by writing a procedure which has the command program modify a graph control, the high order language program can reconfigure dynamically the graph during execution.

Figure 10 in the appendix shows a command program for a dynamically reconfigurable graph.

CHAPTER V

CONFIGURING THE EMSP

The Enhanced Modular Signal Processor (EMSP) is the next generation embedded real-time signal processor for the U.S. Navy. At its system level, the EMSP operates as a dataflow computer. It uses a single command program processor and a single scheduler to oversee the operations of multiple processors, memories, and data interconnections.

The Need for Minimal EMSP Systems

The EMSP must operate in real-time: that is, it must produce its output at a rate equal to or greater than its respective input rates. The EMSP must be embedded: that is, it must fit into a confined space. In some ways, these two criteria conflict. Increasing the functionality of the computer tends to increase the number of modules needed to obtain that functionality, and thus tends to increase the size of the machine to the point where it does not fit in the space available.

The real-time portions of the computer are described by the number of operations per second which the computer must execute. The projected throughput requirements to

process the data from a submarine large-aperture array is 4 trillion operations per second, with 2.4 trillion of those operations being complex multiplications. Even small airborne signal processors have requirements of 300 million complex operations per second for the 1990 time frame [25]. Failure to meet these requirements in an operational signal processor will result in potential disaster.

The embedded characteristics of the computer are described by the size of the environments into which it must fit. In this case, the anticipated environments are locations in aircraft and submarines. In every instance, space is at a premium and the larger the computer, the less space for other critical needs.

As the EMSP is a modular computer, matching the number of modules to the needs of its signal processing application will result in the smallest physical size, and thus allow maximal operational functionality in the smallest overall package.

The Present Approach

The EMSP is designed to execute EMSP Common Operational Software (ECOS). ECOS is a graph-based programming methodology where arcs represent flow of signal values and nodes represent signal processing transformations. When the EMSP application programmer produces an EMSP application, the programmer specifies the

functionality of the application, but not the hardware modules needed to carry out that functionality.

Because of the embedded real-time nature of EMSP systems, it is critical that an EMSP system be as small as possible. In other words, it is critical that the machine have the fewest numbers of processors, memories, and data interconnections needed to execute the signal processing application.

The present approach to identifying the smallest system is based on information obtained by executing the signal processing application. The approach is: 1) find an existing application with functionality similar to the new application, and then 2) modify the machine which executes the existing application by adding or deleting processor, memory, and data interconnection modules to obtain a new machine which will execute the new application. Unfortunately, there are at least three difficulties with the present approach.

Problems with the Present Approach

First, the present approach requires finding an existing similar application. If the new application has functionality dramatically different from all existing applications, or if the new application is the first application (as is the present case, EMSP being a new signal processor with no ECOS programs for it yet), the present approach fails.

Second, to obtain any configuration information, the present approach requires executing the application. Based only on dynamic information obtained from executing the application, the present approach is expensive.

Third, even if a modified machine were obtained which executes the new application, the present approach gives no information as to how close that machine configuration is to the minimal configuration.

The Lower Bound Approach

Given the universe of possible EMSP configurations, the lower bound configuration for a particular signal processing application is the configuration for which there can be no smaller configuration which will execute the application.

The lower bound approach to identifying the lower bound configuration is based on static information obtained from the well-formed ECOS graph. The approach is: 1) for each type of hardware module, determine the minimum needed capacity for its functionality, be that minimum in cycle rates, memory space, data rates, or transfer rates, and then 2) divide the minimum needed capacity by the EMSP specified capacity for that type of module.

The lower bound approach does not rely on previously existing applications or machines. It uses only static information from the graph, the Primitive Interface

Definition (PID), the application, and the characteristics of the EMSP to identify the lower bound configuration.

CHAPTER VI

THE ALGORITHM

The Well-formed ECOS Graph

A well-formed ECOS graph satisfies a number of criteria. First, the graph has correct syntax. Second, the graph has no deadlocked cycles. Third, the graph contains only consistent node execution frequencies.

A correct solution to an ECOS signal processing problem has a graph which is well-formed. Having a well-formed graph does not imply that the solution to an ECOS signal processing problem is correct, but having a graph which is not well-formed does mean that the solution is incorrect. It is wasteful to configure an EMSP for a solution which is wrong. Therefore, it is important to assure that the graph is well-formed before continuing. Computer Assisted System Engineering (CASE) tools can traverse the graph and identify if it is well-formed.

The Graph Parser

A graph having correct syntax satisfies the conditions that each node has the correct numbers and data types of input queues, output queues, and primitive parameters, and that each queue of the graph has exactly

one source node and one sink node. The definition of each primitive available to an ECOS graph node is contained in the Primitive Interface Definition (PID). A Graph Parser can check each node of the graph against the definition of the primitive of that node and report discrepancies. It also can report the names of queues used too many or too few times.

The Deadlock Detector

A graph has a deadlocked cycle if there exists a cycle such that each node in the cycle needs data from a predecessor in the cycle before that node can execute. If there is a deadlocked cycle in a graph executing on a dataflow processor, no node in the deadlocked cycle can obtain its needed data. The nodes in a deadlocked cycle will never go over threshold; the nodes in a deadlocked cycle will never execute.

A graph has no deadlocked cycles if there is no cycle where each node in the cycle needs data from a predecessor in the cycle before that node can execute. Nodes in deadlocked cycles can be identified by the following process: create a reachability matrix where a 1 in (a,b) indicates a direct path from node a to node b; zero out fully initialized paths; form the transitive closure of the resulting matrix (Warshall's Algorithm [26] is appropriate for this process); a 1 in (a,a) indicates that node a is in a deadlocked cycle. Fully initializing one

of the queues can eliminate the deadlocked cycle. A graph Deadlock Detector can report the names of nodes which are contained in deadlocked cycles.

The Produce Calculator

A requirement of a correctly executing graph is that a queue does not overflow. In other words, the Node Execution Parameters (NEPs) specified by the ECOS application programmer must coordinate the Produce amounts of a predecessor node and the Consume amounts of the current node such that the input queue of the current node does not overflow. The programmer does not specify the Produce amounts of a node when creating a graph; the Primitive Interface Procedure (PIP) calculates the Produce amounts as the graph executes based upon the NEPs and the execution rules of the primitive in the Primitive Interface Definition (PID). A Produce Calculator, using information from the graph and from the PID, can calculate the Produce amount for each output queue for one execution of a predecessor node. The Produce amount is needed to calculate the frequencies of a node. Figure 11 in the appendix shows an example of the operation of the Produce Calculator.

The Frequency Calculators

A graph contains only consistent node execution frequencies if the graph contains only consistent

assignments of Node Execution Parameters (NEPs), contains only consistent assignments of input data rates, and contains no unachievable performance requirements.

The Relative Frequency Calculator. Nodes communicate via queues. An output queue from one node is an input queue to another node. The relative frequency of a node describes how relatively often the node must execute and consume data to keep up with the produced data of its initial input node. Relative frequencies are in terms of the symbolic rate of the initial input node frequency, freq(i). If a predecessor node executes two times as often as its initial input node frequency, producing 1024 data items from each execution, and the current node consumes 512 data items at each execution, then the current node must execute four times as often as its initial input node frequency (4*freq(i)) to keep up with the produced data and prevent queue overflow.

```
relative frequency of current node =
   relative frequency of predecessor node
   MULTIPLIED BY  produce amount of predecessor
         node for queue from predecessor to
         current
   DIVIDED BY  consume amount of current node
         for queue from predecessor to current
```

A node of the graph may have multiple paths to it from one input node. The programmer must specify consistent NEPs for each queue of the graph such that each node executes at a consistent frequency relative to each of its initial input nodes. A node can execute only at

one frequency; if the relative frequencies for a node imply that the node must consume data from one of its input queues three times as often as it must consume data from another of its input queues, then at least one specified NEP is incorrect. Figure 12 in the appendix shows an example of the operation of the Relative Frequency Calculator.

The Required Frequency Calculator. The required frequency of a node describes how often the node must execute and consume data to keep up with the produced data of its predecessor node. Required frequencies are in terms of actual executions per second. If a predecessor node executes two times a second, producing 1024 data items from each execution, and the current node consumes 512 data items at each execution, then the current node must execute four times a second to keep up with the produced data and prevent queue overflow.

```
required frequency of current node =
   required frequency of predecessor node
   MULTIPLIED BY  produce amount of predecessor
         node for queue from predecessor to
         current
   DIVIDED BY  consume amount of current node
         for queue from predecessor to current
```

If the node is an initial node, then the required frequency of the current node describes how often the initial node must execute and consume data to keep up with the data provided by its sensor input. If a sensor is providing data at a rate of 32768 data items per second,

and the initial node consumes 1024 data items at each execution, then the initial node must execute 32 times a second to keep up with the sensor data.

required frequency of initial node =
input data rate
DIVIDED BY consume amount of input node for queue from sensor to current

A node of the graph can have multiple predecessor nodes. The programmer must specify consistent NEPs for each queue of the graph such that each node executes at a consistent frequency relative to each of its predecessor nodes. A node can execute only at one frequency; if the required frequencies for a node imply that the node must consume data from one of its input queues three times as often as it must consume data from another of its input queues, then the required frequencies of the node are inconsistent. If all the relative frequencies of a graph are consistent and the required frequencies of the node are inconsistent, then at least one specified NEP is incorrect or at least one input data rate is incorrect. Figure 13 in the appendix shows an example of the operation of the Required Frequency Calculator.

The Maximum Frequency Calculator. The maximum frequency of a node describes the maximum number of times a node can execute in a particular EMSP configuration. If the clock rate of an arithmetic processor (AP) is 100,000 cycles per second, and the node requires 4000 cycles to

execute once, then the node can execute at most 25 executions per second.

```
maximum frequency of node =
   clock rate of arithmetic processor
   DIVIDED BY  cycles needed to execute node
```

If the required frequency for a node is greater than its maximum frequency, then the EMSP cannot execute the application as it is written. There is at least one node which is too large and must be divided into smaller nodes that can execute concurrently, or there is at least one initial input data rate which is too great and must be reduced. Figure 14 in the appendix shows an example of the operation of the Maximum Frequency Calculator.

Ready to Configure

Prior to configuring, Computer Assisted System Engineering (CASE) tools can traverse the graph and identify if it is well-formed. A Graph Parser can check each node of the graph against the definition of the primitive of that node and report discrepancies. It also can report the names of queues used too many or too few times. A Deadlock Detector can report the names of nodes which are contained in deadlocked cycles. A Produce Calculator can calculate the Produce amount for each output queue for one execution of predecessor node. Frequency Calculators can check that a graph contains consistent assignments of Node Execution Parameters,

contains consistent assignments of input data rates, and contains no unachievable performance requirements. It is now reasonable to identify the lower bounds on each type of functional unit needed to execute the graph.

The Lower Bounds

This section contains 4 sub-sections. Each sub-section describes one part of the complete algorithm which identifies the lower bound on the number of each type of hardware module needed to execute a signal processing application on the Enhanced Modular Signal Processor (EMSP). The algorithm uses only static information from the graph, the Primitive Interface Definition (PID), the application, and the characteristics of the EMSP to identify the attributes and resultant hardware needs of the application. The sub-sections are in the order of the functional units they address and are in the order: Arithmetic Processors (APs), Global Memories (GMs), Input Output Processors (IOPs), and Data Transfer Networks (DTNs).

Each sub-section identifies the lower bound for one type of functional unit. The lower bounds assume a well-formed graph and 100% utilization of the functional units. The first sub-section identifies the lower bound on the number of APs needed to execute the graph in real-time. The second sub-section identifies the lower bound on the number of GMs needed to hold the instruction streams and

the data. The third sub-section identifies the lower bound on the number of IOPs needed to handle the input and output in real-time. The fourth sub-section identifies the lower bound on the number of DTNs needed to support the data communications among the APs and GMs in real-time.

The descriptions are in separate sub-sections for clearness of explanation. For efficient execution, the implementation of the algorithm can combine the sub-sections appropriately so as to gather all the information in one graph traversal, and later make the necessary computations to identify the lower bounds. Each sub-section is described using successive decomposition of the unknowns until the lower bound is defined completely by known quantities.

The Lower Bound on Arithmetic Processors

Sub-section one identifies the lower bound on the number of Arithmetic Processors (APs).

The lower bound on the number of Arithmetic Processors needed is the total number of machine cycles per time-unit needed to execute the graph divided by the number of machine cycles per time-unit obtainable from one Arithmetic Processor. In other words, it is the total needed cycle rate divided by the Arithmetic Processor cycle rate.

```
lower bound on the number of APs =
   CEILING OF  (
      total needed cycle rate
      DIVIDED BY  cycle rate of AP )
```

A graph consists of a number of nodes. The total cycle rate needed to completely execute the graph is the total of the cycle rates needed per node for all the nodes in the graph.

```
total needed cycle rate =
   SUM OVER  all nodes
   OF  cycle rate per node

all nodes =
   SPECIFIED IN  the graph
```

An individual node may be required to execute a number of times to complete one execution of the graph. Therefore, the total cycle rate needed for a node is the cycles needed for one node execution multiplied by the number of times the node is executed per time-unit.

```
cycle rate per node =
   cycles for one node execution
   MULTIPLIED BY  node execution rate
```

The cycles needed for one node execution is a function of the Read amount for the underlying primitive of the node. Looking up the primitive name in the Primitive Interface Definition (PID) provides the formula to calculate the number of cycles. Inspecting the graph provides the Read amount to substitute into the formula.

```
cycles for one node execution =
   CALCULATE USING  Read amount and PID formula

Read amount =
   SPECIFIED IN  the graph

PID formula =
   CHARACTERISTIC OF  the primitive
```

The node execution rate is the required frequency of the node.

```
node execution rate =
   required frequency of node

required frequency of node =
   CALCULATE USING  the Required Frequency
         Calculator
```

All the APs in a particular EMSP operate with the same cycle time. The cycle rate for an AP is a characteristic of the particular EMSP.

```
cycle rate of AP =
   CHARACTERISTIC OF  the EMSP
```

Arithmetic Processor Summary. The following operationally summarizes the AP sub-section of the Lower Bound Algorithm.

At each node, calculate the cycles needed for one execution of that node by using the Read amount in the graph and the formula in the PID. Also calculate the execution rate of that node by using the Required Frequency Calculator. Multiplying the cycles needed for one execution by the execution rate gives the cycle rate

needed for that node to execute the graph. Add that amount to a needed cycle rate counter.

After traversing all nodes, divide the needed cycle rate by the cycle rate of an AP to get the lower bound on the number of APs needed.

Figure 18 in the appendix shows an example of the computation of the lower bound on the number of APs using the graph of Figure 15, the computed required frequencies of Figure 16, and the node data of Figure 17. The example is small compared to a complete ECOS application; the computed lower bound on the number of APs for the lower bound example equals one.

The Lower Bound on Global Memories

Sub-section two identifies the lower bound on the number of Global Memories (GMs).

The lower bound on the number of Global Memories needed is the total number of memory bytes needed to store the graph divided by the number of bytes available in each Global Memory.

```
lower bound on the number of GMs =
   CEILING OF  (
      total needed memory space
      DIVIDED BY  memory space in GM )
```

A graph contains both instruction streams and data. The total memory space needed to completely store the graph is the sum of the space needed for the instruction streams and for the queues.

```
total needed memory space =
   memory space for instruction streams
   PLUS  memory space for queues
```

The total memory space needed for the instruction streams is the total of the space needed per instruction stream for all the nodes in the graph. Looking up the instruction name in the PID provides the space needed for the instruction stream in bytes.

```
memory space for instruction streams =
   SUM OVER  all nodes
   OF  space for instruction stream per node

all nodes =
   SPECIFIED IN  the graph

space for instruction stream per node =
   SPECIFIED IN  the PID
```

The total memory space needed for the queues is the total of the space needed per queue for all the queues in the graph.

```
memory space for queues =
   SUM OVER  all queues
   OF  space per queue

all queues =
   SPECIFIED IN  the graph
```

The EMSP Principles of Operations (POPS) manual defines the space for a queue to be three times the Threshold amount for the queue. For each queue, inspecting the graph to obtain the Threshold amount and multiplying that number by three gives the space per queue.

```
space per queue =
   three
   MULTIPLIED BY  Threshold amount

three =
   CHARACTERISTIC OF  the EMSP

Threshold amount =
   SPECIFIED IN  the graph
```

All the GMs in a particular EMSP contain the same number of bytes. The available memory space for a GM is a characteristic of the particular EMSP.

```
memory space in GM =
   CHARACTERISTIC OF  the EMSP
```

Global Memory Summary. The following operationally summarizes the GM sub-section of the Lower Bound Algorithm.

At each node, look up the memory space needed for the instruction stream in the PID. Add that amount to a needed memory space counter. At each queue, calculate the memory space needed for the queue by multiplying the Threshold amount by three. Add that amount to the needed memory space counter also.

After traversing the graph, divide the needed memory space by the memory space in a GM to get the lower bound on the number of GMs needed.

Figure 19 in the appendix shows an example of the computation of the lower bound on the number of GMs using the graph of Figure 15 and the node data of Figure 17.

The computed lower bound on the number of GMs for the lower bound example equals one.

The Lower Bound on Input Output Processors

Sub-section three identifies the lower bound on the number of Input Output Processors (IOPs).

The lower bound on the number of Input Output Processors needed is the total data rate needed to handle the input and output data divided by the maximum data rate obtainable from one Input Output Processor.

```
lower bound on the number of IOPs =
   CEILING OF  (
      total needed input-output data rate
      DIVIDED BY  data rate of IOP )
```

A graph receives input and produces output. The total data rate needed to completely handle the input and output data is the sum of the rates needed for the input data and for the output data.

```
total needed input-output data rate =
   input data rates
   PLUS  output data rates
```

A graph may have multiple inputs. The total data rate needed for the input data is the total of the data rates needed per input queue for all the input queues in the graph. The rate per input queue is specified in the signal processing application.

```
input data rates =
   SUM OVER  all input queues to graph
   OF  data rate per input queue

all input queues to graph =
   SPECIFIED IN  the graph

data rate per input queue =
   SPECIFIED IN  the application
```

Similarly, a graph may have multiple outputs. The total data rate needed for the output data is the total of the data rates needed per output queue for all the output queues in the graph.

```
output rates =
   SUM OVER  all output queues from graph
   OF  data rate per output queue

all output queues from graph =
   SPECIFIED IN  the graph
```

Each output queue has data produced to it by an output node which executes at a certain rate. The Produce amount of a node is the amount of data produced in one node execution. The data rate needed per output queue is its Produce amount multiplied by the number of times its output node is executed per time-unit.

```
data rate per output queue =
   Produce amount of output node
   MULTIPLIED BY  execution rate of output node

Produce amount of output node =
   CALCULATE USING  the Produce Calculator
```

The node execution rate is the required frequency of the node.

```
execution rate of output node =
   required frequency of node

required frequency of node
   CALCULATE USING  the Required Frequency
         Calculator
```

All the IOPs in a particular EMSP operate at the same maximum data rate. The data rate for an IOP is a characteristic of the particular EMSP.

```
data rate of IOP =
   CHARACTERISTIC OF  the EMSP
```

Input Output Processor Summary. The following operationally summarizes the IOP sub-section of the Lower Bound Algorithm.

At each input queue, look up the data rate needed for the input data in the application specification. Add that amount to a needed data rate counter. At each output queue, calculate the amount of data produced to it in one output node execution by using the Produce Calculator. Also, calculate the execution rate of the output node by using the Required Frequency Calculator. Multiplying the data produced in one execution by the execution rate gives the data rate needed for the output queue. Add that amount to the needed data rate counter also.

After traversing the graph, divide the needed data rate by the data rate of an IOP to get the lower bound on the number of IOPs needed.

Figure 20 in the appendix shows an example of the computation of the lower bound on the number of IOPs using

the graph of Figure 15 and the computed required frequencies of Figure 16. The computed lower bound on the number of IOPs for the lower bound example equals one.

The Lower Bound on Data Transfer Networks

Sub-section four identifies the lower bound on the number of Data Transfer Networks (DTNs).

The lower bound on the number of Data Transfer Networks is the total number of transfer cycles per time-unit needed to support the data communications divided by the number of transfer cycles obtainable from one Data Transfer Network.

```
lower bound on the number of DTNs =
   CEILING OF  (
      total needed transfer rate
      DIVIDED BY  transfer rate of DTN )
```

A graph contains many APs and GMs. The total transfer rate needed to support the data communications among APs and GMs is the total of the data transfer rates needed per node for all the nodes in the graph.

```
total needed transfer rate =
   SUM OVER  all nodes
   OF  transfer rate per node

all nodes =
   SPECIFIED IN  the graph
```

An individual node may be required to be executed a number of times to complete one execution of the graph.

Therefore, the total transfer rate needed to support a node is the transfer traffic for one node execution multiplied by the number of times the node is executed per time-unit.

```
transfer rate per node =
   traffic for one node execution
   MULTIPLIED BY  node execution rate
```

The communication traffic for one node execution is the sum of the traffic which results from the instruction stream transfers and from the data transfers.

```
traffic for one node execution =
   instruction stream traffic
   PLUS  data traffic
```

For one node execution, the communication traffic resulting from the instruction stream transfer is the size of the instruction stream. Looking up the instruction name in the PID provides the size of the instruction stream.

```
instruction stream traffic =
   instruction stream size

instruction stream size =
   SPECIFIED IN  the PID
```

For one node execution, the communication traffic resulting from the data transfer is the sum of the input data to the node and the output data from the node.

```
data traffic =
   input data traffic to node
   PLUS  output data traffic from node
```

A node may have many input queues associated with it. The total amount of input data to the node for one node execution is the total of the Read amounts per input queue for all the input queues of the node.

```
input data traffic to node =
   SUM OVER  all input queues per node
   OF  Read amount per queue

all input queues per node =
   SPECIFIED IN  the graph

Read amount per queue
   SPECIFIED IN  the graph
```

Similarly, a node may have many output queues associated with it. The total amount of output data from the node for one node execution is the total of the Produce amounts per queue for all the output queues of the node.

```
output data traffic from node =
   SUM OVER  all output queues per node
   OF  Produce amount per queue

all output queues per node =
   SPECIFIED IN  the graph

Produce amount per queue =
   CALCULATE USING  the Produce Calculator
```

The node execution rate is the required frequency of the node.

```
node execution rate =
   required frequency of node

required frequency of node =
   CALCULATE USING  the Required Frequency
         Calculator
```

All the DTNs in a particular EMSP operate at the same transfer rate. The transfer rate for a DTN is a characteristic of the particular EMSP.

transfer rate of DTN =
CHARACTERISTIC OF the EMSP

Data Transfer Network Summary. The following operationally summarizes the DTN sub-section of the algorithm.

At each node, obtain the instruction stream transfer traffic by looking up the instruction stream size in the PID. Add that amount to a needed transfer traffic counter. At each input queue of the node, obtain the input data transfer traffic by looking up the Read amount in the graph. Add that amount to the needed transfer traffic counter. At each output queue of the node, calculate the amount of the output data transfer traffic by using the Produce Calculator. Add that amount to the needed transfer traffic counter. The transfer traffic counter now contains the total of the needed transfer traffic for one execution of the node. Calculate the execution rate of the node by using the Required Frequency Calculator. Multiplying the needed transfer traffic by the execution rate gives the transfer rate needed for the node. Add this amount to a needed transfer rate counter, and zero out the needed transfer traffic counter.

After traversing all nodes, divide the needed transfer rate by the transfer rate of a DTN to get the lower bound on the number of DTNs needed.

Figure 21 in the appendix shows an example of the computation of the lower bound on the number of DTNs using the graph of Figure 15, the computed required frequencies of Figure 16, and the node data of Figure 17. The computed lower bound on the number of DTNs for the lower bound example equals one.

CHAPTER VII

SUMMARY AND CONCLUSIONS

The Enhanced Modular Signal Processor (EMSP) is an embedded real-time signal processor. Because of the nature of EMSP systems, it is critical that an EMSP system have enough resources to meet the time requirements of the signal processing application and also be as small as possible.

The present approach to finding the smallest system is based on information obtained by executing the signal processing application. The present approach requires finding a similar application even though there may be no similar application. It requires time-consuming and costly trial-and-error simulation. Further, the present approach gives no information as to how close an obtained machine configuration is to the minimal configuration.

The lower bound algorithm developed in this thesis does not require finding a similar application. The lower bound algorithm identifies the lower bound configuration using only static information from the graph, the Primitive Interface Definition (PID), the application, and the characteristics of the EMSP. An implementation of the

algorithm can traverse the graph once and obtain results quickly and inexpensively.

The algorithm requires a well-formed graph. A graph which has correct syntax, no deadlocked cycles, and only consistent node execution frequencies is a well-formed graph. Requiring a well-formed graph is reasonable; an application having a graph which is not well-formed means the solution is incorrect and cannot execute, regardless of the machine configuration.

The lower bound approach is based on the capacities of by the hardware modules, be those capacities in machine cycles for the Arithmetic Processors (APs), memory bytes for the Global Memories (GMs), data rates for the Input Output Processors (IOPs), or transfer rates for the Data Transfer Networks (DTNs). The algorithm divides the total capacity needed for the application by the capacity which can be supported by one hardware module to identify the lower bound on the number of that hardware module needed.

This dissertation shows how to use only static information from the graph, the Primitive Interface Definition (PID), the application, and the characteristics of the EMSP to identify the lower bound configuration.

Further Work

Assignment and Contention

The lower bounds identify the minimum numbers of hardware modules below which the application is guaranteed

to fail to execute. However, given an EMSP with the lower bound numbers of hardware modules, the application still may be unable to execute. In other words, the lower bound configuration may not be a minimal configuration. Further work is necessary to identify the minimal configuration. This work has at least two component parts: the assignment problem, and the contention problem.

The assignment problem is the problem that more functional units may be needed for the minimal configuration than are required for the lower bound configuration. The assignment problem arises because there may be EMSP requirements and application requirements which prohibit full use of Global Memories (GMs) and Input Output Processors (IOPs).

For example, there may be three queues which, when their space requirements are totaled, could be stored by two GMs. The lower bound algorithm will identify two as the number of needed GMs. However, the EMSP architecture requires that a queue cannot be split across GMs. Rather than needing only two GMs (the lower bound number), the EMSP needs three GMs.

There may be four sensors which, when their data rates are totaled, could be handled by two IOPs. The lower bound algorithm will identify two as the number of needed IOPs. However, the application may require that each sensor be assigned to a different IOP. Rather than needing only two IOPs, the EMSP needs four IOPs.

The two preceding examples showed instances of the assignment problem. The problem exists for assigning instruction streams and queues to GMs and assigning sensors and output devices to IOPs. The assignments for instruction streams, queues, sensors, and output devices are static assignments. The initial part of the problem is to devise suitable assignment algorithms, or an optimal assignment algorithm, for assigning sizes to modules. The subsequent part of the problem is to consider how to include those assignments when identifying the numbers of hardware modules in which the application is guaranteed to execute.

The contention problem is the problem that free Arithmetic Processors (APs) or free Data Transfer Network (DTN) paths may not be available when needed. The contention problem arises because APs and communication paths through the DTNs are assigned dynamically by the Scheduler (SCH) as the EMSP executes and thus are not candidates for static assignment.

When a node has all its operands and is ready to execute, the Scheduler (SCH) looks for a free AP. If there is a free AP, the Scheduler schedules the node to the AP. If there is no free AP, the node waits.

When a GM receives the message to send an instruction stream to an AP, it places the instruction stream onto the DTN. If there is a free path from the GM to the AP, the

DTN transfers the instruction stream. If there is no free path, the instruction stream waits.

The two preceding examples showed instances of the contention problem. The problem exists for assigning nodes to APs and for assigning communication paths through DTNs. The contention for free APs and the contention for free DTN paths are dynamic events. The initial part of the problem is to devise suitable measurements for specifying the amounts of contention. The subsequent part of the problem is to consider how to include those measurements when identifying the numbers of hardware modules in which the application is guaranteed to execute.

Suppose there is no free path from a GM to an AP. There may be no set of assignments of instruction streams to GMs which would allow contention free operation. However, there may be certain sets of assignments which would lead to less contention than other assignments. Although the assignments are static, the assignments have a dynamic effect. In other words, the assignment problem and the contention problem interact. This interaction complicates both the assignment problem and the contention problem.

Prediction

The lower bounds identify the minimum numbers of hardware modules below which an application is guaranteed to fail to execute. Often, having the ability to predict

the effect of future changes is important. The prediction problem is the problem of predicting the effects of changes.

Many signal processing applications require fault tolerance, the ability of a system to continue operating correctly even in the presence of a fault. Being able to predict the space cost of fault tolerance is important for an embedded system. Further work is necessary to identify the minimum numbers of functional units needed to provide specified levels of fault tolerance.

Unlike the present approach which gives no guidelines as to where to start modifying an existing machine so it can execute a new application, the lower bound algorithm gives a base below which it is useless to try. Still, there are many possible combinations of processors, memories, and data interconnections. Further work is needed to quantify the relationship between application functionality and the numbers of functional units.

Just as with other systems, real-time signal processing systems are modified in the field. An application may benefit from a small increase in functionality. Being able to predict the size cost implied by a small increase in functionality is important for a potential field modification. Further work is needed to quantify the relationship between incremental increases in application functionality and incremental increases in machine size.

An enclosure for an embedded system may contain some unused space, space which could hold additional functional units in preparation for field enhancements. Being able to predict which functional unit(s) would best be held in that unused space is important. Further work is needed to identify the improvements in processing power which result from increases in the numbers of functional units.

REFERENCES

[1] S. T. Allworth, Introduction to Real-Time Software Design. New York: Springer-Verlang, 1981.

[2] R. L. Glass, Real-Time Software. Englewood Cliffs, New Jersey: Prentice-Hall, 1983.

[3] J. A. Kowal, Analyzing Systems. Englewood Cliffs, New Jersey: Prentice-Hall, 1988.

[4] S. A. Ward, "An approach to Real Time Computation," in Proceedings of the Seventh Texas Conference on Computing Systems, Oct. 1978, pp. 5.26-5.34.

[5] K. G. Shin, "Introduction to the Special Issue on Real-Time Systems," IEEE Transactions on Computers, vol. C-36, no. 8, pp. 901-902, Aug. 1987.

[6] A. V. Oppenheim, A. S. Willsky, and I. T. Young, Signals and Systems. Englewood Cliffs, New Jersey: Prentice-Hall, 1983.

[7] S. D. Stearns and R. A. David, Signal Processing Algorithms. Englewood Cliffs, New Jersey: Prentice-Hall, 1988.

[8] F.S. Wong and M. R. Ito, "A Large-Scale Data-Flow Computer for Parallel Signal Processing," in Proceedings of the 1982 Conference on Circuits and Computers, Sep. 1982, pp. 590-593.

[9] E. B. Hogenauer, R. F. Newbold, and Y. J. Inn, "DDSP -- A Data Flow Computer for Signal Processing," in Proceedings of the 1982 International Conference on Parallel Processing, Aug. 1982, pp. 126-133.

[10] I. Watson and J. R. Gurd, "A Pratical Data Flow Computer," IEEE Computer, vol. 15, no. 2, pp. 51-57, Feb. 1982.

[11] K. Kronlof, J. Skytta, I. Hartimo, and O. Simula, "Performance of an Experimental Data Flow Architecture for Signal Processing," in Proceedings of the 1982 International Conference on Acoustics, Speech and Signal Processing, May 1982, pp. 695-698.

[12] K. Kronlof, I. Hartimo, and O. Simula, "Simulation of a Digital Signal Processing Architecture Based on the Data Flow Principle," in Proceedings of the 1982 IEEE International Symposium on Circuits and Systems, May 1982, pp. 1053-1056.

[13] K. Kronlof, I. Hartimo, and O. Simula, "On the VLSI Implementation of a Data Flow Processor," in Proceedings of the 1982 IEEE International Conference on Circuits and Computers, Sep. 1982, pp. 594-597.

[14] I. Hartimo, K. Kronlof, O. Simula, and J. Skytta, "DFSP: A Data Flow Signal Processor." IEEE Transactions on Computers, vol. C-35, no. 1, pp. 23-33, Jan. 1983.

[15] K. Kronlof, "Execution Control and Memory Management of a Data Flow Signal Processor," in Proceedings of the Tenth International Symposium on Computer Architecture, 1983, pp. 230-235.

[16] M. M. Jamali, G. A. Jullien, W.C. Miller, and S. I. Ahmad, "A Real Time General Purpose Signal Processor," in Proceedings of the 1984 International Conference on Acoustics, Speech and Signal Processing, May 1984, pp. 16.4.1-16.4.4.

[17] Y. S. Wu, A. G. Constantinides, T. E. Curtis, and L. J. Wu, "Architectural Approach to Alternate Low-Level Primitive Structures (ALPS) for Acoustic Signal Processing," IEE Proceedings, vol. 131, part F, no. 3, pp. 327-333, June 1984.

[18] C. B. Robbins, "Navy Real-Time Signal Processor Development: Second Generation Planned Service Standard," Real-Time Signal Processing IV, vol. 298, pp. 216-224, Aug. 1981.

[19] N. H. Brown, "The EMSP Data Flow Computer," in Proceedings of the Seventeenth Hawaii International Conference on System Sciences, Jan. 1984, pp. 39-48.

[20] F. H. Bloch, "The Enhanced Modular Signal Processor," in Proceedings of the Seventeenth Annual Pittsburgh Conference on Modeling and Simulation, Apr. 1986, pp. 829-836.

[21] R. M. Jordan, "Performance Analysis of a New Computer Architecture by Event Time Simulation," in Proceedings of the Seventeenth Annual Pittsburgh Conference on Modeling and Simulation, Apr. 1986, pp. 837-838.

[22] --, ECOS/ACOS Common Operational Support Software Methodology Specification. Washington, D. C.: Naval Sea Systems Command, 1983.

[23] --, ECOS Tutorial. Washington, D. C.: Naval Sea Systems Command, 1983.

[24] --, ECOS/ACOS EMSP/ASP Common Operational Support Software Methodology Specification, Version 3.0. Washington, D. C.: Naval Sea Systems Command, 1984.

[25] --, Enhanced Modular Signal Processor (EMSP) Principles of Operations (POPS). Washington, D. C.: Naval Sea Systems Command, 1985.

[26] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, Data Structures and Algorithms. Reading, Massachusetts: Addison-Wesley, 1983.

APPENDIX

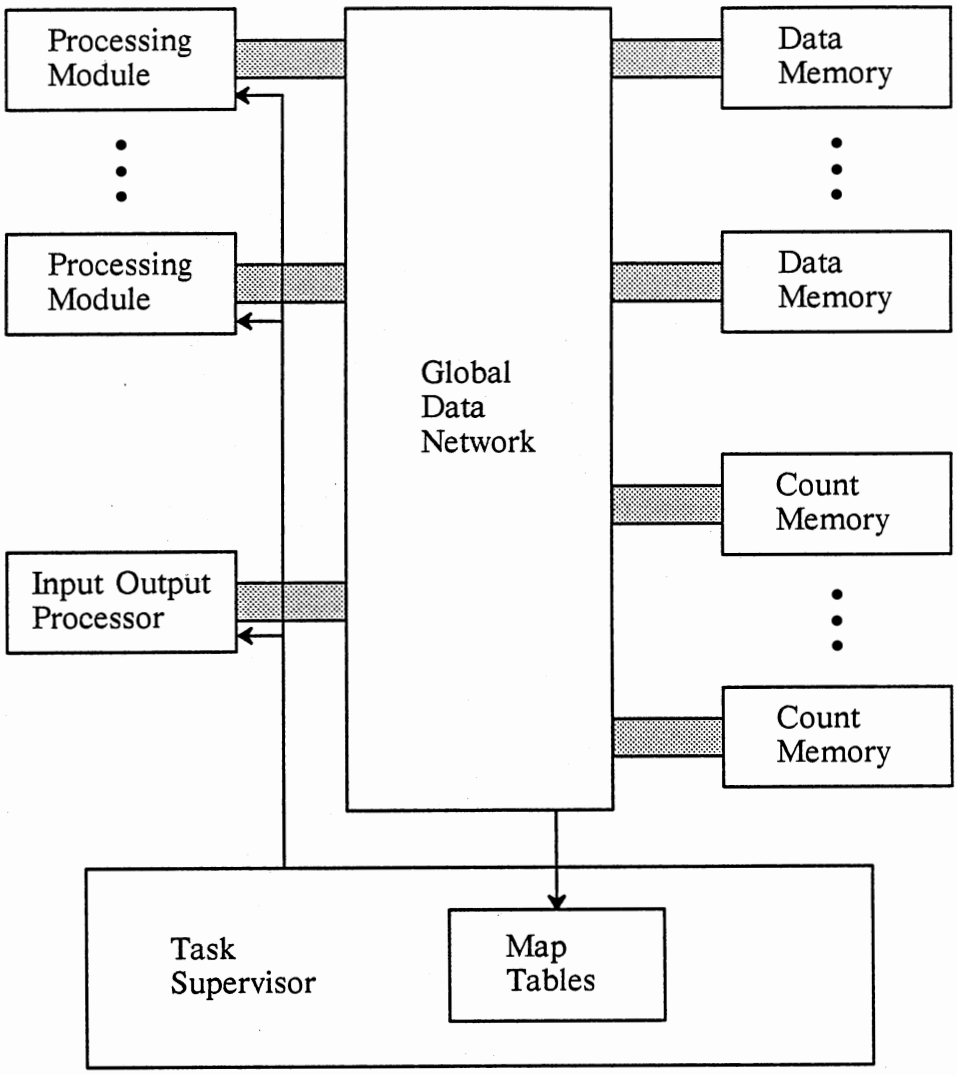


Figure 1. The Dataflow Computer

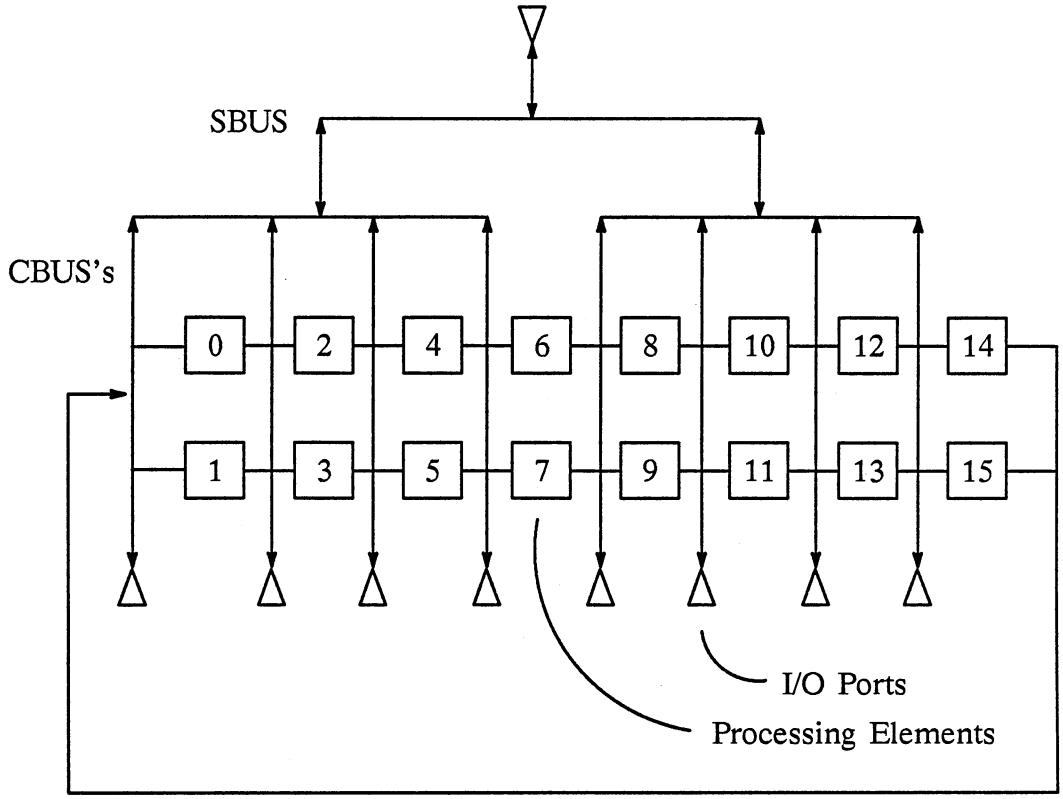


Figure 2. The Data Driven Signal Processor

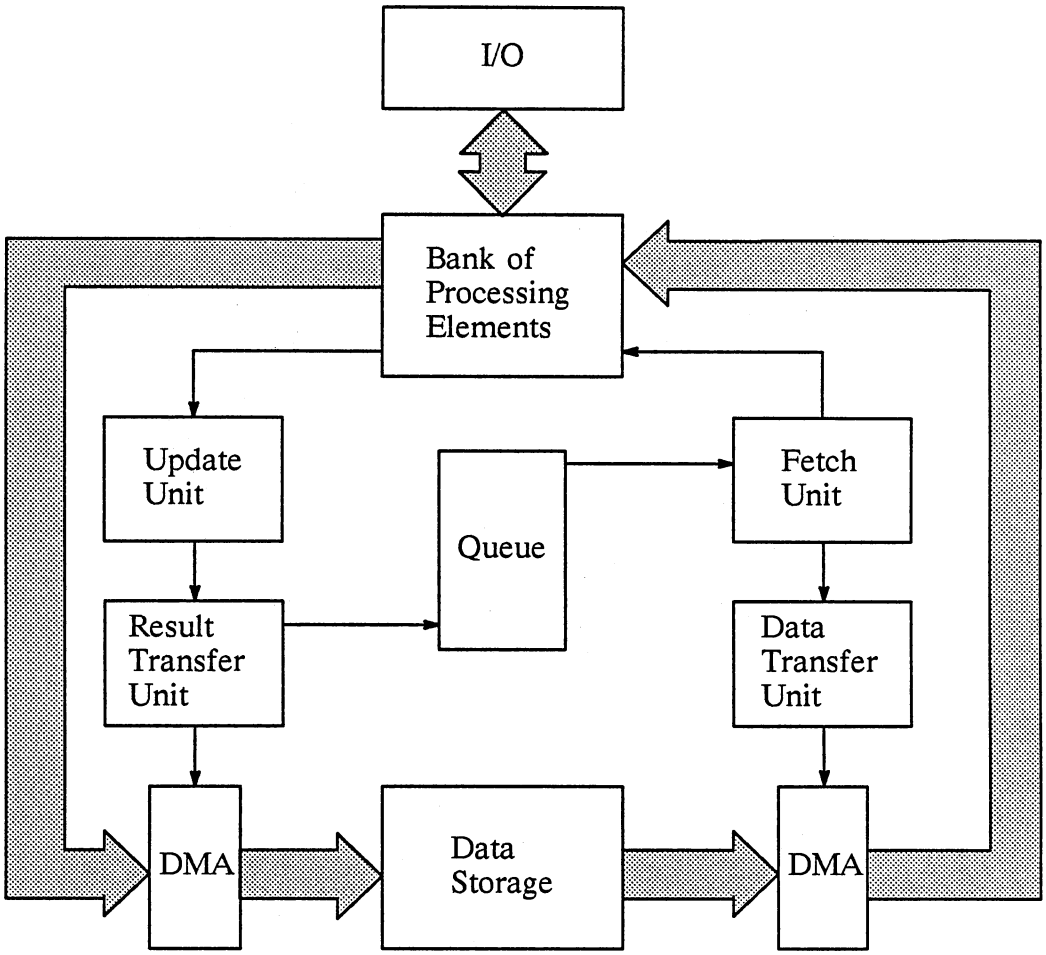


Figure 3. The Data Flow Signal Processor

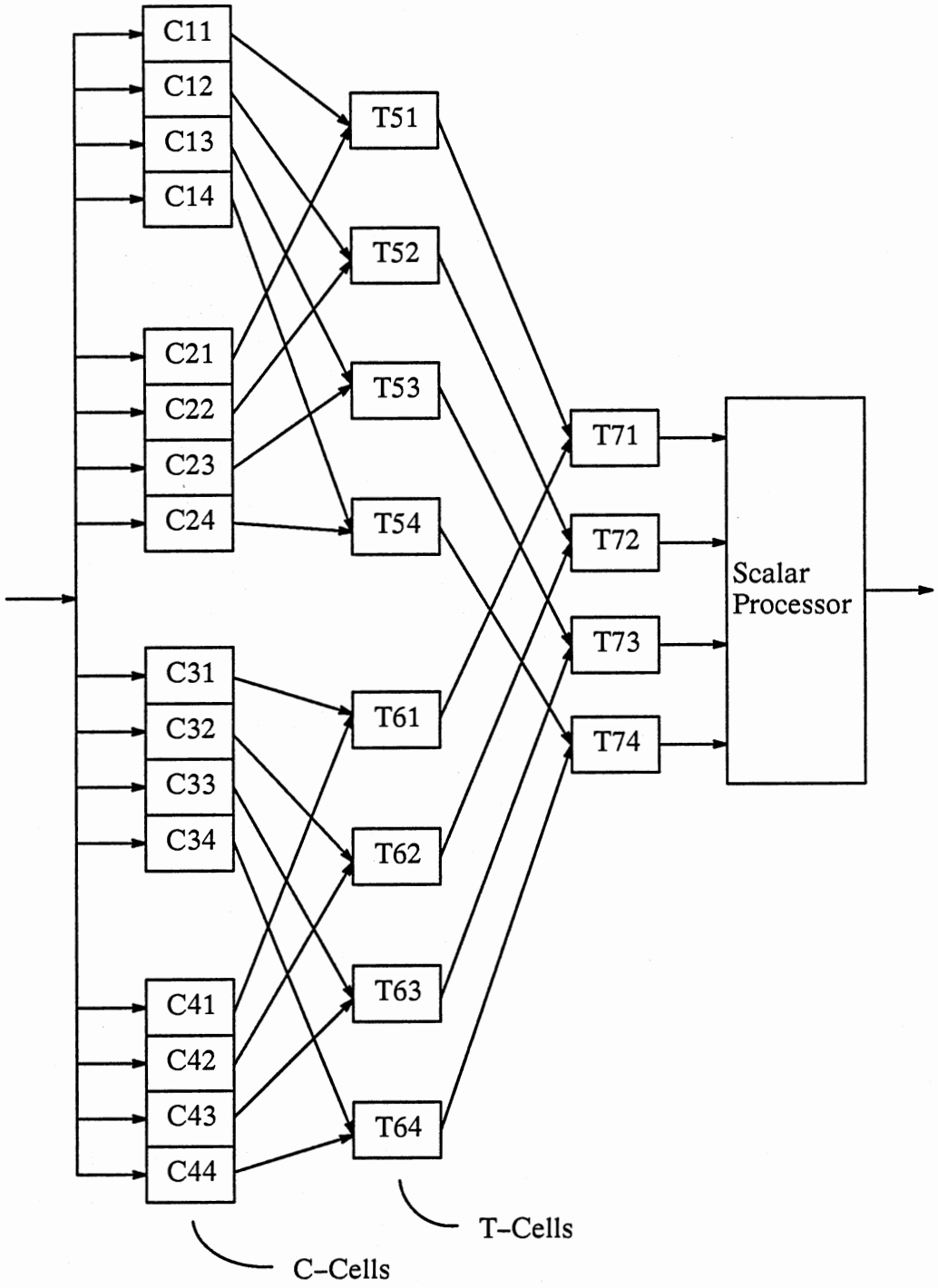


Figure 4. The Dataflow Binary Tree Processor

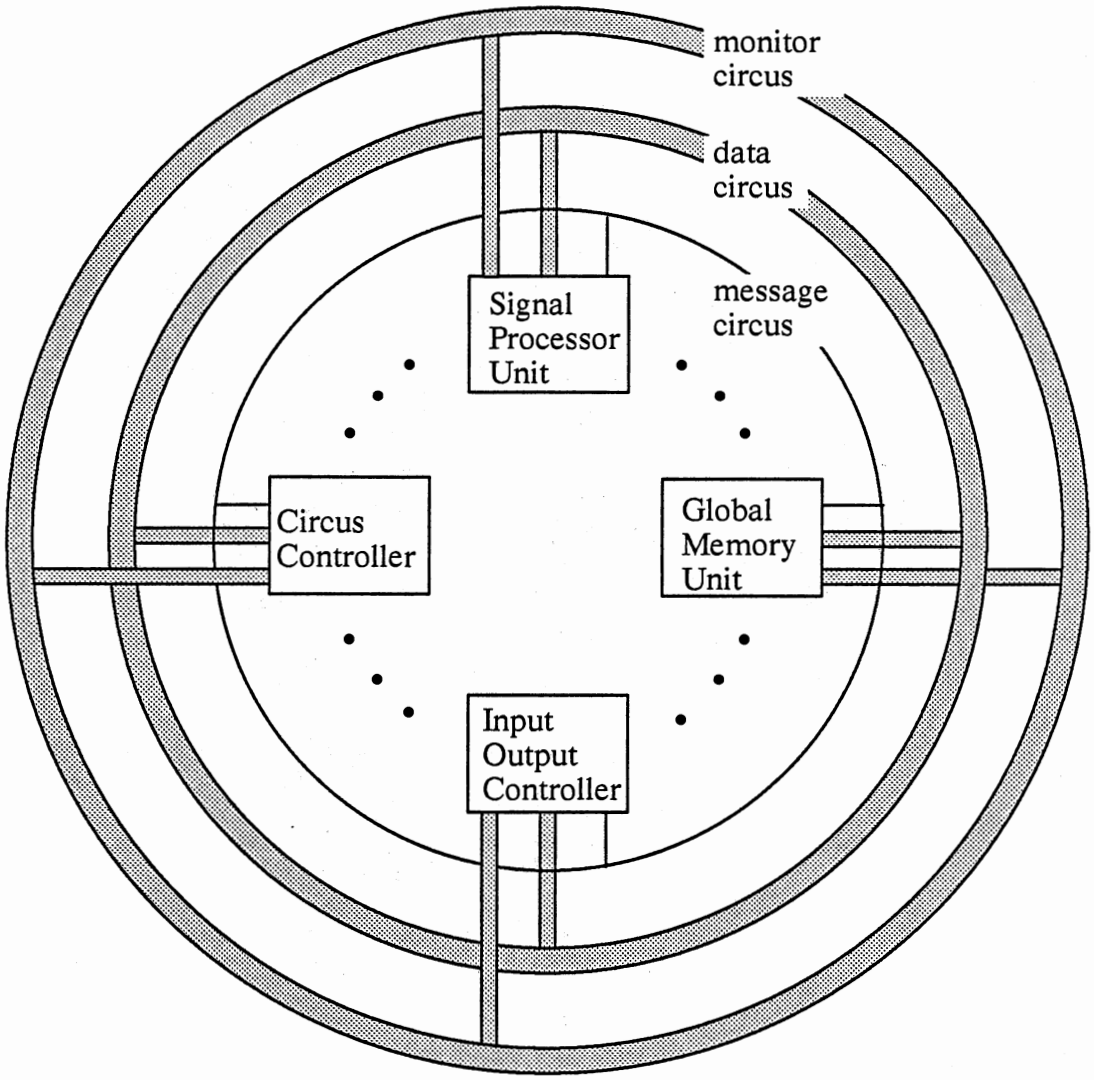


Figure 5. The Roman Circus System

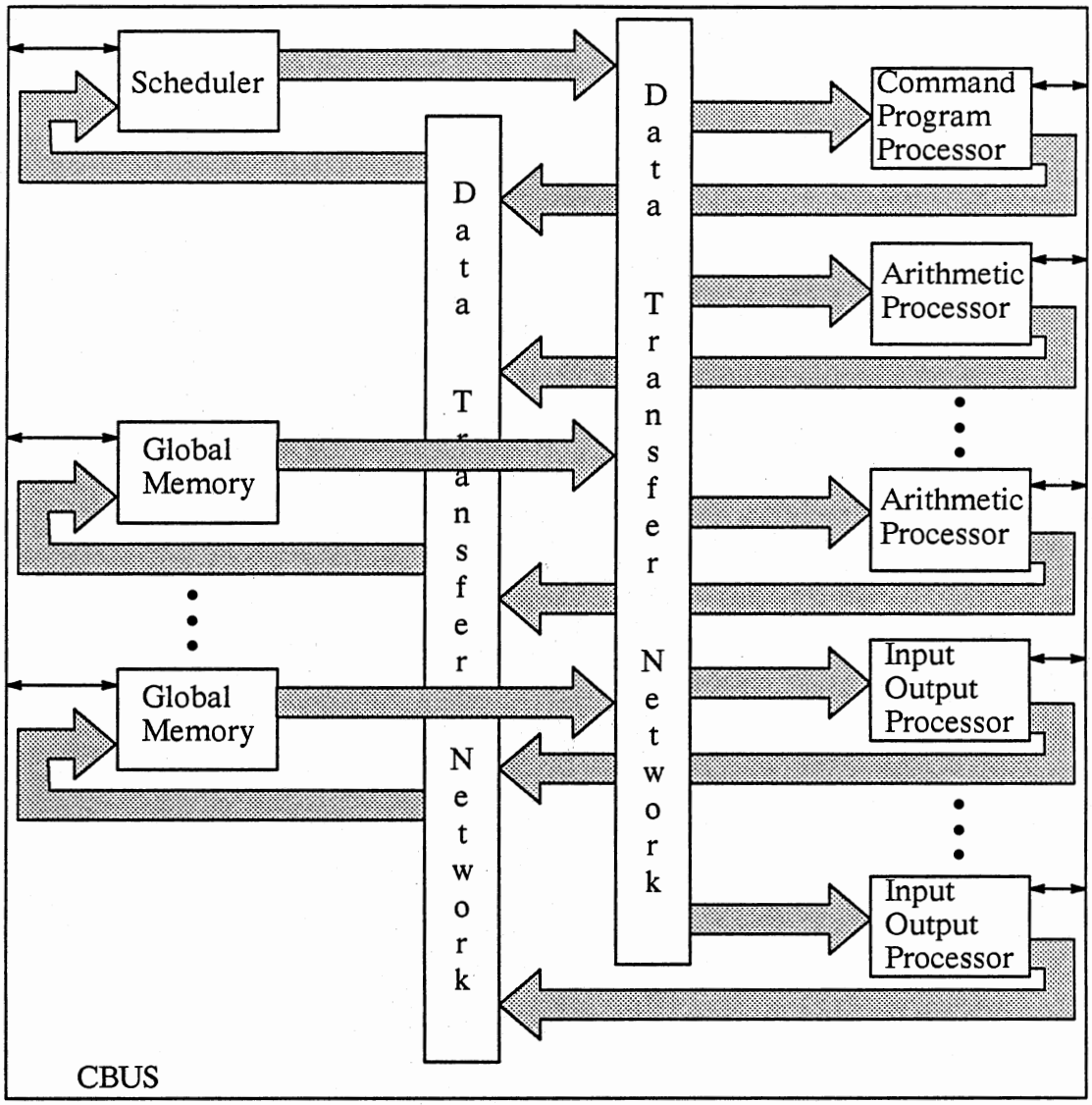


Figure 6. The Enhanced Modular Signal Processor

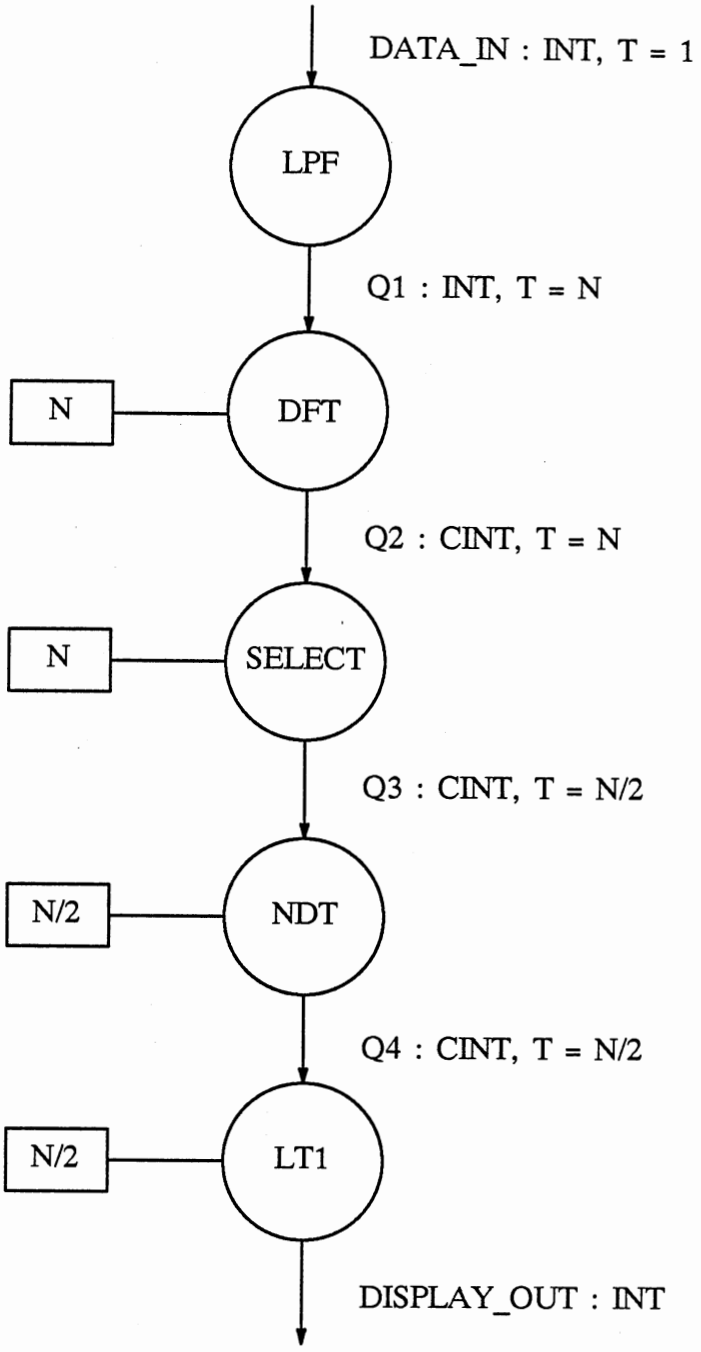


Figure 7. A Graph

```
%GRAPH ( SAMPLE_GRAPH
         GIP = N : INT
         INPUT_Q = DATA_IN : INT
         OUTPUT_Q = DISPLAY_OUT : INT )
   %% declare the internal queues
         %QUEUE ( Q1, Q4 : INT )
         %QUEUE ( Q2, Q3 : CINT )
   %% define the graph topology
   %%
         %NODE ( LPF1_NODE
                 PRIMITIVE = LPF
                 PRIM_IN = DATA_IN
                           THRESHOLD = 1
                           READ = 1
                           CONSUME = 1
                 PRIM_OUT = Q1 )
   %%
         %NODE ( DFT_NODE
                 PRIMITIVE = DFT
                 PRIM_IN = N
                           Q1
                           THRESHOLD = N
                           READ = N
                           CONSUME = N
                 PRIM_OUT = Q2 )
   %%
         %NODE ( SELECT_NODE
                 PRIMITIVE = SELECT
                 PRIM_IN = N
                           Q2
                           THRESHOLD = N
                        %% if read not specified then
                        %%    read = threshold
                        %% if consume not specified then
                        %%    consume = threshold
                 PRIM_OUT = Q3 )
   %%
         %NODE ( NDET_NODE
                 PRIMITIVE = NDT
                 PRIM_IN = N/2
                           Q3
                           THRESHOLD = N/2
                 PRIM_OUT = Q4 )
   %%
         %NODE ( LT1_NODE
                 PRIMITIVE = LT1
                 PRIM_IN = N/2
                           Q4
                           THRESHOLD = N/2
                 PRIM_OUT = DISPLAY_OUT )
%ENDGRAPH
```

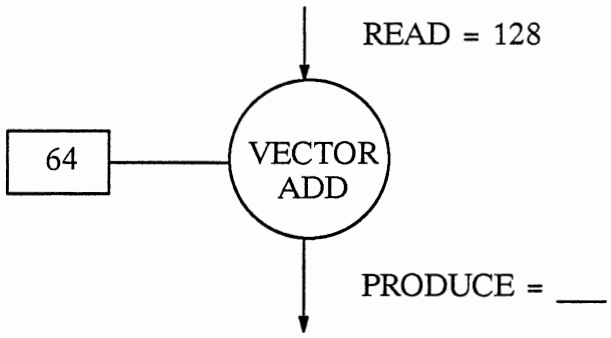
Figure 8. ECOS SPGN for the graph in Figure 7

```
%COMMANDPROG ( SAMPLE_COMMANDPROG ) ;
   %% declare the variables
         GRAPH : GRAPH_ID ;
         INPUT_Q, OUTPUT_Q : QUEUE_ID ;
         INPUT_PROC, DISPLAY_PROC : IO_PROC_ID ;
   %% create the input and output queues
         INPUT_Q := %CREATEQ ( INT ) ;
         OUTPUT_Q := %CREATEQ ( INT ) ;
   %% initialize the input and output procedures
         INPUT_PROC := %INITIO ( INPUT_PROC_NAME
                                 INPUT = INPUT_Q ) ;
         DISPLAY_PROC := %INITIO ( DISPLAY_PROC_NAME
                                   OUTPUT = OUTPUT_Q ) ;
   %% start the graph
         GRAPH := %START ( SAMPLE_GRAPH
                           GIP = 100
                        %% to be reconfigurable then
                        %%    the gip would be a control
                           INPUT = INPUT_Q
                           OUTPUT = OUTPUT_Q ) ;
   %% start the output and input procedures
         %STARTIO ( DISPLAY_PROC ) ;
         %STARTIO ( INPUT_PROC ) ;
   %% do not run off the end or you stop all your graphs
         WAIT FOREVER ;
%ENDPROGRAM
```

Figure 9. Command Program for the Graph in Figure 7

```
RECONFIGURABLE:
BEGIN
   %% create queues, procedures, and start the graph
         START_UP ;
   %% execute the graph until stop
         LOOP
            COMMAND := GET_COMMAND ;
   %% change the graph during runtime
            IF COMMAND = RECONFIGURE THEN
               RECONFIGURE_GRAPH ;
            IF COMMAND = STOP THEN
               STOP_GRAPH ;
               EXIT;
         END LOOP ;
END RECONFIGURABLE ;
```

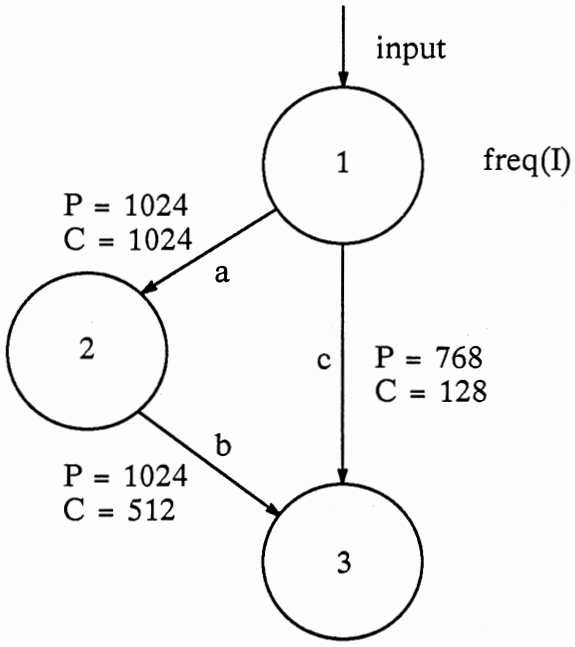
Figure 10. Command Program for a Dynamically Reconfigurable Graph



GIP of 64 tells VECTOR ADD to add 64 data points together when producing output

Therefore, PRODUCE = 2

Figure 11. Produce Calculator



f(2) = freq(I) * 1024 / 1024
= freq(I)

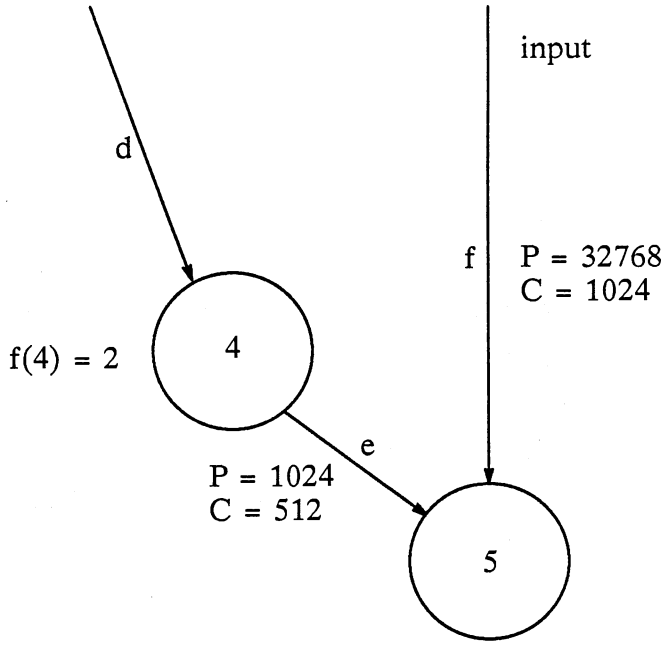
f(3)b = freq(I) * 1024 / 512
= freq(I) * 2

f(3)c = freq(I) * 768 / 128
= freq(I) * 6

f(3)b does not equal f(3)c

Therefore, there is an inconsistency in relative frequencies

Figure 12. Relative Frequency Calculator



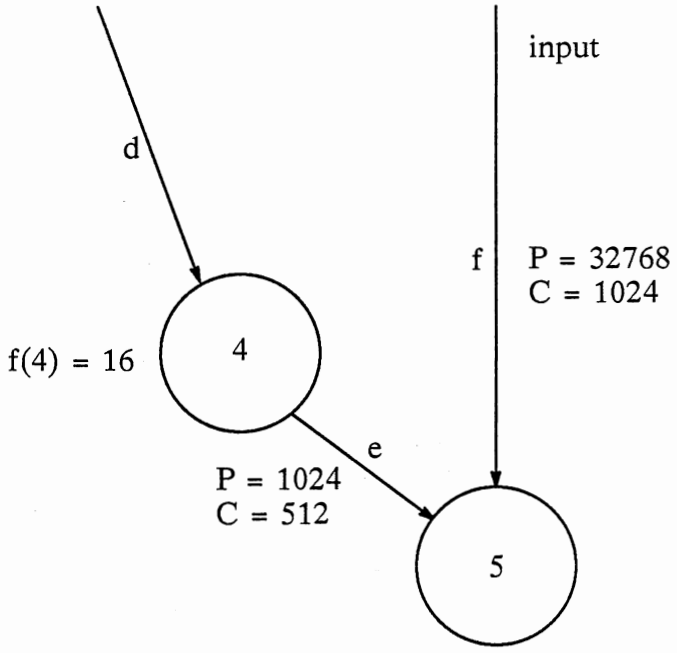
f(5)e = 2 * 1024 / 512
= 4

f(5)f = 32768 / 1024
= 32

f(5)e does not equal f(5)f

Therefore, there is an inconsistency in required frequencies

Figure 13. Required Frequency Calculator



AP executes 100,000 cycles/sec
(EMSP characteristic)

node 5 takes 4000 cycles to execute
(PID information)

maximum frequency of node 5 = 100,000 / 4000
= 25 executions per sec

maximum frequency is less than required frequency

Therefore, the graph will not execute in real-time

Figure 14. Maximum Frequency Calculator

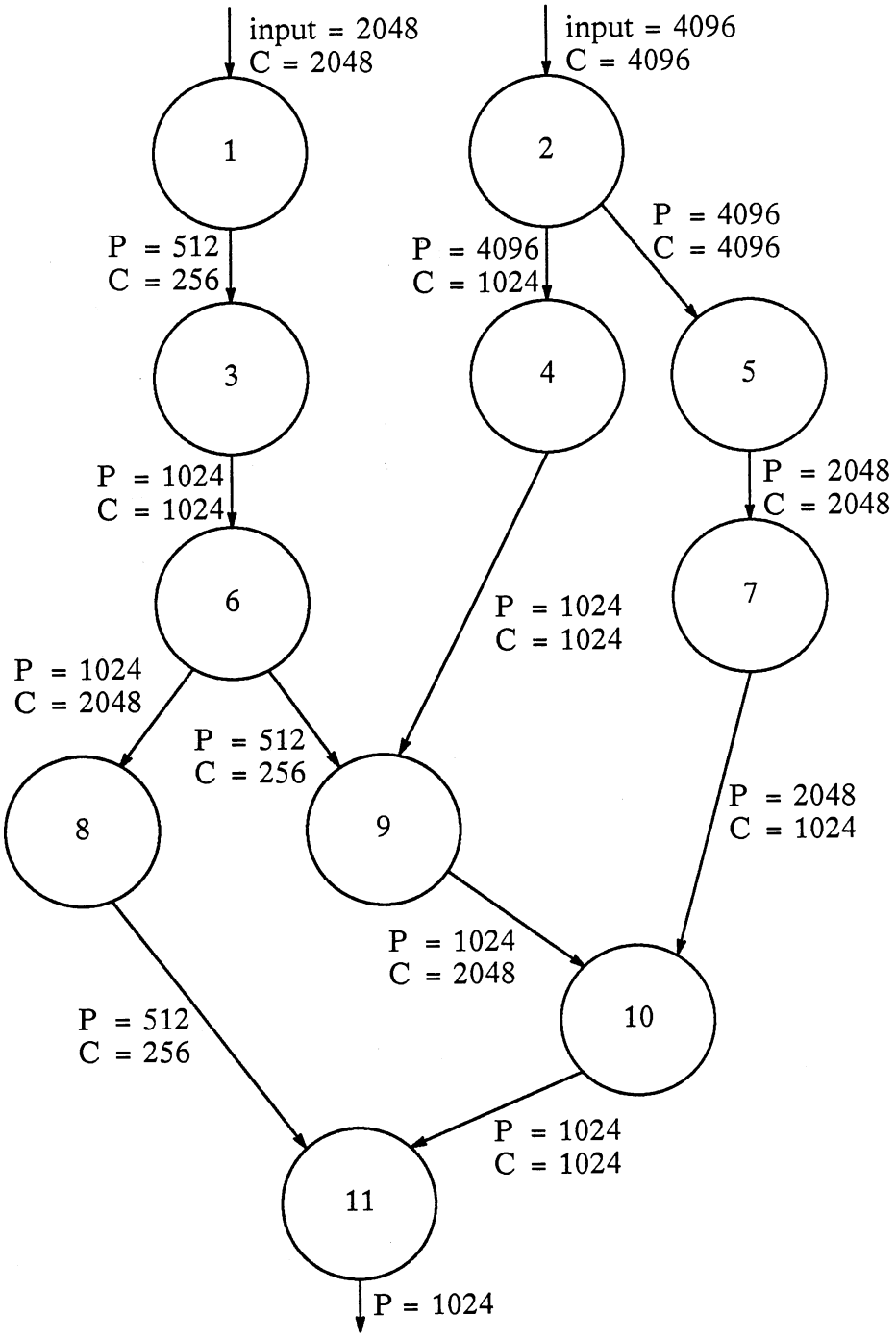


Figure 15. Graph for Lower Bound Example

Node	Required Frequency			
1	2048 / 2048	= 1	=	1
2	4096 / 4096	= 1	=	1
3	f(1) * 512 / 256	= 1 * 2	=	2
4	f(2) * 4096 / 1024	= 1 * 4	=	4
5	f(2) * 4096 / 4096	= 1 * 1	=	1
6	f(3) * 1024 / 1024	= 2 * 1	=	2
7	f(5) * 2048 / 2048	= 1 * 1	=	1
8	f(6) * 1024 / 2048	= 2 * 1/2	=	1
9	f(6) * 512 / 256	= 2 * 2	=	4
	f(4) * 1024 / 1024	= 4 * 1	=	4
10	f(9) * 1024 / 2048	= 4 * 1/2	=	2
	f(7) * 2048 / 1024	= 1 * 2	=	2
11	f(8) * 512 / 256	= 1 * 2	=	2
	f(10) * 1024 / 1024	= 2 * 1	=	2

Figure 16. Required Frequencies for Lower Bound Example

Node	Cycles for one Node Execution	Space for Instruction Stream
1	4450	6
2	5000	8
3	3900	18
4	3200	19
5	4000	10
6	4000	22
7	3800	17
8	4350	14
9	3700	14
10	4800	10
11	4450	21

Figure 17. Node Data for Lower Bound Example

```
  Total Needed Cycle Rate
-----------------------------
  required      cycles for one
  frequency     node execution
  ---------     --------------
      1     *       4450
+     1     *       5000
+     2     *       3900
+     4     *       3200
+     1     *       4000
+     2     *       4000
+     1     *       3800
+     1     *       4350
+     4     *       3700
+     2     *       4800
+     2     *       4450
-----------------------------
  83500 cycles per second
```

Cycle rate of AP = 100000 cycles per second

Lower bound on the number of APs =
ceiling (83500 / 100000) = 1 Arithmetic Processor

Figure 18. Lower Bound on the Number of APs

Total Needed Memory Space

memory space for instruction streams		memory space for queues
6		3 * 2048
+ 8		+ 3 * 4096
+ 18		+ 3 * 256
+ 19		+ 3 * 1024
+ 10		+ 3 * 4096
+ 22		+ 3 * 1024
+ 17		+ 3 * 1024
+ 14		+ 3 * 2048
+ 14		+ 3 * 2048
+ 10		+ 3 * 256
+ 21		+ 3 * 1024
		+ 3 * 256
		+ 3 * 2048
		+ 3 * 1024
		+ 3 * 1024
159	+	69888

70047 words

Memory space in GM = 262144 words

Lower bound on the number of GMs =
ceiling (70047 / 262144) = 1 Global Memories

Figure 19. Lower Bound on the Number of GMs

```
Total Needed Input Output Data Rate
-----------------------------------
input data rate   output data rate
--------------    ----------------
      2048             2 * 1024
    + 4096
--------------    ----------------
      6144       +       2048
-----------------------------------
      8192 cycles per second
```

Input output data rate of IOP = 80000 cycles per second

Lower bound on the number of IOPs =
ceiling (8192 / 80000) = 1 Input Output Processor

Figure 20. Lower Bound on the Number of IOPs

Total Needed Transfer Rate

	required frequency		instruction stream traffic		data traffic			
	1	* (	6	+	2048 + 512)			
+	1	* (	8	+	4096 + 4096 + 4096)			
+	2	* (	18	+	256 + 1024)			
+	4	* (	19	+	1024 + 1024)			
+	1	* (	10	+	4096 + 2048)			
+	2	* (	22	+	1024 + 1024 + 512)			
+	1	* (	17	+	2048 + 2048)			
+	1	* (	14	+	2048 + 512)			
+	4	* (	14	+	256 + 1024 + 1024)			
+	2	* (	10	+	2048 + 1024 + 1024)			
+	2	* (	21	+	256 + 1024 + 1024)			

65865 words per second

Transfer rate of DTN = 1048576 words per second

Lower bound on the number of DTNs =
ceiling (65865 / 1048576) = 1 Data Transfer Network

Figure 21. Lower Bound on the Number of DTNs

VITA

Arlen N. Long

Candidate for the Degree of

Doctor of Philosophy

Thesis: THE LOWER BOUND ALGORITHM FOR THE ENHANCED MODULAR SIGNAL PROCESSOR

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Fountain Hill, Pennsylvania, January 3, 1947, the son of Norman R. and Martha G. Long.

Education: Graduated from Liberty High School, Bethlehem, Pennsylvania, in June 1964; received Bachelor of Science degree in Mathematics from Moravian College, Bethlehem, Pennsylvania, in May, 1968; received Masters of Science degree in Computer Science from Iowa State University of Science and Technology, Ames, Iowa, in May 1979; completed requirements for the Doctor of Philosophy degree at Oklahoma State University, in July, 1988.

Professional Experience: Teaching Assistant, Department of Computer Science, Iowa State University of Science and Technology, Ames, Iowa, November, 1977, to September, 1979; Associate Programmer, General Technologies Division, International Business Machines Corporation, East Fishkill, New York, October, 1979, to August, 1981; Manager of Software Development, Time Management Software, Cushing, Oklahoma, August, 1981, to August, 1983; Assistant Professor, Department of Computing and Information Sciences, Oklahoma State University, August, 1983, to Present.

Professional Organizations: Member, Association of Computing Machinery.