NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

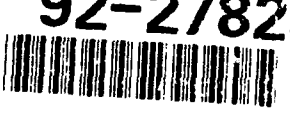
A COMPILE-TIME APPROACH FOR
CHAINING AND EXECUTION CONTROL IN
THE AN/UYS-2 PARALLEL SIGNAL PROCESSOR

by

Harold A. Bell
June, 1992

Thesis Advisor: Shridhar B. Shukla

Approved for public release; distribution is unlimited



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (*If applicable*) 32	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000	7b. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS Program Element No. / Project No. / Task No. / Work Unit Accession Number

11. TITLE (*Include Security Classification*)

A Compile-time Approach for Chaining and Execution Control in the AN/UYS-2 Parallel Signal Processor

12. PERSONAL AUTHOR(S) Bell, Harold A.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*year, month, day*)	15. PAGE COUNT
Master's Thesis	From To	1992, June	79

16. SUPPLEMENTARY NOTATION

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS (*continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUBGROUP	Digital System Processing, Compile-time Analysis, Revolving Cylinder, AN/UYS-2, Data-flow Processing, Processing Graph Methodology, Signal Processing, Scheduling, Large-grain Data-flow Architecture

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

The AN/UYS-2 represents the U.S. Navy's effort to meet the signal processing demands of the 21st century. It is programmed using the Processing Graph Methodology (PGM), where signal processing applications are represented as graphs and the nodes specify library primitives. Presently the AN/UYS-2 incorporates a First-Come-First-Serve run-time technique to allocate system resources to support large-grain data-flow execution. While this technique results in low run-time overhead, the system throughput degrades rapidly under high system load. To provide uniform output even under high load, a compile-time technique, called Revolving Cylinder (RC) analysis, is developed further to identify optimal chains and restructure the graph. It is shown by simulation that such chaining and restructuring improve the overall system performance.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS REPORT ☐ DTIC USERS	Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Shridhar B. Shukla	22b. TELEPHONE (*Include Area code*) 408-646-2764	22c. OFFICE SYMBOL EC/Sh

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Approved for public release; distribution is unlimited.

A Compile-Time Approach
for
Chaining and Execution Control
in the
AN/UYS-2 Parallel Signal Processor

by

Harold A. Bell
Lieutenant, United States Navy
B.S.E.E., Villanova, 1985

Submitted in partial fulfillment
of the requirements for the degree of

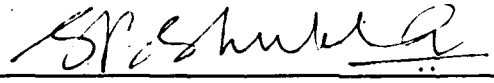
MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

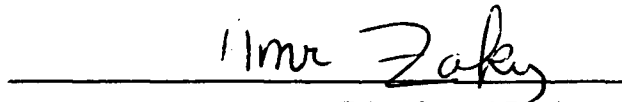
NAVAL POSTGRADUATE SCHOOL
June 1992

Author: 

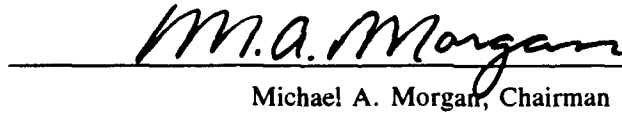
Harold A. Bell

Approved by: 

Shridhar Shukla, Thesis Advisor



Amr Zaky, Second Reader



Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

The AN/UYS-2 represents the U. S. Navy's effort to meet the signal processing demands of the 21st century. It is programmed using the Processing Graph Methodology (PGM), where signal processing applications are represented as graphs and the nodes specify library primitives. Presently the AN/UYS-2 incorporates a First-Come-First-Serve run-time technique to allocate system resources to support large-grain data-flow execution. While this technique results in low run-time overhead, the system throughput degrades rapidly under high system load. To provide uniform output even under high load, a compile-time technique, called Revolving Cylinder (RC) analysis, is developed further to identify optimal chains and restructure the graph. It is shown by simulation that such chaining and restructuring improve the overall system performance.

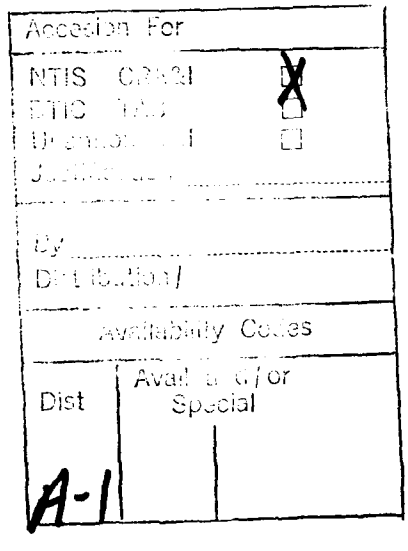


TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. THE AN/UYS-2

The AN/UYS-2 Digital Signal Processor (DSP) represents a new generation of signal processors designed to meet the needs of the U. S. Navy into the 21st Century. With weapon and delivery systems becoming more complex, the need to perform high speed calculations to counter the threat becomes more imperative. With the advantage of high speed processing, the U. S. Navy will be able to quickly detect, localize, identify, attack, and counter the threat in closer to real time fashion. The AN/UYS-2 was designed towards achieving this goal as a "Navy Standard" to be incorporated into air, sea, and shore assets that rely upon signal processing applications.

The hardware development of the machine is based upon the use of Standard Electronic Modules (SEMs) incorporating off-the-shelf processor, and state-of-the-art micro-circuit devices. Presently there are two SEM versions available: type "B" and type "E". The "E" format module differs from the "B" in it is lighter, smaller, and has improved power performance. The development of the "E" version was driven by these factors for implementation in aircraft. [RICE 90, pp.2]

The SEMs are used to build autonomous and asynchronous functional elements (FEs). Currently six FE types exist.

They are the Arithmetic Processors (APs), Global Memories (GMs), Scheduler (SCH), Command Program Processor (CPP), and Input/Output Processors (IOPs), and the Input Signal Conditioner (ISC). The AP executes signal processing primitives, GMs provide data storage and execute memory management functions, the SCH performs node scheduling, the CPP acts as an overall control unit along with user interface, IOPs provide formatting of input and output data and buffering, ISCs are equivalent to IOPs with the addition of input signal conditioning and generating output for sensor control.

Communication between FEs is supported by the Control Bus (CBUS), and the Data Transfer Network (DTN). The CBUS provides a transfer medium for control messages, while the DTN provides a transfer medium for the movement of data queues between FEs. The AN/UYS-2 architecture is given in Figure 1.1. [RICE 90] The details of its operations are described in [POPS 90] and [RICE 90].

The AN/UYS-2 is programmed using the Navy-sponsored Processing Graph Methodology (PGM) with Signal Processing Graph Notation (SPGN). Since the AN/UYS-2 carries out large-grain data-flow execution, computations are best represented as a graph. Using PGM, the application to be implemented is developed as a set of graphs.

The graphs make use of pre-defined signal processing functions called primitives from a library [PRIMLIB 90].

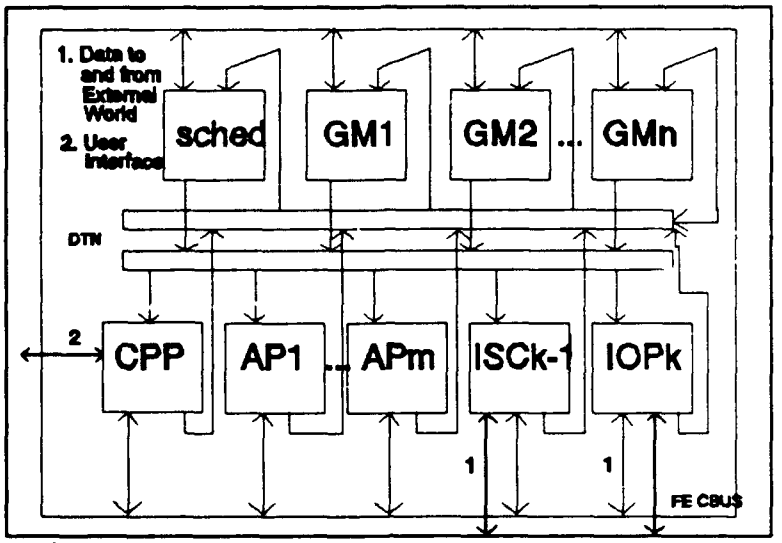


Figure 1.1 The AN/UYS-2 Architecture

Figure 1.2 shows a simple graph description.

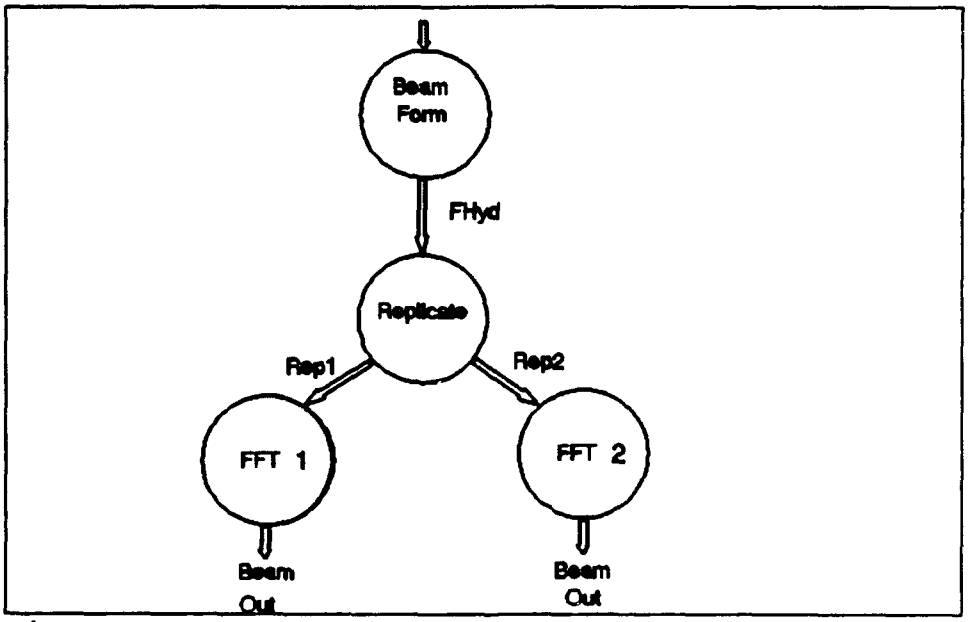


Figure 1.2 A Sample PGM graph

Circles represent nodes, the basic signal processing entity of PGM. Nodes in turn represent primitives. Arrows represent

first-in-first-out (FIFO) data queues, which provide logical data storage and transfer medium between nodes. A graph is executed when the user, through the CPP, invokes it. [RICE 90] With the application invoked, multiple instances will be executed dependent upon input data rate and data flow.

The AN/UYS-2, also known as the Enhanced Modular Signal Processor (EMSP), is currently operational in the acoustic system onboard the P3-C "Orion" aircraft, the BSY-2 Sonar on the SSN-21 "Seawolf" class submarine, and has been proposed for the S3-B "Viking" aircraft electronics upgrade. All of the above shall remain operational into the 21st Century, and will be required to perform their missions to the utmost in an everchanging political environment.

B. OBJECTIVES

The AN/UYS-2 presently uses the First-Come-First-Served (FCFS) strategy of scheduling an application's nodes to a processor. Under high loads, the system can become congested with data, resulting in a degradation of throughput. The FCFS strategy does not yield any execution control at run-time and is difficult to analyze in terms of performance, because of data flow. Since DSP applications are very specific in terms of the amount of computation required on each node, they lend themselves to compile-time analysis easily. However, presently, there exist no design tools to optimize the machine

performance using such analysis in order to ensure the required throughput.

In this thesis, a compile-time strategy is developed to analyze a graph and determine if and how any performance improvement is possible. In particular, chains of primitives are identified and are then incorporated by restructuring the original graph.

A set of algorithms is developed that builds upon and refines the revolving cylinder (RC) technique of controlling node execution sequence to enhance system performance. [SLZ 92]

1. Scope of the Thesis

In [LIT 91] the RC approach to determine node execution sequence is first suggested to enhance the system throughput. This thesis proposes the refinement of the RC approach and includes

- Critical analysis of previous work [LIT 91]
- Development of a graph preprocessor
- Determination of chains in a graph based on RC analysis, and
- Determination of dependencies to order node execution

C. ORGANIZATION

Chapter II describes the RC approach and the previous work accomplished in the area. Improvements that can be incorporated are pointed out, and the flow diagram for a graph

preprocessor is given. Chapter III describes the chaining of graph nodes, the issues related to chaining, in general, and within the RC framework. The chapter is concluded with the presentation of a chaining algorithm that meets the design specifications. Results of executing an unchained graph using FCFS and RC scheduling are compared with the chained graph executed similarly. Chapter IV discusses execution control within the AN/UYS-2 data-flow architecture and how it can be achieved based on the RC scheduling approach. The creation of dependencies is described. The chapter ends with a discussion of implementation of the artificially created dependencies on the machine. Chapter V draws conclusions from the work accomplished along with suggestions for future work to be done.

II. BACKGROUND

A. REVOLVING CYLINDER ANALYSIS

In [LIT 91], a technique for analyzing a PGM graph is presented. This technique, called the Revolving Cylinder (RC), performs compile-time analysis and results in the restructuring of the graph. The restructuring is determined assuming a given AN/UYS-2 configuration and application graph. First we describe briefly the RC approach.

The cylinder refers to a logical data structure created in a systematic approach to determine whether a given graph can be mapped on to a given number of processors and satisfy the required data rates. It is constructed by summing the execution times of all nodes and dividing it by the number of APs present. The result corresponds to the maximum throughput supportable by the machine and defines the circumference of the cylinder. The reason why this quantity is called circumference becomes clear when it is linked to the periodic arrival of data and the resulting periodic invocation of the graph. Each AP is given a portion on the cylinder equal to a band of unit width. This is equivalent to holding the cylinder with its axis horizontal and slicing it vertically into bands of unit width. Each band represents an equal partitioning of work to be accomplished. Further, each band

of the cylinder is divided into slots of a width equal to the smallest node size. In Figure 2.1 a simple PGM graph is depicted, where the numbers represent execution times. The cylinder corresponding to this graph is given in Table 1 [LIT 91]. Each node of the graph occupies a portion of the cylinder equal to its execution time. No nodes are preempted and only one node is assigned in a slot.

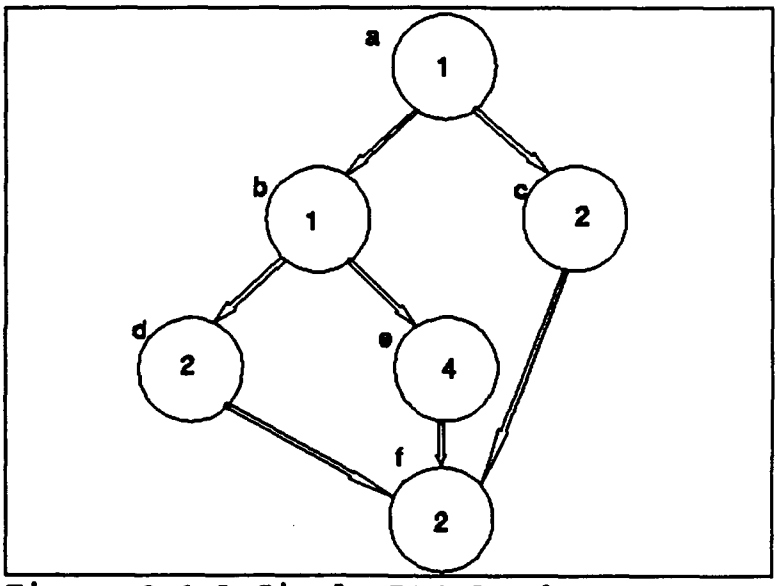


Figure 2.1 A Simple PGM Graph

The cylinder constructed as above, represents a schedule for one instance of the PGM graph. Another instance of the graph can be overlapped with the first instance after six clock cycles. To preserve the correctness of the graph when instances are overlapped, nodes are assigned indices. For the example presented in Table 1, e_1 cannot execute at the same time as a_1, however, e_0 can. With the incorporation of indices

and the divide-by-circumference nature of the node to cylinder successive instances of a graph can be assigned without conflict. [SLK 92, pp.4]

Table I: RC ASSIGNMENT FOR A SAMPLE PGM GRAPH

Cycle # (i ≥ 1)	AP1	AP2
6i - 5	a(i)	e(i-1)
6i - 4	b(i)	e(i-1)
6i - 3	c(i)	f(i-1)
6i - 2	c(i)	f(i-1)
6i - 1	d(i)	e(i-1)
6i	d(i)	e(i-1)

Cylinder assignment of nodes is not enforced strictly, rather, it is enforced by restructuring the original graph by inserting dependencies. The dependencies are used to enforce node execution order to optimize throughput. The dependencies between nodes are determined based upon their location in the cylinder along with their relationship to each other in the original graph.

Each dependency consists of a source node and a destination node along with an initial number of data items that is initialized based on the nodes' relative positions on the cylinder. These dependencies carry tokens instead of

data. At run time, tokens are produced and consumed when the node that is a source or destination of a dependency is executed.

When the number of tokens is greater than the threshold, the scheduler is informed that the node is ready for execution provided that data requirements for the node are also met.

1. Potential of the RC Analysis

The RC technique of restructuring the application provides an improvement over the presently used FCFS scheduling in that the dependencies enforce node execution order to provide more uniform throughput. The FCFS method makes uniform throughput unachievable because the nodes receiving external data are ready for execution independent of the status of other nodes in the graph. If external data arrives more frequently than the execution frequencies of the lower nodes, they fall behind the upper nodes of the graph. This results in the upper nodes' output queues going over capacity, preventing them from entering the ready node list. [POPS 90]

Another benefit of the RC approach is that, by inspecting the projected execution trace represented by the cylinder assignment, optimization schemes such as chaining of nodes can be realized automatically, instead of manually, to reduce node setup and breakdown overhead. With the chains

realized, data queues can be combined as well to reduce GM contention.

With the construction of the cylinder and assignment of nodes to specific AP's, the execution order can be enforced strictly or loosely. The preferred approach is to run the system in a *fully dynamic* mode, that is, the nodes are scheduled at run time only. When all input for a node is available, the node is assigned to a free processor. The cylinder in this mode is used as a compile-time data structure that is not used during run-time. The RC analysis and subsequent restructuring simply enhance the fully dynamic mode.

Enforcing the cylinder strictly would make the system *fully static*. With the AP determined for the execution of each node in the cylinder and the exact time to begin processing of the nodes given by the cylinder, it can be enforced directly by the scheduler to yield the fully-static mode. With the presence of a dedicated processor to schedule nodes in the AN/UYS-2, running an application in the fully-static mode is unwarranted. This mode of operation is usually limited to machines that do not have a dedicated run-time scheduler.

The dependencies determine execution order for the nodes. With all AP's assumed to be identical, it becomes unnecessary to assign a specific processor to a specific node. If a node is ready for execution, it is assigned to the first

available processor. This reduces the amount of time a node waits for a processor in the fully static case, and provides flexibility and optimal utilization of system resources. [LB 90, pp.334]

2. Refining the Current RC Implementation

The implementation of the RC approach can be improved in the following manner.

The first improvement is in the area of node assignments to the cylinder. In order to minimize the time a node waits to be placed on the RL due to dependencies, the number of tokens required should be kept minimal. In order to accomplish this, the size of the cylinder must be kept to a minimum. In the previous work, the circumference of the cylinder was doubled when there was insufficient space on the cylinder for a node to be assigned, and the assignment was started with the first node of the graph down to the last one. With this scheme, the last node to be assigned may be sufficiently large to cause the cylinder to double in order to accommodate that final node. An improvement to the assignment of nodes to the cylinder is proposed which assigns nodes based on their execution time, and only increases the size of the cylinder by incremental slot sizes until adequate space in the cylinder is available to fit the next node. When the circumference of the cylinder is derived, the execution time of the largest node is stored, and as nodes are considered for

assignment to the cylinder the largest nodes are assigned first. With all nodes equal to this size assigned, the next largest node size is then considered. This process continues until all nodes of the graph are placed within the cylinder.

In the previous work with RC analysis, the advantages of chaining were mentioned; however, any chaining of nodes that had been accomplished was done by hand prior to assignment of nodes to the cylinder. This represents the second improvement. When chaining was done by hand combining of nodes took place regardless of how they were assigned to the cylinder and without regard to how the chain might increase the size of the cylinder. An obvious improvement here is a chaining algorithm that identifies chains and assigns them to the cylinder within the limits of its existing size. Chains are, therefore, identified to produce optimal throughput without increasing dependency token queue size.

Finally, node assignments were previously based on an earliest start time of a node. This means that nodes that were ancestors were moved until their respective predecessors had completed processing. If a node's earliest start time could not be met, the index of the node was decremented indicating a previous instance of that node. This resulted in node indices varying by only one. This was achievable for all nodes because of the large cylinder size, and resulted in dependencies only being created between nodes with equal indices. By attempting to meet a node's earliest start time,

the assignment of dependencies is not optimal since it results in a large cylinder and token size. A proposed improvement for index assignment for nodes on the cylinder is that the index can vary by more than one, and the assignment of that index is determined by examining the graph and position of a node in the cylinder with reference to its parents and children.

Another pitfall to earlier determination of dependencies was that they were not allowed if one node was an ancestor of another, which made the assumption that the data dependency would compensate for the lack of an artificial dependency. A modification to improve upon this would be to assign the dependency regardless of ancestor status. The key here is to control execution independent of data status and optimize the throughput.

B. STRUCTURE OF A GRAPH PREPROCESSOR

All the improvements described above are incorporated in a graph preprocessor. The first step in the preprocessor, as depicted in Figure 2.2, is determining the chains that are within the input graph. Once the chains have been determined by the chaining algorithm, the user must incorporate them manually, and run the chaining algorithm multiple times, if required, until no chains are output. The reason for multiple executions is that after the nodes are chained, a new cylinder

assignment may result in an appearance of previously unforseen chains due to a different assignment of nodes to the cylinder.

With all chains incorporated in the graph, the dependencies are then determined.

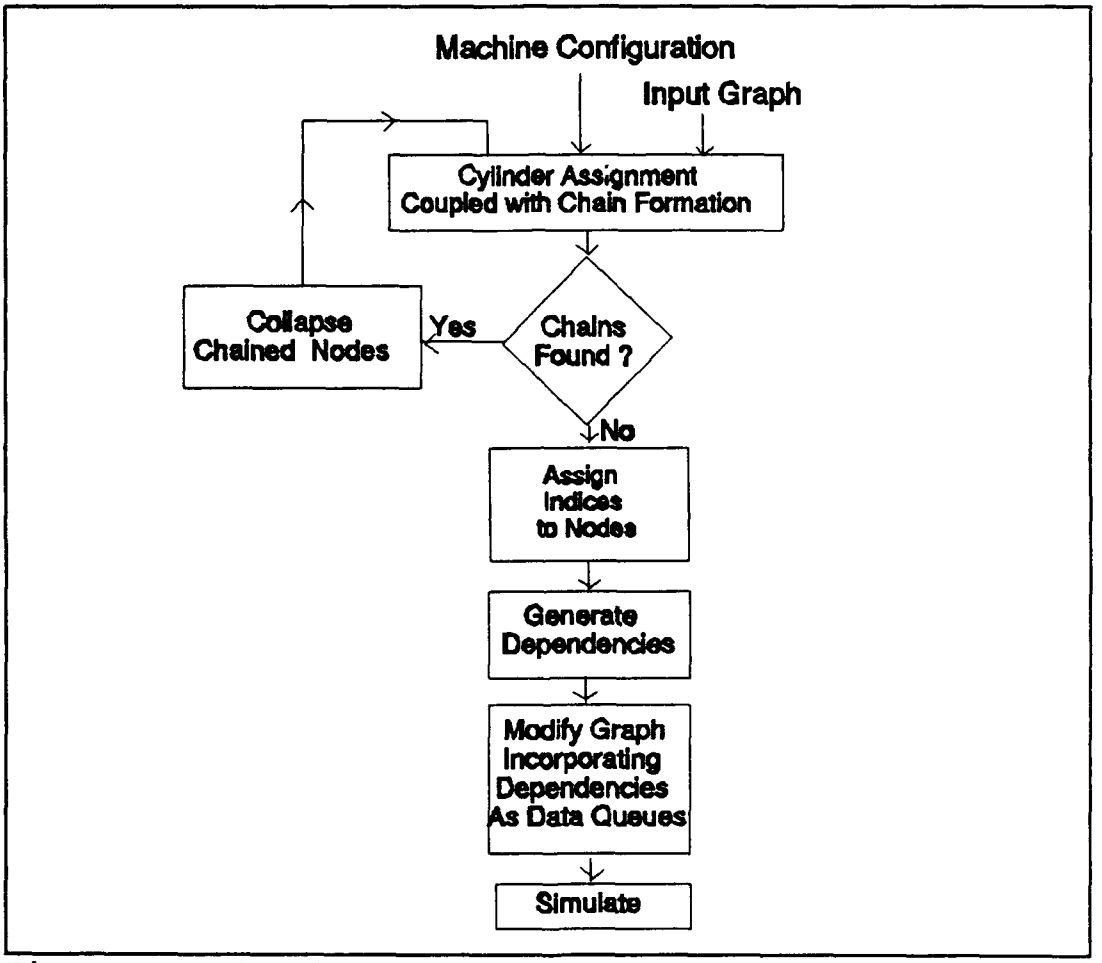


Figure 2.2 Structure of Graph Preprocessor

The more the nodes that can be combined, the fewer the dependencies. Using the output of the dependency algorithm, the graph is again restructured. The dependencies are to be

incorporated as additional data queues whose input and output nodes are designated based on the algorithm.

The structure of each of the individual components of the preprocessor is described in the remaining chapters of this thesis.

III. CHAINING OF SIGNAL PROCESSING PRIMITIVES

A. THE CHAINING PROBLEM

The possibility of chaining for a given graph represents an easily exploitable performance improvement due to the large-grain nature of the AN/UYS-2. However, the problems associated with chaining are not obvious. Assume that the input graph is simply a sequential execution of primitives. This would lead to a single chain of all nodes in the graph. With this chain, all the primitives must be executed on one AP, thereby eliminating the advantages of a multi-processor machine. We note here that the AN/UYS-2 distributed operating system allows only one copy of a node to be executing at any time.

Another difficulty encountered in determining chains is that, if a node's structure is such that more than one input or output queue is associated with it, when the node has completed execution all queues must be written or read. This produces the problem of determining all applicable queues and their locations and relationships to other nodes in the graph. The queues are not directly related.

The question also arises as to which node to begin chaining from. Input and output nodes cannot be used for chaining since they must provide formatting and conversion of

external data. Starting with the smallest node would seem to be a reasonable approach since the ratio of execution time to setup and break-down times would yield a large improvement. However, the chain still must not contain multiple input or output data queues. Starting the chain at the largest node may under-utilize other processors if the chain becomes too large.

Finally, although a chain may be inherently obvious when the graph is visually examined, its size may be too large for the local memory of the APs to process.

All of these questions and difficulties are addressed in the chaining algorithm developed. It will be shown that the RC technique readily lends itself to the determination of optimal chains.

1. Chaining in the RC Context

Given a graph to be executed in the RC context the graph's node execution times are first summed, excluding input and output nodes. The sum is then divided by the number of AP's. This number is equal to the circumference of the cylinder. Examining the correlator application for the AN/UYS-2, depicted in Figure 3.1 [ECOS 89], we can see that the node execution times vary from five thousand to one hundred thousand microseconds. Previously, the slot size was determined using the smallest node size, but as can be seen in the graphical representation of the correlator graph, node

twelve has an execution time is seventy five hundred. This node would require two cylinder entries with half of one actually empty.

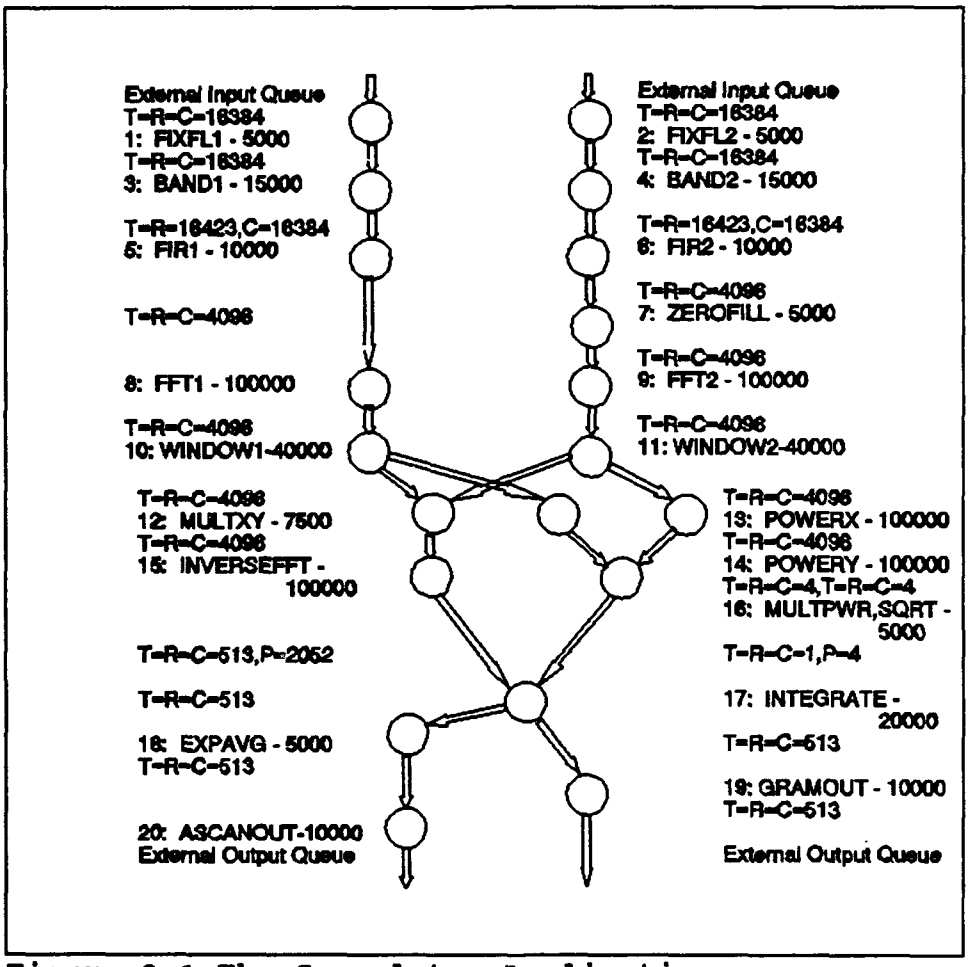


Figure 3.1 The Correlator Application

This situation represents the first computational problem, called graph partitioning, that needed to be addressed. The chaining algorithm corrects this problem by computing the greatest common divisor (GCD) of all nodes. This value is then used as the slot size of the cylinder, and

with the GCD computed, the circumference of the cylinder is then adjusted to be a multiple of the GCD.

A second computational difficulty is the packing of the cylinder while keeping the circumference at a minimum size for node execution. If the graph application is short and many AP's are available for processing, the cylinder becomes short and wide. Here, the problem of packing the cylinder becomes apparent if one node is considerably larger than the others. The cylinder is increased until the largest node can be fit onto one AP. It is assumed the circumference is never less than the largest node's execution time. This case rarely should occur but it does need to be handled appropriately.

If the cylinder is constructed where the largest node is not equal to the circumference, the problem of node assignment can be handled by simply starting with the top of the graph and moving to the bottom. Handling of assignments in this manner benefitted from simplicity, but if the bottom nodes were large the cylinder size would have to be increased unnecessarily. A more efficient packing algorithm was developed that assigns largest nodes first, and as these nodes are assigned to the cylinder, the next largest node size is determined for next assignment. In this way, the smallest nodes are assigned last and, if the cylinder size does need to be increased, it is done in small increments.

With the improvements to the cylinder consolidated the complete cylinder for the correlator application is depicted in Figure 3.2. The actual code that implements the refinement of the cylinder assignment is given in Appendix A.

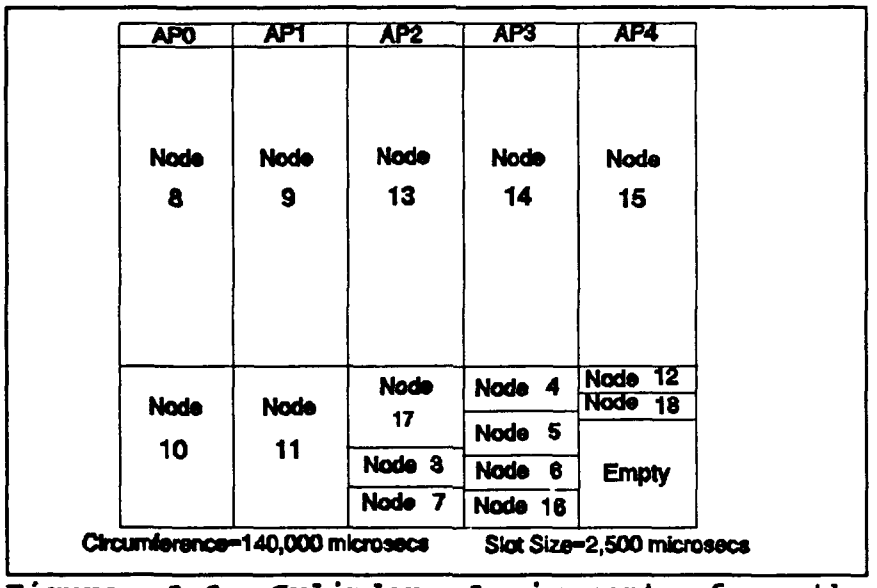


Figure 3.2 Cylinder Assignment for the Correlator

B. THE CHAINING ALGORITHM

1. Parameters of the Algorithm

The input for the algorithm was made to be compatible with the NPS simulator developed and is depicted in Figure 3.3. Only the key elements of the PGM representation are read in, that is, node information along with queue information. Column header should be ignored and the required data should be in an ASCII file "graph". [LIT 91, pp.32]

After the graph is read in and the circumference is properly determined, the slot size is set to the GCD of all nodes.

```
9                    Constitutes the total number of queues
Queue Node  Node  Arrival Threshold Production OverCapacity
  ID   In    Out  Period   Value      Value        Value
  1    -1    a      6       1024       1024         8192
  2     a    b      0       1024       1024         8192
  3     a    c      0       1024       1024         8192
  4     b    d      0       1024       1024         8192
  5     b    e      0       1024       1024         8192
  6     c    f      0       513        513          4096
  7     d    f      0       1024       1024         8192
  8     e    f      0       1024       1024         1024
  9     f   -1      0       1024       1024         1024
6                    Constitutes the total number of nodes
Node  IOP  AIS Execution Number Input  Number Output
 ID  Node Size   Time    of In Queue  of Out Queue
                         Queues  ID   Queues   ID
 a     1   256     1       1      1      2     2 3
 b     0   256     1       1      2      2     4 5
 c     0   256     2       1      3      1     6
 d     0   256     2       1      4      1     7
 e     0   256     4       1      5      1     8
 f     1   256     2       3      7 8 6  1     9
```

Figure 3.3 Input Format for the Chaining Algorithm

With the GCD determined, it must be ensured that a slot size as large as possible is used. If the nature of the graph is such that the slot size is one, large amounts of memory will be required and the program may terminate execution abnormally. To avoid this situation a manual smoothing of execution times may be essential. Although this smoothing may result in wasted cylinder space, the overall effect on the algorithm is expected to be minimal.

Mapping of the graph nodes to the cylinder is accomplished by assigning all the largest nodes first. Once the node to be assigned is established, the algorithm examines the number of input queues and output queues. Based upon this information, it attempts to chain down the graph, up the graph or down and then up. In Figure 3.1, the actual correlator application for the AN/UYS-2 was shown in PGM form prior to determination of any chains. Figure 3.4 shows the cylinder assignment as identified by the algorithm for the correlator graph with the chains incorporated. Comparing Figure 3.2 and Figure 3.4, it should be noted that the node execution sizes are not to scale, and the sum of all empty cylinder slots in both figures is equal.

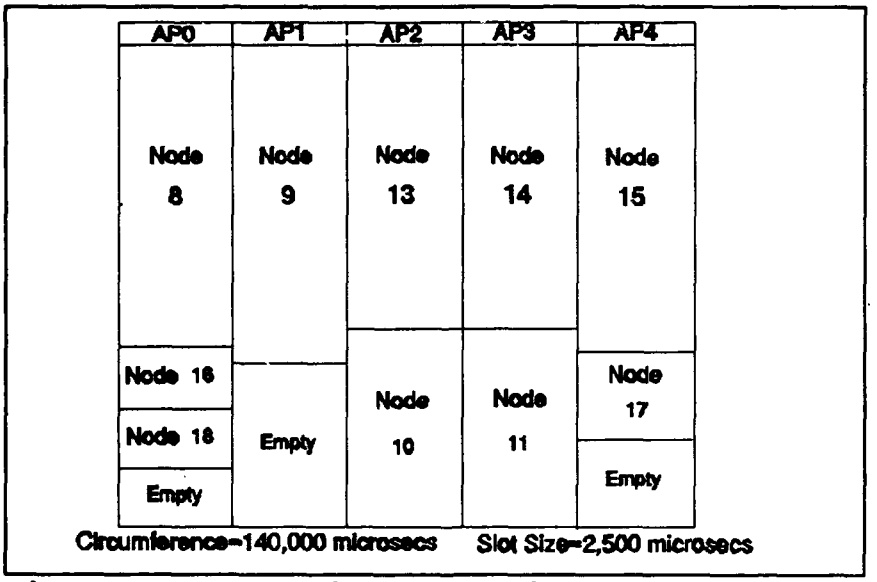


Figure 3.4 Cylinder Assignment for the Correlator with Chaining

With three means of traversing the graph, the algorithm is sufficiently robust to determine all possible chains contained

therein. Figure 3.5 shows the actual restructuring of the application in PGM form.

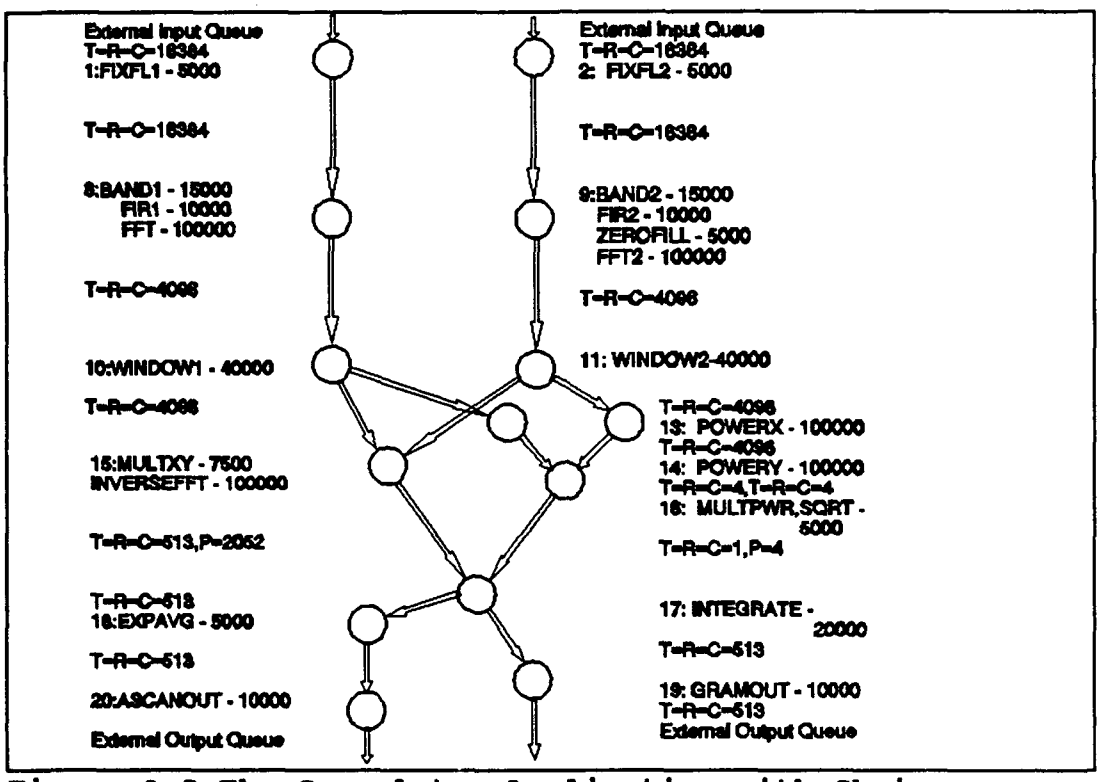


Figure 3.5 The Correlator Application with Chains

2. Explanation of the Algorithm

As seen in Figure 3.6, the main procedure "chaingraph" simply identifies which subroutine to call based on the number of input and output queues. If none of the cases indicated applies to the given node, it is not considered for chaining, and is assigned to the cylinder by itself. Examining the "chaindown" portion of the algorithm depicted in pseudo code of Figure 3.7, the first statement gets the output queues associated with the node.

From here the output node of the queue is obtained and the node that the function was entered with is flagged as being assigned to the cylinder.

```
procedure Chaingraph(node, cylinder, slot_size);
   if ((number of input queues for node is not 1
       and (number of output queues is 1));
         chaindown(node, cylinder, slot_size)
   end(if);
   if ((number of input queues for node is 1)
      and (number of output queues is not 1));
         chainup(node, cylinder, slot_size)
   end(if);
   if ((number of input queues for node is 1)
      and (number of output queues is 1));
         chaindwnup(node, cylinder, slot_size)
   end(if);
end(procedure)
```

Figure 3.6 Procedure Chaingraph

While there is sufficient cylinder space available and the number of input and output queues is one, and the end of the graph has not been reached, the chain becomes longer. As more nodes are added to the chain, the execution size and AIS are summed with the entry node. As each node is added to the chain, it is flagged as assigned to avoid assigning a node multiple times to the cylinder and eliminate the possibility of it being incorporated in other chains. The last "if" in the "chaindown" subroutine handles the case when a chain is terminated by a node whose number of input queues is one and number of output queues is not one. This node still meets the criteria for chaining and is added with this piece of code.

Upon exiting the "chaindown" procedure, the node that entered the subroutine now is of a size equivalent to the size of the chain and is assigned to the cylinder.

```
procedure Chaindown(node, cylinder, slot_size);
   determine node's output queues
   determine children of the node
   assign node to the cylinder
   while((child's number of input queues = 1) and
         (number of output queues = 1) and
         (child not an iop) and
         (sufficient space in the cylinder ));
             sum node and child's execution times
             sum node and child's AIS
             ancestor assigned to cylinder
             get next child
   end(while);
   if((child's number of input queues = 1) and
      (number of output queues not = 1) and
      (child not an iop) and
      (sufficient space in the cylinder ));
          sum node and child's execution times
          sum node and child's AIS
          child assigned to cylinder
   end(if);
end(procedure)
```

Figure 3.7 Algorithm for Chaining Down a Graph

The routine for chaining up the graph is provided in Figure 3.8. The only difference is that the input queue for the node that the routine was entered with is now examined and the successor node is considered for chaining. All other conditions and assignments of variables remain the same as those found in "chaindown".

"Chaindwnup" is essentially the combination of "chaindown" and "chainup". For the sake of brevity, its algorithm is not included here.

```
procedure Chainup(node, cylinder, slot_size);
   determine node's input queues
   determine parent of the node
   assign node to cylinder
   while((parent's number of input queues = 1) and
         (number of output queues = 1) and
         (parent not an iop) and
         (sufficient space in the cylinder));
              sum node and parent's execution times
              sum node and parent's AIS
              parent assigned to cylinder
              get next parent
   end(while);
   if((parent's number of input queues not = 1) and
      (number of output queues = 1) and
      (parent not an iop) and
      (sufficient space in the cylinder));
          sum node and parent's execution times
          sum node and parent's AIS
          parent assigned to cylinder
   end(if);
end(procedure)
```

Figure 3.8 Algorithm for Chaining Up a Graph

The routine will add nodes to the chain in the same manner, and under the same restrictions previously discussed. When one of the conditions for addition of a node to a chain is not met while chaining down the graph, it will reverse direction and attempt to find nodes to be added by searching up the graph.

By summing the execution times of all nodes contained within a chain, the algorithm assigns a block of size equal to

the sum on the cylinder. By determining chains within the confines of the cylinder, we are guaranteed that they are optimized for throughput, and are divided over all AP's so as to distribute the work to be done evenly. The algorithm also ensures that the nodes chained can be assigned to the same AP to produce the desired effect of minimizing communication overhead.

It should be noted that the chaining algorithm never increases cylinder size to accommodate a chain. The cylinder size can only be increased by the assignment of an individual node. By assigning nodes from largest to smallest the algorithm ensures that the cylinder size will not be increased by a large node as long as the number of large nodes does not exceed the number of APs.

The output of the algorithm is located in the file "cylchain". In this file, the chain number is provided along with the nodes contained within the chain, the input and output queues for the chain, and the new AIS for the chain. After the information pertinent to the chains is output the cylinder assignment by AP for the graph is produced.

The algorithm also allows for the chaining algorithm to be disabled, thereby allowing the user to examine the cylinder assignment before chaining and after chaining. If no chains for a graph are generated, this will be indicated in the output file.

Exact code for the algorithm is provided in Appendix B.

C. SAMPLE RESULTS

With the chaining algorithm developed, a simulation of the correlator graph and the chained version of the correlator were executed on the NPS simulator, and in the RC executions the dependencies utilized were arrived at as described in [LIT 91]. The results of the simulation for the correlator are depicted in Figures 3.9 and 3.10. In Figure 3.9, the AP Efficiency is plotted against the Normalized Input Data Interval, where the normalized input data interval refers to the theoretical maximum throughput rate for the application, assuming no internal delays. This value is arrived upon by dividing the total execution time for the graph by the number of APs for the system configuration.

As seen in the plot for the correlator graph, the AP efficiency for the FCFS peaks and quickly falls off under high system loads, whereas the efficiency is constant for the RC run regardless of the system load. The AP efficiency results are supported in Figure 3.10, where the number of graph instances completed for a fixed time interval are given. The results indicate that the number of graph instances executed varies directly with the AP efficiency.

By examining the AP efficiency for the execution of the chained version of the correlator given in Figure 3.11,

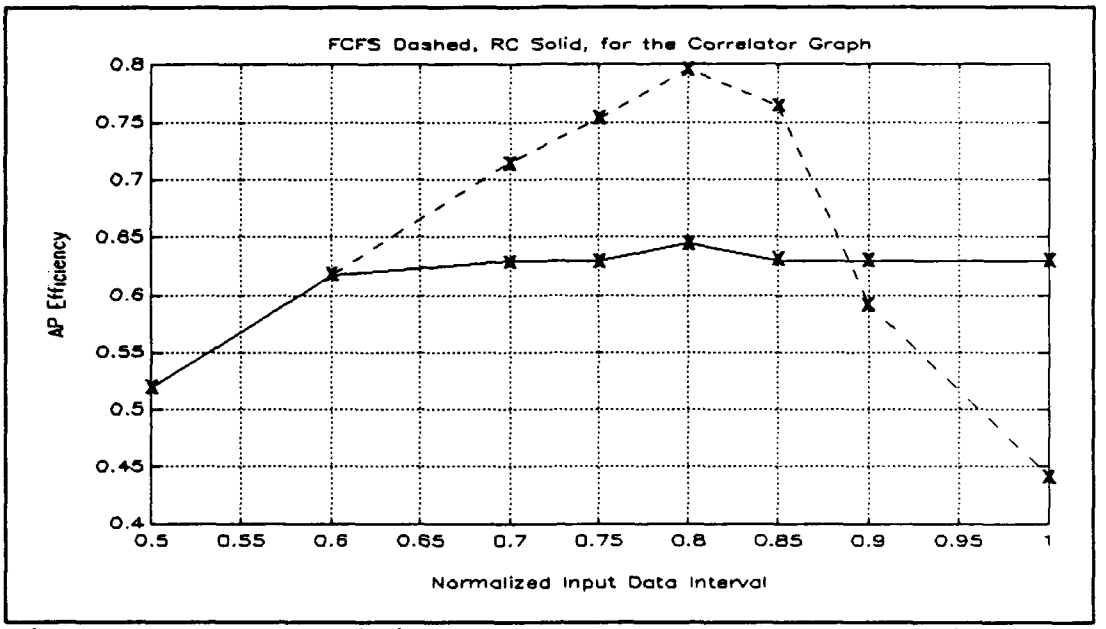


Figure 3.9 AP Efficiency for the Correlator Application

it is seen that, once again, the efficiency for the FCFS execution drops off quickly under heavy system loads. However, the efficiency begins to improve when the normalized input data interval ranges between .9 and 1.0. The RC execution reaches a maximum efficiency below the peak of FCFS, but maintains its maximum efficiency even when the input data rate is high. In Figure 3.12 the number of graph instances completed varies directly with the AP efficiency for both simulations, thereby confirming the results obtained for AP efficiency.

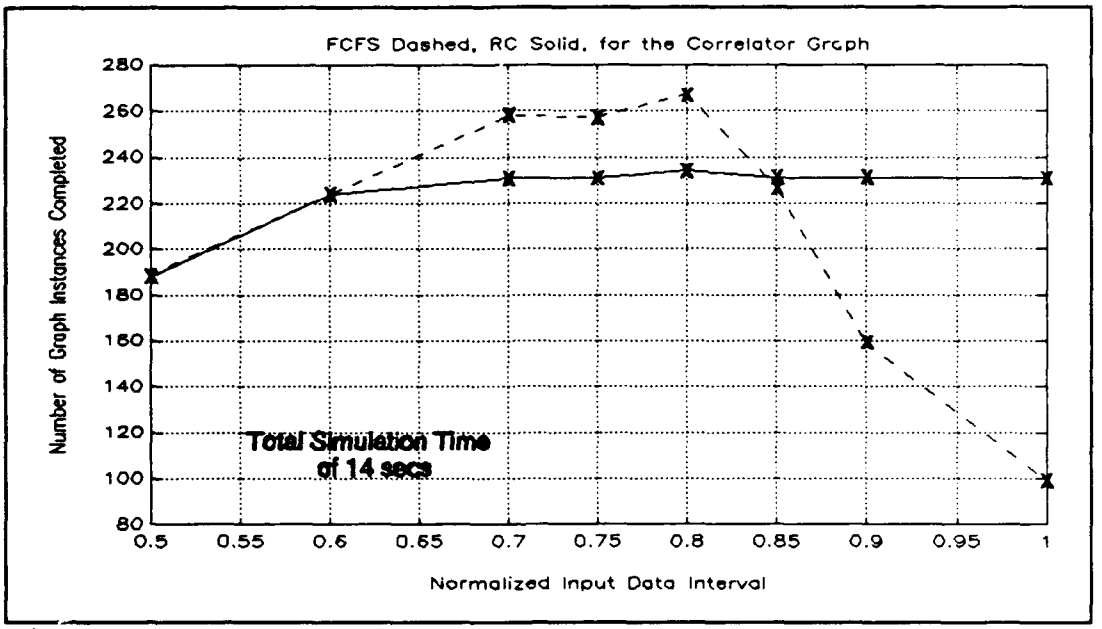


Figure 3.10 Number of Instances Completed for the Correlator Application

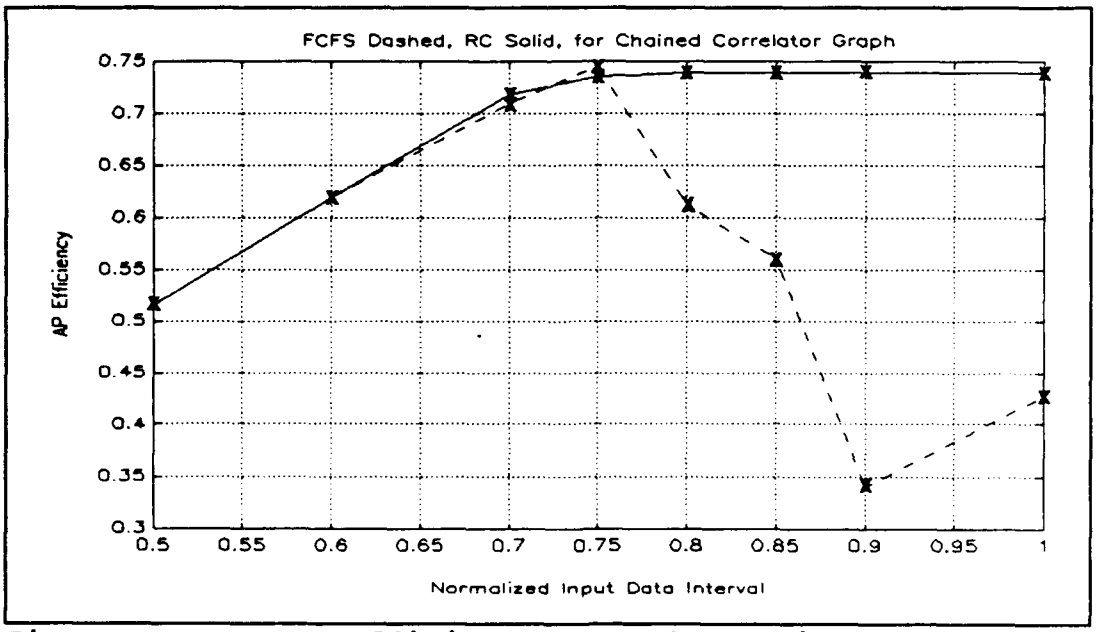


Figure 3.11 AP Efficiency for the Chained Correlator Application

In the FCFS mode of execution, the incorporation of optimal chains produced a reduction in AP efficiency when the demand on the system was high. This result indicates that the chains developed were not suited for FCFS execution.

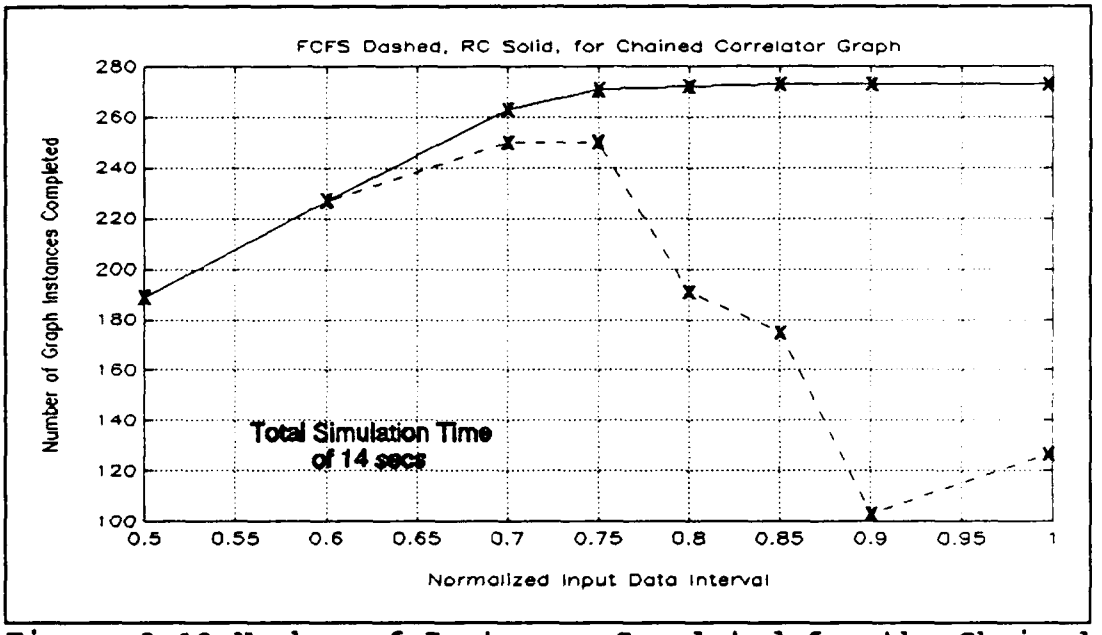


Figure 3.12 Number of Instances Completed for the Chained Correlator Application

In examining the AP efficiency for the RC simulation, however, it stabilizes at a level ten percent greater for the chained correlator than that for the unchained version. Therefore, in order to realize the highest system throughput in RC scheduling, the incorporation of optimal chains is necesssary.

IV. EXECUTION CONTROL

A. AN/UYS-2 EXECUTION

The AN/UYS-2 utilizes a distributed run-time operating system that incorporates a hybrid data-flow and control-flow technique at the task level and the control-flow approach at the elementary processing level. [POPS 89, Rice 90]

The data-flow organization is implemented when a task(node) has the machine resources available to it, and the input data is available. Sequencing is performed by the flow of data in an asynchronous manner, thereby eliminating the need for a program counter or central control. All input data is consumed and the output results are passed directly to subsequent tasks as input data. By using data-flow at the task or functional level communication and bookkeeping overheads are minimal compared with the gain in concurrent processing, and the parallelism inherent in signal processing applications can be exploited.

Control-flow processing is implemented at the fine grain level to eliminate the communication and bookkeeping overheads that would result if data-flow organization was used. All control-flow is contained within each individual processing element(node). [RICE 90]

With data-flow execution of nodes implemented on the AN/UYS-2, the only fixed time frame for arrival of data is the data rate associated with input queues. Once this data is consumed by follow-on nodes and the entire graph is involved in execution, the status of individual queues, set-up and break-down of nodes, and available APs are all independent of the original graph structure. The only remaining way to evaluate performance of the machine is by examining the throughput. Since the process of executing a graph on the AN/UYS-2 is nondeterministic, a compile time prediction of performance is infeasible.

The SCH presently being employed on the AN/UYS-2 uses a FCFS scheduling scheme. Scheduling is performed by matching a ready node to a free AP by maintaining four tables: the ready-node list, the free AP list, the node status table, and the queue-to-node table. The SCH receives queue information from the GMs. As queues exceed threshold levels, the GMs send queue over threshold messages to the SCH. When all of a graph node's queues are over threshold, the SCH attempts to match free APs to ready graph nodes. If a match is found, and the node is not currently executing, scheduling data is sent to the GMs and database tables are updated to indicate the match. If a match is not achievable, the node status table and ready-node list are updated to reflect that a ready node is waiting to be executed. When an AP does become available, the first node on the list is served without regard to priority or

optimization of throughput. Throughout the process of scheduling a node, there exists no execution control other than the queues going over threshold or capacity. [POPS 90, RICE 90]

1. Enforcing the RC Assignment in the AN/UYS-2

Within the constraints of the AN/UYS-2 programming and run-time environment, a way of controlling graph execution has been developed. The RC approach of assigning dependencies to the graph to enforce the desired scheduling of nodes is realized by consuming and adding tokens and is given in [LIT 91]. The graph is modified by inserting artificial dependencies to optimize throughput. The creation of dependencies in the previous work is improved upon by the cylinder being filled more efficiently and the incorporation of chains. Since the size of the cylinder has been reduced significantly the instances of nodes now placed on the cylinder are no longer the same and represent more than one previous instance of the graph. A new algorithm to determine from which instance of the graph a node is placed on the cylinder is developed based upon the improved assignment of nodes. Since the instances of a graph on the cylinder can now be substantially separated, a new dependency generation algorithm was needed.

B. ASSIGNMENT OF DEPENDENCIES

1. Assignment of Node Indices

As described earlier, because of the data-flow execution of the graph, the assignment of nodes to the cylinder must be analyzed to determine which instance of a node is actually to be executed. The cyclic interpretation of the cylinder assignment represents one instance of the graph. However, a node within the graph cannot be executed until its parents have executed, and produced the data required for the present node's execution. The procedure to assign indices for each node is given in Figure 4.1. This algorithm, "assignindex" is contained within the code to determine dependencies and the exact code is given in Appendix C.

The code is entered with one of the largest nodes in terms of execution time. It is assigned an index of zero to represent the fact that it is the current instance of this node that is being executed. Every parent of the node entered with is examined with respect to their relative positions on the cylinder. If the parent's completion time on the cylinder is greater than the present node's start time on the cylinder, the parent must be executing a previous instance. The algorithm is then called again with an initial index plus one; however, the procedure is now entered with the parent node, and its index is assigned one greater than the starting node. If the condition of the "if" statement does not hold, the

parent is assigned an index equal to the one of the node which entered the procedure.

```
procedure Assign_Index(node, index)
   node's index <- index
   for every parent of node
      if parent's index not determined
         if ((parent's finish time on the cylinder) >
             (node's start time on the cylinder))
               Assign_Index(parent, index+1)
         else
               Assign_index(parent, index)
         end(if)
      end(if)
   end(for)
   for every child of node
      if ((child's index not determined) or
      (child's index >= index))
         if ((child's start time on the cylinder) <
             (node's finish time on the cylinder))
                Assign_Index(child, index-1)
         else
                Assign_Index(child, index)
         end(if)
      end(if)
   end(for)
end(procedure)
```

Figure 4.1 Algorithm for Index Assignment

With all parents of the original node assigned their indices, the algorithm now examines all child nodes of the initial node. If a child has not been assigned an index or its index is greater than or equal to the present index, the second conditional "if" is examined. If the child's start time on the cylinder is less than the present node's completion time, then the index of the child is the index minus one. This index indicates that the child is an instance

behind the node that the algorithm was entered with. When the comparison of cylinder start and finish times is not met the node instance is the same as the node entered with.

The assignment of indices represents a "snap shot" of one graph instance. By giving each node a respective index based on its position in the cylinder, it can be determined for each node as to how far ahead or behind it is. A positive index indicates that a node is that many instances ahead of the reference node, while a negative index indicates that the node is that many instances behind the reference node. The decision to make the algorithm recursive was to ensure multiple "visits" to each node and ensure that no node was preempted from receiving an index. The cylinder assignment for the chained version of the correlator graph is given in Figure 4.2 with the indices assigned for each node.

2. Creation of Dependencies

With the cylinder assignment determined and the indices for each node assigned, the required dependencies for the RC schedule are now determined. The algorithm is given in Figure 4.3. The input for the procedure is the node listing and circumference of the cylinder. Again, all input is compatible with the input for the NPS simulator. Since dependencies may be created for all nodes, the graph is examined from top to bottom.

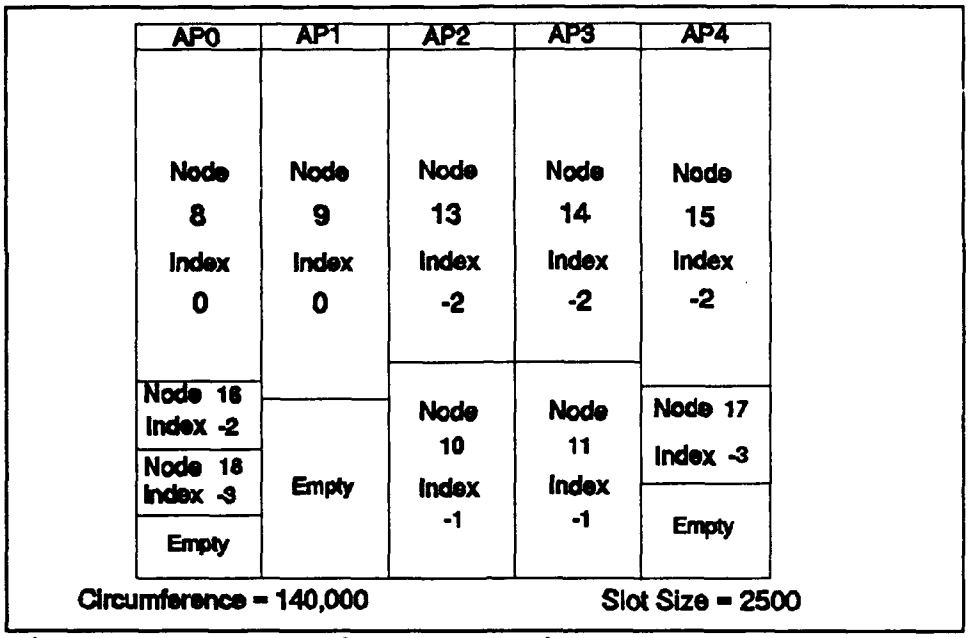


Figure 4.2 Cylinder Assignment for Chained Correlator with Indices

```
procedure create_dependencies(node list, circumference);
   /*n_r, n_s are nodes of graph, G*/
   for all nodes, n_r
       check index i of n_r
       find the latest node, n_s, that ends before
       n_r starts on the cylinder
       check index j of n_s
       /* if index i is not equal to j, and */
       /* a dependency does not already exist.*/
       add a dependency from n_s to n_r
       if i >= j
           put i - j tokens on the arc
           set threshold = 1, consume = 1
       else
           put 0 initial tokens on the arc
           threshold = j - i, consume = 1
       end(if)
   end(for)
end(procedure)
```

Figure 4.3 Algorithm to Generate Dependencies

Upon entering the algorithm, the node's index is stored, then the latest of all other nodes that end before the entry node starts on the cylinder is found. There can be more than one of these nodes since multiple nodes may end on the same slot entry in the cylinder, and the node's end time is modulo circumference to ensure nodes at the bottom of the cylinder are also considered. If these conditions are met and the two nodes have different indices, a dependency is created.

Having determined that two nodes require a dependency, it is now a matter of finding the initial tokens, threshold value and consume value for the dependency. These values are determined by using the index of the two nodes. If the index of the initial input node is greater than or equal to the node found to require a dependency, then the initial token size is the difference of the indices. This results in a node that will complete more instances, not executing until the other node has executed as many times as the initial token size. Otherwise, no tokens are placed and the threshold value is equal to the difference of the indices. The threshold amount determines the number of tokens that must be present before the destination is eligible for execution.

Dependencies are created for each node, but there can be no more than one dependency for each node pair, and no node can have a dependency to itself. In Figure 4.4 the chained version of the correlator application is restructured incorporating some of the dependencies identified by the

algorithm. The dependencies are identified by arcs between nodes, and the 3-tuple associated with each indicates the tokensize, threshold, and consume values for that dependency. The implementation of the algorithm in its coded form is given in Appendix D.

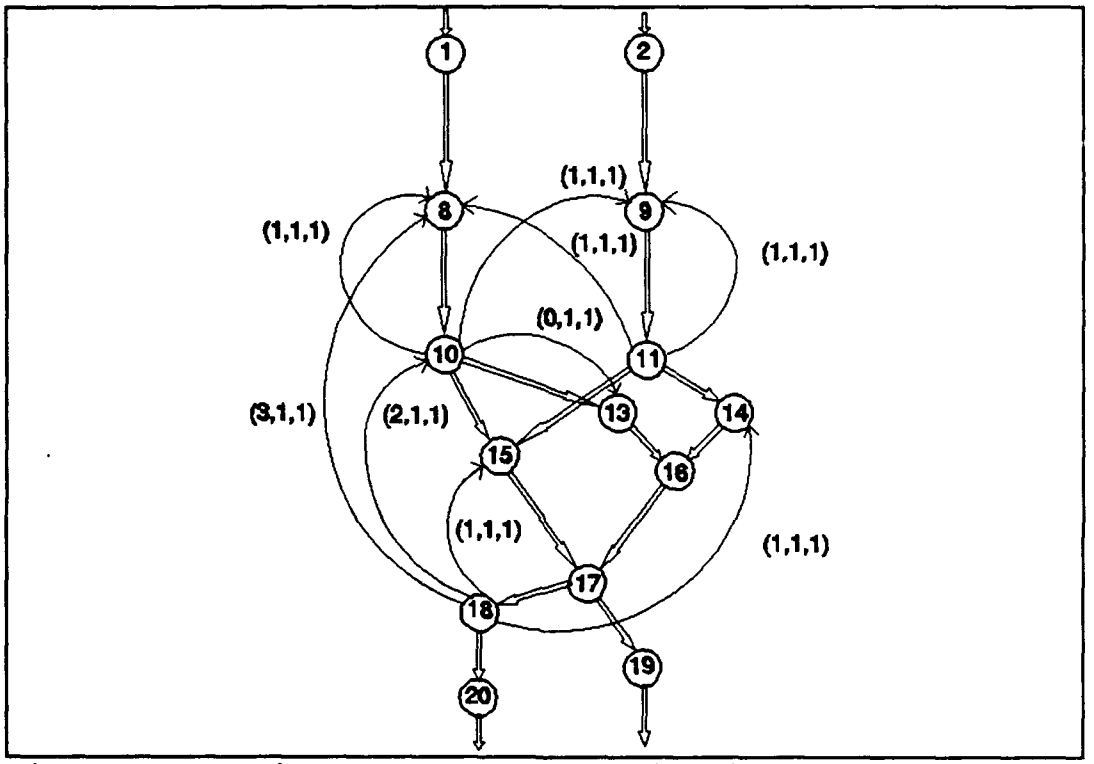


Figure 4.4 Chained Correlator Application Restructured by Dependencies

Output of the dependency program is found in the file "tokens". The node pairs are identified along with the token sizes, threshold values, and consume values. All of these values are given in bytes to interface with the NPS simulator whose construction and execution constants are based upon bytes per second.

C. IMPLEMENTATION OF DEPENDENCIES

There are two possible approaches for actual implementation of the dependencies in the AN/UYS-2. The first is equivalent to the implementation described in the previous work completed [LIT 91]. The second is to use the information provided by the dependency program to add additional data queues for the node pair involved. Each of these approaches has its advantages and disadvantages.

The original enforcement of dependencies was accomplished by the maintaining of a dependency list. When a node was considered for execution not only were its queues checked for exceeding the threshold, but the number of tokens for the dependency were also checked. The advantage with this approach is its simplicity. Until the required number of tokens has been accumulated, the node is not executed. The disadvantage of this method is that the data-flow execution of nodes presently incorporated in the AN/UYS-2 does not allow the checking of tokens in the SCH. To actually implement this scheme, a complete rewriting of SCH code would be required to check the tokens and the operating system would also require rewriting.

By implementing the dependencies as data queues the data-flow execution in the AN/UYS-2 can be used as it already exists. When the graph is instantiated by the user the dependencies can be determined prior to execution and the dependency queues are constructed as standard data queues.

Instead of initializing these queues to be empty, the token size can be inserted, and as with any data queues, the threshold and consume values can be established.

Although there is an additional latency involved in creating more queues and more memory is required, the dependencies are incorporated within the original structure of the AN/UYS-2. There is no need for operating system or application modifications. The new queues are transparent to the user along with being cost effective.

With the dependencies now incorporated the graph preprocessor is complete.

V. CONCLUDING REMARKS

A. CONCLUSIONS

In the thesis, a graph preprocessor was developed that identifies optimal chains within the RC context and determines improved artificial node dependencies to optimize system throughput.

The preprocessor was used to modify the application identified as the Correlator. With the appropriate chains for the Correlator determined by the RC analysis of the graph, the restructured graph was then simulated on the NPS simulator [LIT 91]. Using both scheduling schemes of FCFS and RC without the improved dependencies, results were presented. The results show that in the FCFS case under high system loads, with and without chaining, performance rapidly deteriorated. While the RC case without chaining maintained a uniform throughput, it was improved upon with the addition of optimal chains.

An improved cylinder assignment was then developed that creates artificial dependencies that will control node execution order more efficiently. With the improved dependencies implemented, the execution of the graph should yield better performance results than previously realized. No simulation results for the improved dependencies are available

since the NPS simulator does not presently have the capability to operate with the implementation of the dependencies as data queues. The ETS++ simulator provided was considered for all simulations, however, once again, the specifics for incorporation of dependencies as data queues with initial tokens contained within presented a dilemma.

With the preprocessor providing compile-time analysis for a given graph, the throughput for the AN/UYS-2 can be optimized by improving processor efficiency even under high system loads. This will result in an extension of service life for presently employed systems at a low cost, since the modification is done in software without changes to the operating system. If hardware upgrades to the AN/UYS-2 are realized the RC analysis can still be utilized.

B. FUTURE WORK

As seen in the cylinder assignments for the Correlator and the chained version of it, empty slots are available to be utilized. With many possible assignments of nodes to the cylinder conceivable, the one that produces the fewest dependencies and smallest token sizes can be identified. The incorporation of such an algorithm will increase latency when dependencies are developed, but the overhead at run-time will be decreased.

Accurate simulation results need to be obtained. In the thesis, all simulation was done on the NPS simulator, which

has yet to be validated against the ETS++ simulator. Benchmarks for the ETS++ simulator must be acquired to perform a valid comparison between the two. With the validation completed, a method for implementation of the dependencies as data queues must be instituted. The nature of signal processing involves delays, therefore a mechanism should be readily available for this to be accomplished.

Using the preprocessor on the benchmarks and restructuring the graph accordingly, an accurate comparison between FCFS and RC scheduling could then be completed.

LIST OF REFERENCES

[ECOS 89]
AT&T Technologies, Report Alpha 890301-01, *ECOS Workstation User Manual,* AT&T Be l Laboratories, 1 March 1989.

[LB 90]
Lee, E. A., and Bier, J. C., "Architectures for Statically Scheduled Dataflow," *Journal of Parallel and Distributed Computing,* v. 10, pp. 333-348, December 1990.

[LIT 91]
Little, B. S., *A Technique for Predictable Real-Time Execution in the AN/UYS-2 Parallel Signal Processing Architecture*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1991.

[POPS 90]
AT&T Technologies, Report 5885401, *Enhanced Modular Signal Processor (EMSP) Principles of Operation (POPS),* AT&T Bell Laboratories, 20 March 1990.

[PRIMLIB 90]
AT&T Technologies, Report 5885404, *AN/UYS-2 Graph Primitives library - Floating Point*, AT&T Bell Laboratories, 17 September 1990.

[RICE 90]
Rice, M. L., "The Navy's New Standard Digital Signal Processor: The AN/UYS-2," paper presented at the Association of Scientists and Engineers 27th Annual Technical Symposium, 23 May 1990.

[SLZ 92]
Shukla, S. B., Little, B. S., and Zaky, A., "A Compile-time Technique for Controlling Real-time Execution of Task-level Data-flow Graphs," presented at the 1992 International Conference on Parallel Processing.

APPENDIX A: IMPROVED CYLINDER ASSIGNMENT CODE

```
// Description  : This code assigns nodes to the cylinder
// Parameters   : temp     - the graph node to schedule
//                circum   - the circumference of the cylinder
//                widthavg - slot size in the cylinder
//                numaps - the number of aps in the AN/UYS-2
//                cyl    - the cylinder itself

void topgraph :: schedulenode(topgraph *temp,long int
                widthavg, int numaps,cyltype *cyl) {

  cyltype       *tempcylinder = cyl;
  cylentrytype *tempcyl,*temp2cyl;
  boolean       scheduled = false;
  boolean       available = false;
  int           nodesize = 0;
  int           blockcount = 0;

  if ((temp->width % widthavg) == 0) {
     nodesize = temp->width / widthavg;
  }
  else {
     nodesize = temp->width / widthavg + 1;
  };
  tempcylinder = cyl;
  while (scheduled == false) {
    for (int i=1;i<=numaps;i++) {
      tempcyl = tempcylinder->cylentrylist;
      while ((tempcyl->nodesch != 0) && (tempcyl != NULL)) {
        tempcyl = tempcyl->nextcylentry;
      };
      temp2cyl = tempcyl;
      while ((blockcount != nodesize) && (temp2cyl !=NULL)){
        if (temp2cyl->nodesch == 0) {
          blockcount = blockcount + 1;
          available = true;
        }
        else {
          available = false;
        };
        temp2cyl = temp2cyl->nextcylentry;
      };
      if ((available) && (blockcount == nodesize)) {
        temp2cyl = tempcyl;
        for (int j=1;j<=nodesize;j++) {
          temp2cyl->nodesch = temp->id;
          temp2cyl = temp2cyl->nextcylentry;
```

```
        };
        scheduled = true;
        break;
      }
      else {
        tempcylinder = tempcylinder->nextcylap;
        blockcount = 0;
      };
    };
  };
};

// Description  : This code implements the improvements to the
//  assigning of nodes to the cylinder discussed in Chapter III.
//  The cylinder is constructed in the code with the circumference
//  determined and the slot size calculated using the GCD
//  algorithm. Largest nodes are assigned to the cylinder first.
// Parameters   : tempgnodelisting - the node graph itself
//                numaps - the number of aps in the AN/UYS-2
//                chainop - chain option selected
void topgraph :: assignrc(gnode *tempgnodelisting,int
                          numaps,int chainop) {

  gnode           *temp2gnodelisting = tempgnodelisting;
  ptrtoptrtoaq    *tempptrtoptr = NULL;
  topgraph        *q = NULL;
  topgraph        *qtemp = NULL;
  topgraph        *temp = NULL;
  int             count = 0;
  int             nodecount = 0;
  int             chainnum = 1;
  long int        gcd = 0;
  long int        tempgcd = 0;
  long int        circumference = 0;
  long int        maxwidth = 0;
  long int        nextmax = 0;
  long int        widthavg = 0;
  long int        j = 0;
  cyltype         *cylinder = NULL;
  cyltype         *tempcylinder = NULL;
  cylentrytype    *cyl2entrylist = NULL;
  boolean         iopinnode = false;
  boolean         iopoutnode = false;

  while (temp2gnodelisting != NULL) {
    tempptrtoptr = tempgnodelisting->getgnodeinputqslist
                   (temp2gnodelisting->getnodeid());
    while (tempptrtoptr != NULL) {
      if (tempptrtoptr->getnodein() == -1) {
         iopinnode = true;
         break;
```

```
      }
      else {
         iopinnode = false;
      };
      tempptrtoptr = tempptrtoptr->getnextelement();
    };
    tempptrtoptr = tempgnodelisting->getgnodeoutputqslist
                   (temp2gnodelisting->getnodeid());
    while (tempptrtoptr != NULL) {
      if (tempptrtoptr->getnodeout() == -1) {
        iopoutnode = true;
        break;
      }
      else {
        iopoutnode = false;
      };
      tempptrtoptr = tempptrtoptr->getnextelement();
    };
    if ((iopinnode) || (iopoutnode)) {
      // do nothing
    }
    else {
      count++;
      if (q == NULL) {
        if (!(q = new topgraph)) {
          fprintf(stderr,"Insufficient memory for topgraph\n");
          exit(1);
        };
        q->id = temp2gnodelisting->getnodeid();
        q->width = temp2gnodelisting->primtime;
        gcd = q->width;
// improve RC assignment using GCD of node execution times.
        if (q->width > maxwidth) {
          maxwidth = q->width;
        };  // note q->est set equal to zero by constructor
        circumference = circumference + q->width;
        qtemp = q;
      }
      else {
        if (!(qtemp->next = new topgraph)) {
          fprintf(stderr,"Insufficient memory for topgraph\n");
          exit(1);
        };
        qtemp->next->id = temp2gnodelisting->getnodeid();
        qtemp->next->width = temp2gnodelisting->primtime;
        tempgcd = qtemp->next->width;
        if (gcd != 1) {
          if ((tempgcd % gcd) != 0) {
            while (gcd * tempgcd != 0) {
              if (tempgcd >= gcd) {
                tempgcd = tempgcd - gcd;
```

```
                }
                else {
                  gcd = gcd - tempgcd;
                };
              };
            };
          };
          if (qtemp->next->width > maxwidth) {
            maxwidth = qtemp->next->width;
          };
          circumference = circumference + qtemp->next->width;
          qtemp = qtemp->next;
        };
      };
      temp2gnodelisting = temp2gnodelisting->getnextgnode();
    };
    widthavg = gcd;
    circumference = circumference / numaps;
    if (circumference < maxwidth) {
// circumference is largest node if smaller than largest node
      circumference = maxwidth;
    };
    if (circumference % widthavg != 0) {
      circumference = circumference+(circumference % widthavg);
    };
// circumference set to multiple of gcd
    for (int i=1;i<=numaps;i++) {
//This loop constructs the cylinder
      if (cylinder == NULL) {
        if (!(cylinder = new cyltype)) {
          fprintf(stderr,"Insufficient memory for cyltype\n");
          exit(1);
        };
        j = 0;
        while (j < circumference) {
          if (cylinder->cylentrylist == NULL) {
            if (!(cylinder->cylentrylist = new cylentrytype)) {
              fprintf(stderr,"Insufficient memory for
                                              cylentrytype\n");
              exit(1);
            };
            cylinder->cylentrylist->widthstarttime = j;
            cyl2entrylist = cylinder->cylentrylist;
          }
          else {
            if (!(cyl2entrylist->nextcylentry = new cylentrytype)) {
              fprintf(stderr,"Insufficient memory for
                                                cylentrytype\n");
              exit(1);
            };
            cyl2entrylist->nextcylentry->widthstarttime = j;
```

```
          cyl2entrylist = cyl2entrylist->nextcylentry;
        };
        j = j + widthavg;
      };
      tempcylinder = cylinder;
    }
    else {
      if (!(tempcylinder->nextcylap = new cyltype)) {
        fprintf(stderr,"Insufficient memory for cyltype\n");
        exit(1);
      };
      j = 0;
      while (j < circumference) {
        if (tempcylinder->nextcylap->cylentrylist == NULL) {
          if (!(tempcylinder->nextcylap->cylentrylist = new
                                        cylentrytype)){
            fprintf(stderr,"Insufficient memory for
                                        cylentrytype\n");
            exit(1);
          };
          tempcylinder->nextcylap->cylentrylist->
                                          widthstarttime = j;
          cyl2entrylist = tempcylinder->nextcylap->cylentrylist;
        }
        else {
          if (!(cyl2entrylist->nextcylentry = new cylentrytype)) {
            fprintf(stderr,"Insufficient memory for
                                            cylentrytype\n");
            exit(1);
          };
          cyl2entrylist->nextcylentry->widthstarttime = j;
          cyl2entrylist = cyl2entrylist->nextcylentry;
        };
        j = j + widthavg;
      };
      tempcylinder = tempcylinder->nextcylap;
    };
  };
  nodecount = count;
  qtemp = q;
  while (qtemp != NULL) {
    if ((qtemp->width == maxwidth) && (qtemp->assigned==false)) {
//Largest nodes scheduled first
      temp = qtemp;
      if (chainop == 1) {  //Chain nodes if option selected
        chaingraph(q,temp,qtemp,cylinder,
                                    nodecount,widthavg,chainnum);
      };
      qtemp->assigned = true;
      qtemp = qtemp->next;
      while (qtemp->assigned == true) {
```

```
          qtemp = qtemp->next;
        };
        while (cylinder->checkfreecylspace(cylinder) <
                                        (temp->width/widthavg)) {
          cylinder = cylinder->
                        increasecylsize(cylinder,circumference);
          circumference = circumference + widthavg;
        };
        schedulenode(temp,widthavg,numaps,cylinder);
        temp->assigned = true;
        nodecount = nodecount - 1;
          if ((qtemp == NULL) && (nodecount != 0)){
            qtemp = q;
            maxwidth = nextmax;
            nextmax = 0;
          };
       }
       else {
         if ((qtemp->width > nextmax) &&
           (qtemp->width < maxwidth)  && (qtemp->assigned ==
                                                 false)) {
            nextmax = qtemp->width;
            qtemp = qtemp->next;
         }
         else{
           qtemp = qtemp->next;
         };
         if ((qtemp == NULL) && (nodecount != 0)){
           qtemp = q;
           maxwidth = nextmax;
           nextmax = 0;
         };
     };
   };
   if ((chainnum == 1) && (chainop == 1)) {
     printf("\nNo chains were found for this graph\n");
   };
};
```

APPENDIX B: NODE CHAINING CODE

```
// Description  : Determines which direction in the graph to chain
//      based on the number of input and output queues of the node.
// Parameters   : temp - the node to begin chaining frcm
//                cylinder - the cylinder
//                nodecount - the number of nodes remaining to be
 //                           assigned to the cylinder
//                widthavg - the slot size of the cylinder
//                chainnum - the number of the chain being built
 void topgraph :: chaingraph(topgraph *q, topgraph *temp,
                        topgraph *qtemp, cyltype *cylinder,
                int &nodecount, long int widthavg, int &chainnum) {

    gnode        *temp3gnodelisting = gnodelisting;

    while (temp->id != temp3gnodelisting->nodeid) {
       temp3gnodelisting = temp3gnodelisting->nextgnode;
    };
    if ((temp3gnodelisting->numinqs !=1)
                &&(temp3gnodelisting->numoutqs == 1)) {
      chaindown(temp3gnodelisting,temp,qtemp,cylinder,
                           nodecount,widthavg,chainnum);
    };
    if ((temp3gnodelisting->numinqs == 1)
                &&  (temp3gnodelisting->numoutqs != 1)) {
      chainup(temp3gnodelisting,q,temp,qtemp,cylinder,
                           nodecount,widthavg,chainnum);
    };
    if ((temp3gnodelisting->numinqs == 1)
                && (temp3gnodelisting->numoutqs == 1)) {
      chaindwnup(temp3gnodelisting,q,temp,qtemp,cylinder,
                           nodecount,widthavg,chainnum);
    };
};

// Description  : Chains nodes down the graph while the conditions
//                 discussed in Chapter III are met.
// Parameters   : temp3gnodelisting - starting node to begin chain
//                q                 - the beginning of the topgraph
//                qtemp               - temporary topgraph
//                cyl                 - the cylinder
//                nodecount          - number of nodes remaining to
//                                       be assigned to the cylinder
//                widthavg            - cylinder slot size
//                chainnum          - the number of the chain being
//                                       built
```

```
void topgraph :: chaindown(gnode *temp3gnodelisting, topgraph
              *temp,topgraph *qtemp, cyltype *cylinder,
              int &nodecount,long int widthavg, int &chainnum) {

   ptrtoptrtoaq    *temptrtoptr = NULL;
   gnode           *temp4gnodelisting = gnodelisting;
   gnode           *temp5gnodelisting;
   boolean         chained      = false;
   int             newaissize   = 0;
   int             gnodeoutqnum = 0;
   int             qnodeoutnum  = 0;

   newaissize = temp3gnodelisting->aissize;
   temptrtoptr = temp3gnodelisting->
                 getgnodeoutputqslist(temp3gnodelisting->nodeid);
   gnodeoutqnum = temptrtoptr->getgqueueid();
   qnodeoutnum = temptrtoptr->getqnodeoutnum(gnodeoutqnum);
   while (qnodeoutnum != temp4gnodelisting->nodeid){
      temp4gnodelisting = temp4gnodelisting->nextgnode;
   };
   qtemp->assigned = true;
   while ((qtemp->id !=temp4gnodelisting->nodeid)
     && (temp4gnodelisting->iopidassigned == 0)) {
      qtemp = qtemp->next;
   };
   if ((qtemp->assigned == false)
     && tempmp4gnodelisting->iopidassigned == 0)
     && (temp4gnodelisting->numinqs == 1)
     && (cylinder->checkfreecylspace(cylinder)
     >= ((temp->width/widthavg)
                  + (temp4gnodelisting->primtime/widthavg)))) {
      printf("\n");
      printf("Chain number: ");
      printf("%d", chainnum);
      printf(" is made up of nodes: ");
      printf(" %d ", temp->id);
   };
   while ((temp4gnodelisting->numinqs == 1)
     && (temp4gnodelisting->numoutqs == 1)
     && (temp4gnodelisting->iopidassigned == 0)
     && (qtemp->assigned == false)
     && (cylinder->checkfreecylspace(cylinder)
     >= ((temp->width/widthavg)
                  + (temp4gnodelisting->primtime/widthavg)))) {
      temp->width = temp->width + temp4gnodelisting->primtime;
      newaissize = newaissize + temp4gnodelisting->aissize;
      nodecount = nodecount - 1;
      printf(" %d ", qtemp->id);
      qtemp->assigned = true;
      chained = true;
      temp5gnodelisting = temp4gnodelisting;
```

```
      temptrtoptr = temp4gnodelisting->
                    getgnodeoutputqslist(temp4gnodelisting->nodeid);
      gnodeoutqnum = temptrtoptr->getgqueueid();
      qnodeoutnum = temptrtoptr->getqnodeoutnum(gnodeoutqnum);
      while (qnodeoutnum != temp4gnodelisting->nodeid){
         temp4gnodelisting = temp4gnodelisting->nextgnode;
      };
      while ((qtemp->id !=temp4gnodelisting->nodeid)
        && (temp4gnodelisting->iopidassigned == 0)) {
         qtemp = qtemp->next;
      };
   };
   if ((temp4gnodelisting->numinqs == 1)
     && (temp4gnodelisting->numoutqs != 1)
     && (temp4gnodelisting->iopidassigned == 0)
     && (qtemp->assigned == false)
     && (cylinder->checkfreecylspace(cylinder)
     >= ((temp->width/widthavg)
                    + (temp4gnodelisting->primtime/widthavg)))) {
      temp->width = temp->width + temp4gnodelisting->primtime;
      newaissize = newaissize + temp4gnodelisting->aissize;
      nodecount = nodecount - 1;
      printf(" %d ", qtemp->id);
      qtemp->assigned = true;
      chained = true;
      temp5gnodelisting = temp4gnodelisting;
   };
   if (chained) {
      printf("\n");
      printf("The new node number is: ");
      printf(" %d ",temp->id);
      printf("\n");
      printf("The new execution time is: ");
      printf(" %ld ",temp->width);
      printf("\n");
      printf("The new AIS size is: ");
      printf(" %d ",newaissize);
      printf("\n");
      printf("The input queue(s) is/are: ");
      temptrtoptr = temp3gnodelisting->
                    getgnodeinputqslist(temp3gnodelisting->nodeid);
      while(temptrtoptr != NULL) {
         printf(" %d ",temptrtoptr->getgqueueid());
         temptrtoptr = temptrtoptr->getnextelement();
      };
      printf("\n");
      printf("The output queue(s) is/are: ");
      temptrtoptr = temp5gnodelisting->
                    getgnodeoutputqslist(temp5gnodelisting->nodeid);
      while(temptrtoptr != NULL) {
         printf(" %d ",temptrtoptr->getgqueueid());
```

```
            temptrtoptr = temptrtoptr->getnextelement();
         };
         printf("\n");
         chainnum = chainnum + 1;
      };
   };

   // Description  : Chains nodes up the graph while the conditions
   //                discussed in Chapter III are met.
   // Parameters   : temp3gnodelisting - starting node to begin chain
   //                q                 - the beginning of the topgraph
   //                temp                - temporary topgraph
   //                qtemp               - temporary topgraph
   //                cyl                 - the cylinder
   //                nodecount           - number of nodes remaining to
   //                                      be assigned to the cylinder
   //                widthavg            - cylinder slot size
   //                chainnum            - The number of the chain being
   //                                      built.
   void topgraph ::chainup(gnode *temp3gnodelisting, topgraph *q,
                   topgraph *temp,topgraph *qtemp, cyltype *cylinder,
                   int &nodecount,long int widthavg, int &chainnum) {

      ptrtoptrtoaq   *temptrtoptr = NULL;
      gnode          *temp4gnodelisting = gnodelisting;
      gnode          *temp5gnodelisting;
      boolean        chained      = false;
      int            newaissize   = 0;
      int            gnodeinqnum = 0;
      int            qnodeinnum  = 0;

      newaissize = temp3gnodelisting->aissize;
      temptrtoptr = temp3gnodelisting->
                     getgnodeinputqslist(temp3gnodelisting->nodeid);
      gnodeinqnum = temptrtoptr->getgqueueid();
      qnodeinnum = temptrtoptr->getqnodeinnum(gnodeinqnum);
      temp4gnodelisting = gnodelisting;
      while (qnodeinnum != temp4gnodelisting->nodeid){
         temp4gnodelisting = temp4gnodelisting->nextgnode;
      };
      qtemp->assigned = true;
      while ((qtemp->id !=temp4gnodelisting->nodeid)
        && (temp4gnodelisting->iopidassigned == 0)) {
         qtemp = qtemp->next;
         if (qtemp == NULL) {
            qtemp = q;
         };
      };
      if ((qtemp->assigned == false)
        && (temp4gnodelisting->iopidassigned == 0)
        && (temp4gnodelisting->numoutqs == 1)
```

```
  && (cylinder->checkfreecylspace(cylinder)
  >= ((temp->width/widthavg)
                  + (temp4gnodelisting->primtime/widthavg)))) {
   printf("\n");
   printf("Chain number: ");
   printf("%d", chainnum);
   printf(" is made up of nodes: ");
   printf(" %d ", temp->id);
};
while ((temp4gnodelisting->numinqs == 1)
  && (temp4gnodelisting->numoutqs == 1)
  && (temp4gnodelisting->iopidassigned == 0)
  && (qtemp->assigned == false)
  && (cylinder->checkfreecylspace(cylinder)
  >= ((temp->width/widthavg)
                  + (temp4gnodelisting->primtime/widthavg)))) {
   temp->width = temp->width + temp4gnodelisting->primtime;
   newaissize = newaissize + temp4gnodelisting->aissize;
   nodecount = nodecount - 1;
   printf(" %d ", qtemp->id);
   qtemp->assigned = true;
   chained = true;
   temp5gnodelisting = temp4gnodelisting;
   temptrtoptr = temp4gnodelisting->
              getgnodeinputqslist(temp4gnodelisting->nodeid);
   gnodeinqnum = temptrtoptr->getgqueueid();
   qnodeinnum = temptrtoptr->getqnodeinnum(gnodeinqnum);
   temp4gnodelisting = gnodelisting;
   while (qnodeinnum != temp4gnodelisting->nodeid){
      temp4gnodelisting = temp4gnodelisting->nextgnode;
   };
   while ((qtemp->id !=temp4gnodelisting->nodeid)
     && (temp4gnodelisting->iopidassigned == 0)) {
      qtemp = qtemp->next;
      if (qtemp == NULL) {
         qtemp = q;
      };
   };
};
if ((temp4gnodelisting->numinqs != 1)
  && (temp4gnodelisting->numoutqs == 1)
  && (temp4gnodelisting->iopidassigned == 0)
  && (qtemp->assigned == false)
  && (cylinder->checkfreecylspace(cylinder)
  >= ((temp->width/widthavg)
                + (temp4gnodelisting->primtime/widthavg)))) {
   temp->width = temp->width + temp4gnodelisting->primtime;
   newaissize = newaissize + temp4gnodelisting->aissize;
   nodecount = nodecount - 1;
   printf(" %d ", qtemp->id);
   qtemp->assigned = true;
```

```
      chained = true;
      temp5gnodelisting = temp4gnodelisting;
   };
   if (chained) {
      printf("\n");
      printf("The new node number is: ");
      printf(" %d ",temp->id);
      printf("\n");
      printf("The new execution time is: ");
      printf(" %ld ",temp->width);
      printf("\n");
      printf("The new AIS size is: ");
      printf(" %d ",newaissize);
      printf("\n");
      printf("The input queue(s) is/are: ");
      temptrtoptr = temp5gnodelisting->
                    getgnodeinputqslist(temp5gnodelisting->nodeid);
      while(temptrtoptr != NULL) {
         printf(" %d ",temptrtoptr->getgqueueid());
         temptrtoptr = temptrtoptr->getnextelement();
      };
      printf("\n");
      printf("The output queue(s) is/are: ");
      temptrtoptr = temp3gnodelisting->
                    getgnodeoutputqslist(temp3gnodelisting->nodeid);
      while(temptrtoptr != NULL) {
         printf(" %d ",temptrtoptr->getgqueueid());
         temptrtoptr = temptrtoptr->getnextelement();
      };
      printf("\n");
      chainnum = chainnum + 1;
   };
};

// Description  : Chains nodes down first until one of the
//   conditions discussed in Chapter III is not met. Then it will
//   chain up the graph until one of the required conditions is
//   not met.
// Parameters   : temp3gnodelisting - starting node to begin chain
//                q                 - the beginning of the topgraph
//                 temp               - temporary topgraph
//                 qtemp              - temporary topgraph
//                 cyl                - the cylinder
//                nodecount          - number of nodes remaining to
//                                       be assigned to the cylinder
//                 widthavg           - cylinder slot size
//                chainnum           - The number of the chain being
//                                       built.
void topgraph::chaindwnup(gnode *temp3gnodelisting, topgraph *q,
               topgraph *temp, topgraph *qtemp, cyltype *cylinder,
               int &nodecount, long int widthavg, int &chainnum) {
```

```
ptrtoptrtoaq    *temptrtoptr = NULL;
gnode           *temp4gnodelisting = gnodelisting;
gnode           *temp5gnodelisting = temp3gnodelisting;
gnode           *temp6gnodelisting = temp3gnodelisting;
boolean         chained      = false;
int             newaissize   = 0;
int             gnodeoutqnum = 0;
int             qnodeoutnum  = 0;
int             gnodeinqnum  = 0;
int             qnodeinnum   = 0;

newaissize = temp3gnodelisting->aissize;
temptrtoptr = temp3gnodelisting->
             getgnodeoutputqslist(temp3gnodelisting->nodeid);
gnodeoutqnum = temptrtoptr->getgqueueid();
qnodeoutnum = temptrtoptr->getqnodeoutnum(gnodeoutqnum);
while (qnodeoutnum != temp4gnodelisting->nodeid){
   temp4gnodelisting = temp4gnodelisting->nextgnode;
};
qtemp->assigned = true;
while ((qtemp->id !=temp4gnodelisting->nodeid)
  && (temp4gnodelisting->iopidassigned == 0)) {
   qtemp = qtemp->next;
};
if ((qtemp->assigned == false)
  && (temp4gnodelisting->iopidassigned == 0)
  && ((temp4gnodelisting->numinqs == 1)
  && (temp4gnodelisting->numoutqs == 1))
  || ((temp4gnodelisting->numinqs == 1)
  && (temp4gnodelisting->numoutqs != 1))
  && (cylinder->checkfreecylspace(cylinder)
  >= ((temp->width/widthavg)
                + (temp4gnodelisting->primtime/widthavg)))) {
   printf("\n");
   printf("Chain number: ");
   printf("%d", chainnum);
   printf(" is made up of nodes: ");
   printf(" %d ", temp->id);
};
while ((temp4gnodelisting->numinqs == 1)
  && (temp4gnodelisting->numoutqs == 1)
  && (temp4gnodelisting->iopidassigned == 0)
  && (qtemp->assigned == false)
  && (cylinder->checkfreecylspace(cylinder)
  >= ((temp->width/widthavg)
                + (temp4gnodelisting->primtime/widthavg)))) {
   temp->width = temp->width + temp4gnodelisting->primtime;
   newaissize = newaissize + temp4gnodelisting->aissize;
   nodecount = nodecount - 1;
   printf(" %d ", qtemp->id);
   qtemp->assigned = true;
```

```
   chained = true;
   temp5gnodelisting = temp4gnodelisting;
   temptrtoptr = temp4gnodelisting->
                 getgnodeoutputqslist(temp4gnodelisting->nodeid);
   gnodeoutqnum = temptrtoptr->getgqueueid();
   qnodeoutnum = temptrtoptr->getqnodeoutnum(gnodeoutqnum);
   while (qnodeoutnum != temp4gnodelisting->nodeid){
      temp4gnodelisting = temp4gnodelisting->nextgnode;
   };
   while ((qtemp->id !=temp4gnodelisting->nodeid)
     && (temp4gnodelisting->iopidassigned == 0)) {
      qtemp = qtemp->next;
   };
};
if ((temp4gnodelisting->numinqs == 1)
  && (temp4gnodelisting->numoutqs != 1)
  && (temp4gnodelisting->iopidassigned == 0)
  && (qtemp->assigned == false)
  && (cylinder->checkfreecylspace(cylinder)
  >= ((temp->width/widthavg)
                + (temp4gnodelisting->primtime/widthavg)))) {
   temp->width = temp->width + temp4gnodelisting->primtime;
   newaissize = newaissize + temp4gnodelisting->aissize;
   nodecount = nodecount - 1;
   printf(" %d ", qtemp->id);
   qtemp->assigned = true;
   chained = true;
   temp5gnodelisting = temp4gnodelisting;
};
temptrtoptr = temp3gnodelisting->
              getgnodeinputqslist(temp3gnodelisting->nodeid);
gnodeinqnum = temptrtoptr->getgqueueid();
qnodeinnum = temptrtoptr->getqnodeinnum(gnodeinqnum);
temp4gnodelisting = gnodelisting;
while (qnodeinnum != temp4gnodelisting->nodeid){
   temp4gnodelisting = temp4gnodelisting->nextgnode;
};
while ((qtemp->id !=temp4gnodelisting->nodeid)
  && (temp4gnodelisting->iopidassigned == 0)) {
   qtemp = qtemp->next;
   if (qtemp == NULL) {
      qtemp = q;
   };
};
if ((qtemp->assigned == false)
 && (temp4gnodelisting->iopidassigned == 0)
 && (chained == false) && (temp4gnodelisting->numoutqs == 1)) {
   printf("\n");
   printf("Chain number: ");
   printf("%d", chainnum);
   printf(" is made up of nodes: ");
```

```
   printf(" %d ", temp->id);
};
while ((temp4gnodelisting->numinqs == 1)
  && (temp4gnodelisting->numoutqs == 1)
  && (temp4gnodelisting->iopidassigned == 0)
  && (qtemp->assigned == false)
  && (cylinder->checkfreecylspace(cylinder)
  >= ((temp->width/widthavg)
                  + (temp4gnodelisting->primtime/widthavg)))) {
   temp->width = temp->width + temp4gnodelisting->primtime;
   newaissize = newaissize + temp4gnodelisting->aissize;
   nodecount = nodecount - 1;
   printf(" %d ", qtemp->id);
   qtemp->assigned = true;
   chained = true;
   temp6gnodelisting = temp4gnodelisting;
   temptrtoptr = temp4gnodelisting->
                 getgnodeinputqslist(temp4gnodelisting->nodeid);
   gnodeinqnum = temptrtoptr->getgqueueid();
   qnodeinnum = temptrtoptr->getqnodeinnum(gnodeinqnum);
   temp4gnodelisting = gnodelisting;
   while (qnodeinnum != temp4gnodelisting->nodeid){
      temp4gnodelisting = temp4gnodelisting->nextgnode;
   };
   while ((qtemp->id !=temp4gnodelisting->nodeid)
     && (temp4gnodelisting->iopidassigned == 0)) {
      qtemp = qtemp->next;
      if (qtemp == NULL) {
         qtemp = q;
      };
   };
};
if ((temp4gnodelisting->numinqs != 1)
  && (temp4gnodelisting->numoutqs == 1)
  && (temp4gnodelisting->iopidassigned == 0)
  && (qtemp->assigned == false)
  && (cylinder->checkfreecylspace(cylinder)
  >= ((temp->width/widthavg)
                  + (temp4gnodelisting->primtime/widthavg)))) {
   temp->width = temp->width + temp4gnodelisting->primtime;
   newaissize = newaissize + temp4gnodelisting->aissize;
   nodecount = nodecount - 1;
   printf(" %d ", qtemp->id);
   qtemp->assigned = true;
   chained = true;
   temp6gnodelisting = temp4gnodelisting;
};
if (chained) {
   printf("\n");
   printf("The new node number is: ");
   printf(" %d ",temp->id);
```

```
            printf("\n");
            printf("The new execution time is: ");
            printf(" %ld ",temp->width);
            printf("\n");
            printf("The new AIS size is: ");
            printf(" %d ",newaissize);
            printf("\n");
            printf("The input queue(s) is/are: ");
            temptrtoptr = temp6gnodelisting->
                          getgnodeinputqslist(temp6gnodelisting->nodeid);
            while(temptrtoptr != NULL) {
               printf(" %d ",temptrtoptr->getgqueueid());
               temptrtoptr = temptrtoptr->getnextelement();
            };
            printf("\n");
            printf("The output queue(s) is/are: ");
            temptrtoptr = temp5gnodelisting->
                          getgnodeoutputqslist(temp5gnodelisting->nodeid);
            while(temptrtoptr != NULL) {
               printf(" %d ",temptrtoptr->getgqueueid());
               temptrtoptr = temptrtoptr->getnextelement();
            };
            printf("\n");
            chainnum = chainnum + 1;
         };
      };
```

APPENDIX C: CODE TO ASSIGN INDICES TO NODES

```
// Description  : This code implements the assigning of node
//   indices as discussed in Chapter IV. It is implemented by
//   checking the graph for parents and children of the
//   input node and calls itself recursively until all
//   nodes receive the proper index.
// Parameters   : tempgnodelisting - The node to start with
//                index       - The index a node is to be assigned.
void gnode :: assignindex(gnode *tempgnodelisting, int index) {

   gnode        *temp2gnodelisting = tempgnodelisting;
   ptrtoptrtoaq *temptrtoptr = NULL;
   int          gnodeinqnum  = 0;
   int          qnodeinnum   = 0;
   int          gnodeoutqnum = 0;
   int          qnodeoutnum  = 0;

   tempgnodelisting->index = index;
   temptrtoptr = tempgnodelisting->
                 getgnodeinputqslist(tempgnodelisting->nodeid);
   while(temptrtoptr != NULL) {
// get the node's parents but not iops
      gnodeinqnum = temptrtoptr->getgqueueid();
      qnodeinnum = temptrtoptr->getqnodeinnum(gnodeinqnum);
      temp2gnodelisting = gnodelisting;
      while(qnodeinnum != temp2gnodelisting->nodeid) {
         temp2gnodelisting = temp2gnodelisting->nextgnode;
      };
      if(temp2gnodelisting->iopidassigned != 0) {
      }
      else {
         if(temp2gnodelisting->index == -10000) {
            if(temp2gnodelisting->finishtime >
                                  tempgnodelisting->startime) {
               assignindex(temp2gnodelisting, index + 1);
            }
            else {
               assignindex(temp2gnodelisting, index);
            };
         };
      };
   temptrtoptr = temptrtoptr->getnextelement();
   };

   temptrtoptr = tempgnodelisting->
                 getgnodeoutputqslist(tempgnodelisting->nodeid);
```

```
   while(temptrtoptr != NULL) {
// now get the node's children but not iops
      gnodeoutqnum = temptrtoptr->getgqueueid();
      qnodeoutnum = temptrtoptr->getqnodeoutnum(gnodeoutqnum);
      temp2gnodelisting = gnodelisting;
      while(qnodeoutnum != temp2gnodelisting->nodeid) {
         temp2gnodelisting = temp2gnodelisting->nextgnode;
      };
      if(temp2gnodelisting->iopidassigned != 0) {
      }
      else {
         if ((temp2gnodelisting->index = -10000)
         || (temp2gnodelisting->index >= index)) {
            if(temp2gnodelisting->startime <
                                 tempgnodelisting->finishtime) {
               assignindex(temp2gnodelisting, index - 1);
            }
            else {
               assignindex(temp2gnodelisting, index);
            };
         };
      };
   temptrtoptr = temptrtoptr->getnextelement();
   };
};
```

APPENDIX D: CODE FOR CREATION OF DEPENDENCIES

```
// Description  : Creates dependencies based on a node's index and
//   start and finish times in the cylinder. Threshold, Consume,
//   and Produce values are resolved as described in Chapter IV.
// Parameters   : tempgnodelisting - The node to start with
//                circum   - The circumference of the cylinder
dependencyqs *topgraph :: createdeps(gnode *tempgnodelisting,
                                long int circum) {

   gnode        *temp2gnodelisting = tempgnodelisting;
   gnode        *temp3gnodelisting;
   gnode        *temp4gnodelisting;
   dependencyqs *headdepq = NULL;
   dependencyqs *tempheaddepq;
   int          indexnr = 0;
   int          indexns = 0;
   long int     largestcylentry = 0;
   boolean      needependency = true;

   while (temp2gnodelisting != NULL) {
      if (temp2gnodelisting->iopidassigned != 0){
         temp2gnodelisting = temp2gnodelisting->nextgnode;
      }
      else {
         indexnr = temp2gnodelisting->index;
         while (needependency == true){
            needependency = false;
            temp3gnodelisting = tempgnodelisting;
            while (temp3gnodelisting != NULL){
               if ((temp3gnodelisting->iopidassigned != 0)
                  ||(temp3gnodelisting->nodeid ==
                                 temp2gnodelisting->nodeid)) {
               }
               else {
                  if (((temp3gnodelisting->finishtime % circum)
                    <= temp2gnodelisting->startime)
                    && (temp3gnodelisting->finishtime
                    >= largestcylentry)
                    && (temp2gnodelisting->index !=
                                       temp3gnodelisting->index)
                    && (headdepq->alreadydeps
                          (headdepq,temp3gnodelisting->nodeid,
                          temp2gnodelisting->nodeid) == false)) {
                     temp4gnodelisting = temp3gnodelisting;
                     largestcylentry =
                                    temp3gnodelisting->finishtime;
```

```
                needependency = true;
             };
          };
        temp3gnodelisting = temp3gnodelisting->nextgnode;
        };
        indexns = temp4gnodelisting->index;
        if (needependency) {
           if (headdepq == NULL) {
              if (!(headdepq = new dependencyqs)){
                 fprintf (stderr, "insufficient memory for
                                              dependencyqs\n");
                 exit(1);
              };
              if (indexnr >= indexns){
                headdepq->deptokensize=(indexnr-indexns) * 4;
                 headdepq->threshold = 4;
//consume initialized to 4 bytes in constructor
                 headdepq->nodefrom =
                                    temp4gnodelisting->nodeid;
                headdepq->nodeto = temp2gnodelisting->nodeid;
              }
              else {
                 headdepq->threshold=(indexns - indexnr) * 4;
// initial tokens set to 0 in constructor
// consume initialized to 4 bytes in constructor
                 headdepq->nodefrom =
                                    temp4gnodelisting->nodeid;
                headdepq->nodeto = temp2gnodelisting->nodeid;
              };
           tempheaddepq = headdepq;
           }
           else {
              if(!(tempheaddepq->nextdepq=new dependencyqs)) {
                 fprintf (stderr, "insufficient memory for
                                              dependencyqs\n");
                 exit(1);
              };
              if (indexnr >= indexns){
                 tempheaddepq->nextdepq->deptokensize =
                                       (indexnr - indexns) * 4;
                 tempheaddepq->nextdepq->threshold = 4;
                           //in bytes
// consume initialized to 4 bytes in constructor
                 tempheaddepq->nextdepq->nodefrom =
                                    temp4gnodelisting->nodeid;
                 tempheaddepq->nextdepq->nodeto =
                                    temp2gnodelisting->nodeid;
              }
              else {
                 tempheaddepq->nextdepq->threshold =
                                       (indexns - indexnr) * 4;
```

```
            // initial tokens set to 0 in constructor
            // consume initialized to 4 bytes in constructor
                    tempheaddepq->nextdepq->nodefrom =
                                        temp4gnodelisting->nodeid;
                    tempheaddepq->nextdepq->nodeto =
                                        temp2gnodelisting->nodeid;
                  };
               tempheaddepq = tempheaddepq->nextdepq;
               };
             };
          };
          temp2gnodelisting = temp2gnodelisting->nextgnode;
          needependency = true;
          largestcylentry = 0;
      };
   };
   tempheaddepq = headdepq;
   printf("\nThe following dependencies need to be assigned\n");
   printf("All values are presented in bytes to interface with NPS
                                                    simulator\n");
   while(tempheaddepq != NULL) {
      printf("From: ");
      printf(" %d ",tempheaddepq->nodefrom);
      printf("   To: ");
      printf(" %d ",tempheaddepq->nodeto);
      printf("   Tokensize: ");
      printf(" %d ",tempheaddepq->deptokensize);
      printf("   Threshold: ");
      printf(" %d ",tempheaddepq->threshold);
      printf("   Consume: ");
      printf(" %d ",tempheaddepq->consume);
      printf("\n");
      tempheaddepq = tempheaddepq->nextdepq;
   };
   return headdepq;
};
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center — 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Library, Code 52 — 2
   Naval Postgraduate School
   Monterey, CA 93943-5002

3. Chairman, Code EC — 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5000

4. Professor Shridhar Shukla, Code EC/Sh — 2
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5000

5. Professor Amr Zaky, Code CS/Za — 1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5000

6. LCDR Steve Kasputis — 1
   Department of the Navy
   Naval Sea Systems Command (PMS 412)
   Washington, DC 20362-5101

7. Mr. David Kaplan — 1
   Commander of Naval Research Laboratory
   4555 Overlook Avenue
   S. W. Washington, DC 20375-5000

8. Mr. Richard Stevans — 1
   Commander of Naval Research Laboratory
   4555 Overlook Avenue
   S. W. Washington, DC 20375-5000

9. Mr. Jerome L. Uhrig, WH 46243 — 1
   American Telephone and Telegraph Bell Laboratories
   67 Whippany Road
   P.O. Box 903
   Whippany, NJ 07981-0903

10. LT Harold A. Bell, USN 1
Air Department, V-2 Division
USS Saratoga (CV-60)
FPO AA 34078-2740