Active

Project #: E-21-F09  
Center #: 10/24-6-R7192-0A0
Contract#: N00014-91-C-2081
Subprojects #: N
Main project #: 

Project unit:  
Project director(s): 
YALAMANCHILI S  
ELEC ENGR  
(404)894-2940  

Sponsor/division names: NAVY  
Sponsor/division codes: 103

Award period:  910401 to 920331 (performance) 920530 (reports)

Sponsor amount  
Contract value  0.00
Funded  0.00
Total to date  51,868.00
Cost sharing amount  0.00

Does subcontracting plan apply #: N

Title: PARTITIONING ALGORITHMS FOR SIGNAL FLOW GRAPHS

PROJECT ADMINISTRATION DATA

OCA contact: E. Faith Gleason  
Sponsor technical contact  
ANDREW FOX  
(202)404-7439

Sponsor issuing office  
CONTRACTING OFFICER, CODE 3230.LR  
(202)404-7439

CODE 8120  
NAVAL RESEARCH LABORATORY  
4555 OVERLOOK AVENUE, SW  
WASHINGTON, DC 20375-5000

Security class (U,C,S,TS): U  
ONR resident rep. is ACO (Y/N): Y
Defense priority rating : N/A

Equipment title vests with: Sponsor X

WRITTEN AUTHORIZATION FROM SPONSOR REQUIRED IF OVER $1,000.
Administrative comments -  
GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION

NOTICE OF PROJECT CLOSEOUT

Closeout Notice Date 03/12/93

Project No. E-21-F09 ________ Center No. 10/24-6-R7192-0A0 ______

Project Director YALAMANCHILI S ________ School/Lab ELEC ENGR ______

Sponsor NAVY/NAVAL RESEARCH LAB, DC ________________________________

Contract/Grant No. N00014-91-C-2081 ________ Contract Entity GTRC

Prime Contract No. __________________________

Title PARTITIONING ALGORITHMS FOR SIGNAL FLOW GRAPHS ________________________________

Effective Completion Date 920331 (Performance) 920530 (Reports)

Closeout Actions Required: Y/N Submitted

Final Invoice or Copy of Final Invoice Y __________
Final Report of Inventions and/or Subcontracts Y __________
Government Property Inventory & Related Certificate Y __________
Classified Material Certificate N __________
Release and Assignment Y __________
Other N __________

Comments EFFECTIVE DATE 4-1-91. CONTRACT VALUE $51,868.

Subproject Under Main Project No. __________

Continues Project No. __________

Distribution Required:

Project Director Y
Administrative Network Representative Y
GTRI Accounting/Grants and Contracts Y
Procurement/Supply Services Y
Research Property Management Y
Research Security Services N
Reports Coordinator (OCA) Y
GTRC Y
Project File Y
Other HARRY VANN-FMD ________________ Y
FRED CAIN-OOD ________________ Y

NOTE: Final Patent Questionnaire sent to PDPI.
Technical Progress Report

for

Partitioning Algorithms for Signal Flow Graphs

submitted to

Naval Research Laboratory

Period Covered: April 1st - May 31st

Technical Contact

Sudhakar Yalamanchili
Tel: (404) 894-2940
e-mail: sudha@eeecom.gatech.edu

Technical Contact

Administrative Contact

Matt Gedney
Tel: (404) 894-4817

School of Electrical Engineering
Computer Engineering Program
Georgia Institute of Technology
Atlanta, Georgia 30332-0250

Georgia Tech
1 Introduction

This report documents progress covering the period from April 1st 1991 through May 31st 1991.

2 Summary

During this period one student Mr. C. deCastro, was supported by this contract at half time. Progress includes,

1. A model of the operation of a multicluster ALPS system and definition of the operation of intercluster Interface Control Units (ICUs).
2. Implementation of a partitioning algorithm based on simulated annealing using adaptive schedules.
3. Some experimentation using a workload generator to create sample signal flow graphs.

The following section briefly describes the work completed during this period.

3 Technical Progress

3.1 System Model

The target architecture for the mapping algorithm is the ALPS system being developed at NRL. The component elements of ALPS may be general purpose microprocessors, signal processors, or ASICs. Collections of the primitive elements are interconnected into "clusters" of processors. Each type of ALPSs primitive within a cluster is assigned a element type - each type being capable of executing some functions, e.g., FFT, filtering, etc. The composition of a cluster is fixed prior to partitioning and mapping of the flow graph. Execution times for each signal flow graph node (referred to as a task) on each element type is tallied in an array, the execution time matrix. In this matrix, element $(i,j)$ indicates the time needed for element $i$ to execute task $j$. Element $(i,j)$ is zero if task $j$ may not be executed by device $i$.

We assume there will be multiple clusters of processors. The use of multiple clusters is often necessitated by the limited bandwidth available on a single clusters' data circus. The interconnection scheme between clusters is pre-specified and fixed at mapping time in the following manner. A cluster adjacency matrix indicates how the various clusters are interconnected, and an additional array indicates what processing resources each cluster contains, i.e., the device types, and their number. Clusters may differ in the number of elements of each type that they are comprised of.

Physically, multiple clusters are connected by intercluster ICUs. Each intercluster ICU will connect two clusters. These ICUs are somewhat different than intracluster ICUs in that they must intercept messages and data from one cluster and route the data to the appropriate target cluster. Sending data from one cluster to an adjacent cluster is a two phase process. Essentially, the intercluster ICU appears, to the source, as the desired destination. In one step the data is transferred to
the intercluster ICU. Next, the ICU must offer its data to the target. The target may be the original intended target, (i.e., a processing node), or it may be yet another intercluster ICU which must forward the data again. Thus, there is a factor of two difference between intercluster and intracluster transfers. Beyond the speed factor, such intercluster ICUs must provide an arbitration mechanism for received data. It is conceivable that an intercluster ICU is unable to forward its data for lack of a suitable recipient. If new data is received prior to the sending of this data, it is desirable to have the necessary buffering and arbitration mechanisms to correctly forward the new data in spite of the fact that the previous data has yet to be delivered.

Flowgraphs that are to execute on this system are also described via an adjacency matrix. Further, associated with each node there are two parameters, the task that the node is to perform and the size of the input queue for the task.

3.2 Partitioning Algorithm

During this reporting period we have implemented a partitioning and mapping algorithm using adaptive schedules for simulated annealing. A workload generator which produces directed acyclic graphs (DAGs) written by Ajay Mohindra has been adapted to our development platform. The annealing algorithm starts with a partition of the flow graph. The number of partitions is equal to the number of clusters, and each partition is assigned to a cluster.

In implementing the simulated annealing algorithm, several different move strategies have been implemented as well as several energy functions. The annealing algorithm may move a randomly selected task node from a cluster to another neighboring cluster within the constraints of the problem (i.e. a node will not be allowed to move to a cluster where there is not adequate processing capability or the right device type.) This move strategy is used to "fine tune" the resulting partition at low temperatures to generate better partitions. Multiple parallel moves may also be made whereby several task nodes are moved at once to permissible neighboring clusters. Such parallel moves allow the annealing algorithm to survey the solution space in a more coarse manner. These moves are useful at high temperatures where the algorithm is essentially picking a region of the search space to focus on.

The energy function being used consists of two parts, a computational energy metric and a communication energy metric. Presently, the compute metric produces a measure of computational work by assuming (worst case) that all tasks assigned to a given cluster must be executed sequentially on the slowest device able to perform them. The metric is further affected by the number of clusters in the target system which have no work assigned to them. This portion of the energy function attempts to force partitions to be balanced. The communication energy metric captures intercluster communication. Presently, two nodes that are in the same cluster are considered to have no communication overhead. This component of the energy function attempts to minimize intercluster communication at the expense of load balancing.

An example of the application of the partitioning algorithm to a flow graph is attached. The nodes of the flowgraph are encoded as follows. The shade represents the cluster that the node is assigned to. The N and T notation are the node number and element type required for execution respectively.
4 Plans for the Next Reporting Period

The effort for the next reporting period will include Mr. C. deCastro at half time and Dr. Yalamanchili at half time. During the next reporting period we plan to,

1. Further experiment with the annealing algorithm to improve its performance.

2. Incorporate bandwidth constraints into the system, i.e., intracluster. This will involve incorporating the input queue sizes of the attached elements to determine a more accurate measure of communication overhead.

3. Complete the implementation of the partitioning algorithm based on the branch and bound heuristic

4. Complete the implementation of additional graph theoretic partitioning heuristics.

By the end of the summer quarter we intend to have completed implementation of all of three partitioning algorithms. The final reporting period will then concentrate on experimentation, improvements, interface to Albert, and if possible, new algorithms.

5 Problems and Potential Problems

None at this time.
EXAMPLE: The following mapping was produced using a 2 phased annealing. The annealing was run four times using multiple parallel moves based on the flowgraph structure. Then, three runs were made using single intercluster moves so as to fine tune the mapping.

FLOWGRAPH

<table>
<thead>
<tr>
<th>TASK</th>
<th>EXECUTION TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>5 units</td>
</tr>
<tr>
<td>T1</td>
<td>10 units</td>
</tr>
<tr>
<td>T2</td>
<td>1 unit</td>
</tr>
<tr>
<td>T3</td>
<td>2 units</td>
</tr>
</tbody>
</table>

Energy of mapping: 31.25 units
Minimum energy encountered: 26.25 units

SYSTEM ARCHITECTURE

Cluster 0
- 3 device types
- Type # 1,2,3

Cluster 2
- 3 device types
- Type # 1,2,3

Cluster 3
- 2 device types
- Type 0,1
Technical Progress Report

for

Partitioning Algorithms for Signal Flow Graphs

submitted to

Naval Research Laboratory

Technical Contact

Sudhakar Yalamanchili
Tel: (404) 894-2940
e-mail: sudha@ee.com.gatech.edu

Administrative Contact

Matt Gedney
Tel: (404) 894-4817

School of Electrical Engineering
Computer Engineering Program
Georgia Institute of Technology
Atlanta, Georgia 30332-0250

Georgia Tech
1 Introduction

This report documents progress covering the period from June 1st 1991 through September 30th 1991.

2 Summary

During this period one student, Mr. C. deCastro was supported one half time, and Dr. Yalamanchili was supported one half time. Progress includes,

1. Implementation of, and experimentation with, a partitioning algorithm based on the branch and bound paradigm
2. Implementation of, and experimentation with, of a partitioning algorithm based on critical path heuristics.
3. Submission of one paper to a workshop on Heterogeneous Architectures
4. Obtaining and beginning study of the draft specifications of the Scalable Coherent Interface Protocol as a interconnect for ALPS-type architectures

The following section briefly describes the work completed during this reporting period.

3 Technical Progress

This reporting period has focused on the implementation of the two remaining partitioning algorithms - the branch and bound algorithm (BBA) and the critical path heuristic (CPH). The BBA starts from the empty partition in which no signal flow graph node is assigned to a target ALPS cluster. This partition is expanded to assign one flow graph node to all possible clusters producing a number of partial partitions. Each partial partition has associated with it a cost that is a lower bound on the cost of any partition that is generated starting from that partial partition. The BBA iteratively expands the partition with the lowest cost. The node selected for expanding the least cost partial partition at each step is the "heaviest" of all of the remaining signal flow graph nodes. The weight of a signal flow graph node is computed as the sum of its longest execution time (i.e., on the slowest target processor), and the communication costs of all incident edges (assuming they represent communication between adjacent clusters). The main loop of the partitioning algorithm appears as shown below.
initialize();

while (!done()){
    partition = get_partial_partition();
    expand_candidates(partition);
}

The present implementation of the BBA takes excessively long run times for larger problems (e.g., 20-30 nodes, 4-6 clusters). Run times exceed several hours on a Sparcstation. We would like to improve the implementation to make the run times compatible with an interactive design environment. However, it is possible that this cannot be achieved without compromising the quality of the solutions that can be found by the BBA. This problem will be addressed in the next reporting period.

The critical path heuristic begins by leveling the signal flow graph. This leveling provides a coarse ordering on flow graph nodes that is used in their placement. All flowgraph nodes that are at level zero will be assigned to the lowest numbered cluster able to provide the required resource. When this operation is completed, level zero is said to have been placed. Then, for each node at level i, successor nodes in the next level, i + 1, are placed. For each node at level i, the successor node at the next level, i + 1, connected via the heaviest edge is placed in the same cluster if possible. If this placement is not possible (perhaps due to a lack of processing power in a cluster), the node is placed in the closest cluster capable of performing the nodes task. The remaining successor nodes at the next level, i + 1, are distributed across the available clusters so as to permit parallelism among the tasks at this level. Specifically, nodes are placed in order of ascending node number and in a manner that attempts to evenly distribute the remaining nodes over the neighboring clusters (in order of increasing cluster number). When such a placement may not be performed on a given node, it is placed in the nearest cluster able to perform its' task. This process is continued until the next to last level when all nodes will have been placed.

This heuristic makes no use of computation overhead nor does it consider edges which transit from level i to level i + j where j \geq 2. This will result in a poor partitioning when such edges are very heavy relative to the remaining edges. A related issue in testing CPH is that the artificially created examples were constructed to enable analysis of the results. The examples created to test the annealing algorithms are not necessarily representative, since they have unique features that make it possible to make some judgment about the performance of the annealing algorithm. CPH was designed with the structure of more general directed signal flow graphs in mind. We feel that its performance will improve as we apply it to more realistic flow graphs. We plan to use the optimal partitions created by BBA to evaluate the performance of this algorithm.
4 Plans for the Next Quarter

The effort for the next reporting period will include Mr. C. deCastro at half time and Dr. Yalamanchili at one quarter time. During the next reporting period we plan to,

1. Improve the implementations, primarily in the area of speed and memory requirements for the branch and bound algorithm
2. Experimentation and performance evaluation over a range of signal flow graphs.
3. Study implementation issues with respect to a SCI based ALPS architecture
4. Integrate the three partitioning algorithms into a single package.

5 Problems and Potential Problems

None
Technical Progress Report

for

Partitioning Algorithms for Signal Flow Graphs

submitted to

Naval Research Laboratory

Technical Contact
Sudhakar Yalamanchili
Tel: (404) 894-2940
e-mail: sudha@eeom.gatech.edu

Administrative Contact
Matt Gedney
Tel: (404) 894-4817

School of Electrical Engineering
Computer Engineering Program
Georgia Institute of Technology
Atlanta, Georgia 30332-0250

Georgia Tech
1 Introduction

This report documents progress covering the period from October 1st 1991 through March 31st 1992.

2 Summary

During this period one student, Mr. C. de Castro was supported one half time, and Dr. Yalamanchili was supported one quarter time. Progress includes,

1. Completion of the three partitioning algorithms.
2. Submission of the program results as an archival journal publication.
3. Acquisition of the Ptolemy simulation environment, and initial design of simulation of the ALPS concept

The following section briefly describes the work completed during this reporting period.

3 Technical Progress

During this reporting period, our research has produced three partitioning algorithms for partitioning signal flow graphs for execution on a class of heterogeneous signal processing architectures. The specific class of architectures embody the ALPS model of computation developed at the Naval Research Laboratories. The first partitioning algorithm is an adaptive variant of simulated annealing. The second is an edge elimination heuristic which follows a critical path through the signal flow graph in assigning flow graph functions to clusters. The last method is an application of the branch and bound state space search paradigm. The traditional paradigm is augmented with a simple heuristic which speeds the search for desirable partitions while sacrificing the guarantee of finding optimal partitions.

Further, we have evaluated the performance of these three algorithms. Our experimental results are encouraging, indicating that the use of such partitioning algorithms can significantly improve the performance of ALPS type architectures. For example, in a simple manual simulation of the execution of a seven node signal flow graph, less than a second of partitioning time reduced the execution time of the flow graph by 25%. These and further results have been compounded into a journal paper which has been submitted to the International Journal of Computer Simulation.

Lastly, the Ptolemy design environment has been installed and studied as a potential design environment for the simulation of architectures embodying the ALPS model of computation.
4 Plans for the Next Quarter

We foresee an environment where the application and system designer will iteratively simulate architectural variations of the target system before implementation. The goal in simulating DSP architectures is to model all of the hardware, software, and I/O phenomena that affect performance so that design decisions can be made to maximize performance.

Toward this end, in the next reporting period, we hope to have a discrete-event simulation of architectures which embody the self-scheduling, self-synchronizing model of computation that is inherent in the ALPS model. The simulation is being developed within the Ptolemy design environment which has been under development at the University of California, Berkeley. Integration of the partitioning algorithms will follow.

With a design environment and partitioning algorithms, we hope to make more accurate performance studies. These studies will allow a better understanding of the effects of application partitioning on application performance. Further, more accurate measures of partition quality may be developed.

5 Problems and Potential Problems

None
Volume in drive A is NRL_PARTIT
Volume Serial Number is OF33-10F1
Directory of A:

SA&EEH <DIR> 06-18-92 9:42a
BB <DIR> 06-18-92 9:42a
  2 file(s) 0 bytes
    1323008 bytes free

Volume in drive A is NRL_PARTIT
Volume Serial Number is OF33-10F1
Directory of A:\SA&EEH

.. <DIR> 06-18-92 9:42a
.
ANNEAL C 16837 06-18-92 9:43a
DEFINES H 724 06-18-92 9:43a
GRAPH C 15256 06-18-92 9:44a
IO C 6727 06-18-92 9:45a
MAP1 C 10306 06-18-92 9:45a
MATRIX C 3002 06-18-92 9:45a
METRIC C 5901 06-18-92 9:45a
RANDOM C 411 06-18-92 9:45a
SA1 C 21491 06-18-92 9:46a
STRUCTS H 3375 06-18-92 9:46a
TIME C 298 06-18-92 9:46a
  13 file(s) 84328 bytes
    1323008 bytes free

Volume in drive A is NRL_PARTIT
Volume Serial Number is OF33-10F1
Directory of A:\BB

.. <DIR> 06-18-92 9:42a
.
ANNEAL C 601 06-18-92 9:48a
BB C 1657 06-18-92 9:48a
BBFUNC C 26395 06-18-92 9:49a
GLOBAL H 3361 06-18-92 9:49a
GRAPH C 4480 06-18-92 9:49a
IO C 9180 06-18-92 9:50a
  8 file(s) 45674 bytes
    1323008 bytes free
Final Report

for

Partitioning Coarse-Grain Signal Flow Graphs for Heterogeneous DSP Architectures

submitted to

Naval Research Laboratory

Technical Contact                Administrative Contact

Sudhakar Yalamanchili            Matt Gedney
Tel: (404) 894-2940              Tel: (404) 894-4817
e-mail: sudha@eeom.gatech.edu

Researchers

C. de Castro and S. Yalamanchili

Computer Systems Research Laboratory

School of Electrical Engineering
Computer Engineering Program
Georgia Institute of Technology
Atlanta, Georgia 30332-0250

Georgia Tech
Abstract

This research addresses a problem in accurately modeling and evaluating the performance of a class of heterogeneous digital signal processing (DSP) architectures. These architectures can be tailored to match the computational requirements of specific applications by utilizing a mix of special purpose hardware and off-the-shelf programmable components. In such an environment, scheduling of computations to minimize communication while maximizing load balancing is often the principal determinant of performance, and is a formidable and challenging task. In this research, we describe three algorithms for partitioning coarse grain signal flow graphs for execution on hierarchically structured, heterogeneous, DSP architectures. Examples are provided to demonstrate how the application of such algorithms during modeling and simulation of a candidate design can produce architectures with improved performance. We conclude with a brief description of current efforts to include these partitioning algorithms as part of an environment for the specification, design, and analysis of high performance, heterogeneous DSP architectures[16].

1 Introduction

This research focuses on application specific, heterogeneous parallel architectures for high performance digital signal processing. These architectures can be tailored to match specific applications by using a mix of custom hardware and off-the-shelf programmable components. However, they also present new problems in the mapping and scheduling of signal processing functions. As a result of communication latency and synchronization overhead, poor schedules will produce dramatically poor performance, regardless of the computational speed of the individual components. Therefore it is necessary to be able to study the effects of such scheduling algorithms as a integral part of environments for the specification, design, modeling, and simulation of signal processing architectures. This report motivates this observation, describes three partitioning algorithms for signal flow graphs, and demonstrates how they may be used as a part of the design process in developing application specific, heterogeneous DSP architectures. We demonstrate how these algorithms can be used to during modeling and simulation to avoid run time performance bottlenecks that arise from resource contention (i.e., for processors, communication links, memories, etc), which in turn are caused by poor scheduling of computations.

Many signal processing applications may be expressed in terms of signal flow graphs where the nodes represent functions, e.g., FFT, FIR filtering, etc., and edges represent data dependencies between the functions [1]. Often a small number of such functions are responsible for a majority of the computations. Heterogeneous architectures become attractive in such applications. Special purpose hardware, e.g., application specific integrated circuits (ASICs), can be used to execute the most compute intensive subtasks, while programmable signal processors (SPUs), commercial array processors, etc., may be used for other less demanding functions. By having a number of different PE types of varying complexity/performance, one can tailor the architecture to match a specific application, or set of applications [10, 3].

Due to the presence of programmable components, functions typically may be performed by more than one processing element (PE). Further, inter-PE bandwidth constraints typically necessitate grouping PEs into clusters of varying compositions of PE types. Scheduling flow graphs for execution must now account for these various constraints, as well as minimize inter-cluster communication. This is a complex and challenging task since the execution time of functions depends upon the cluster in which they are placed as well as the composition of PE types within each particular
cluster. As a result, there will be an increased reliance on automated tools to support the partitioning and scheduling of flow graphs. We demonstrate how partitioning algorithms can be utilized to improve the performance of such architectures. The scheduling mechanisms within a cluster is a novel self-scheduling, self-synchronizing scheme that is implemented in hardware[1].

The following section describes the target architecture. Section 3 presents the overall performance model with a description of the target architecture, the algorithm model, and the metrics used to evaluate partitions. Section 4 presents the three partitioning algorithms used in this study. The first is an adaptive variant of simulated annealing. The second is an edge elimination heuristic. This heuristic follows a critical path through the flow graph, using this information in placing functions in clusters. The last method is an application of the branch and bound state space search paradigm. The traditional paradigm is augmented with a simple heuristic to speed up the search. Section 5 describes the results of the experimental evaluation of these heuristics. We conclude with some recommendations and a brief description of ongoing work. This includes the integration of these partitioning algorithms into an environment for the specification, design, and analysis of high performance, heterogeneous DSP architectures[16].

2 The ALPS Architecture

The class of heterogeneous signal processing architectures that motivate this study implement the Alternative Low-level Primitive Structures (ALPS) paradigm[3]. In this paradigm, the basis for the architecture is a small extensible set of high speed processing elements (PEs) designed specifically for signal processing applications. In the ALPS design methodology, mixtures of off-the-shelf and custom hardware tailored to specific signal processing applications are interconnected to form heterogeneous, hierarchically structured systems.

Presently, real-time signal processing applications demand tremendous computational bandwidths which are are not attainable with general purpose processing hardware. The ALPS philosophy represents a compromise between an application-specific custom design and a general-purpose design. A complete ALPS system is comprised of collections of PEs sufficient to perform a variety of signal processing applications. The hardware primitives may be general purpose microprocessors, microprogrammed digital signal processors, or application specific integrated circuits (ASICs) which embody specific functions, e.g. FFT, or filtering operations. PEs are interconnected by a common backplane, referred to as a circus in the original ALPS model. All primitive PEs share three circuses - the message circus for passing control information, the data circus for data, and the monitor circus for real-time monitoring of cluster activity. The composition of a cluster is fixed prior to the partitioning and mapping of a flow graph. In the remainder of this report we use the terms PE and hardware primitives interchangeably.

Processing primitives are bound to the circuses by a standardized system interface, referred to as an Interface Control Unit (ICU). The ICU implements all communication protocols necessary for communication with other ICUs and which queues incoming/outgoing data. Further, ICUs control the activation/de-activation of attached primitive PEs. Each ICU has a complete description of the signal flow graph to be executed, along with a specification of the tasks within the flow graph that can be executed by its associated PE. When a primitive completes the execution of a function, its ICU places a service request on the message circus for execution of the next function in the signal flow graph. The ICU of each free PEs monitors the message circus for service requests
Locally executable functions

![Diagram of a Functional Model of the ICU]

Figure 1: Illustration of a Functional Model of the ICU

it is able to service. All able ICUs bid for the function. A hardware arbitration mechanism[1] selects a single ICU to perform the function, and initiates the transfer of data between ICUs. In situations where a function requires data from multiple sources, the ICU selected to perform the function becomes the only valid candidate in bidding for subsequent data sets required for its execution. This hardware arbitration mechanism provides completely decentralized control and run-time scheduling. Thus, computations are self-scheduling and self-synchronizing. A functional model of the ICU is illustrated in figure 1. The hash table shown in the figure contains pointers to the programs for implementing each of the functions that can be executed by the associated PE. The PE operates as a slave to ICU, executing functions identified by the ICU.

Bandwidth constraints within a single cluster will necessitate the use of multiple cluster configurations. Figure 2 depicts an example of such a configuration. Specialized ICUs are used to interconnect clusters of PEs. Specifically, the inter-cluster ICUs will interconnect the message and data circuses of the interconnected clusters. To a source ICU within a cluster, the inter-cluster ICU appears to control all of the hardware primitives in the second cluster. Physically, the inter-cluster ICU must first intercept messages and data from one cluster, and then forward the data along the circus in the second cluster. Thus, inter-cluster data transfers occur in two phases, and there is a factor of two difference in overhead between intra-cluster and inter-cluster transfers. There is also the problem of synchronizing the transmission of multiple data sets between clusters. There are many complex synchronization, correctness, and performance issues that need to be resolved in specifying the correct operation in an inter-cluster ICU. Many of these issues have yet to be addressed. However, for the purposes of this study, it is not necessary to know the exact behavior of inter-cluster ICUs, but is sufficient to model their performance impact.
3 The Performance Model

We are concerned with modeling and simulation of architectures embodying the ALPS architecture. Accurate performance modeling and simulation must include algorithms for specifying how partitions of the signal flow graph are to be assigned to the individual clusters. In general, such graph partitioning problems are non-trivial, and many well known partitioning problems belong to the class of NP-Complete problems [11]. However, an underlying assumption of our approach is that in an interactive environment, optimal solutions are often not necessary. Solutions which meet performance specifications are sufficient. The overall context of the work described in this report is illustrated in figure 3. Candidate architectures and algorithms are first modeled and evaluated via simulation. Then the algorithms are implemented on a test bed architecture that emulates various ALPS architectures. If necessary, simulation and emulation are repeated until closure on an acceptable design is reached. The target ALPS system is now ready for implementation.

The work reported here fits in the modeling and simulation stage shown in figure 3. In particular, we are pursuing the integration of these algorithms into the Ptolemy [16] environment developed at the University of California, Berkeley. The remainder of this section describes the algorithm and architecture models upon which this performance evaluation is based.

3.1 Architecture Model

This heterogeneous parallel architecture is represented by a undirected graph \( G_c = (V_c, E_c) \). The nodes \( V_c \) correspond to clusters in the system and the edges \( E_c \) correspond to inter-cluster ICUs. It is assumed that the inter-cluster communication is symmetric and balanced. Our representation further includes a collection of sets \( R_i, 1 \leq i \leq C \), each of which specifies the primitives within cluster \( i \) and where \( C \) is the number of clusters in the system. Since many primitives may perform a given signal processing function, a matrix \([p_{ij}]\) records the execution cost for flow graph node \( j \) on hardware primitive \( i \). A matrix \([w_{ij}]\) contains the weight of edges in the signal flow graph which represents data communication from node \( i \) to node \( j \). Finally, the cluster adjacency matrix, \([s_{ij}]\), records the distance, i.e., the number of inter-cluster ICUs traversed, between primitives in clusters \( i \) and \( j \). An example target architecture model is depicted in the lower graph of figure 4.
Figure 3: Development Environment for ALPS Architectures
3.2 Algorithm Model

A signal processing flow graph is a weighted directed acyclic graph, $G_p = (V_p, E_p)$, where the nodes correspond to functions to be executed, e.g., FFT, filtering, etc., and the edges correspond to communication between tasks. The weights associated with the nodes represent the computational overhead of tasks expressed in arithmetic/logical operations. Similarly, weights on edges correspond to communication overhead between functions. Flow graph nodes are represented by a node number and function type. The former is simply a unique node address. The latter represents the function being computed by that node. The upper graph in figure 4 illustrates an example signal flow graph used in our experiments. In general, these graphs are expressed and constructed in the Signal Processing Graph Notation (SPGN) developed at the Naval Research Laboratory[2].

3.3 Objective Functions

A partition of a signal flow graph is implicitly associated with a cluster in the architecture. This partitioning is represented by an ordered list, $\vec{v}$, whose $i^{th}$ entry indicates the cluster number that flow graph node $i$ has been placed in. An objective function, consisting of two components, is used to quantify the quality of the generated partition. The first component measures the degree of load balancing achieved in the present partition, and the second measures the communication overhead of the partition. The sum of these two metrics is the desired objective function.

The following is the expression used to compute the load balancing component.

$$E_{comp} = \max_{c \in V_c} \left\{ \sum_{i \in c} \left( \max_{j \in R_c} p_{ji} \right) \right\}$$

When the load is balanced, $E_{comp}$ is minimized. Also note that $E_{comp}$ is computed using the worst case execution time for functions within a cluster, i.e., each of the functions mapped to a given cluster are assumed to execute serially on the slowest primitive able to execute the function.

Communication overhead is measured by the following expression.

$$E_{comm} = \sum_{i \in V_p} \sum_{j \in V_p} w_{ij} * D_{\vec{u}_i, \vec{u}_j}$$

where

$$D_{\vec{u}_i, \vec{u}_j} = \begin{cases} 1, & \vec{u}_i = \vec{u}_j \\ 2 * s_{\vec{u}_i, \vec{u}_j}, & \text{otherwise} \end{cases}$$

Note that communication can be minimized simply by placing all of the nodes in the flow graph in the same cluster if each cluster contains at least one PE that can compute each function. However, this would increase the load balancing component. Alternatively, a uniformly balanced load can result in high communication overhead since the structure of inter-task communication is neglected. Thus, minimizing the sum represents finding a reasonable compromise between minimizing either component individually.

The quality of a partition is evaluated as
Figure 4: Example Signal Flow Graph and Target Architecture
\[ E = E_{\text{comp}} + E_{\text{comm}}. \]

This expression represents a pessimistic view in that it is based on the worst possible schedule for the execution of functions within a cluster, and for inter-cluster communication. However, this is partially offset by erring on the side of optimism in computing the load balancing component by assuming that all partitions are executing simultaneously.

### 3.4 Effect of Partitioning Algorithms on Performance

Intuitively the assignment of flow graph nodes to clusters affects the execution time of the flow graph. In general, this can be significant. In designing signal processing architectures for computationally demanding applications, it is useful to experiment with partitioning algorithms during the design to help fine tune the architecture and maximize the performance. The problem is too complex to perform adequately manually. In conjunction with simulation tools, iterative application of partitioning and simulation can be used to fine tune a design and minimize execution time. This research focuses on the partitioning algorithms that would be part of such a process. The fact that partitioning algorithms can indeed affect execution time is evident from the following example.

Consider the example shown in figure 4. This figure shows a heterogeneous partitioning problem. The graph shown is to be partitioned across the four cluster architecture shown. There are clearly many alternatives in assigning functions to clusters. Consider the assignment in which nodes 0, 1, 2, and 6 are in cluster 0, node 3 is in cluster 1, node 4 is in cluster 2, and node 5 is in cluster 3. This assignment has an objective function value of 98. Now consider the assignment in which nodes 0 and 2 are in cluster 0, nodes 1 and 4 are in cluster 2, node 6 is in cluster 2, and nodes 3 and 5 in cluster 3. This assignment has an objective function value of 53. The execution times of each of these partitions is 55 time units, and 44 time units respectively. This represents a 25% decrease in simulated execution time due to a few seconds of partitioning prior to simulation. These execution times were derived using a simplified manual simulation. From these simulations, it is evident that partitioning algorithms can benefit design environments for systems implementing the ALPS execution model.

Partitioning signal flow graphs on heterogeneous systems can produce performance problems not found on homogeneous systems. In the latter case, partitioning is simplified since measures of the execution time of a function, and communication overhead can often be computed or approximated statically. In heterogeneous architectures based on the ALPS paradigm, the execution time of a function is dependent upon the PE which computes the function. The specific PE upon which a task is scheduled depends upon the cluster in which the function is scheduled to execute, the number of PEs that can compute that function, and relative timing of all other functions to be computed in that cluster. These choices are not made until run-time in the ALPS model. Similarly, the data dependent delay of communication between two flow graph nodes is also determined at run-time. This non-determinism precludes determining computation and communication overhead a-priori. Without partitioning tools, it is extremely difficult for the user to know how to best implement an algorithm on such heterogeneous architectures, or the performance implications of the choices that are made.
4 Partitioning Algorithms

Throughout this report we will use the phrase "partitioning of the flow graph" to be synonymous with "assignment of a node to a cluster". The partitioning algorithms assume that the goal is to partition the graph into a number of partitions equal to the number of clusters in the target architecture. Each partition is implicitly associated with a specific cluster. Placing node in a partition is therefore synonymous with assigning a node to the associated cluster. Figure 4 illustrates an example flow graph and associated four cluster ALPS architecture. Flow graph nodes are labeled with the function they are to compute (e.g., T4). This is equivalent to a flow graph node type. Clusters are similarly labeled indicating which functions may be computed within the cluster, e.g., the list (T1, T2, T3, T4) associated with cluster 0. The following discussion will frequently refer to this figure for illustrative examples of ideas and definitions. We propose three algorithms for partitioning flow graphs for ALPS type architectures - simulated annealing, edge elimination, and branch and bound. These are briefly described in the following.

4.1 Simulated Annealing (SA)

Simulated annealing (SA) is probabilistic, hill-climbing, Monte Carlo optimization technique based on Metropolis' algorithm[4, 5]. Unlike greedy algorithms and other simple iterative improvement techniques, SA provides a mechanism for the search procedure to get out of local minima. The algorithm is described in figure 5. In the application to the partitioning problem, the state is represented as the assignment of each signal flow graph node to a partition, which in turn is assigned to an ALPS cluster. Starting from an initial partition, a move generates another partition. An objective function provides a quantitative estimate of the quality of this partition. This new partition is accepted if it is better, or probabilistically accepted if it is worse. When an equilibrium state has been reached, the temperature is reduced, and the process is repeated. The initial state at this new temperature is the best state found at the previous temperature, and not the last state encountered. Temperature serves as a control parameter. This iteration is continued until the algorithm meets a termination condition. The decisions concerning the form of the equilibrium condition, termination condition, state change, and acceptance function collectively constitute the annealing schedule.

It has been recognized that adaptive schedules can provide much better performance for specific problems at a fraction of the computational cost[5]. The motivation to derive an adaptive schedule is based on the observation that the behavior of SA is very different at high and low temperatures. At high temperatures, it is the number of acceptances that dictate equilibrium, while at low temperatures it is the number of rejected states that dictates equilibrium. Another important factor is the number of iterations required to reach the final state. If a new state is created by changing the assignment of a single flow graph node, the number of iterations can be very large. However, if multiple assignments can be changed at each iteration[6], the total number of iterations can be decreased, resulting in a faster convergence. Such changes of state correspond to large jumps in the state space. However, it is not desirable to make such large jumps in the vicinity of global optima, i.e., at low temperatures. We propose to reconcile such conflicting demands by adopting an adaptive schedule.

The schedule we propose here was derived after experimentation with several partitioning and mapping problems[8]. The following describes the various components of the proposed annealing
1. Define an objective function $E(x) = E(x_1, x_2, ..., x_n)$ where $x_1, x_2, ..., x_n$ represent the partitions that nodes 1, 2, ..., $n$ respectively are assigned.

2. Start with a random state and high temperature $T_H$.

3. Reduce the previous temperature by a small factor, i.e., $T_i = \alpha T_{i-1}$. Set the initial state to the best state found at $T_{i-1}$.

4. At each temperature $T_i$, repeat the following moves until a set of equilibrium conditions are met:
   - Select a change of state, $x'$, from the current state $x$.
   - Compute $\Delta E = E(x') - E(x)$.
   - If $\Delta E \leq 0$, accept the new state $x'$.
   - If $\Delta E > 0$, accept the new state with probability $e^{-\Delta E / T}$.
   - Update the best partition observed at this temperature.

5. Repeat steps 3 and 4 until a set of terminating conditions are met.

Figure 5: Partitioning with the Simulated Annealing Paradigm

Generating the Initial Partition: The initial partition is generated randomly, subject to execution constraints, i.e., the cluster must contain a PE that can execute the tasks corresponding function.

State Changes: The following five move strategies were considered in this study. The moves made by the various move strategies are always made with consideration of execution restraints.

1. Single random inter-cluster move - a single random node is selected and moved to a random cluster (possibly the cluster it presently resides in).
2. Multiple random inter-cluster moves - a random number of nodes are selected to move. Each node is moved to a random cluster.
3. Single node to cluster containing a neighboring node - a single random node is selected and moved to a cluster which contains a neighboring (with respect to the flow graph) node.
4. Single node to neighboring cluster - a single random node is moved from its present cluster to a neighboring cluster.
5. Type 3 or 4 move - Two possible moves are considered. Two nodes are randomly chosen. Using the first node chosen, a type 3 move is attempted placing the node in a cluster containing one of its neighbors in the flowgraph. Then, using the second node chosen, a move of type 4 is attempted placing that node in a cluster which neighbors the cluster it presently resides in. The move yielding the lowest objective function value is selected as the move to be implemented.
Strategies 1 and 2 enable the algorithm to randomly walk through the solution space using either small steps, as in strategy 1, or large steps, as in strategy 2. The drawback to these strategies is the lack of correlation to the structure of the signal flow graph. There is no reason to believe that one move is better than another, or that progress is being made towards a globally (or even locally) optimal solution. Strategies 3, 4, and 5, however, take advantage of the structure of the signal flow graph and target ALPS configuration by operating on the intuitively appealing assumption that moving nodes closer to their neighbors in either sense described above may decrease communication time or improve load balancing. Strategies 3 and 4 tend to be relatively more sensitive to the initial partitioning. When used individually, both strategies cause all nodes to accumulate in a single cluster making load balancing impossible. Strategy 5 has the advantages of strategies 3 and 4, and appears to provide the best compromise.

- **Equilibrium/Termination Conditions:** At a given temperature $T$, the algorithm is said to be at equilibrium if the number of acceptances exceeds $10 \times N$, where $N$ is the number of nodes in the flow graph, or if the number of rejections at temperature $T$ exceeds $100 \times A$, where $A$ is the number of acceptances. Further, the minimum number of acceptances required for equilibrium is decayed at the same rate as the temperature making convergence at low temperatures comparable with convergence at high temperatures without any appreciable compromise in the quality if the results. If the partitions computed cannot be improved over five consecutive temperatures, the algorithm terminates.

- **Initial Temperature:** The initial temperature is set at twice the value of the objective function computed for the initial state. The temperature decays geometrically by a factor of 0.9 at each successive temperature.

### 4.2 Edge Elimination Heuristic (EEH)

In this section we describe an application of a greedy, critical path approach to realize an Edge Elimination Heuristic (EEH). In general, critical path heuristics are based on the observation that there are sets of nodes in flow graphs that must be executed sequentially, i.e., there is no parallelism to exploit within these sets and little is to be gained by placing them in separate clusters. Determination of these sets is useful because they also indicate which tasks may be performed in parallel, subject to flow graph precedence constraints. Further, there is some degree of flexibility in scheduling nodes not contained in distinct sets. A high level description of EEH is shown in figure 6. Generally, EEH attempts to ensure communication corresponding to the heaviest edges occur within a cluster. It is in this sense that it is "critical path like." Flow graph nodes not on this path are spread among neighboring clusters to promote parallelism. Thus, EEH is expected to perform best on flow graphs representing DSP algorithms operating on large data sets. A detailed description of EEH requires the following definitions.

We have the following definitions relative to signal flow graph node $n_i$.

- **Level($n_i$)** is the level of node $n_i$, i.e., the length of the longest path from an input node.
- **Perform($n_i$)** is a list of all clusters in which node $n_i$ may be executed, i.e., there is at least one PE type that can compute the function corresponding to node $n_i$.
- **Valid($n_i$)** represents the lowest numbered cluster in which node $n_i$ may be executed.
1. Level the flow graph.
2. \( \forall n_i \in V \text{ s.t. } \text{Level}(n_i) = 0 \) do \( v_i = \text{Valid}(n_i) \).
3. for \( i = 0 \) to \( \text{levels-1} \) do \\
   for \( i = 0 \) to \( \text{nodes-1} \) do \\
     if \( \text{Level}(n_i) = i \) then \\
       \( j = \text{Heavy}(n_i) \) \\
       if \( j \neq 0 \) then \\
         if \( v_i \in \text{Perform}(n_j) \) then \\
           \( v_j = v_i \) \\
         else \\
           \( v_j = \text{ClosestValid}(n_j, v_i) \) \\
     \( N = \text{Neighbor}(v_i) \) \\
     \( l = \text{Succ}(n_i) \) \\
     Distribute nodes of \( l - n_j \) over \( N \).

Figure 6: Outline of the Edge Elimination Heuristic

- \( \text{Succ}(n_i) \) is an ordered list of nodes, in ascending order of node number, which are successors of node \( n_i \) in level \( \text{Level}(n_i) + 1 \).
- \( \text{Heavy}(n_i) \) is the heaviest edge from node \( n_i \) to a node in level \( \text{Level}(n_i) + 1 \). The function returns the terminal node number.
- \( \text{ClosestValid}(n_i, c) \) is the closest cluster to cluster \( c \) which may perform node \( n_i \)'s task.

We have the following definition relative to cluster \( c_i \).

- \( \text{Neighbor}(c_i) \) is an ordered list of clusters which are adjacent to cluster \( c_i \). The list is ordered in increasing cluster number. \( c_i \) is the last element of this list.

EEH begins by leveling the signal flow graph. This leveling provides a coarse ordering on flow graph nodes that is used to place them in a partition. Each flow graph node, \( n_i \), at level zero will be assigned to the lowest numbered cluster in \( \text{Perform}(n_i) \). When this operation is completed, level zero is said to have been placed. Nodes at successive levels are now placed. Consider the situation where all nodes at level \( i \) have been placed. For each node \( n_k \) at level \( i \), the successor node at the level \( i + 1 \) connected via the heaviest edge is selected for placement placed in the same cluster. If this placement is not possible due to the unavailability of a PE type that can compute that function, the node is placed in the nearest cluster that can compute that function. The remaining successor nodes in \( \text{Succ}(n_k) \) at level \( i + 1 \) are distributed across the available clusters so as to promote parallel execution. This process in continued over succeeding levels. This heuristic makes no use of computation overhead nor does it consider edges which connect nodes between level \( i \) and level \( i + j \) where \( j \geq 2 \). This will result in a poor partitioning when such edges are
very heavy relative to the remaining edges. We point out that the decisions made at every step is complicated by the need to consider compatibility between function types and PE types available within a cluster. Many of these issues are best illustrated with an example.

**Example:** Consider the signal flow graph and target architecture graph depicted in figure 4. The flow graph is leveled resulting in nodes at 4 levels. Let us refer to the placement of nodes at each level as a *phase*. The will be 4 phases in the partitioning algorithm. During phase 0, nodes at level 0 will be placed. In phase 0, node 0 is placed in the lowest numbered cluster able to perform its function - in this case it is cluster 0. Now phase 1 begins by considering each node at level 0, and the nodes that succeed them at level 1. An ordered list, $l$, is made of clusters which neighbor the cluster containing the node at level 0, i.e., cluster 0. We have $l = <1, 3, 0>$. Note that the neighbor relation is reflexive, i.e. clusters are neighbors of themselves, but are placed last in the list $l$. The heaviest edge from node 0 to a node at level 1 is $e_{0,1}$. The destination node, Node N1, is placed in the same cluster as node 0 - cluster 0. Using the list $l$, the remaining nodes are placed in clusters which neighbor cluster 0. Specifically, node N2 is placed in cluster $l_0 = 1$, and N3 is placed in cluster $l_1 = 3$. EEH allows parallelism within flow graph level 1 to be exploited, i.e., between nodes N1, N2, and N3. The process is continued for levels 2 (nodes N4 and N5) and 3 (node N6).

We can make a few observations about the general behavior of EEH. Note that the edge $e_{0,4}$ with weight 5 was never considered. This is a consequence of the fact that only edges traversing a single level are considered for placement. The rationale behind doing so is that nodes should be placed as late as possible in the algorithm. Thus, more information concerning the location of neighboring nodes is potentially available to aid in efficiently placing a node. Further, no use of execution time information was explicitly considered. This is partly due to the inability to determine the execution time without placement information. Finally, nodes that have multiple inputs are placed multiple times, once for each predecessor, with the final result being the placement produced by the predecessor considered last. It is relatively straightforward to force a choice among these alternatives. However, there does not seem to be any advantage in doing so.

### 4.3 Branch and Bound Heuristic (BBH)

This section describes an algorithm based on a well known state space search technique - branch and bound state space search[12]. Our implementation differs from previous approaches in two principal ways[13, 14, 15]. First, the problem we are faced with here differs from previous studies - partitioning onto a heterogeneous, hierarchically structured, self scheduling machine. The constraints are different and are reflected in the cost function, as well as the formulation. Second, unlike traditional implementations of branch and bound paradigms, we adopt a simple technique for further reducing the size of the search tree. The degree of reduction can be adjusted to speedup the search, while potentially sacrificing the guarantee of finding optimal partitions.

The structure of the algorithm is shown in figure 7. This approach distinguishes between partially constructed partitions and complete partitions. The former is one where only a subset of the flow graph nodes have been assigned to a cluster. The algorithm starts with the empty partition on the list OPEN. The initial partition is then expanded in steps until a complete partition has been produced. An expansion of a partially constructed partition assigns one of the unassigned flow graph nodes to a cluster to produce a new partial partition. The algorithm iteratively selects a partial partition with the lowest cost, selects an unassigned node, and expands it to all possible partial partitions using this node. The cost assigned to each of these new partitions is a lower bound
1. Put the initial empty partition with a cost of 0 in a list called OPEN.

2. Remove from OPEN the partial partition with the lowest cost.

3. If this is a complete partitioning of the graph, remove and stop.

4. (a) Select an unassigned flow graph node \( n_i \). Expand this partial partition, creating all possible new partial partitions. A new partial partition is created by assigning this node to a cluster in \( \text{Perform}(n_i) \).

   (b) Compute a lower bound on the cost of the final partition that can ultimately be created from each new partial partition.

   (c) Place all of these new partial partitions on OPEN.

5. Go to step 2.

Figure 7: Partitioning with the Branch and Bound Paradigm

on the final partition that can be realized from this partial partition. The cost for an expanded partition is computed as specified in section 3.3. Only flow graph nodes that have been assigned to a cluster can contribute to the load balancing component. Only edges between such assigned nodes contribute to the communication component. This cost is a trivial lower bound on the cost of any partition generated from this partial partition. At this time we do not utilize any other additional lower bound computation. The process continues until a minimum cost complete partition of the graph is obtained. This formulation is guaranteed to find the minimum cost partition. Though the worst case behavior is exponential, in practice branch and bound implementations have found good average case behavior in many applications. We refine the basic paradigm in the following manner. It is these refinements and the cost function which distinguishes this implementation from traditional implementations.

Node Selection: When a partial partition is to be expanded, a choice must be made in selecting the flow graph node to expand this partition. Nodes are selected in the order defined by their level in the graph. Nodes at level 0 are selected first, followed by nodes at level 1, etc. Nodes within a level are selected in an arbitrary order. An experimental evaluation of three selection functions clearly indicated that selecting nodes by level produced the best results. In all of our test cases optimal partitions were found with a minimum number of generated states using this selection function. Therefore, our implementation and analysis expands partial partitions in this order.

Size of OPEN: We use a simple greedy approach to reducing the number of intermediate partial partitions. The size of OPEN is fixed at some maximum - MAXLENGTH. OPEN is also maintained as sorted list with entries ordered in increasing cost. This has the effect of taking the state space search tree and pruning it to include only those MAXLENGTH number of leaf nodes with the least cost. When MAXLENGTH has no bound, the search tree is equivalent to that searched by traditional branch and bound implementations. When MAXLENGTH is very small, the search approaches a purely greedy search. With the test cases we have used we have found a MAXLENGTH on the order of 1000 partitions finds optimal partitions. In general, we would like
MAXLENGTH to be large enough to ensure that at least one partial partition that will expand to an optimal partition will always be within the MAXLENGTH number of minimum cost states. It appears that expanding nodes according to their level encourages this to be the case with relatively a small values of MAXLENGTH - on the order of several thousand partitions.

Example: A better understanding can be gained with an example. In this example a state is represented by an array where ARRAY[i] is the cluster in which node i executes. A value of -1 indicates that i has not yet been assigned a cluster. Consider the example graph shown in figure 4. The set of states that are generated when partitioning this graph for execution on the architecture shown in figure 4 is shown in figure 8. The initial partition is the empty partition denoted by the state with all elements equal to -1. The flow graph is leveled. Node NO is at level 0, nodes N1, N2, and N3 are at level 1, nodes N4 and N5 are at level 2, and node N6 is at level 3. The empty partition is removed from OPEN and node NO is selected to expand the empty partition. From the types associated with each cluster in figure 4, we see that node NO can only be executed in cluster 0. The new partial partition < 0, -1, -1, -1, -1, -1, -1 > is added to OPEN. Assuming unit execution costs, the partial cost associated with this partition is 1. This partition is removed from OPEN, and from the leveling of the graph, any one of the nodes at level 1 is selected for expanding this partition. Let us assume it is node N1. This node can execute in clusters 0, 1, or 3. The corresponding partial partitions are generated as shown in figure 8 and the associated costs computed. From the figure, it is evident that the next state that will be selected from OPEN will be < 0, 0, -1, -1, -1, -1, -1 > which has the minimum cost of 5. This process is repeated until a complete partition is removed from the head of the list. The effect of MAXLENGTH is to prevent all of the leaf nodes of the tree shown in figure 8 to be added into OPEN. An interesting variant of this algorithm is to have MAXLENGTH vary dynamically. For example initially MAXLENGTH could relatively small and increased as the search movers to lower levels in the tree. A better understanding of the structure of the space of partitions is necessary to have a successful approach for dynamically varying MAXLENGTH.
5 Experimental Results

One of principal obstacles to the evaluation of partitioning and mapping algorithms is that for arbitrarily structured graphs, the optimal solutions are not known. It therefore becomes difficult to evaluate the quality of the solutions produced by any proposed heuristic. We have addressed this problem by using test cases for which the optimal partitions are known to help us in developing and understanding the algorithms. For example, consider the graphs in figures 9 - 11. These graphs were constructed such that the optimal partitions could be derived.

The following experiments were performed on a Sun Sparcstation. 100 trials were conducted for each graph for each partitioning algorithm. Flow graph nodes were randomly relabeled prior to each trial to ensure that partitions generated were not dependent upon node labels. Table 1 indicates the objective function values obtained for each of the algorithms. Additional information includes the number of iterations taken by SA, the number of states examined in BBH, and the average execution times of the partitioning algorithms. The Random column represents the average objective function value observed in 100 random partitions. Max indicates the largest objective function value that was found, and Opt indicates the objective function value of the optimal partitions when they are known. The table values are drawn from the 100 trials for each algorithm. For each application of SA, move strategy 5 was used.

All node and edge weights in the flow graph shown in figure 9 are equal to 1, and all nodes are of type 0. This graph is partitioned for execution on a two cluster architecture where each cluster supports types T0 and T1. This represents a homogeneous case. The optimal partitioning corresponds to the two natural partitions apparent from the figure. Both SA and BBH found the optimal partition in comparable times. SA computed the optimal partition in 90% of the trials. EEH found reasonable partitions but did not fare as well as SA and BBH. A heterogeneous two cluster variant of this graph has one node in each partition that is constrained to be mapped onto one cluster. Figure 10 shows the same graph augmented with one node, N20, whose weight is equal to the sum of the weights of all other nodes. The optimal partition places this node in one cluster and all the other nodes in the other cluster. Finally, figure 11 shows a flow graph with four natural clusters representing the optimal partition. This graph is partitioned onto a four cluster architecture that can support the execution of the optimal partition, but also includes other constraints on node/cluster combinations. The target architecture for this graph is shown in figure 4. In all three graphs G1-G3, SA determined partitions which were consistently within 20% of the optimal partition.

BBH consistently computed optimal partitions for all of these graphs using a MAXLENGTH of 1000. The total number of partial partitions generated was relatively small averaging in the low 1000s. The number of partial partitions generated even without the MAXLENGTH heuristic was not significantly larger. EEH did not fare quite as well, though on average the objective function values of the computed partitions were not much greater than that of the optimal partition.

Finally, figures 12 and 13 illustrate two synthetically generated problem graphs with varying degrees of parallelism - small, average spread 1.6, and relatively high, average spread 4.3, respectively. The optimal partitions for these two graphs are not known apriori. The best partitions found by SA are depicted in the respective figures. Partitions that are easy to visualize are highlighted and those which are not are specified below the respective figure. BBH requires significantly longer run times for the latter (and larger) graph, while the quality of the solution is not quite as good as that generated by SA. We found early implementations of BBH to incur excessively long run times.
This was the primary motivation for further reducing the size of the search tree as described in the previous section. EEH produces the relatively poorest partition, but runs an order to several orders of magnitude faster. This could be desirable for very large graphs if the solutions generated meet performance specifications, even though they are not necessarily optimal.

We make the following observations from our experiments. For graphs where there are a small number of functions that require a disproportionate amount of compute power (e.g., G2) BBH works well. This node places a strong constraint on the cost of partial partitions and forces faster convergence to a solution. However, BBH does not perform as well on homogeneous cases since it is not possible to discard many of the partial partitions as most of them have comparable costs. SA consistently provides the best performance across a range of graphs. EEH provides the fastest convergence to a partition but produces correspondingly poorer partitions. However, for very large applications where it is sufficient that solutions are “good enough”, this difference in execution speed may be desirable. All of the algorithms consistently compute partitions that are better than the randomly created partitions. These relationships between partitioning algorithms and flow graph structure underscores the need to have them available to the system designer for iterative, interactive, experimentation.

These experiments establish that significantly better partitions can be obtained with well structured partitioning algorithms. Therefore, in order to accurately and reliably model run-time execution performance these partitioning algorithms must be available during modeling and simulation as part of the designers repertoire of tools.
Figure 11: Test Case 3 - Partitioning onto Four Clusters, Graph G3

Cluster 0 = (N0, N4, N5, N9)

Cluster 1 = (N0, N4, N5, N9)

Figure 12: Test Case 3 - Partitioning onto Four Clusters, Graph G4
6 Concluding Remarks and Future Work

This research is concerned with an environment for the design of a class of application-specific, heterogeneous DSP architectures. The goal in simulating DSP architectures is to model all of the hardware, software, and I/O phenomena that affect performance so that design decisions can be made to maximize performance. One area that has received comparatively little attention in the context of coarse grain parallel architectures for signal processing is that of scheduling parallel computations. Though this phenomena is critical to performance, they are not included in most modeling and simulation environments. This is the focus of this research.

We foresee an environment where a designer will iteratively simulate architectural variations of the target system before proceeding to an implementation. We argue that partitioning algorithms for coarse grain signal flow graphs are instrumental to maximizing the performance of such architectures. Poor schedules for the execution of signal processing functions will result in poor performance regardless of the speed of individual components. Partitioning algorithms are an integral part of the system and therefore should be also a part of the specification, design, modeling, and simulation environment.

We have presented three candidate partitioning algorithms for a specific class of heterogeneous architectures, and an evaluation of their performance. Our experimental results are encouraging, indicating that the use of such partitioning algorithms can significantly improve the performance of ALPS type architectures. Plans for the immediate future include integrating these algorithms with an existing environment specifically tailored to the design and simulation of heterogeneous DSP architectures[16]. We are particularly interested in modifying the simulator to reflect the self-scheduling, self-synchronizing mechanism in ALPS type architectures. Integration of the parti-
tioning algorithms will follow. We will then have an environment that reflects accurate, and reliable simulation of application-specific DSP architectures using the ALPS approach.

References


A Software provided

The following section briefly describes the software which comprises the partitioning algorithms. The section begins with a description of the component source files and concludes with a description of operational principles.

A.1 Simulated Annealing and EEH

Source File
sal.c Contains the user interface. This is the main program module.
random.c Contains some random number generation routines which may have to be changed for use on different platforms.
matrix.c Contains some matrix manipulation routines used for dealing with adjacency matrices.
graph.c Contains some graph theoretic routines.
io.c Contains the routines which perform I/O. That is, these routines read the various configuration files (.fg, .cl, and .ex).
metric.c Contains routines to compute computation and communication costs for given partitions.
anneal.c Contains the annealing algorithm, i.e., mode strategies, etc.
map1.c Contains heuristic partitioning algorithm, EEH.

To construct the main executable program, merely compile sal.c.
Run the program from the Unix prompt. The software is menu driven and is self-explanatory. Before partitioning, the three files which specify the algorithm, architecture, and the execution-time statistics must be read-in using the Read flow graph option and the Modify System architecture options.

A.2 Branch and Bound Partitioning Software

The following source files are used in constructing the executable module.

Source File
bb.c Contains the user interface. This is the main module.
bbfunc.c Contains the branch and bound algorithm internals and other functions.
graph.c Contains some graph theoretic routines.
io.c Contains the routines which perform I/O. That is, these routines read the various configuration files (.fg, .cl, and .ex).
anneal.c Contains a routine to test the validity of a state change.
To construct the main executable program, merely compile bb.c.

Run the software from the Unix prompt. The software will ask for the name of the files containing the algorithm, architecture, and execution-time information, i.e., the .fg, .cl, and .ex files, respectively. It will also ask for the maximum size of OPEN.

B File Formats

The following is a brief description of files used by the partitioning software. There are three files required to uniquely specify a partitioning problem, i.e., the algorithm flow graph and the architecture onto which the algorithm is to be partitioned. The files will have extensions .fg, .cl, and .ex.

Files having an extension .fg specify the algorithm signal flow graph. Essentially, the flow graphs adjacency matrix is stored in this file. For a graph consisting of \( n \) nodes, the file will contain \( n + 1 \) rows with the first line in the file considered to be row 1. The first row will contain an integer indicating the number of nodes in the flow graph, i.e., \( n \). Nodes are numbered from 0 to \( n - 1 \). The remaining \( n \) rows will consist of \( n + 1 \) integers separated by spaces. The first integer in each row indicates the function number the corresponding node is to perform. The following \( n \) integers indicate the edge weights in the signal flow graph for all nodes. Specifically, row \( i, j > 1 \) indicates the edge weight from node \( i - 2 \) to node \( j - 2 \).

The next file describes the target architecture. A target architecture will consist of clusters (perhaps only 1) interconnected in some configuration. Further, each cluster will contain certain processing elements. The .cl file indicates both the interconnection strategy and the composition of the individual clusters. For an \( n \) cluster system there will be \( 2 \times n + 1 \) lines in the file (beginning with line 1). Line 1 will contain an integer indicating the number of clusters in the target system. The next \( n \) lines will indicate the composition of the clusters. Specifically, line \( i \), for \( 1 < i < n + 2 \), will indicate the device numbers which are present in cluster \( i - 2 \). The first integer in each line \( i \), which we call \( d_1 \), indicates the number of devices which are present in cluster \( i - 2 \). The next \( d_1 \) integers, namely \( d_2 \ldots d_{d_1} \), indicate the device numbers. The last \( n \) lines (rows) in the file indicate the connectivity of the clusters, i.e., the adjacency matrix for the clusters. Line \( n + 2 \) in the .cl file is the first row of the adjacency matrix and consists of \( n \) integers separated by spaces. The remaining \( n - 1 \) rows are defined accordingly.

The last file indicates the execution time of the \( f \) functions on each of the \( d \) device types. The file will consist of \( d + 2 \) lines beginning with line number 1. The first line will contain an integer indicating the number of device types present. The next line indicates, with an integer, the number of functions available. Finally, the remaining \( d \) lines will consist of \( f \) integers indicating the execution time of the various functions on each of the device types. A time of zero indicates that the device may not perform the function. Specifically, line \( i \) for \( 2 < i < d + 3 \), column \( j \), for \( 1 \leq j \leq f \) indicates the execution time for task \( j \) on device type \( i - 2 \).