RASSP Data Flow Graph Design Appnote

Abstract

Data Flow graphs combined with autocoding techniques are an advanced graphical method for programming networks of processors for signal processor applications. Currently, complete development of a signal processor system beginning with system engineering through the actual software can be accomplished using methodologies and simulation and software design tools developed under the Rapid Prototyping of Application-Specific Signal Processors (RASSP) program. In this application note the key issues and concerns involved in developing such an application are explored so the software developed by the autocoding tools will yield a cost effective signal processor (minimum number of processors and memory usage). The design techniques presented in this paper are based upon Data Flow techniques and implementation. These will be illustrated by two applications the Synthetic Aperture Radar (SAR) signal processor and the Sonar Echo Tracker and Classifier (ETC) processor.

Purpose

The presentation of procedure and design methodology that can take a general signal processing problem and convert it to a data flow design is the purpose of this application note. The Data Flow graphs are implemented by a Hardware/Software codesign process where portions of the Data Flow graph will be directly mapped to software running on DSP processors and portions such as input processing directed to hardware. This application note will also discuss a tool from Lockheed Martin ATL which provides the functional simulation as well as the autocoding to target DSP processors called GEDAE™. Using the Data Flow graph technology and the associated tool, the complete software development cycle including documentation and coding is shown to be significantly reduced. The resultant software product is also more maintainable than a conventional software approach.

Roadmap

RASSP Data Flow Graph Design Application Note

1.0 Executive Summary

Data flow representation of an algorithm is an effective technique for code development so that the algorithm may be executed in parallel on multiple signal processors. Usually, in signal processing applications, parallelization is required to meet a latency constraint. The issues involved in usage of the parallel constructs such as: Split and Merge, Mux and Demux, and Separate and Concatenate are all discussed in reference to the final autocoded result. The impact of additional issues associated with Data Flow graph primitive design and the impact upon the resultant code are then developed. The issues are: State, Memory management, construction and usage of the queues and static run schedules. These are the practical issues that affect the ability of the autocoding system to efficiently convert the graph to usable code. By properly designing the Data Flow graph data structure, the amount of processing and the size of the buffers (memory) required can be controlled. This paper presents guidelines for the usage of these concepts in the construction of the Data Flow graph. All the concepts will be illustrated by examples taken from the SAR and the ETC4ALFS on COTS Processors benchmark program applications. Case Studies on these two benchmarks programs are available by following the links provided.

Next Up Previous Contents
Next: 2 Introduction to Graphical Data Flow Description **Up:** Appnotes Index **Previous:**Appnote Data Flow Graph Index

 Dennis Basara

RASSP Data Flow Graph Design Application Note

2.0 Introduction to Graphical Data Flow Description

Graphical representations of computations have been developed to provide an insight into sequence of computations. A complete process has been developed that begins with the translation of the requirements for a signal processing problem description into a Data Flow (DF) graph. This process allows the designer to rapidly progress from the signal processing algorithm to executable code that is automatically generated to run on the target COTS DSP processors. A supporting set of tools have also been developed to simulate and verify the design and to provide timing and memory sizing of the final executable code. These tools automatically autocode the DF graph and generate executable code that loads and runs on the target DSPs. While the tools automate most of the mechanical procedures involved, there are important design considerations in the design of the DF graph that affect the final executable code. This paper presents these considerations and discusses the design options in the DF graph design:

- parallelization of the signal processing algorithm
- memory management
- processor allocation.

These are design options that must be explored as part of the design process in order to be able to generate efficient and compact code for the target DSPs. While it may be desirable to be able to perform these processes automatically, the tools that automate these tasks have not yet been developed to a mature state so they can be used routinely used. The mature tools, like GEDAE™, that are availabel now assist the intelligent designer to be more productive in considering more design options. The basic graphical tool used to develop the Data Flow design is the Data Flow graph. In order to understand how this graphical concept captures the complete functionality of the signal processing algorithm, we must first re-examine the basic method of expressing a mathematical description of a signal processing algorithm by a graphical description. The resultant graphical description provides the insight into the structure of the equations and aids the parallelization of the algorithm. This section provides a description of a computational graph and then shows what is required to convert this to a Data Flow Graph.

2.1 Computation Graphs

The graphical notation that is used to express a set of computations follows a functional description of these computations. We use c = func(a, b) to represent an arithmetic operation performed upon the elements a and b to obtain c. The elements may be single numbers, vectors or matrices. The elements may also be structures that are mixtures of numbers represented as integers and/or floating point numbers with different precision. It is also possible to have the elements represent characters.

The functional representation incorporates both binary and unitary operators. For example, if we have c = a + b we can change this representation to be c = add(a,b) or using the symbol + for the function add, we have c = +(a,b). In the case of the unitary operation we have c = -a be represented by c = -(a) where - is the unitary negation operation. In the latter case the input to the function is only a single element. The functional description can be extended to multiple input and multiple output functions. In a notation borrowed from MATLAB we have [x y z] = func(a, b, c) for a three input element (a, b, c) and 3 output elements (x, y, z) of the function. A complex series of operations for the computation of a quadratic root such as

$$X = \frac{(b^2 - 4\,ac)^{1/2} - b}{2\,a}$$

can be represented using functional notation

x= div(sub(pow(sub(pow(b,2),mul(mul(4,a),c)),1/2),b),mul(2,a))

The functional representation expressed in the above equation when diagrammed, gives insight into the parallel structure of the computations.

This is clear from the resulting computational graph shown in Figure 2-1. In this figure the input elements are the variables a , b ,c and the constants 2, 4, and 1/2. The function operations are *, ^ , - and / which are mul, pow, sub and div respectively. These operations are represented in the graph by nodes We attached an additional label to the operations to distinguish the instances of the same operations. As is evident from the graph, the longest chain of computations beginning with mul(a,c) will require six operations to complete the computation. The operation mul(2,a) only needs two operations. While this computation graph gives effective insight into the computation required by the algorithm, other refinements in the graphical representation can be added to give a complete description of the computational behavior of the algorithm and to include the temporal nature of the computations. The computation graph as presented, just shows the precedence relationships among the individual computations. When the computational graph is executed the temporal nature of the computations comes into play.

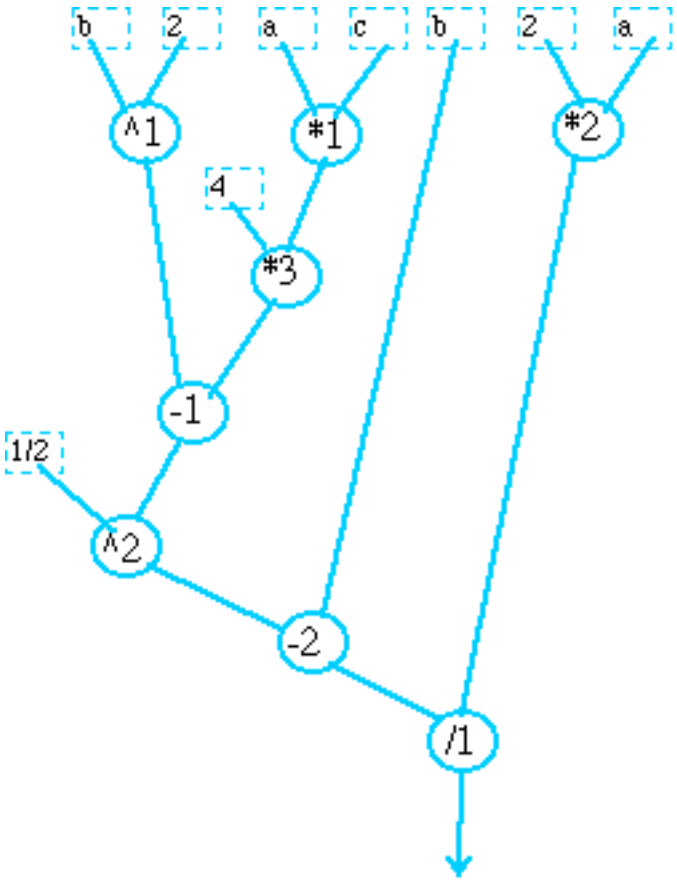


Figure 2 - 1: Computational Graph for Quadratic Root

For our example let us assume that all the operations take a unit time to execute. We begin the development of the timeline of the execution by assuming at time = 0 that the variables and constants have been assigned values. Figure 2 - 2 is a timeline of the executions showing the maximum parallelization of the computations.

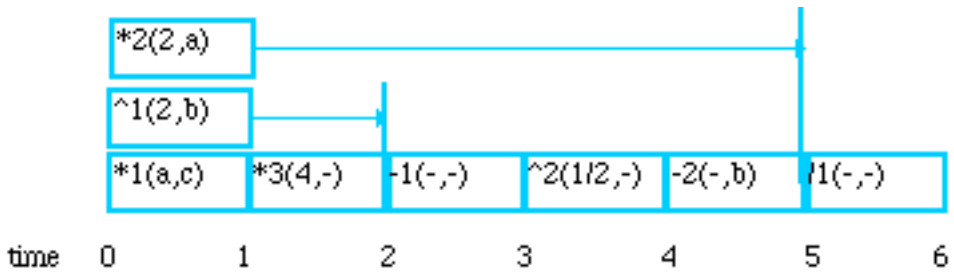


Figure 2 - 2: Timeline for Quadratic Root Computation

In this figure each operation is given a unit execution time. We have assumed that the data transfer as indicated by the arrows from the parallel computations is executed in zero time.

2.2 Data Flow Description

While the computation graph expresses some of the aspects of the computation we extend the graphical description of the computation to encompass additional attributes of the computation. The graph that results is called a Data Flow (DF) graph.

The computation graph expresses precedence among operations. In order to reuse the computation graph on successive sets of input variables we need a new form of graph. In order to formalize the expression of this successive computation mechanism we develop the DF graph. The values of the input and intermediate variables which are represented by an arc in the computation graph become the storage locations for these variables. The mechanism of storage of these variables is First In First Out or a FIFO queue. The quadratic root DF graph is set up by initially loading an input set of queues with values of the variables a, b, and c. The nodes of the DF graph now represent the computations that are performed upon the set of data items obtained from the input arc or input queues associated with this node and the output data items are placed on the output queue. In order to allow the temporal nature of the computation to be expressed in the DF graph the concept of threshold is introduced. Each node has the capability of checking all its input queues and when there is sufficient data on all the queues that exceed the threshold, the node executes and produces the output. After the node executes or fires the input data is removed from the input queues. With this firing mechanism implied in the graph, the DF graph representation now contains, in addition to the data values, control information that controls the execution of the graph. The actual sequence of computation represented by the DF graph is now not a simple linear execution of the nodes as it was with the computation graph.

A simple example illustrates the basic nature of the DF graph as differentiated from the computation graph. This DF graph consists of a two nodes shown in Figure 2-3. The first node is a node has a input threshold of 1 and produces 10 pieces of data. The next node in the Data Flow graph has a threshold of 1. When the first node fires 10 data items are placed on the interconnecting arc or data queue that this arc represents. The second node will now have to fire 10 time to remove the ten data items from the intermediate queue. The actual computation graph would now show the first node and then 10 second nodes as Figure 2-4.

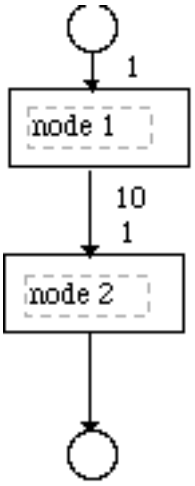


Figure 2 - 3: Two node DF Graph

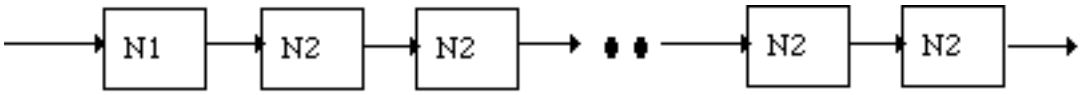


Figure 2 - 4: Correponding Computation Graph

The next issue addressed is when the first node fires, the 10 data items it produces are now available. These items can be processed in cascade creating 10 firings (see Figure 2-4) of the second node or the 10 data items can be placed in 10 queues and the second node can be replicated 10 time so that we may process all the data in parallel. We can draw a new DF graph to represent the required parallel computations as shown in Figure 2-5. The individual second node computation nodes must then be mapped to individual processors in order to enable the parallel computation. The purpose of this parallelization and mapping of the computations is to reduce the computational latency of the algorithm execution. In the execution order of the first DF representation of Figure 2-3 it takes 11 execution times to produce the output from the 1 input data item. If the executions were performed in parallel as shown in figure 2-5 then the computational latency can be reduced to 2. This simple example shows that in order to be able to achieve parallelization the DF graph must be altered to express the parallelization.

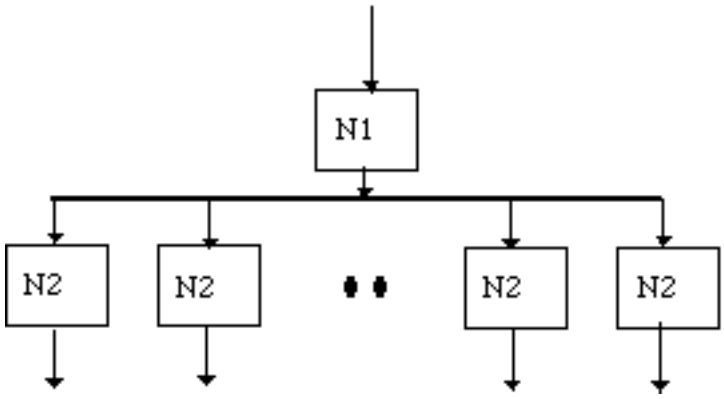


Figure 2 - 5: Correponding Computation Graph

This example illustrates the task of the graph designer. Even though the two DF implementations produce equal functional results, the two node Data Flow graph originally proposed must now be altered to produce an 11 node graph in which the first node produces 10 items and each output of the first node is followed by a single second node. This example illustrates that the design of a Data Flow graph is affected by the requirement of computational latency. When the problem requires parallelization in order to achieve this

requirement, the design process or capture process of the algorithm to the DF graph must include the parallelization constructs or nodes. The next section discusses the alternative type of node specifically needed in the DF graph to implement parallel forms of the algorithm.

 Dennis Basara

RASSP Data Flow Graph Design Application Note

3.0 Introduction to Data Flow Graph Design

In this section, we will show how a set of algorithms that make up a signal processing algorithm can be converted to

1. a data flow graph, and
2. a meaningful set of primitives that support the resultant graph.

In the introductory discussions of DF graphs, we have only introduced the processes of data distribution. When the data resides on the same processor that performs the computation, the data is not moved but a pointer to the data is transferred between the consecutive processing nodes. When the processing nodes are separated on multiple processors, the data is actually copied from one processors memory space to the other. The copying process actually takes a finite time to execute. The designer must be aware of this time but usually, in the early stages of the design, the data movement time is assumed to be negligible or easily accommodated by adding an additional processing time to each node's computation time. As the design progresses the actual transfer time is included in the simulation to verify that the system has met its latency specification. In the next section we introduce the set of specialized nodes that are used to copy and replicate the data so the DF graph can express the parallelism in the algorithm.

3.1 Parallel Constructs

A DF graph is an ideal tool for representation of course grain parallel data processing. Coarse grain processing is best illustrated by example. In the example discussed in Section 2.0, the graph represented operations upon single data items such as a, b, and c and produced one result, x, the root. In general, the setup time for the functional calls needed to implement the computations represented by the nodes of the graph, will exceed the time it takes to execute the single functional operation performed by the node. Thus using a series of nodes to perform a sequence of operations will increase the overall execution time of the graph. In order to spread the setup time for a single node computation over many computations the graph is made coarse grained or each node is made to operate on a group of data each time it is fired. An example of this coarse grained operation would be the operation of an FFT node. The FFT function would operate on a data set or a vector and produce a vector of data for each FFT operation. Very often, even single operations as those implied by the computation graph in Section 2.0, can be converted to vector operations. Each of the of a items of a, b and c can be grouped into a queue of a, b and c and a single computation node can be made to include a loop to operate on each of the input items sequentially. From a graph point of view, the new graph is now coarse grained. Once the processing is put on a coarse grain basis each of the node operations can now be distributed. By assigning different nodes to different processors the processing can be performed on multiple processors simultaneously thus parallelizing the signal processing problem. The graphical notation used to express this data separation is the node constructs of SPLIT or SEP(aration) and the rejoining primitives of MERGE or CAT (concatenation). The use of each of these techniques will be discussed and illustrated by examples in the following sections. Other data organization elements such as MUX, and DEMUX will also be discussed.

3.2 Splits and Merges-Sequential Splitting

Usually data arrives at the input queue in a steady rate and this data must be processed sequentially by a

computation node as shown in Figure 3-1 There are two methods of processing blocks of data so that the processing can keep up with the input data rate. The first method assumes that the node can complete the processing of the first block before the second block shows up. In this case, as shown in Figure 3-2, the latency or the time it takes to produce the results of processing the first block of data is less than it takes to process a block of data. If there exists a nonlinear trade off curve between block size and processing time, a block size can be determined that allows the computation rate to keep up with the data arrival rate.

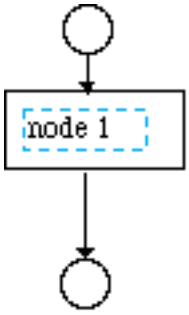


Figure 3 - 1: DF Graph

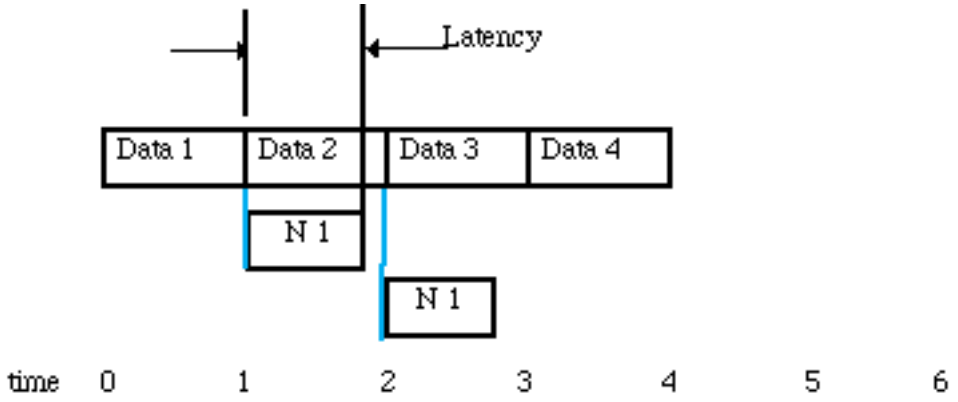


Figure 3 - 2: Timeline and Latency Computation

This block size can be found by equating the block arrival rate to the block processes time as shown in Figure 3-3. In the case where there is no solution for any block size or it is impractical to use the solution block size, then the processing must be parallelized as shown in Figure 3-4. The parallelization procedure begins by inserting a Splitter node which will send the first block of data to node1 and the second block to node2 etc. If the nodes are distributed to separate processors then the time line shown in Figure 3-5 results. For this method of parallelization the timeline shows that the latency that is greater than one block processing time. In this case, the latency can be decreased by using a smaller block size and using more processors and this is shown in Figure 3-6. This figure shows that we have cut the latency by a factor of two by increasing the number of processors by a factor of two. This is possible when the processing time is a linear function of the block size.

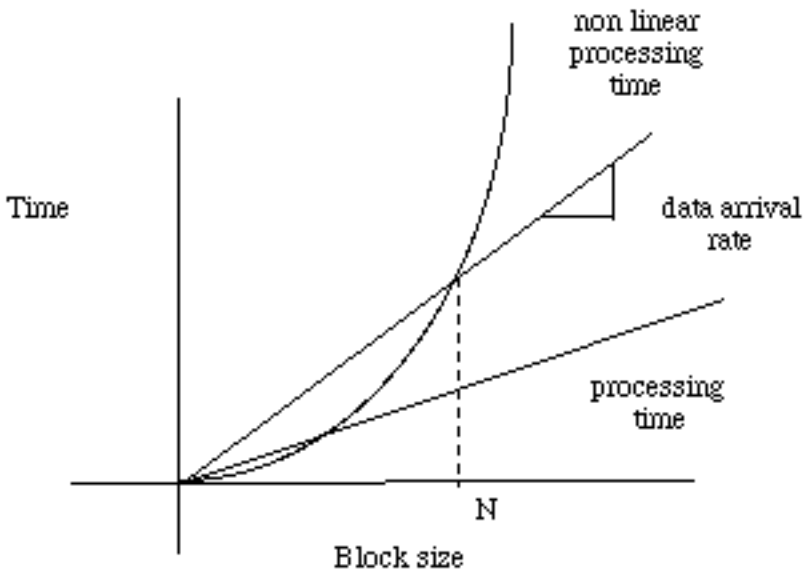


Figure 3 - 3: Nonlinear Solution Block Size

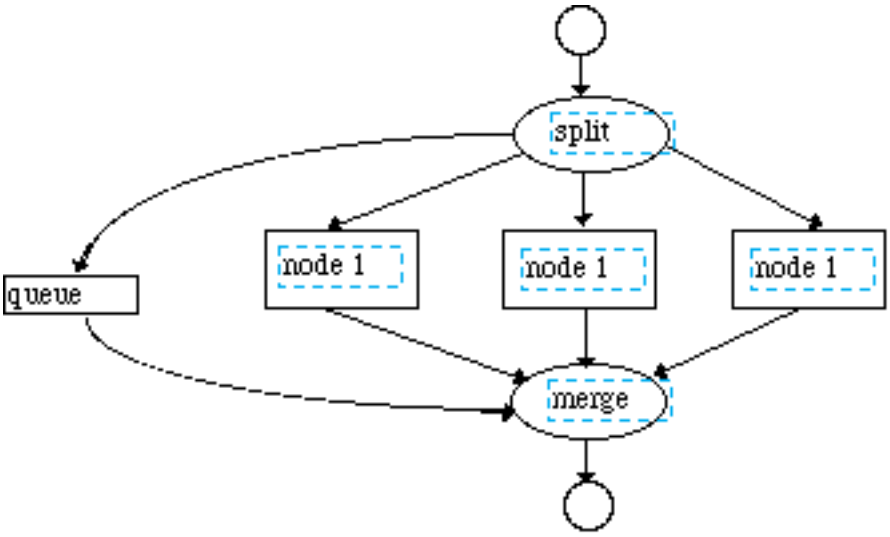


Figure 3 - 4: Split Merge DF Graph

The other node that is shown in Figure 3-4 collects the data stream and is called the merge node. This node actually fires when a block of data arrives on one of its indicated input nodes. Strictly speaking, the merge node does not behave as an ordinary DF node. The rule for firing a DF node is that the node fires or computes an output when all the input queues are over threshold. In the case of a merge node, only one of the input nodes will be over threshold at a given time. In practice, the merge node uses an additional queue of control data generated by the split node that indicates which of the input queues should be examined by merge node for the queue over threshold signal. This additional queue is shown in the graph of Figure 3-4. This control node assures us that the merge node will reassemble the data blocks in the same order that they were split apart by the splitter node.

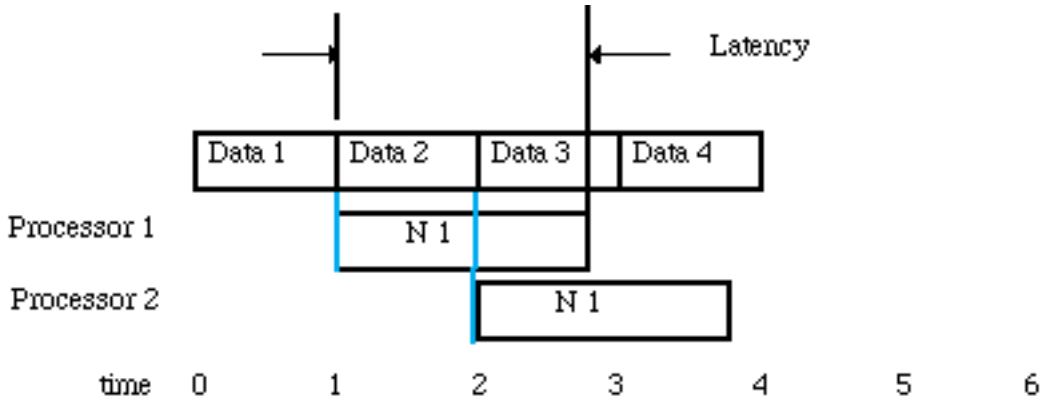


Figure 3 - 5: Computation Latency

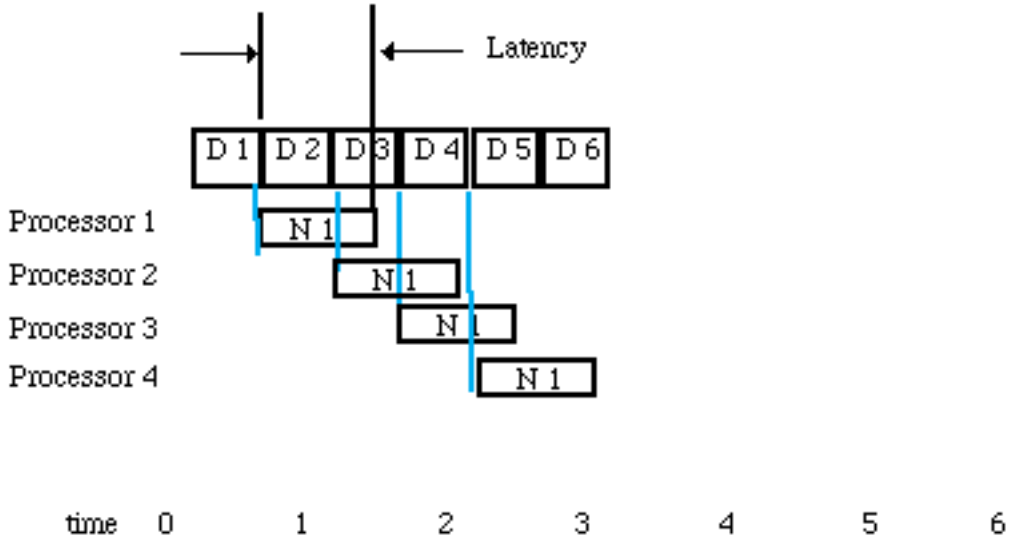


Figure 3 - 6: Reduced Computation Latency

In the SAR example (see SAR Case Study) the split - merge pair was used to distribute the range processing data to each node. The graph also made use of the Family notation so the discussion of this graph will be deferred to Section 3.4.

3.3 Sep(aration) and Cats(concatenation)

When the input rate is large compared to the processing rate we can assume that the input queue is filled instantaneously. In this situation, the split and merge can be replaced by the Separate and Concatenate node. These operations are functionally similar to the split and merge nodes but the Sep node requires that the N blocks are present in the input queue before the node fires. N is the number of streams that the input block will be separated by the Sep processing node. Similarly the Cat node takes the N input nodes and following the DF rules fires when all the N input queues to this node are present. This combination of nodes is shown in Figure 3-7. In the figure the sep_a node uses the dfc_sep primitive and the mux_a node uses the dfc_cat primitive. The usage of the Sep - Cat pair is similar to the Split - merge pair but the Sep- Cat obeys the Data Flow rules and does not require the additional control queue that insures that the proper firing sequence is maintained. There is another primitive node that can be used to separate data streams. This node is called DEMUX. The multiplex operation separates an N block queue into N streams but the data is not taken in contiguous blocks from the input data stream as in the Sep node. There is a stride applied to the selection of the data items from

the input steam. The resultant output block now contains a group of data in which each consecutive item comes from the input stream and was separated by the stride from the next data item in the input stream. The corresponding concatenate operation is called MUX and reassembles the data stream.

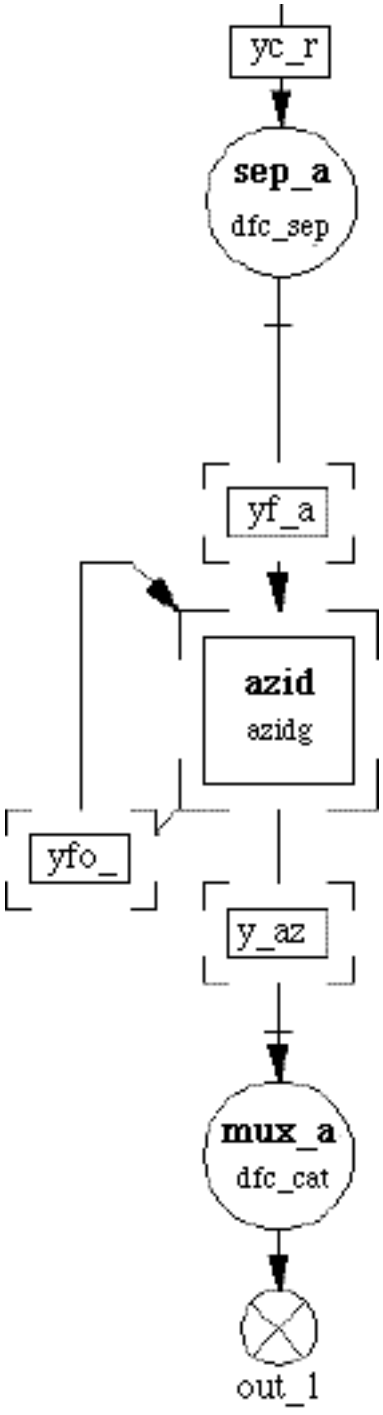


Figure 3 - 7: SEP/CAT Graph

3.4 Family Notation

In DF graphs we introduce an important notational construct that supports parallelization and it is family construction. By the use of this construct, a group of nodes and or storage queues may be vectorized. The individual entities are directly addresses by an index. This convenient notational technique allows the graphs

to be simply constructed. For example if a Sep and Cat operation are used, the intervening node can be put into a family index notation. The family notation is shown conceptually in Figure 3-8. In this case the output of the Sep node is put into inq family of queues indexed by i = 1..n. In this example we feed [i]inq to [i] node and feed the output queues [1..n]outq to the CAT node. The Cat node is set to take a family of queues. The Family notation can be used on a graph that has routing complexity that is greater than the simple one to one routing shown in the example given. To accommodate this complexity a routing box can be inserted after inq to route the output of queue [i]inq to any node in accordance with a routing formula on the index i or expressed using a routing graph that can be explicitly drawn.

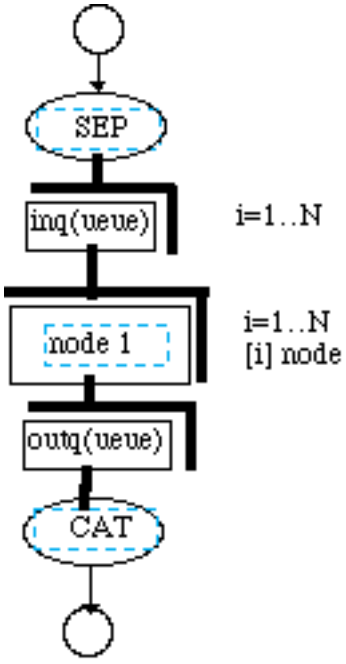


Figure 3 - 8: Family Notation DF Graph

3.4.1 Examples of Family Notation.

In the SAR example the split - merge pair was used to distribute the range processing data to each node. In this manner each set of range cells could be processed in parallel. The graph was developed with the split node because the range processing time was greater than the time it took for one range block data to arrival. Figure 3-9 shows the overall graph for the SAR processing. In this case the range

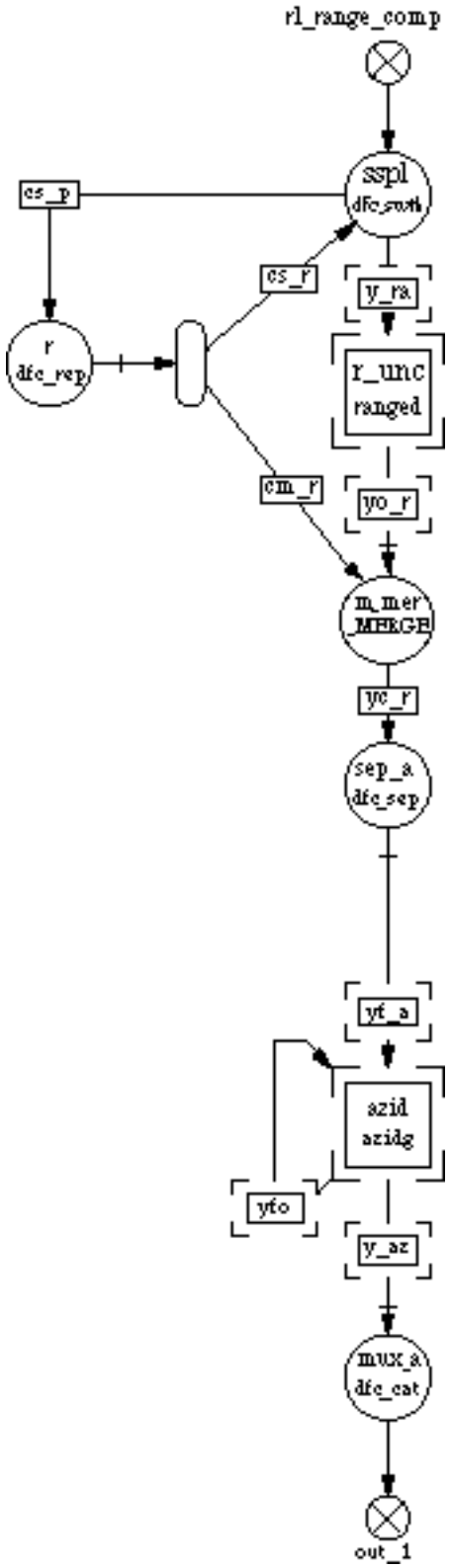


Figure 3 - 9: SAR Processing Graph

processing is shown is executed between the split and merge nodes and a Family notation is used for the range processing nodes. The external feed-forward queues are shown that feeds the control queue from the split node (dfc_swth) to the merge node. The addition of the dfc_rep node in this path was used because of the configuration of the dfc_swth node that was used to implement the control feed-forward. Since the azimuth

processing could not proceed until a full frame of range data was processed, the azimuth processing uses a Sep and Cat pair. This pair was used to implement the corner turning required by the SAR processing. The processed range data could only be distributed only after the queue following the merge node contained a complete frames worth of data. A Transpose node could have been used to perform the corner turning but in order to perform the azimuth processing in parallel, a Sep node would still have to follow the transpose node. Therefore the transpose operation was performed by the Sep node in addition to the parallelization of the data stream. Again the azimuth processing nodes are represented using family notation.

Because of the setup time for a single block of range processing was large for each individual block, the input blocks of data were grouped together and the range computation node was performed on a group of blocks at one time. The range processing node included a loop for each block of range data even though the inner processing was still performed on a range block by range block basis. Since all the computations were performed on one processor the successive range computations could be pipelined so that the setup time was only present for the first range block and the overall processing time reduced. The additional latency caused by the range computation being performed over several blocks at a time was acceptable. This latency is small compared to the amount inherent in the corner turning operation of the SAR computation. A full frame (range by azimuth block data size) latency exists as a minimum even in addition to the azimuth computation time.

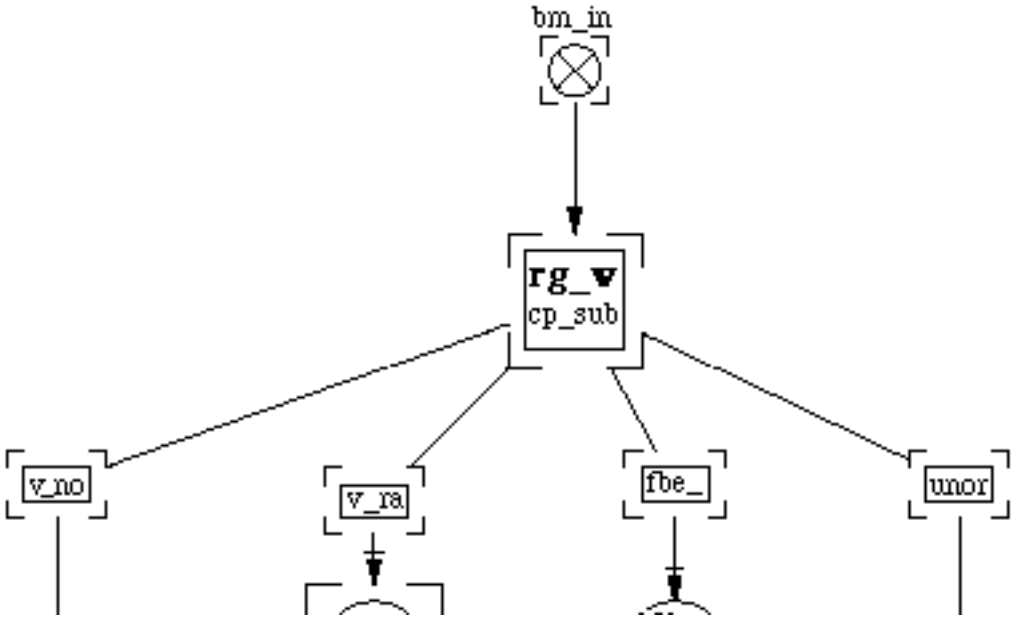
3.5 Data Flow Design Verification

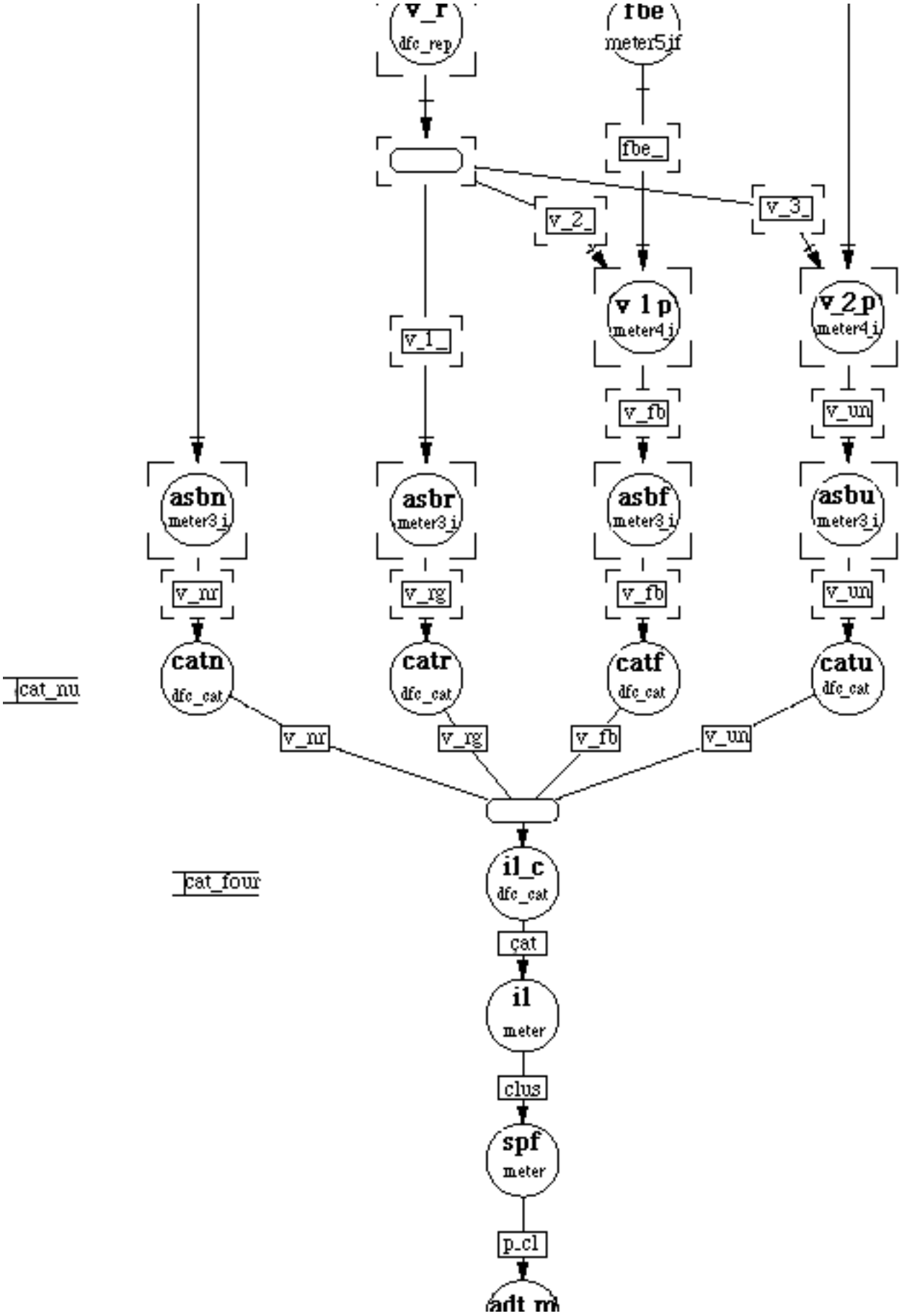
After the DF graph has been constructed using the appropriate parallel constructs, a simulation should be set up to verify that the data stream is operating as desired. A graph can be set up using:

1. the Parallel construction nodes (Split-Merge, Sep- Cat, Mux Demux)
2. the replicate node which creates identical streams
3. replacing all the computational nodes by identity nodes that reproduce the input stream at the output

The initial input data streams can be made to be a simple sequence of numbers. As the data is passed through the node of the initial Data Flow design the integrity of the data stream can be verified.

This technique is illustrated by the ETC graph.(see the ETC4ALFS on COTS Processor Case Study for additional information regarding this application). Figure 3-10 and 3-11 shows how the DF was set up and tested for the complete ETC algorithm The data input arrives as a block of range data that has been processed by the matched filter and there are [1.. Num_beams]input streams. The per beam processing represented by the family of rg_w processing is shown in the subgraph of Figure 3-11. In all cases shown in the figure, the identity processing nodes are represented by the Meter nodes.




v_r
dfc_rep
fbe
meter5jf
fbe_
v_2_
v_3_
v_1_
v 1 p
meter4_j
v_2 p
meter4_j
v_fb
v_un
asbn
meter3_j
asbr
meter3_j
asbf
meter3_j
asbu
meter3_j
v_nr
v_rg
v_fb
v_un
catn
dfc_cat
catr
dfc_cat
catf
dfc_cat
catu
dfc_cat
cat_nu
v_nr
v_rg
v_fb
v_un
cat_four
il_c
dfc_cat
çat
il
meter
clus
spf
meter
p_cl

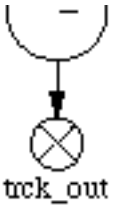


Figure 3 - 10: ETC Simulation DF Graph

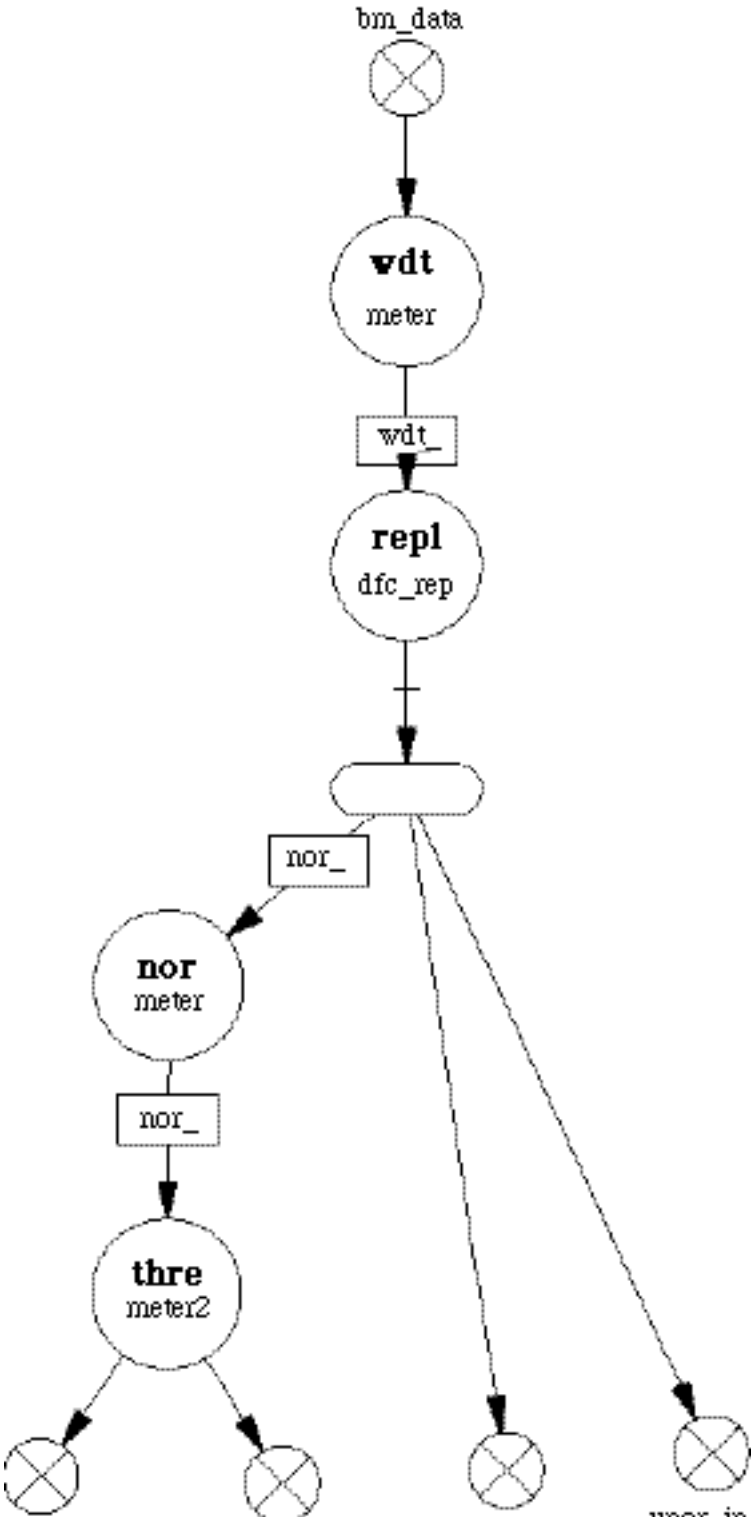


Figure 3 - 11: ETC Channel DF Graph

The family output data streams were then combined as shown using the CAT node. The Processing was then completed by as set of sequential nodes. This DF graph design was tested with sequential data streams and the integrity of the DF design established. By using a performance simulation tool and assigning representative processing time for the processing nodes (meter nodes) a execution time line was developed. This is shown inn figure 3-12.

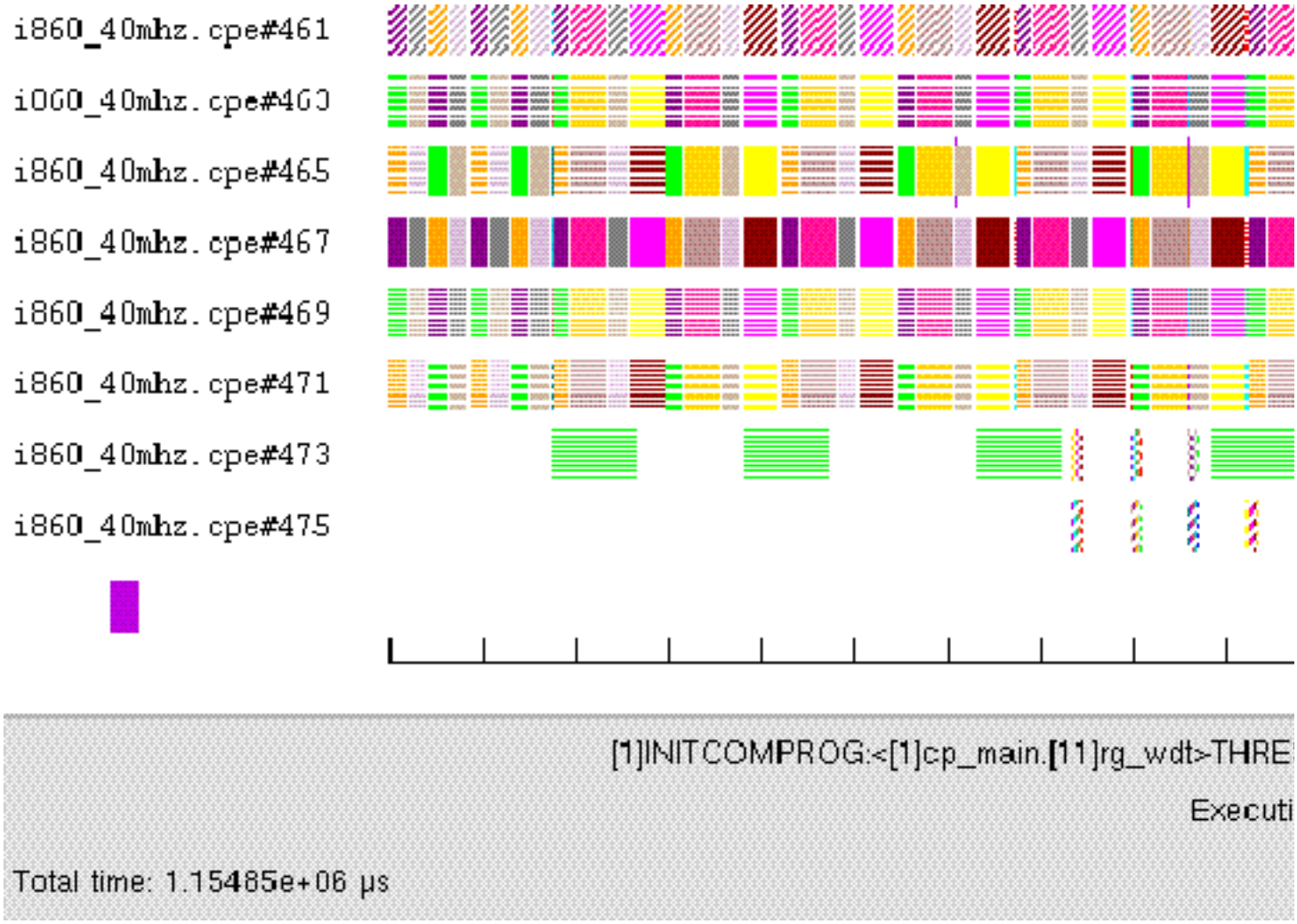


Figure 3 - 12: Timeline for ETC DF Graph

The Processing was divided onto 8 processors. In this initial design the node processing times were approximated by a simple linear time formula as a function of the block size N. Processing time for a node is t_{node} = a + b N formula where a and b are processing times and N is the number of data item processed per node firing.

Next Up Previous Contents

Next: 4 Concept of State **Up:** Appnotes Index **Previous:**2 Introduction to Graphical Data Flow Description

RASSP Data Flow Graph Design Application Note

4.0 Concept of State

In general, a pure DF graph architecture is greatly simplified if all the input data elements to a node can be processed the same way. In a general signal processing problem, however, the processing of data packets depends upon their position in the data stream. Usually the first and last packets of a data stream have to be augmented by additional data. For example: when running a FIR filter on a data stream, the filter must be initialized and terminated properly.

There are essentially two methods available in a DF graph to accomplish this data packet dependency processing. The first technique makes use of a feedback queue to keep the value of a counter. The processing performed internally by the node is made to depend upon the current value of this counter which is taken from this queue. For this implementation, the internal node processing was made to depend upon the node state i.e. the count of the data packet that is currently being processed. The code that implements the state update and processing of the state is in the node processing wrapper.

There are several tools that simulate and generate code for DF graphs. The tool that was used for RASSP was GEDAE™. Using GEDAE™, this state processing becomes part of the GEDAE™ methods. Alternately, the node processing state may be stored as a local variable as part of the GEDAE™ methods instead of being a queue variable where it would be a formal part of the graph.

The other technique that is used to mark the state of a data packet is to embed in each data packet its state. Each node will now have to extract the state marking to determine the processing required. This, in theory, this makes the DF graph non pure, i.e., data and control information are mixed in the same data stream. This technique may often be a preferred solution to the feedback queue because it insures synchronization of the processing with the data packet. The difficulty with the implementation of this technique is the actual insertion and removal of the state header. In a complete signal processing system, the data stream is usually prepared by an input processor. In many cases the state information can be inserted into the data stream as a header. However, each node that processes this packet with the header is now required, as a minimum, to remove the header. Each node that is used in the graph is now required to have a primitive method to remove the header from the data before the core processing associated with this node can be accomplished and then the header must be reinserted into the output stream.

A practical solution with the exiting DF software development systems would be to use the feedback queue to implement the state dependency during the initial design of the DF graph. This will allow the use of a maximum amount of library primitive nodes in the design. As soon as the basic DF ideas are implemented and tested on the workstation version of the graph implementation, the graph can be redesigned with the header packets. In practice, the state queue feedback method requires the designer to maintain synchronization of the processing of all the nodes with the correct packet state. This is made more difficult since the packet count must be kept locally on each node. The header method, which keeps the state information with the packet, is a direct method of synchronization.

4.1 Data Processing by Slices

An example of state dependent processing is illustrated by the ETC norm processing. In this case the data stream for each ping was divided up into slices which were batch processed. The first and last slice of each

ping is required to be processed differently because the first slice data stream must be augmented by folding the first n data points and adding them to the beginning of the slice. Similarly, at the last slice of the ping the data is folded and added to the end of the last slice's data set. Figure 4-1 shows the feedback data queues that maintain the state or the slice count.

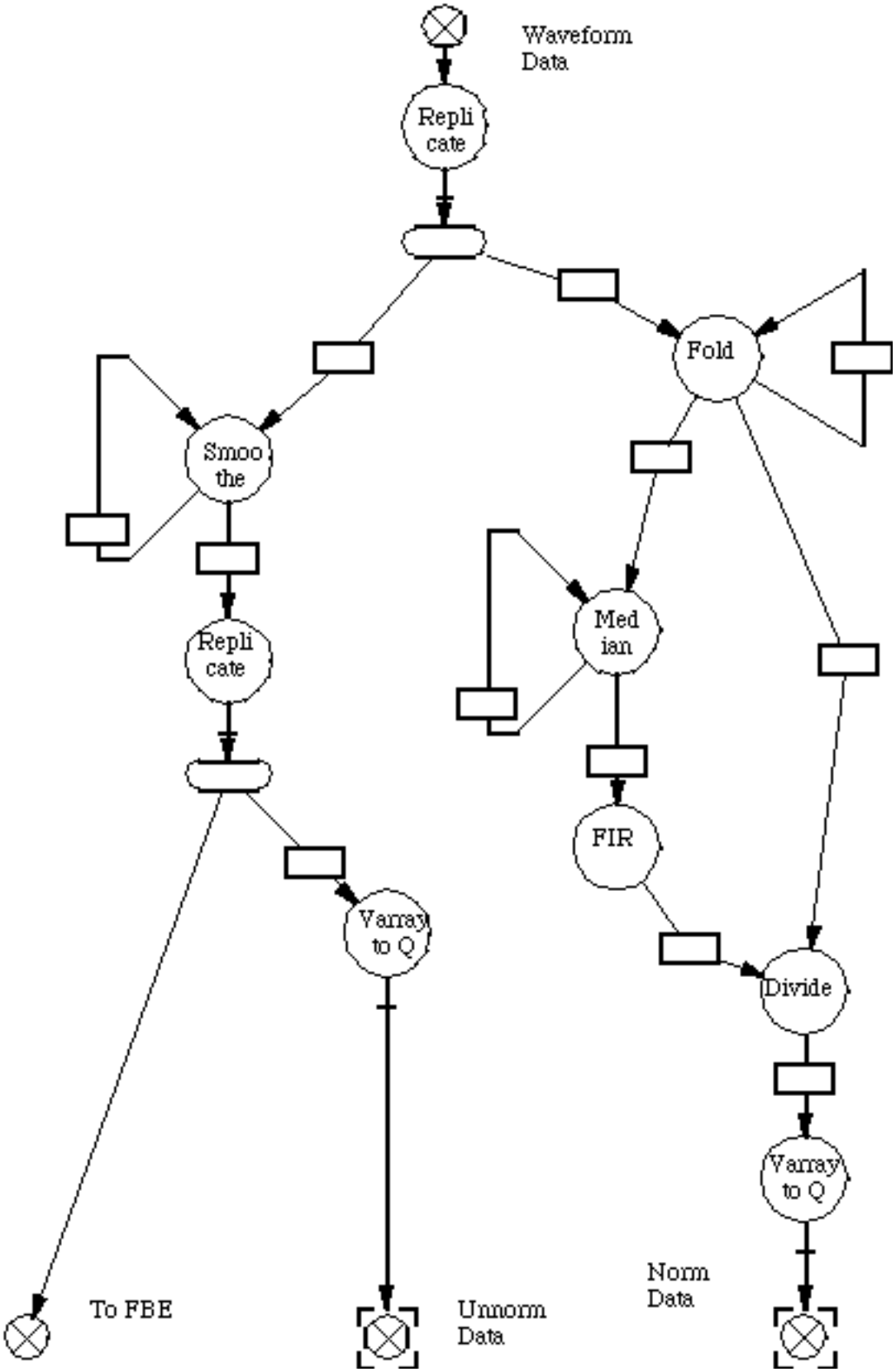


Figure 4 - 1: ETC Graph

The folder, median and smooth require the slice count and include the counter update code as:

counter = counter + 1 modulo number_slices

where counter is the value (state) stored on the feedback queue. The internal computation in the node makes use of the counter value to determine which slices require the additional folding to be processed.

RASSP Data Flow Graph Design Application Note

5.0 Memory Management

In a DF environment, the queues perform the function of local data buffering. In a practical DSP implementation, buffers must be sized so the implementation of the queues will have sufficient size to provide for the maximum expected size of the queue. If a selected set of the queues are sized too small, as the DF graph is being executed, there will be blockage at the output of the node and this blockage will be propagated upward to the beginning of the graph. This blockage will affect the timeline by preventing execution of certain of the nodes. The expedient of allocating larger storage for the queues to prevent this blockage, trades between storage requirements and execution times. As the execution time decreases on the down stream nodes this will allow the downstream data to be removed faster and thus unblocking the upstream queues.

There are other design techniques that can help reduce the required memory such as, small packet sizes and additional parallelization. In the former case, the small packet size will allow a reduction in the queue length since the maximum size of the queue required for operation of the graph is usually a multiple of the packet size. Reducing the packet size reduces the maximum queue size required. Adding more paths through the graph as a result of additional parallelization of the graph will decrease the time it takes the total graph to execute thus cutting down the data that must be stored in the graph to complete a full cycle of the graph execution. Parallelization will reduce the data storage and sizes of the queues in the graph.

5.1 Allocating Queue Sizes

Peak sizes of the queues and hence the memory required for the graph is a dynamic quantity . This can usually be determined by a performance simulation of the DF graph. Using GEDAE™, the graph is run and the peak value of all the queues can be recorded and the memory allocation required can be determined.

 Dennis Basara

RASSP Data Flow Graph Design Application Note

6.0 Queue Data Structures

In a DF environment the queues are used to store the data items locally and when the processing is distributed, this data must be sent from one node to another in order to continue the processing. The data types stored on each queue are usually of one type in "C" notation i.e. int, float, double, etc. These data items can be single words or arrays (one to n dimensional). Both the queue of stat and the array are stored contiguously in memory. When arrays are stored, the association is a logical grouping. In this sense, the logical grouping of the queue affects the way the node inputs data. The physical data is always a stream of data and the node following the queue can be set to threshold on n data items in this stream. If we logically organize the data into vectors, the threshold must be set to the number of vectors. The stream threshold would be n while the vector threshold would be 1. Another logical grouping is the matrix. For a matrix of size 'n x m' the threshold for the matrix is 1 and for the stream the threshold must be set to 'n * m' data elements where 'n' is the row and 'm' is the column count respectively.

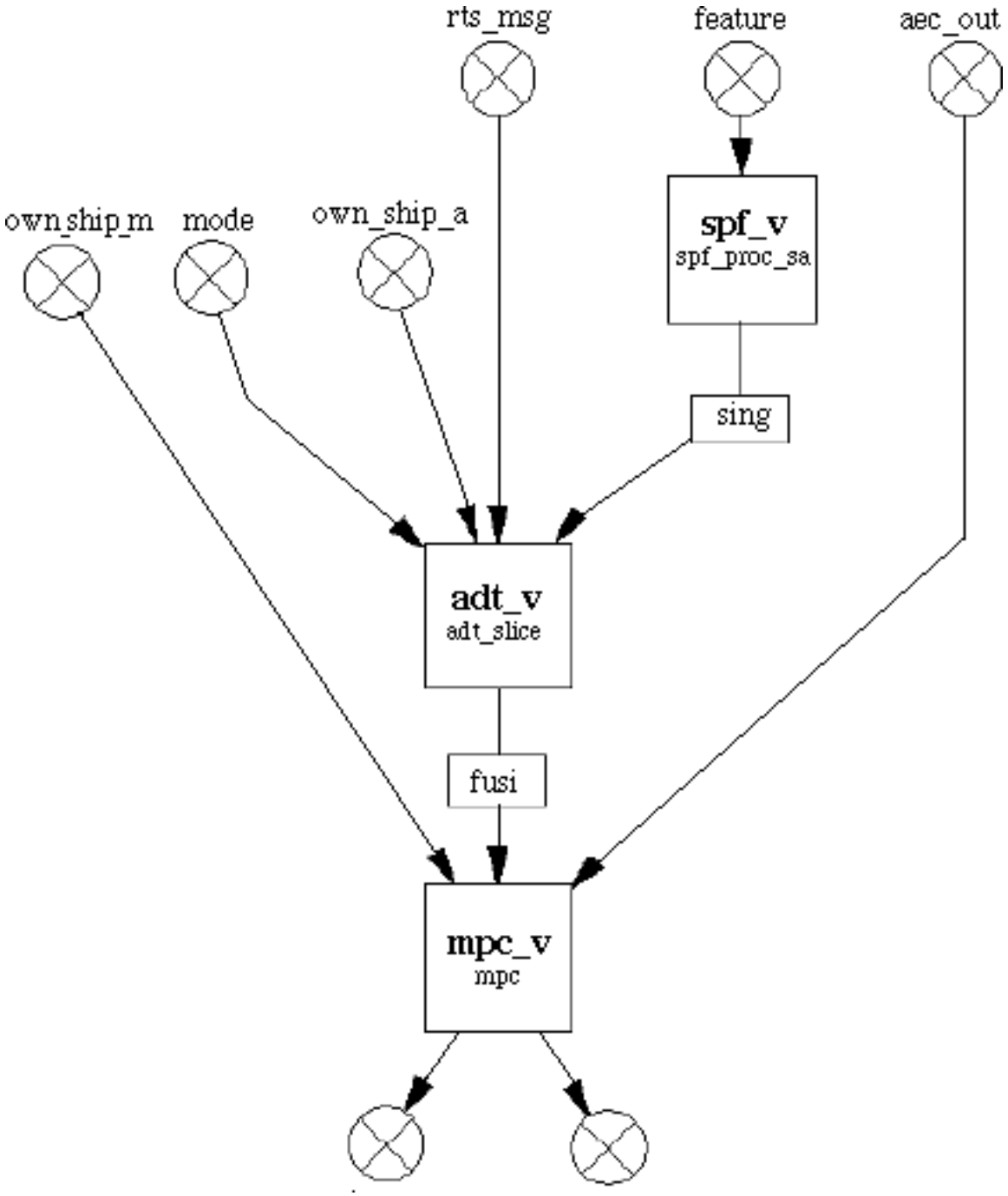
The threshold either in terms of the stream or vector grouping of the data is used to control the packet behavior of the DF graph. We now examine the situation when a node produces a variable amount of data. If we desire to maintain a consistent firing order of the downstream nodes we can either dynamically change the threshold to correspond to the data amount produced by upstream node or we can have this node produce a V array. In the case of the V array, the threshold can be kept constant to one V array and now the node will always produce one V array every time it is fired. The sequence of node firings is maintained for any amount of data produced by each of the individual nodes using the V arrays. The data stream of a V array contains count items. The first item is the actual amount of data produced and the second item is the maximum amount of data that could be produced followed by the number of data items. The V array is useful for building embeddable graphs that requires the firing order to be determined at compile time of the embeddable code. The V array keeps the node execution sequence of the graph static and fixed at compile time, while allowing for internal dynamic data length variability.

An useful extension to the queue structure would be a queue that contains a structure. Even though the current DF implementations do not support this data structure directly this would be a worth while improvement. The structure definition would contain a mixture of floats, ints and doubles. The structure can be considered as a generalized vector and allow structures in the"C"sense to be passed between processing nodes. An additional extension to the DF graph would be the ability to pass a block of structured memory such as a heap with a pointer to one of the data items in the heap. This queue type would be useful for moving an entire linked list between processing nodes without flattening the list at the output of the send node and then restoring the flattened list to a heap structure format on the receive node. This form of queue would not require the graph to include the additional operations and processing time needed to flatten the list to a stream and then restoring the list into list processing format. The decrease in processing time would come at the expense of increasing the memory size needed to store the entire heap area rather than just the data items that are used in the list.

6.1 Using V Arrays

Often the DF mechanism is used to implement a set of data processing algorithms. These processes are very often needed to complete the processing of data sets that have been created by conventional signal processing algorithms. For example in the ETC example, the data set have been produced by a series of beam forming and filtering operations. After these operation, the data set that results is processed by clustering and tracking

algorithms. In the ETC example shown in Figure 6-1 is the so called back end processing operated upon cluster data. The three nodes represented by the spf_v, adt_v and mpc_v nodes sequentially process the feature queue. The contents of the feature queue is 0 .. n sets of data of vectors of length seg_size. The feature queue is formed for each slice of processing and the number of clusters contained in this array is data dependent. There is a limit on the number of clusters that may exit per slice. The V array was used as the queue element. In this case, as the node spf_v fires it consumes one V array and produces one V array marking the end of the slice processing. Using V arrays we can also send an empty packet which results when there are no clusters in this slice of data. In order to complete the back end processing, a queue needs to be formed for every slice processed. If a marker was not present when there were no clusters for this particular slice, a synchronization problem would develop for the downstream processes. The adt_v node has three other queues in addition to the main cluster V array and the data in the queues must all pertain to the same slice. By the use of the empty V array, the place marker, the adt_v processing is kept in sync even when the slice contains zero clusters.



class_w class_f

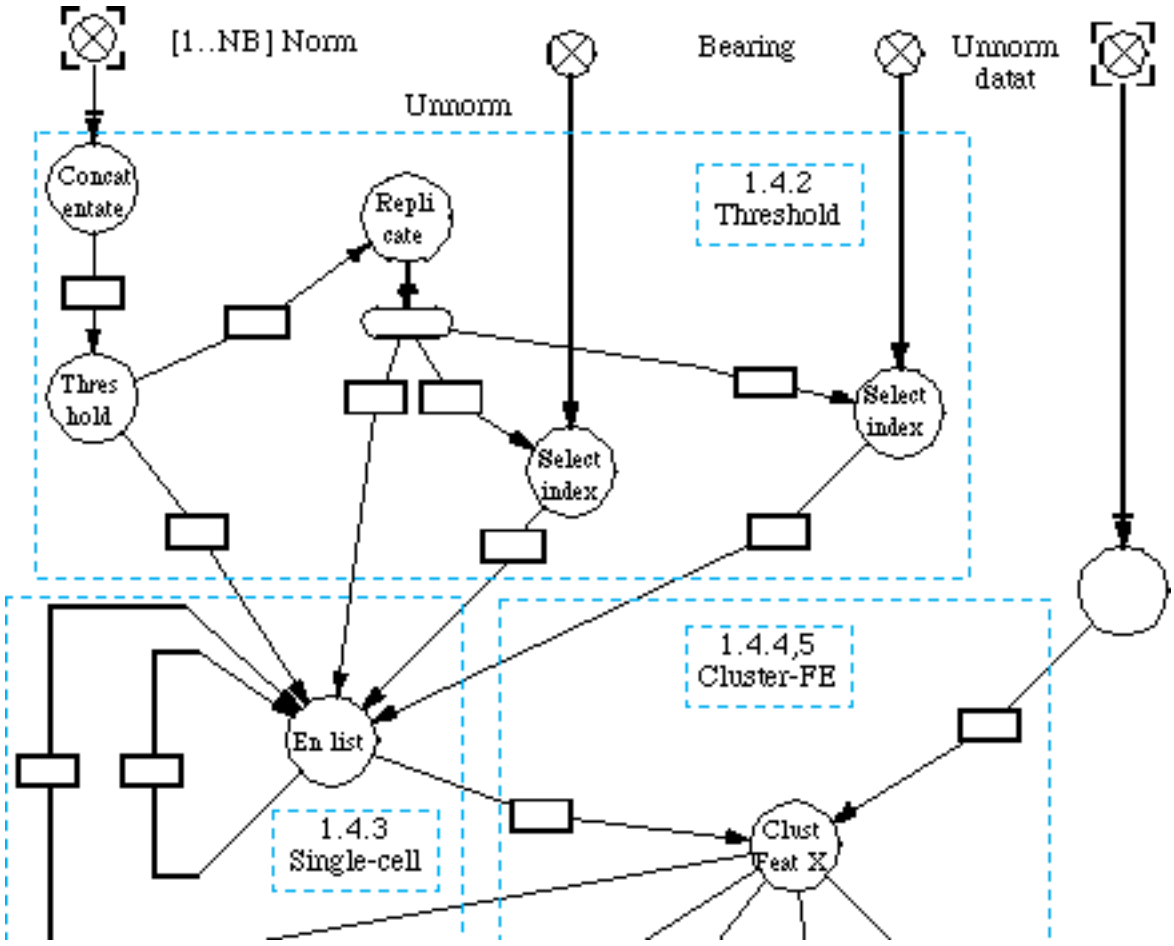
Figure 6 - 1: Back End Graph

6.2 Structures

Very often, the data items in the queue are not all of one type, i.e. integers of floats, but are grouped by a"C"code typedef of structure which allows the mixing of data types. In early DF implementations a work around was used where the structure was passed as a binary array and the data reinterpreted by the receiving and sending nodes sharing the same header with the typedef of the structure. The complications introduced by this approach is that every time the binary queues had to be viewed, programs that read the binary queues and interpreted the results according to the typedef were used. In other cases the mixed types were separated into int and float arrays but from a programming documentation point of view this is a logically disruptive process. In the future DF implementations this structure definition should be implemented.

6.3 Dynamic Structures and Lists

On occasion, the structure used to process data is a linked list. The first node takes a stream of data and creates a linked list of cells that were dynamically allocated from the heap. Figure 6-2, Cluster Feature Extractor (CFE) Graph, shows that the En list node creates a linked list and when the list is completed the clustering algorithm is run in the node ClustFeat X. In this DF graph when the node En list executed, only a token was passed to Clust Feat X so this node could begin clustering. The actual data structure remained in the heap and the clustering began at the beginning of the list as indicated by the list pointer. Since there was only one copy of the heap the two nodes En list and ClustFeat X, have to be on the same processor. DF processing is not supposed to work this way. If the processing performed by Clust Feat X had to be performed in parallel then there would have to be copies of the list sent to each one of the clul processes.



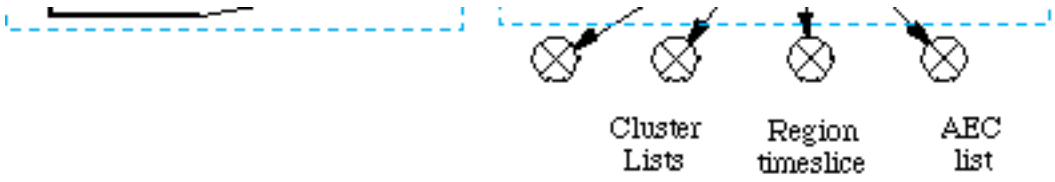


Figure 6 - 2: Cluster Feature Extractor (CFE) Graph

One method of accomplishing this is to develop a set of new() or malloc() programs which perform the dynamic cell allocation from the heap. However, in this case, the designated block of memory is input to the specialized malloc() so that the cells of the list are from the buffer area rather than the heap. The output queue of the node will now have to support a new data type which will be the entire buffer area and a pointer to the beginning of the list. If the ClustFeat X nodes are separate processors, each node will get a copy of this buffer area and the pointer. The merging of these data sets of this new created type will now have to be defined. When we want to merge many of these buffer areas, we usually wish to combine cells from each one of the lists contained in the input buffers so that a new list can be formed. The input queues would be the n buffer areas and the merge node would combine the lists and put out an updated buffer area with cells of data from the input cells in the form of a new list. This then would be a full support for list processing that would obey the DF paradigm.

Next Up Previous Contents

Next: 7 Primative Construction **Up:** Appnotes Index **Previous:**5 Memory Management

RASSP Data Flow Graph Design Application Note

7.0 Primitive Construction

In a DF graph the concept of a node or primitive forms the largest reusable element in the DF graph design process. In an ideal world, any signal processing problem can be decomposed into interaction between these primitive nodes. In order to allow for customization of the primitive nodes for a particular application the concept of the PIP (Primitive Interface Procedure) as well as the NEP (Node Execution Parameters) are introduced. Their purpose is to adapt a general purpose primitive to the particular application without directly customizing the primitive code. Very often these mechanisms to customize the node are not sufficient and a new custom primitive needs to be constructed. The construction involves the creation of a new node method.

7.1 Using PIPS and NEPS

These two concepts are implemented in the interface between the core node computation module or"C"function and the DF graph directly. These two items are supported in the GEDAE™ method procedure construction. In this section we discuss the usage of this facility in the proper operation and embedding of this computation on DSP processors.

The PIP has already been discussed in reference to graph state. Any computations that are involved with the setting of control variables that affect the operation of the node are called the PIP computations. This could involve the incrementing of the state counter as described in Section 4 or changing the input Node Execution Parameter (NEPS) to produce amounts on the output of the node.

The second node specialization are the NEPS (Node Execution Parameters). We have already discussed the major DF control variable called the threshold. In the basic operating mode of the DF graph a threshold is associated with the input queue. This threshold value is used to trigger the node computations downstream of the queue. When there is sufficient data or there is a block of data in the input queue of the length of the threshold, the node is fired. This amount of data is then removed from the input queue upon the completion of the node computation. In order to support a more complex operation of the DF graph we can separate the threshold in four quantities. The NEPS are:

1. Threshold Amount (T)
2. Read Amount (R)
3. Consume Amount (C)
4. Offset Amount (O)

When we were using a single threshold, we were assuming that T = R = C and O =0. However, other settings of these parameters can be used to accomplish operations that are needed for signal processing.

An important operation of signal processing is the Finite Impulse Response (FIR) Filter. This filter operation DF graph is shown in Figure 7-1 and the timeline of the operation of this node is shown in Figure 7-2. The input stream threshold is set, for example, to operate on 8 items and the FIR filter length is set to 4. The output as shown on the P1 output is obtained by taking the product of the first 4 input items and multiplying by the 4 hi (i = 1..4) values and summing the four products to produce the 1 output. If the entire data stream were processed in one batch then we would not require a data overlap. However, if we break up the input into blocks, the input of the preceding block must be processed to produce output for the next block of data. This

produces an overlap situation. The first execution of the filter node fires when there are 8 item in the input queue, and the

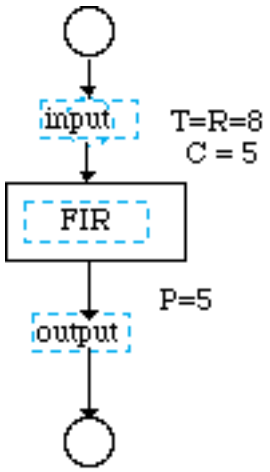


Figure 7 - 1: Fir Filter DF Graph

node produces 5 output elements. If the NEPS were set to T=R=C then when the node fired for the second time, the 1st output from the second firing or the 6th data item overall, is a result of the node execution upon the last 3 data items from the first block and the 1st one from the second block as indicated in the tiimeline of Figure 7-2. There are two alternate methods of storing these three data items from the first computation to be used by the second firing. The first technique just stores the 3 values as a node state and set the NEPS to T=R=C. In this case the first node firing produces 5 data items and the second and subsequent times we produce 8. The data set is entirely used if the total input data length is a multiple of T=8.

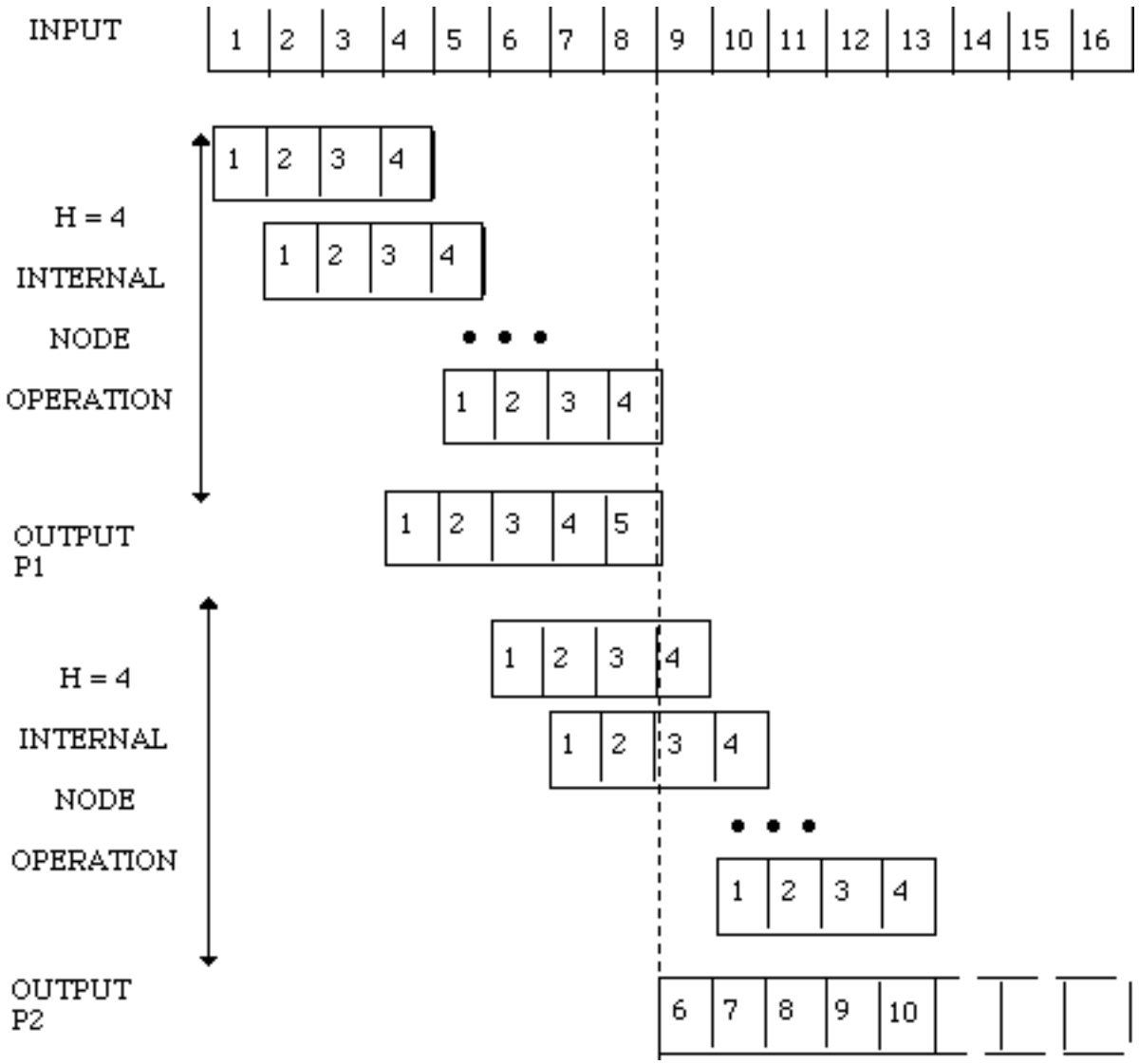


Figure 7 - 2: Fir Filter Timeline

The other method is to use the NEPS T=R=8 and C=5. In this case the produce amount is always P=5 until the last time. The last number of data items will not be able to be read because we do not exceed the threshold and consequently we need to have the NEPS change for the last firing of the node. By the use of NEPS we are in effect keeping the overlap data on the input queue. The selection of the method that is used for storing overlap data is dependent upon the capability of the scheduling algorithm used when the DF graph is embedded on the DSPs. Very often the use of NEPS rather than a constant threshold value and dynamically changing the values of the NEPS forces the embedded graph to actually support dynamic scheduling of the node. This increases the run time overhead. The first method which locally stores the overlap data, requires additional work to specialize the primitive for the overlap operation. From a strictly DF graph point of view, the use of NEPS explicitly shows what is being attempted in the graph rather than the technique where we use constant threshold and make primitive change to accomplish the overlap.

The inclusion of the Offset parameter in the NEPS gives the user additional flexibility to have the read amount offset from the beginning of the input data block. In this case the primitive could have been altered to read in the entire amount of O+R and then select only the R portion. This is again an effective conceptual capability that aids in the construction of the DF graph.

The last item associated with the threshold is a variable threshold. In the case of the FIR filter we have seen that there is a startup transient when the input data stream is first read. Depending upon the implementation, a

change in the threshold value could occur at the beginning or at the end segment of the data set. A simple solution when using NEPS would be to preload the queue with data and thus in effect avoiding the ending transient. However, when the next input stream starts we must re-initialize the input queue and thus start the graph again. When the graph must be set up for continuous data operation of input streams of data of varying total length, the NEPS must be adjusted on a segment by segment basis and we will also need a state counter to signal when these changes in the NEPS must be made.

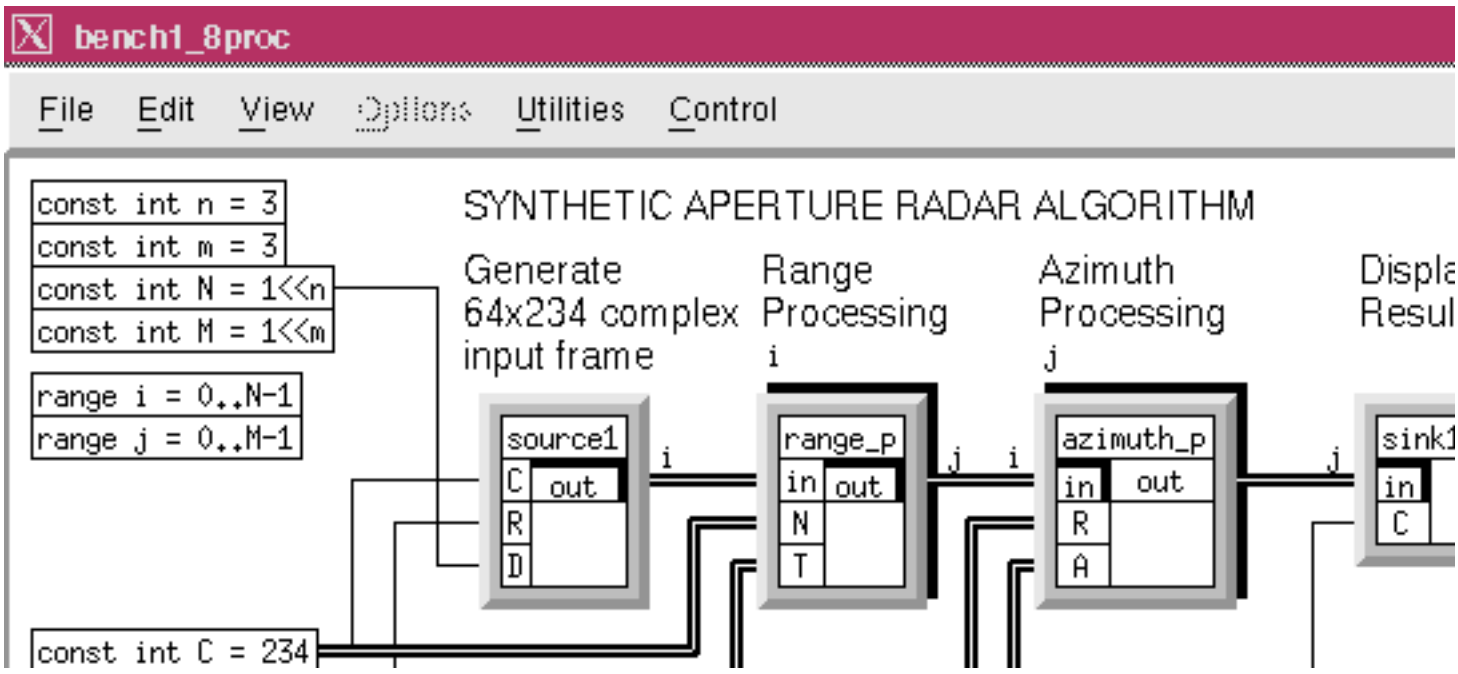
RASSP Data Flow Graph Design Application Note

8.0 Selected Graph

In the previous section we have discussed portions of the graphs from the two implementation projects in order to illustrate some of the points associated with the design of a general DF graph. In this section we will review the complete graph for each problem and show how the final graph was implemented. The first graph, created for a SAR algorithm, illustrates of a class of projects that can be implemented using data flow graphs with out many extensions. The second example, shows how the data flow graph can be extended to handle a data processing algorithm. This project implemented a set of algorithms that are used to processes a Sonar pulse, produces clusters of data and then aggregates these clusters to produce a track. The figures shown throughout this section were generated using the GEDAE™ tool.

8.1 SAR Graph

The SAR problem (see SAR Case Study) has been implemented using DF methodology. The SAR processing is divided up into two set of processes, the range and azimuth processing. The range processing is performed a column at a time and the number of columns processes by one node depend on the number of processors available. The number of range processing nodes are now grouped into a family whose size is selected by the number of processors available to perform the range processing. The azimuth or row processing is similarly divided into a family of nodes whose size is the number of azimuth processors used. This family notation is illustrated in Figure 8 - 1 through the shadowed boxes for the range and azimuth processing functions. This graph also shows the source and sink boxes. The source node provides a frame of data to the family of range processing and the sink displays the processed image. The input data to the range processing arrives at a fixed rate. Within the Source box there is a Split node which is used to distribute the data for the range processing to the various processors.



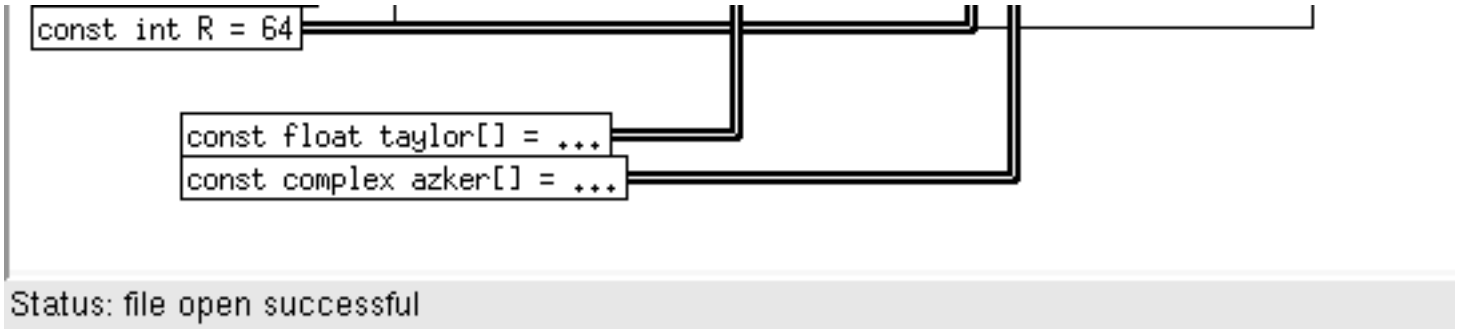


Figure 8 - 1: Top-Level Graph for the SAR Algorithm

The subgraphs for the SAR range processing are shown in Figure 8 - 2.

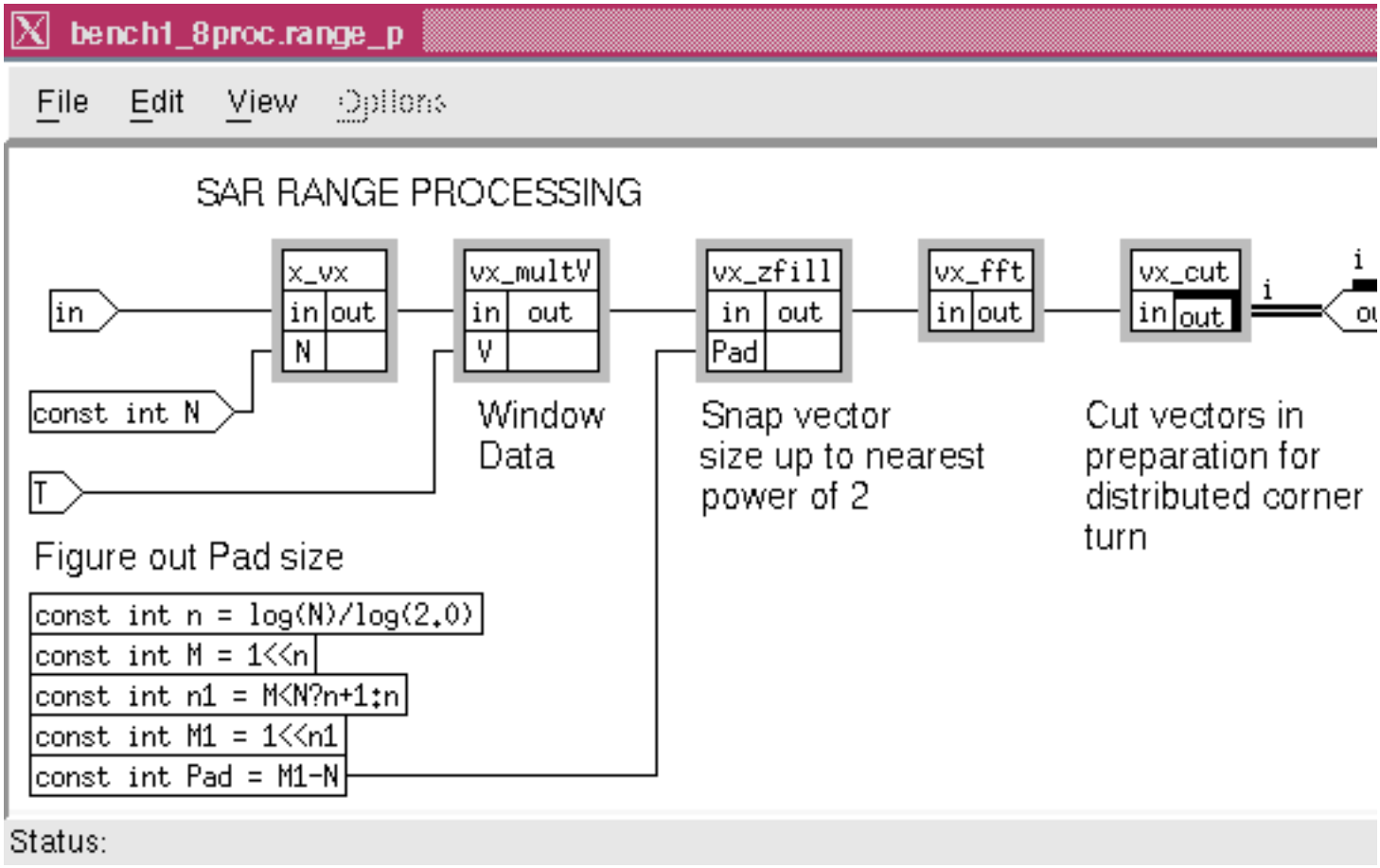


Figure 8 - 2: SAR Range Processing Graph

This shows that the data is first separated into Vectors by nodes x_vx and vx_multV and then filled by node vx_xfill to the FFT size. The output of the FFT computations is then segmented into vectors whose number correspond to the number of azimuth processors available. This operation is performed by the vx_cut node. Examination of the SAR algorithm shows that the corner turning processing of the data can be made efficient. Initially this processing was implemented by first collecting the data back into a stream and then corner turning the data after the whole family of range processing was completed. This process was changed so that as each column of range processing node is complete we distribute the data into a family of buffer queues. The number of queues match the number of azimuth processors. When the complete data set, consisting of all the columns that are needed for one of the families of the azimuth node and its corresponding queue is filled, the azimuth processing can begin. This procedure implements a distributed corner turning of the data. This corner

turning implementation is readily expressed in a graphical form by using of index families of nodes and queues. The overall DF graph in Figure 8 - 1 shows that the range and azimuth processing which are families of dimension i and j respectively are connected by a family interconnect labeled i==j. This is the short hand notation used in GEDAE™ for a routing box. The routing box shown in Figure 8 - 3 is the expansion of the routing symbol on the main graph. Close examination of this figure shows how the i range j computation is routed to the j azimuth i computation. In effect, this operation block transposes the i row by j column range processing output to the j row and i column of the azimuth input. The data in the individual blocks is then transposed by including a transpose node operation in the azimuth processing graph before the azimuth processing takes place. This is shown in Figure 8 - 4 by the mx_transpose node.

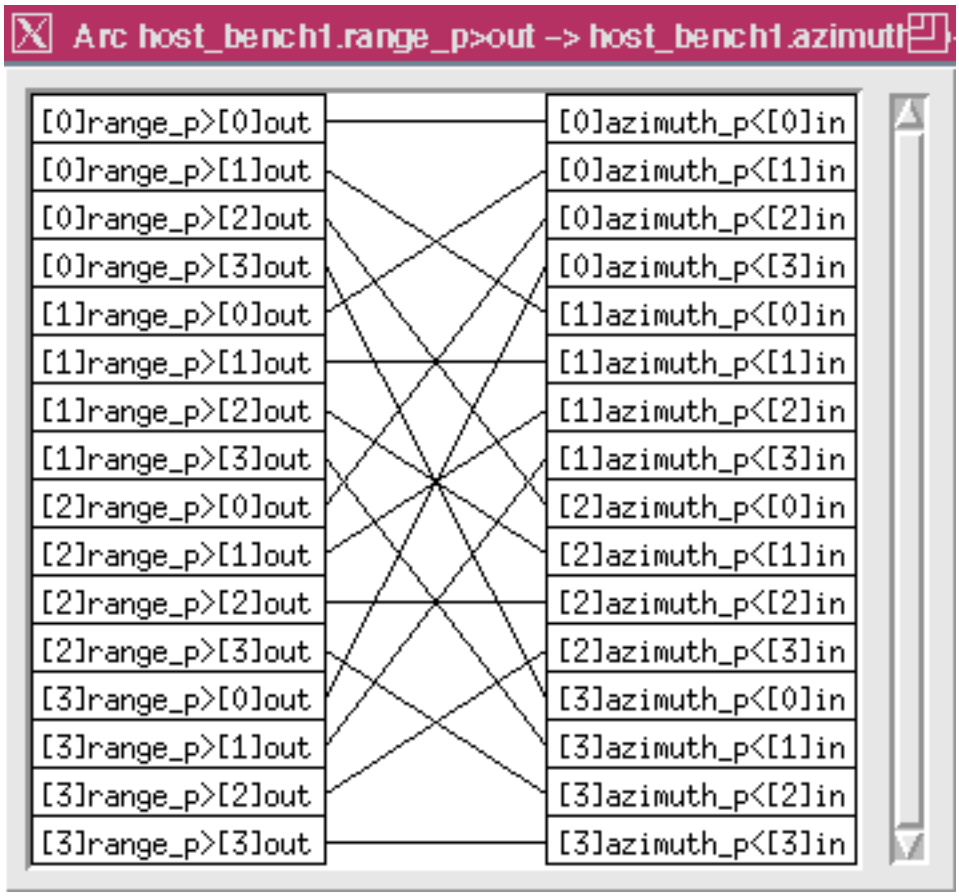
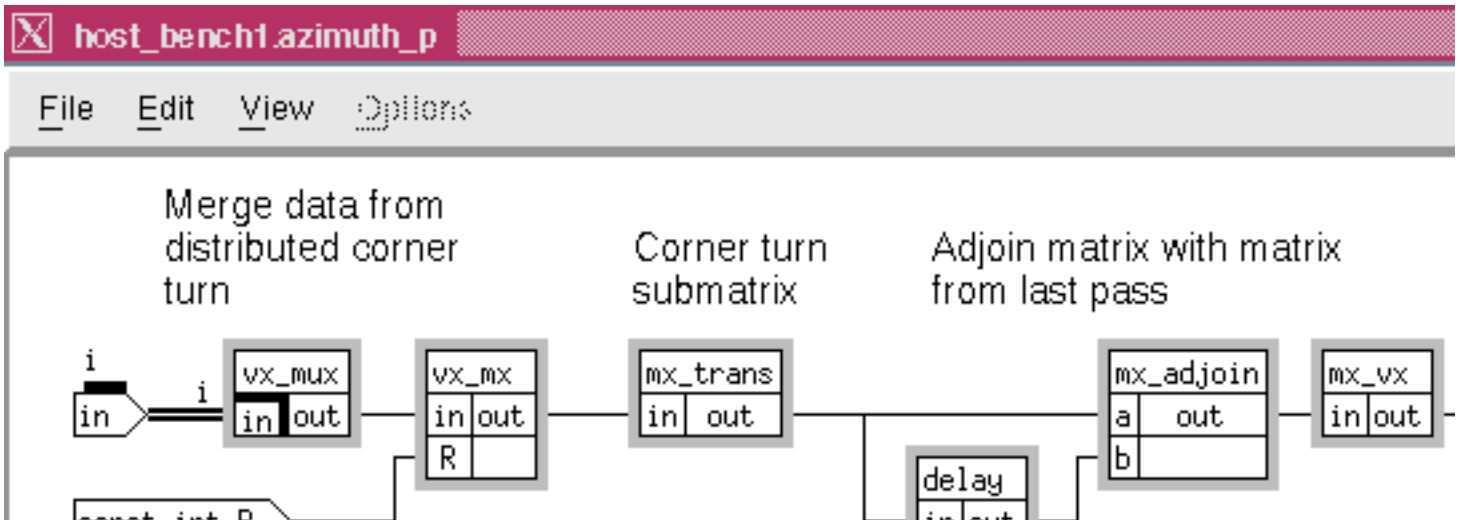


Figure 8 - 3: GEDAE™ Routing Box



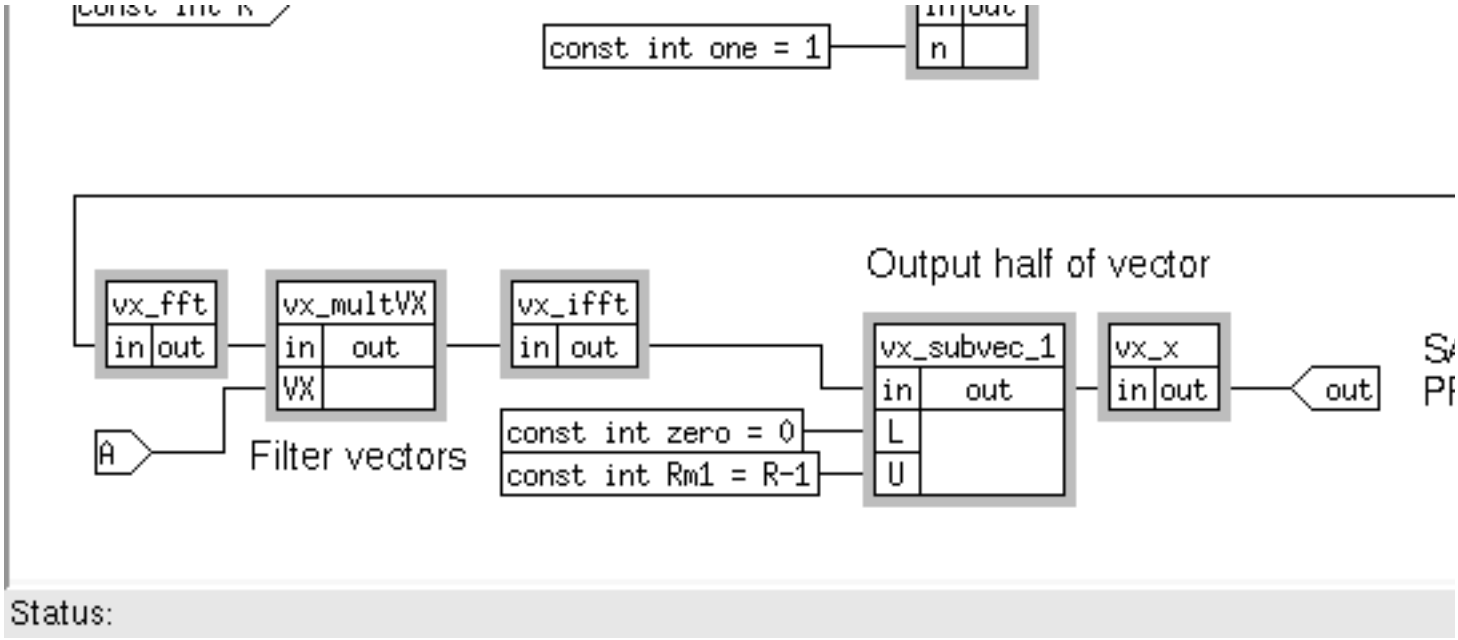
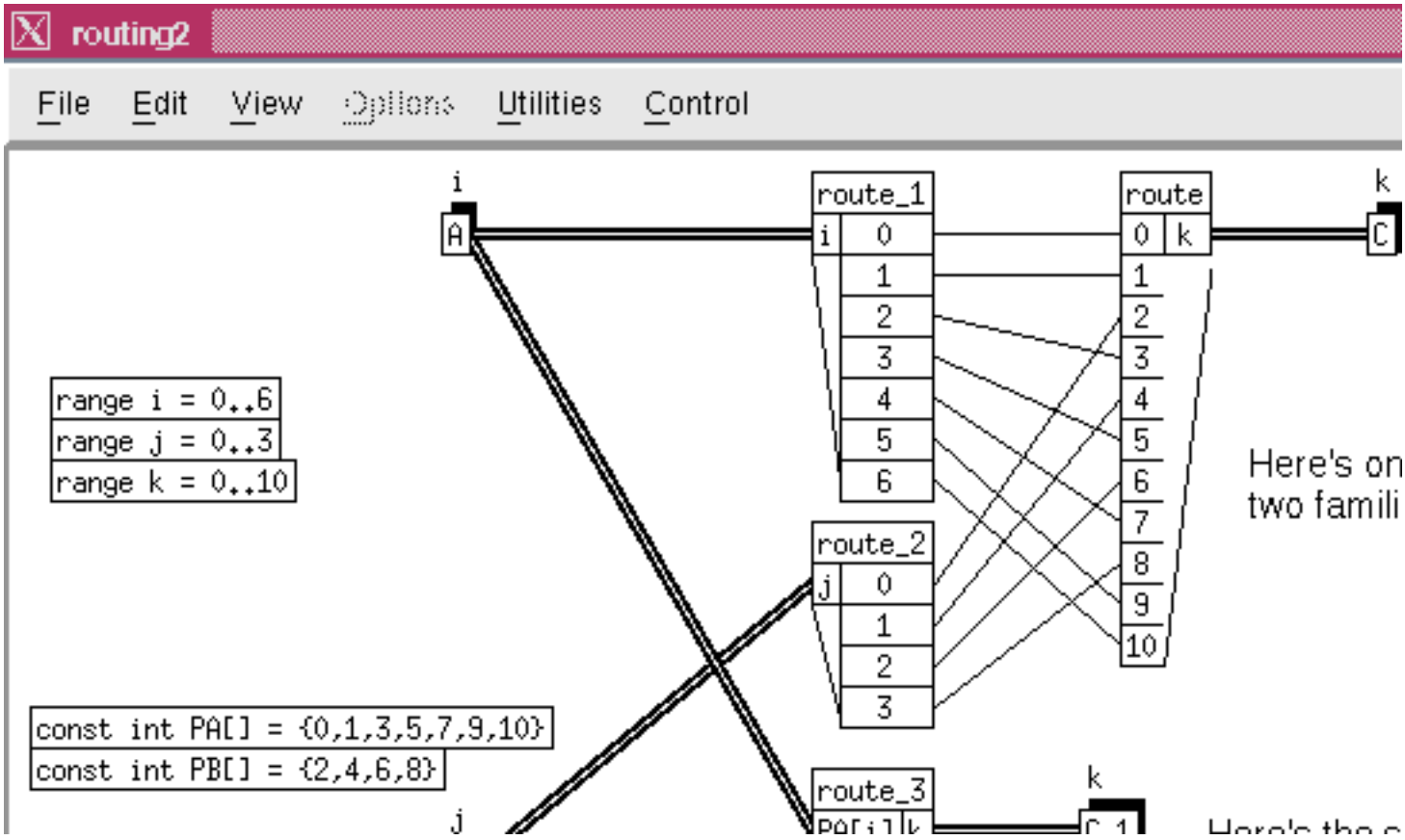


Figure 8 - 4: SAR Azimuth Processing Graph

The routing operation can be generalized and we may use graphical expression of tabular information in addition to using indices as described above. Figure 8 - 5 shows a comparison of the graphical and the tabular methods. The tabular method forms a table with the input index sequential and the output index is expressed as a function with the values as shown in the table. The indexing method used for the SAR example was made use of a simple index interchange. In other situations a more complex expression could be needed to express the input and output indices required and this is supported by the DF graph notation.



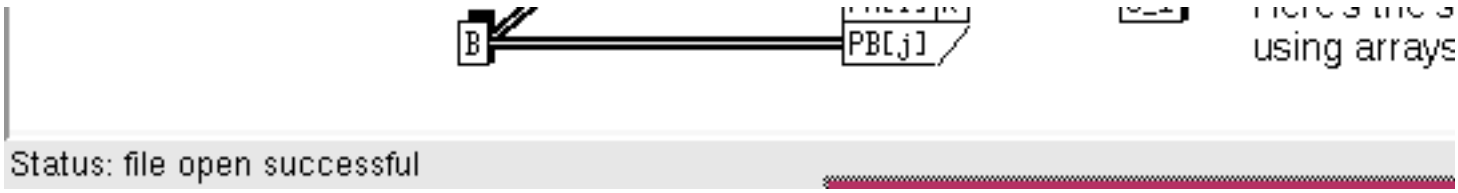


Figure 8 - 5: Graphical and Tabular Methods Comparison

8.2 Cluster Processing

The Echo Tracker Classifier (ETC) problem (see ETC4ALFS on COTS Hardware Case Study) has been implemented using Data Flow. The overall DF graph consists of two major subgraphs, the front end and back end. The ETC graph has been set up hierarchically with the top level graph shown in Figure 8 - 6

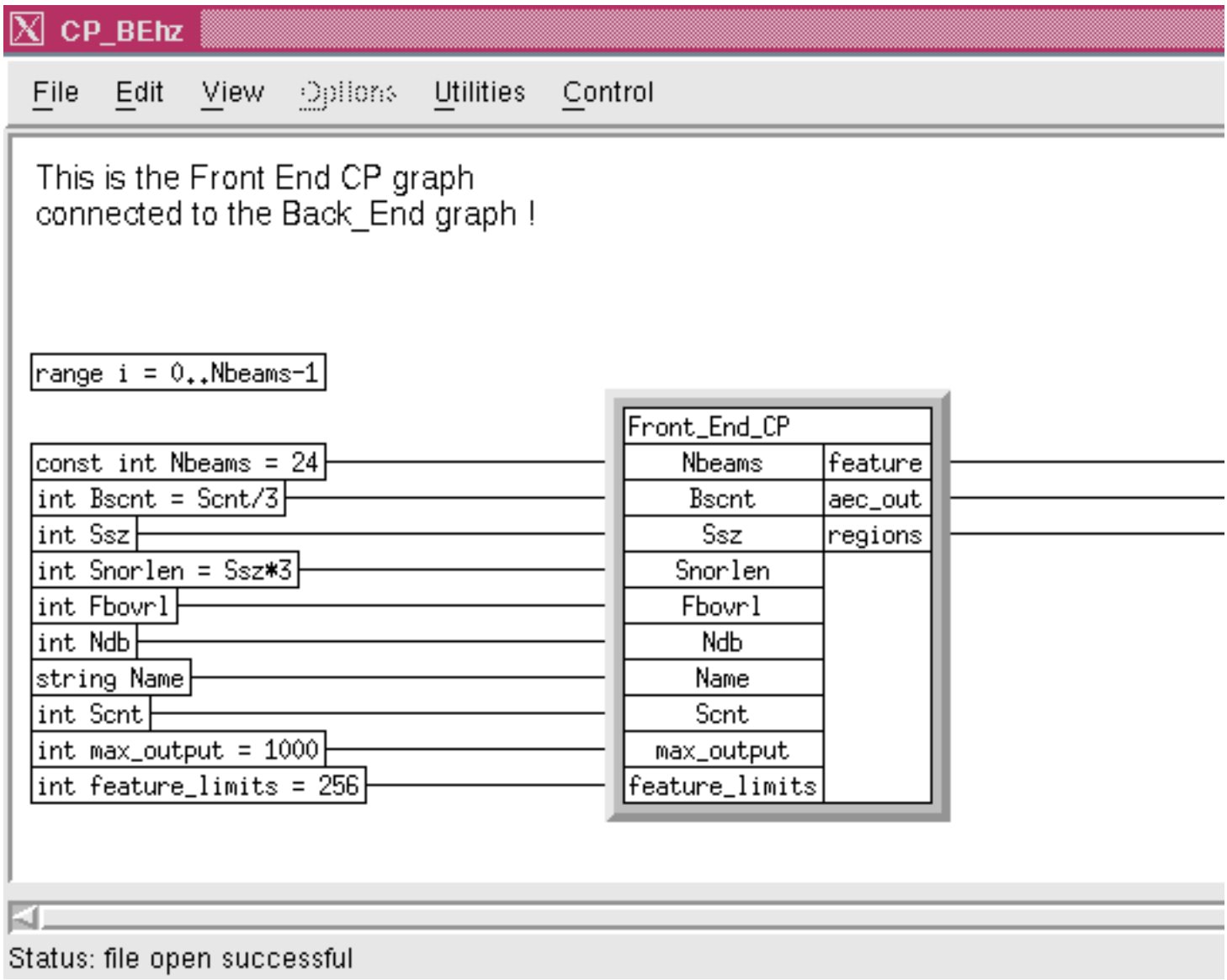
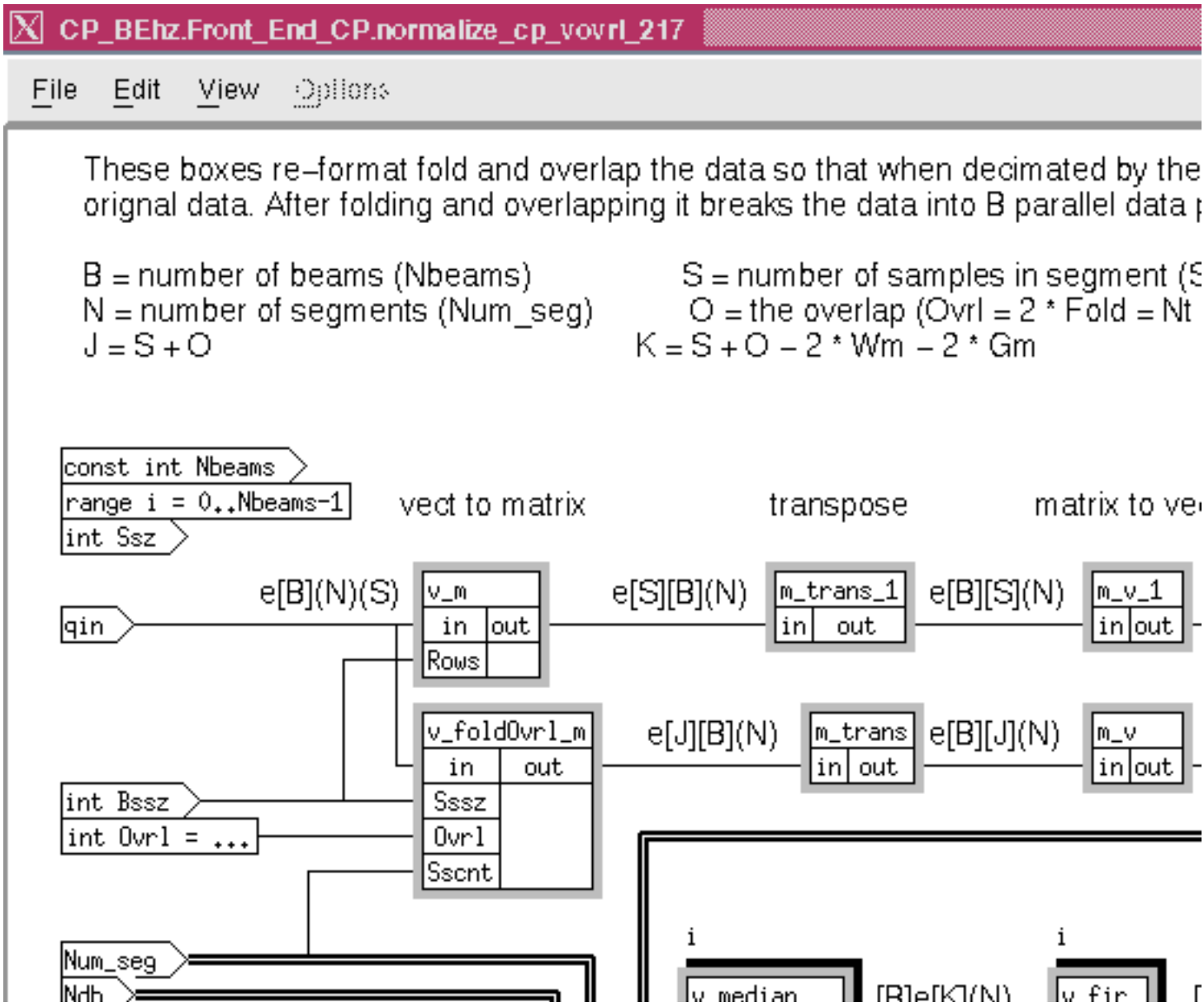


Figure 8 - 6: Top-Level Graph for the ETC Algorithm

The first sub-graph, called the front end, is made up of a series of filtering operations upon the data called normalization and fine bearing estimation. These computations are terminated by a node that produces a data

set that represents the clusters. Since the number of clusters produced for a given length of input data is dynamic, the V array is used to capture the clusters output from the CFE node. Each cluster consists of 54 data items and there could be from 0 to n clusters per slice. The number of clusters per slice depend upon the number of data items that exceed the threshold at the input of the CFE node.

Figure 8 - 7 shows the normalization or NORM processing. All the nodes shown are primitive nodes. The data is brought in as qin vector and converted to a set of vectors of S sample length and there are B (Nbeams) of these vectors per segment (slice) and the are N slices per ping. In this case S = 768 and B = 24. This rearrangement and movement of the input data set into 24 parallel 768 sets of data is represented by the upper path consisting of m_trans_1, m_v_1 and v_demux_1. A parallel path generates the same data sets but augments the data by an overlap amount due to the action of the folder, v_fold0vr1_m. The folder operates differently for each slice therefore the state counter repeat count of Num_seg is used to reset the counter at the end of each ping. The filtering operations of v_median and v_fir perform the NORM filtering and the upper data stream is divided by the results of the filtering operation to perform the normalization. In this graph the nodes that process the filter operations output vectors of length K which depends upon the parameters Wm, Gm and O. The processing is completed by merging that data in a v_mux_l operation. The vector length is also divided by 3 to 256 to have a stream of 256 x 24 set of data per slice as required by the next portion of the graph.



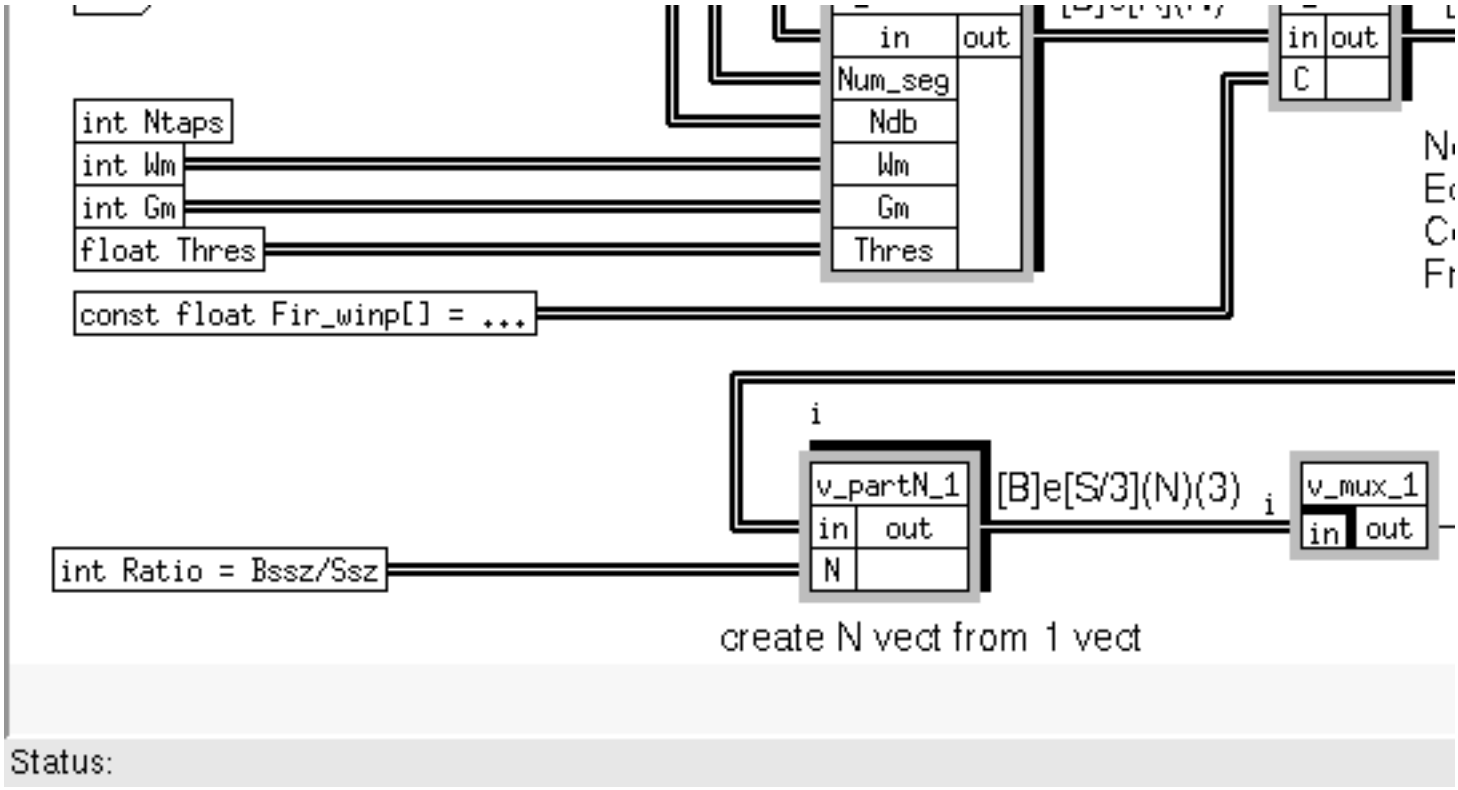
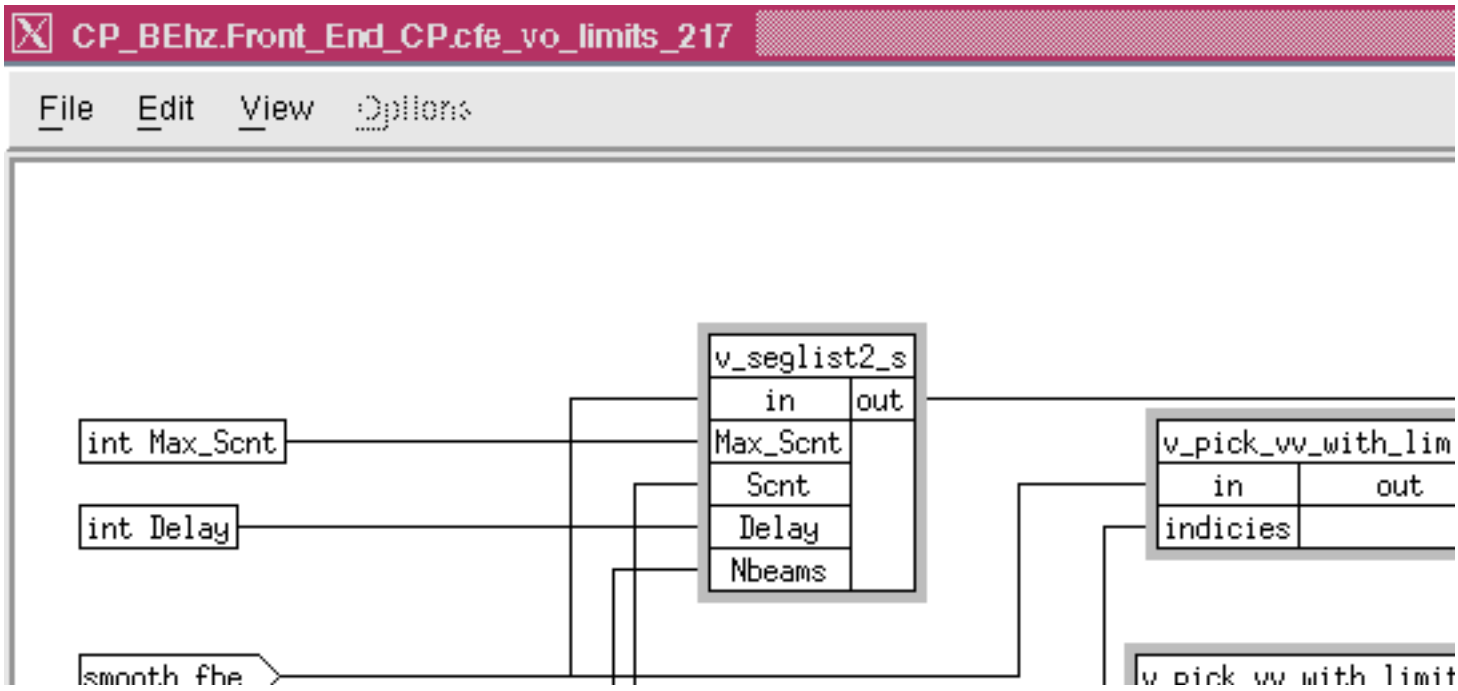


Figure 8 - 7: Normalization Processing

The CFE graph decomposes the vector data into V arrays and outputs a set of clusters of V arrays. These operations are shown in Figure 8 - 8. The input is one vector per slice and a V array is output by v_thres_vv_with_limits. The latter node thresholds the data at a level set by the parameter Tu_tlp. Therefore, the number of the values that exceeded the threshold depend on the data . The V array was used to contain the output of the threshold node Similar operations were performed by the v_pick_vv_with limits nodes. The last processing node was the cfelimt_cp_with_limits which inputs the V arrays and outputs a set of clusters in a V array per slice



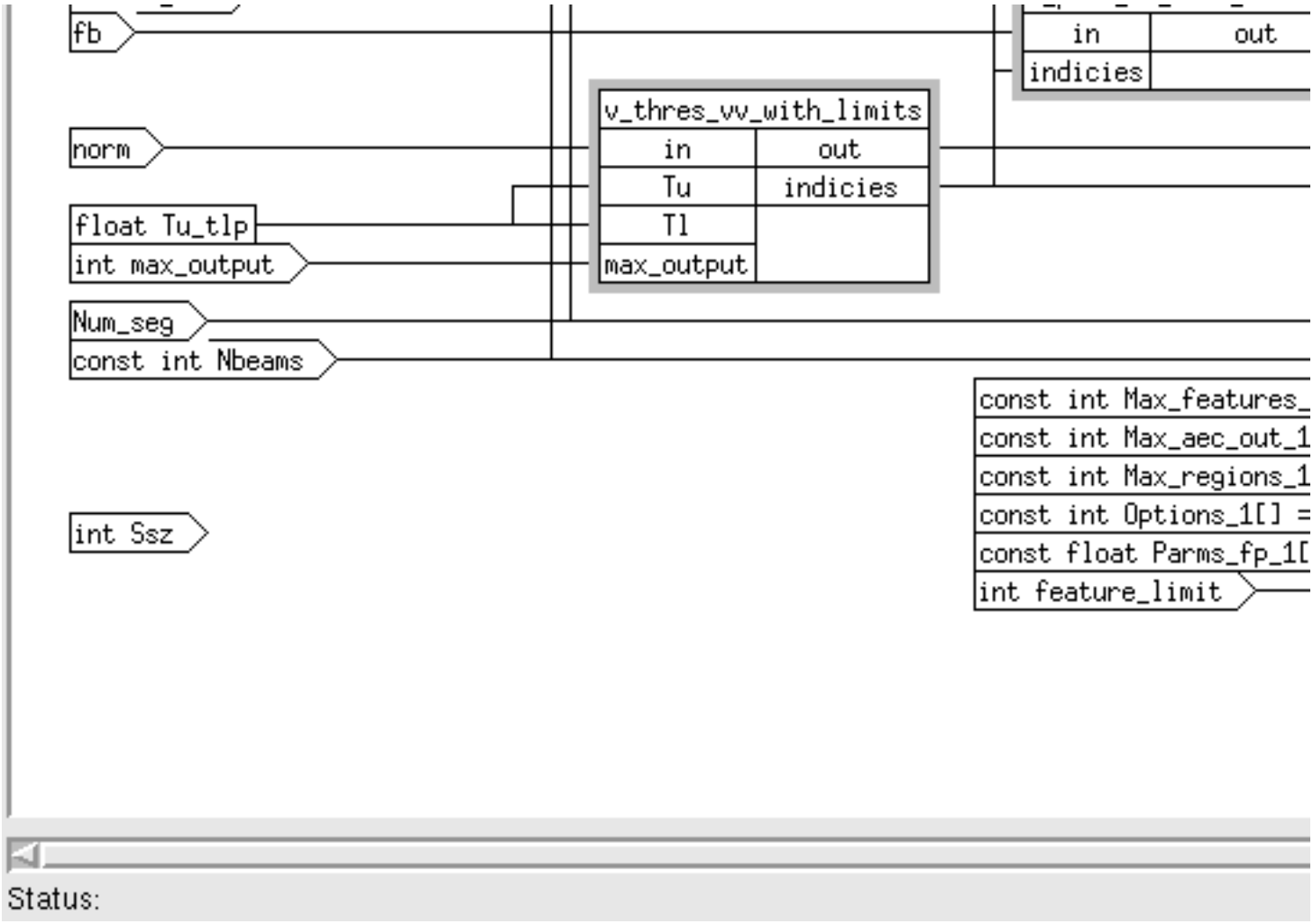
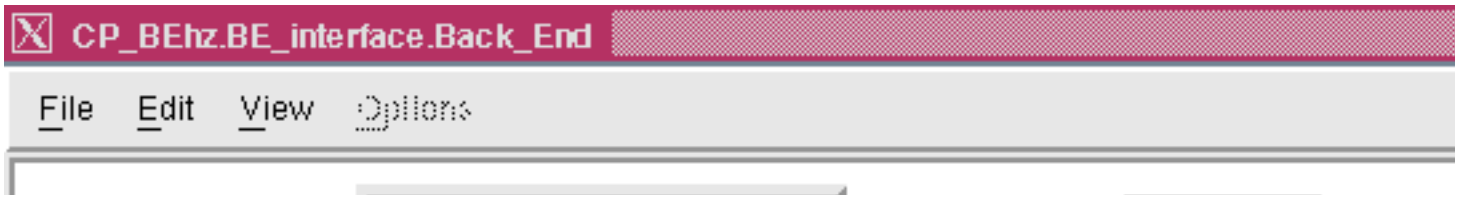
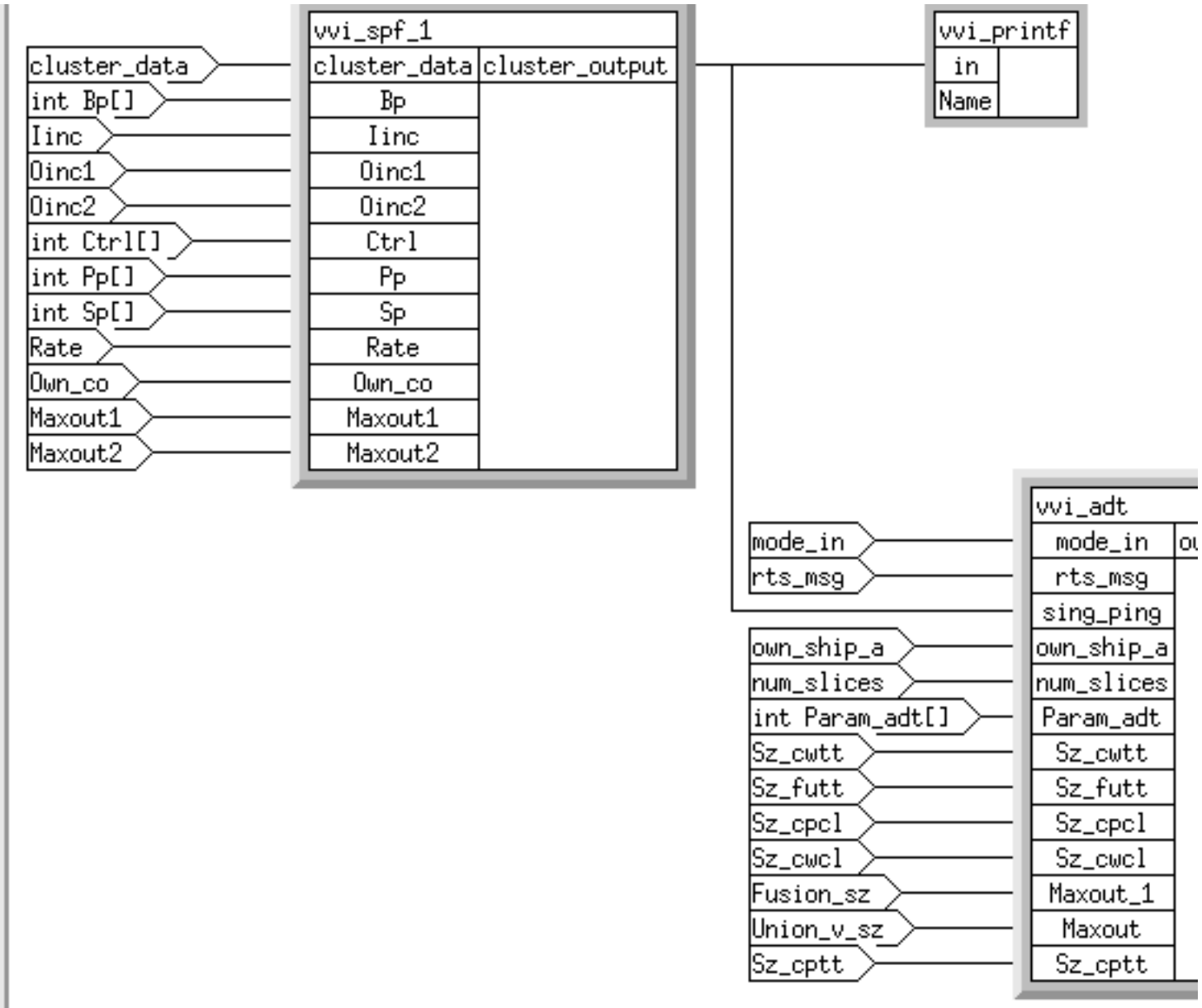


Figure 8 - 8: Vector Data Decomposed into V arrays

The processing called the back end, begins with the output of CFE and contains the processing of SPF, ADT and MPC. The cluster output of CFE is processed into tracks which are in the same form as the clusters data set. Each track is a data set of length 226 items with 0 to n tracks per slice. Again V arrays are used to transfer the data between the ADT computation node and MPC. The final output is a track report and there is a dynamic amount of track reports per slice depending on the number of valid tracks and a V array is used. The back end sub-graph is shown in Figure 8 - 9 and has a simpler overall graph topology since the graph uses V arrays for the inter element (SPF, ADT and MPC) transfers. At the lower level of the backend hierarchy is the individual sub-graphs for the three elements. Figure 8 - 10 shows the sub-graph for the ADT processing. This graph consists of two nodes. The Load-data node input is the clusters data in the form of a V array and other auxiliary data sets in the form of a vector or V arrays. One set is received for each slice. Functionally, the Load-data node puts all of the input data into global data structures and then passes a token to initiate the track processing. The expedient way to convert this code for DF operation was to mimic the way the original workstation version of the code operated. This DF architecture was used this way for the ADT process because the entire process was able to be run on one processor. There was no need to parallelize the operations in ADT in order to speed up its operation.




vvi_spf_1
cluster_data
cluster_output
cluster_data
int Bp[]
Iinc
Oinc1
Oinc2
int Ctrl[]
int Pp[]
int Sp[]
Rate
Own_co
Maxout1
Maxout2
Bp
Iinc
Oinc1
Oinc2
Ctrl
Pp
Sp
Rate
Own_co
Maxout1
Maxout2
vvi_printf
in
Name
vvi_adt
mode_in
rts_msg
sing_ping
own_ship_a
num_slices
int Param_adt[]
Sz_cwtt
Sz_futt
Sz_cpcl
Sz_cwcl
Fusion_sz
Union_v_sz
Sz_cptt
mode_in
rts_msg
sing_ping
own_ship_a
num_slices
Param_adt
Sz_cwtt
Sz_futt
Sz_cpcl
Sz_cwcl
Maxout_1
Maxout
Sz_cptt

Status:

Figure 8 - 9: Back-end Processing Sub-Graph

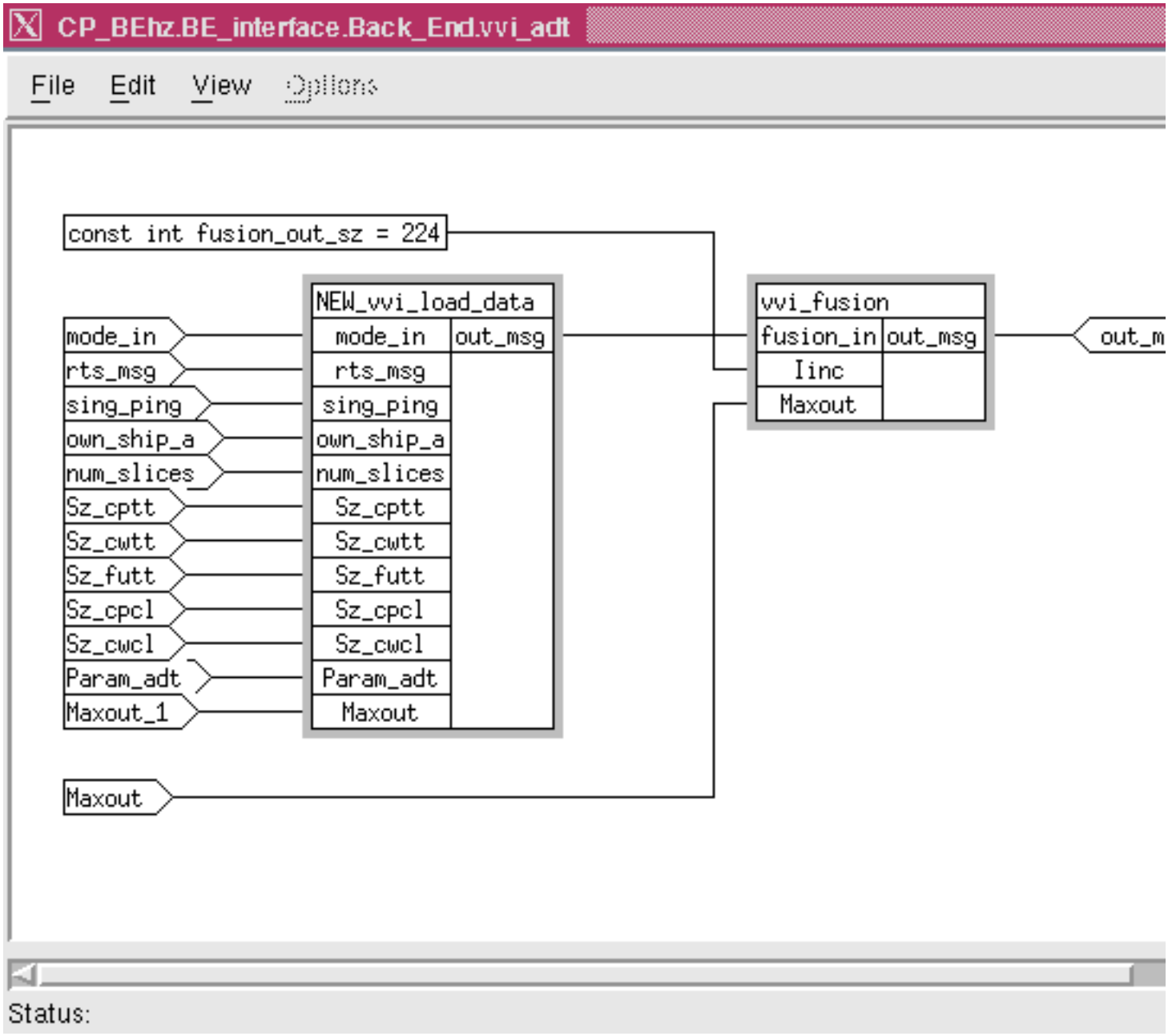


Figure 8 - 10: ADT Processing Sub-Graph

Next Up Previous Contents

Next: 9 Conclusions **Up:** Appnotes Index **Previous:**7 Primitive Construction

RASSP Data Flow Graph Design Application Note

9.0 Conclusions

The current state of the art of the graphical signal processing code design has progressed from a concept to a practical methodology that includes procedures as well as tools to implement the complete design process. However, it is still incumbent upon the intelligent designer to use the specialized Data Flow graph technology to create graphs that result in code that is efficient and uses minimum memory for the target DSPs. This code now meets the throughput and latency requirements in addition to functionally implementing the signal processing algorithm. Automatic parallelization, memory management and processor allocation tools have not yet been developed to a mature state so they can be used routinely to perform this task. These operations are still in the hands of the designer and in this paper we presented techniques and examples so that we may accomplish these operations for a DF graph.

 Dennis Basara

RASSP Data Flow Graph Design Application Note

10.0 References

Pridmore, J., Schaming, W. " RASSP Methodology Overview " Processing of the First Annual RASSP Conference, pg 71 - 86, Arlington, VA, August 15 - 18, 1994. [PRIDMORE_94]

Hillson, Roger, " Support Tools for the Processing Graph Method " Proceedings of the Fifth International Conference on Signal Processing Applications and Technology (ICSPAT 94), Dallas, TX, 18 - 21, Octover 1994. [Reference Not Available]

Mielke, R., Soughton, J., et. al., " Algorthm to Architecture Mapping Model (ATAMM) Multicomputer Operating System Functional Specification, " NASA CR 4339, November 1990. [Reference Not Available]

Evans, Brian, Kamas, Alan, Lee Edward A, " Design and Simulation of Heterogeneous System Using Ptolemy ", Proceedings of the First Annual RASSP Conference, pg 97 - 105, Arlington, VA, August 15 - 18, 1994. [EVANS_94]

GEDAE™ Web Site

Processing Graph Method Specification: Version 1.0, Navy Standard Signal Processing Program, (PMS - 500), December 1987
[To obtain these documents and other information regarding PGM refer to the PGMT home page presented by the Processing System Section, Advanced Infomation Technology Branch at the Naval Research Laboratory, Washington, D.C. www.ait.nrl.navy.mil/pgmt/pgm2.html or contact Mr. David Kaplan at phone number (202) 404-7338 or e-mail kaplan@ait.nrl.navy.mil]

Processing Graph Method Tutorial, Navy Standard Signal Processing Program, (PMS - 500), January 1990.
[To obtain these documents and other information regarding PGM refer to the PGMT home page presented by the Processing System Section, Advanced Infomation Technology Branch at the Naval Research Laboratory, Washington, D.C. www.ait.nrl.navy.mil/pgmt/pgm2.html or contact Mr. David Kaplan at phone number (202) 404-7338 or e-mail kaplan@ait.nrl.navy.mil]

Zuerndofer, B., Shaw, G. A., " SAR Processing for RASSP Application, " Proceedings of the First Annual RASSP Conference, pg 71 - 86, Arlington, VA, 1994. [ZUERNDOFER_94]

10.1 Case Studies

Synthetic Aperture Radar (SAR)
ETC4ALFS on COTS Processors
