JOINT ROYAL MELBOURNE INSTITUTE OF TECHNOLOGY AND
COMMONWEALTH SCIENTIFIC AND INDUSTRIAL RESEARCH ORGANISATION
PARALLEL SYSTEMS ARCHITECTURE PROJECT

The RMIT Data Flow Computer
The Architecture

TR 112 061 R

D. Abramson †
G.K. Egan ‡
M. Rawling ‡
A. Young ‡

† Division of Information Technology
C.S.I.R.O.
‡ c/o Department of Communication and Electronic Engineering
Royal Melbourne Institute of Technology
P.O. Box 2476V
Melbourne 3001
Australia.

Version 1.0 January 1987

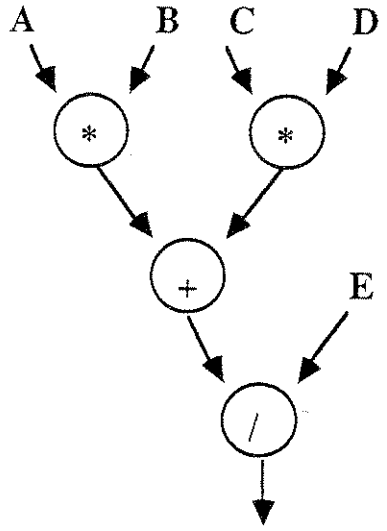
ABSTRACT:

This paper provides an overview of a particular data-flow architecture which is being developed and implemented at the Royal Melbourne Institute of Technology in Australia. The project began at Manchester University, UK, where a prototype machine was built in 1976. The original configuration was built from four small microprocessor boards connected via an exchange. The current system consists of a number of processing elements built from 68000 processors connected via a high speed shuffle exchange communication network.

The basic concepts are described as well as the current hardware emulation facility. Some experimental results are presented which predict the performance of the architecture.

1. INTRODUCTION

Data-flow machines are multiprocessors which execute parallel program graphs rather than sequential programs. The order of execution of the nodes in the graph (or instructions) is determined by the availability of their operands rather than the strict sequencing of instructions in a von Neumann machine. Consequently the program statements are executed in a non-deterministic manner, and concurrency is obtained if more than one node executes at the same time. Figure 1 shows a sample data-flow graph for an arithmetic expression and figure 2 shown a model for the hardware required to execute such data-flow programs. In this hardware, the program graph is distributed to the processing elements so that the computation of A*B can proceed at the same time as C*D. The results of a computation are sent from the processor that holds the source node to the processor that holds the destination node. When the result arrives at the destination processor it waits in the matching unit until all of the operands for the destination node are ready before the next result is computed. Thus the addition is performed when both A*B and C*D have been computed and division is computed once the addition has completed.



F = (A * B + C * D)/E

Figure 1 - A data-flow graph.

This paper provides an overview of a particular data-flow architecture which is being developed and implemented at the Royal Melbourne Institute of Technology in Australia. The project began at Manchester University, UK, where a prototype machine was built in 1976. The original configuration was built from four small microprocessor boards connected via an exchange. The current system consists of a number of processing elements built from 68000 processors connected via a high speed shuffle exchange communication network.

A good review of existing dataflow systems can be found in [14]. This paper highlights the main features of the RMIT architecture. The RMIT data-flow machine is characterised by the following attributes:

- The architecture combines a static execution model with a dynamic model.
- Node-functions are weakly typed.
- Tokens are strongly typed and of variable length.
- The system supports shared sub-graphs which facilitate multiple recursions.
- Graphs are partitioned and the partitions are allocated statically to processing-elements.
- Storage nodes are provided to allow the graph to retain 'semi-permanent' information.
- An Object Store is provided for large structures or persistant objects (i.e.files).
- Exceptions can be handled using a special token type
- Streams are supported in sufficient generality to allow streams of streams.
- Input-output is accomplished using pre-defined nodes.

- Nodes may send a token to many destinations either by building a tree of special duplicate nodes, or emitting the token many times with different destinations.

These points will be described in further detail in the remaining sections of the paper.

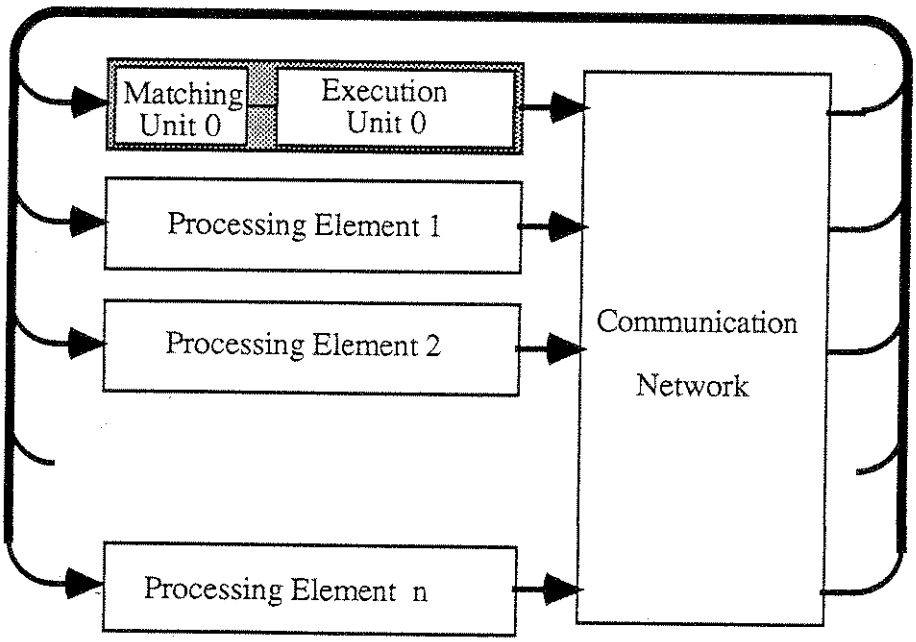


Figure 2 - Hardware model for data-flow machine.

2. STATIC AND DYNAMIC ARCHITECTURES

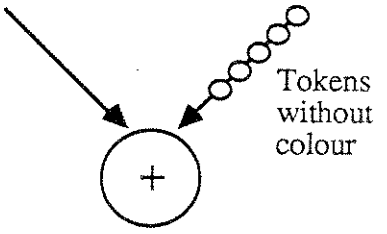
There are currently two main classifications for data-flow architectures, *static* and *dynamic* . The static scheme was first proposed by Dennis [1, 2, 3], and has been used by various research architectures such as TI's DDP [4] and the LAU architecture [5]. The dynamic scheme is used in Arvind's research group at MIT [6,7], at Manchester University [8], the DDM architecture [9] and the EDFG system [10]. In the static data-flow model only *one* token (or instruction operand) is allowed on a program arc at any time. In the dynamic model many tokens are allowed on arcs, and their order is determined by special *tag* fields. A good overview of these architectures can be found in [14].

Static architectures are easier to implement than dynamic ones because the matching of operands can be performed with a simple table lookup. Each input to a two input node either has no tokens or one pending token. When the second token arrives the node is *fired* and the tokens are consumed. The main disadvantage of the static model is that the concurrency in a graph is determined by the width of the graph and the amount of data pipelining involved. These two parameters are low in some computation intensive tasks. They are however quite high in algorithms which process sets of data. Thus static architectures can provide very good performance for particular algorithms.

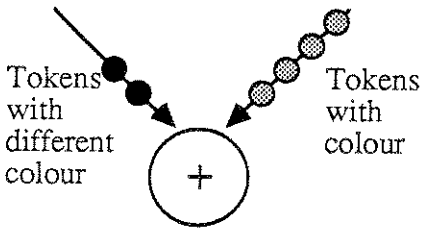
In a dynamic architecture many tokens may be pending on the input of a node, and they are only consumed when one with the correct *colour* or *tag* value arrives on the other input. Thus the matching process is more complex than for the static architecture. The network traffic is higher because the colour information must be carried with the tokens. Also special colouring nodes must be placed in graphs, which makes programs run slower. The main advantage of the dynamic model is that a particular node may consume more than one token pair simultaneously, thus increasing the concurrency in the data-flow graph. A disadvantage is that algorithms which cannot use this feature must inherit the cost of tagging the tokens.

The RMIT architecture provides a hybrid scheme in which a modified form of the static model coexists with the tagged dynamic scheme. Tokens may either be tagged or untagged. If there is no

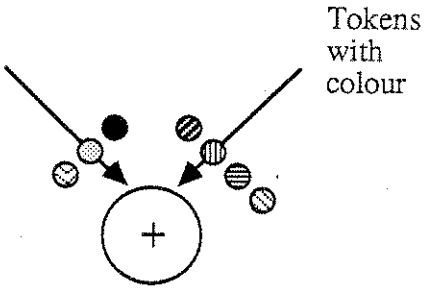
tag present, or there is already a token with the same tag value present on an input, the tokens are queued on the input until a partner arrives. A separate queue is maintained for each different tag value. One queue is used for all those tokens without a tag. When a partner does arrive a token is removed from the head of the appropriate queue. In this way tokens are sequenced to a particular node. If all tokens for a node do not have tag values then there is no possibility of more than one instance of the node exisiting. However, if the tag field is present then more than one instance of the node may exist. In this way the concurrency may be increased. The advantage of this hybrid arrangement is that the cost of tagging the tokens is not present when it is not required. Because the static mode of operation does not demand that each arc only hold one token the potential concurrency is higher than in Dennis' static model. Figure 3 shows how the hybrid combines the static queued model and the dynamic model. A more complete discussion of the advantages of the hybrid architecture, together with an implementation strategy, can be found in [22]. Later in the paper, we present some simulation results which demonstrate these advantages.



Static Queued Model - tokens without colour



Static Queued Model - tokens with different colours on each input



Dynamic Model - tokens with different colours on same input

In the hybrid architecture all of these systems can co-exist

Figure 3 - the hybrid model

3. GRAPHS AND SUBGRAPHS

Data-flow program *graphs* can be constructed using one or more *subgraphs*. Subgraphs may be declared separately from the main program graph and linked together to form one large program. Subgraphs may be *called* with paramaters and can *return* results. Subgraphs aid the constuction and maintenance of large data-flow programs, just as subroutines do for conventional von Neumann programs. They also help reduce object code size because common code structures do not need to be duplicated.

Two types of subgraphs can be used, *normal subgraphs* and *shared subgraphs*. When a normal subgraph is invoked a copy of the code is placed into the program graph. The input parameters are linked to the formal parameters of the subgraph, and the nodes which generate return data are linked to the receiving nodes. Normal subgraphs do not help reduce object code size because each invocation causes a new copy to be included in the final program. However, because the source

code for the subgraph appears only once, they aid program construction and maintenance. There is no extra cost incurred when a normal subgraph is called because there is no need to maintain return linkage information. In general, normal subgraphs cannot be recursive because the number of calls must be known at compile time.

When a shared subgraph is invoked only one copy of the graph is included in the program. Parameters and return linkage information are passed using *argument and return entry* nodes. Because more than one invocation of the same shared subgraph may be active at a time, it is necessary to tag, or colour, the tokens as they enter the subgraph to distinguish the different data sets. This tagging information is used by the matching units to determine whether a node has sufficient operands to execute. Thus, operands must have the same tag values before they can match and be consumed by a node.

The tag value is composed of two portions, a processing element identifier and a within element value. Each processing element is responsible for generating its own tag values. Thus, when a value is created the processing element number is placed in the most significant bits of the tag value. In this way there is no contention for the creation of colours in the multiprocessor. The unique tag value is simply an incremented version of the last unique value which was generated. A separate counter is maintained for each shared subgraph in the program graph. When a subgraph is called a unique value may be created by the processing element performing the call. Using this scheme tag values cannot easily be re-used. Thus the tag field is large enough to 'guarantee' that a given computation will not exhaust the available tag address space.

4. NODES AND TOKENS

All data-flow programs are composed of *nodes*. Nodes are the equivalent of the machine instructions in a conventional von Neumann processor. In the RMIT machine nodes are either *monadic*, or single operand, or *diadic*, or two operand. Because monadic nodes do not require matching, the node is enabled when an operand arrives. Diadic nodes require two operands before they can execute, and the pending tokens are held in a special matching unit. Nodes are composed of two separate sections, a *matching function* and an *execution function*. A matching function controls the input data to a node and forwards the tokens to an execution function. An execution function processes the data and emits the results. This general structure is shown in Figure 4. This approach allows any matching function to be combined with any execution function and provides great flexibility in node functions.

4.1 Matching Functions

A node's match class dictates how tokens arriving on the inputs to the node should be matched. In all diadic match classes the match is qualified by the colour of the arriving tokens. The variety of match classes is extensive and includes:

i) matching two simple tokens on alternate inputs. In this case a token with the same tag value must be present on each input of the node.

ii) matching elements of streams arriving on alternate inputs. In this case each token in the stream on one input is matched with a corresponding stream token on the other input. Steams are discussed further in section 7.

iii) matching a scalar on one input to all elements of a stream arriving on the other input. In this case the scalar is sent to one input of the execution function for each token of the stream present on the other input.

iv) **Cons, Get, Head, Rest, Append** and **Concatenation** operations on streams. **Cons** takes a token on one input and prepends it to the stream at the other input before sending the new stream to the execution function. **Get** removes the head token from the stream and forwards the head to one argument of the execution function and the remaining stream to the other argument. **Head** takes the first token from a stream and forwards it to the

execution function and absorbs the remaining stream. **Rest** absorbs the head of a stream and forwards the rest to the execution function. Append takes a token on one input and appends it to the steam at the other input before sending the new stream to the execution function. **Concatenate** combines two streams on the inputs to the matching function and sends them to the execution function.

(v) Initialization functions. These functions allow a graph to initialize various nodes. They are used in implementing shared subgraphs in which various graph sections must hold predefined tokens.

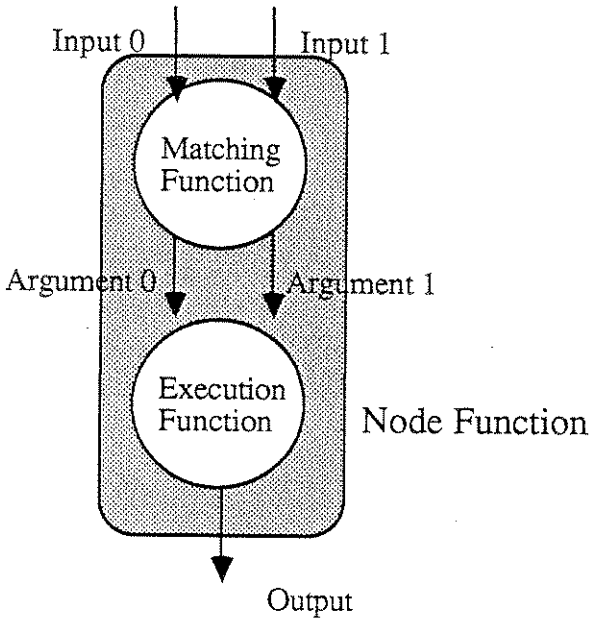


Figure 4 - Matching function and execution function.

In general a match function which produces two arguments for the associated execution function may be used with any diadic execution function and a function class that produces a single argument may be used with any monadic execution function. The result however may not however always be 'sensible'.

4.2 Execution Functions

The description of each node contains a function field, a match class, optional literal data fields and zero or more destination fields. The function field determines what operation the execution must perform. The literal data fields hold any constant data associated with the node itself. The destination fields indicate which nodes should receive the output data.

The node functions are divided into five main classes; computational, type coercion, structure manipulation, path control functions and colour functions.

The computational nodes include arithmetic nodes, logical and set data nodes, relational nodes and sequence nodes. The arithmetic nodes are used for calculating numeric results. Some of these are diadic, for example the **ADD** node, and some are monadic, for example **SQR**. The logical and set functions are used for manipulating boolean or set data. Example are the **AND** node and the **NOT** node. The relational nodes are used for comparing data values. They produce a boolean result which can then be used for input to switching nodes.

The type coercion functions are used for converting a data value of one type to another. Since most nodes perform automatic type converion if two inputs are of different type, these nodes are

not commonly required. A general conversion by example function is provided as well as explicit functions for the more frequent conversions. Examples of these functions are *ord* and *chr*.

The structure manipulation nodes are used for accessing structured data sets. These data sets either take the form of *streams* of data values, *vectors* of data values or *stored structures*. These data types will be discussed in more detail later in the paper. All nodes may accept either simple data values or streams of data values. There are some special nodes for manipulating streams, for example converting data values to a stream and back again. The vector nodes allows simple vectors of data values to be manipulated. The nodes for manipulating stored structures allow structures to be created, accessed and destroyed in a shared object storage unit. Stored objects will be discussed later in the paper.

The path control nodes are used for creating conditonal control loops in programs. They include nodes for passing data values depending on a control input, testing the type of data value present, duplicating data values and creating eager and lazy *if-then-else* structures. Some nodes are also available for manipulating the destination fields of data values.

The colour functions are involved in computing and setting the tag information on tokens. They allow a unique colour value to be created as well as the manipulating of already existing colour values. They are used extensively in calling shared subgraphs as discussed above.

4.3 Token Structure

Data is transmitted between nodes by tokens. Tokens are composed of a destination field, an optional colour or tag field, and one or more data fields. The destination field contains a user process number, a processing element number, an input point number and an object number. The user process number is used for distinguishing different user processes which are concurrently executing on the data flow machine. The processing element number indicates which processor holds the appropriate partition of the program graph. The input point number indicates which input of a diadic node should receive the token. If the token is being sent to a node, the object number determines which node within the processing element should receive the token. If the token is being sent to the object store the object number determines which object in the object store should receive the token.

The colour field is used for distinguishing different invocations of a section of code. Thus many tokens may be executing concurrently and independently in the one code section. The creation of colours is described above in Section 2. The data carried by a token may contain either *simple* datum, a *vector* of data or *compound* data. *Compound* data is used to carry several datum of different types in a single token. e.g records. Because the tokens in the RMIT machine are variable in length, any data size can be easily transmitted.

4.4 Token Replication

There are two methods of replicating tokens for distribution to multiple receiving nodes. First, the output of a node can be sent to a balanced tree of duplicate nodes. Each duplicate node produces two copies of the token colour and data fields. The depth of the tree is logarithmic with respect to the number of token copies. Because the duplicate nodes will be distributed across the multiprocessor, the cost of generation is also distributed. Second, each node can directly generate a number of token copies. In this mode each of the destination fields held in the node description is used to label a token. This method decreases the number of nodes required in a graph, but increases the time taken to execute of the node function. Both forms may be used by compilers in combination using suitable heuristics.

5. ITERATION AND RECURSION

5.1 Simple Iteration

The hybrid architecture of the RMIT machine allows three main forms of loop control. The simplest scheme involves producing a cycle in the data-flow graph, as shown in the example in figure 5. In this case the program iterates until the counter value reaches zero. The loop is executed sequentially because tokens generated in the loop body are queued for particular processing elements. However, it should be noted that the various nodes that constitute the loop body will be distributed to different processing elements, and thus there may be concurrency involved in the execution of the loop body itself.

If the loop is part of a shared subgraph then each invocation of the subgraph will have tokens of a different colour. Each invocation executes independently as tokens with different colour values form different queues, possibility in different matching units.

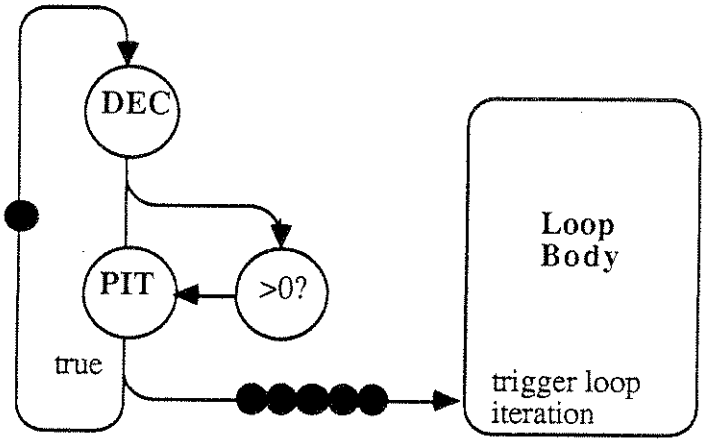


Figure 5 - Simple loop iteration scheme.

5.2 Generating Multiple Iteration Control Values

The logic required for constructing loops in data-flow systems has traditionally been more complex than in conventional von Neumann machines [16]. If the number of loop iterations is known when the loop body starts then the control structure may be greatly simplified. The RMIT architecture allows a block of control tokens for such loops to be generated in a burst. Thus, if a loop is to execute 100 times, then 100 **true** control tokens are generated followed by one **false** token. The **false** token value terminates the loop. These are created by a special node and can be injected into the loop cycle and queued until required. The same node can also inject the value of the loop counter. Using this approach the comparison, switch and decrement nodes can be eliminated because the correct number of control tokens is aready waiting in the matching unit. Such a loop is shown below in Figure 6. A problem with this scheme is that if too many control tokens are generated the matching store may overflow, and the machine may deadlock. Thus, only a small number of true tokens should be generated in one step. Larger loops can use multiple blocks of true tokens.

5.3 Recursion and Loop Unfolding

The simple loop constructs described above do not allow each iteration of the loop body to execute concurrently. A technique called *loop unfolding* has been used in dynamic data-flow machines for exploiting the maximum concurrency from algorithms[12]. In this scheme each iteration is treated as a separate invocation of the loop body, and providing there are no data dependencies between iterations of the loop, they can execute in parallel.

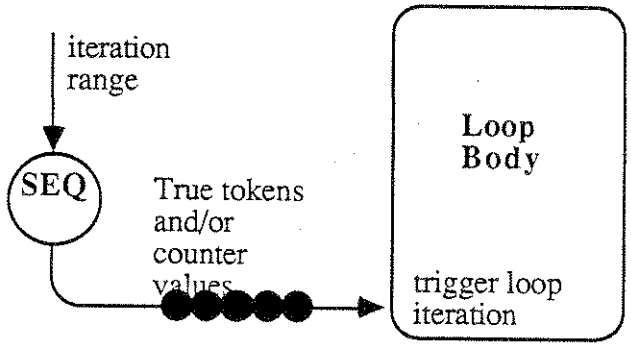


Figure 6 - Optimised loop iteration scheme.

The RMIT machine allows loop unfolding by constructing the loop from a tail-recursive shared subgraph. Each time the shared subgraph is entered a new tag value is created, and the instance of the subgraph may execute in parallel with other instances. An example of such a loop is shown in Figure 7. In this case each iteration of the loop executes in parallel. This technique is very powerful. Subgraphs may include more that one recursion, creating trees of subgraph invocations. These trees can exploit a very high degree of parallelism and execute algorithms which would otherwise have linear complexity in logarithmic time.

6 GRAPH ALLOCATION

In a perfect data-flow machine there would be one processing element per graph node. In this way the graph concurrency would be extracted optimally. However, most real data-flow machines have many fewer processing elements than nodes in a program, thus is is necessary to allocate nodes of the graph to the available machines.

6.1 Graph Partitioning

Ideally the data-flow program graph should be allocated to the multiple processing elements of a data-flow machine in a way which maximises the available parallelism. However, the information required to perform an ideal allocation is only available when the graph is executing. Even if such information were available (for example from previous program executions) the problem is also extremely complex. Consequently, the RMIT machine allocates nodes using a uniform random distribution algorithm. If the graph is large enough, this allocation can be shown to give very good performance. Simulation runs on a large number of programs indicates that this strategy provides about 80% of optimal allocation for large graphs [17].

Using this static graph allocation the processing element number can be placed directly in the destination fields of the nodes. When a token is generated, this element number is used directly by the communication network for selecting the required processor.

6.2 Dynamic token distribution

Multiple invocations of a shared subgraph can only execute concurrently if the subgraph is available on more than one processing element. Thus, shared subgraphs are distributed to all processing elements. When a token is generated the processing element number must be determined dynamically in order to distribute the workload around the machine. The current algorithm hashes the tag field of the token and generates a processing element number directly. This can then be placed in the token and used by the communication network. Using this technique all tokens with the same tag value are directed to the same processing element. Experimental results indicate that that a simple hashing algorithm which folds bits with an **exclusive or** operator provides a satisfactory randomising effect. The **exclusive or** operator was chosen because it is faster than alternative hashing functions, such as the **mod** operator.

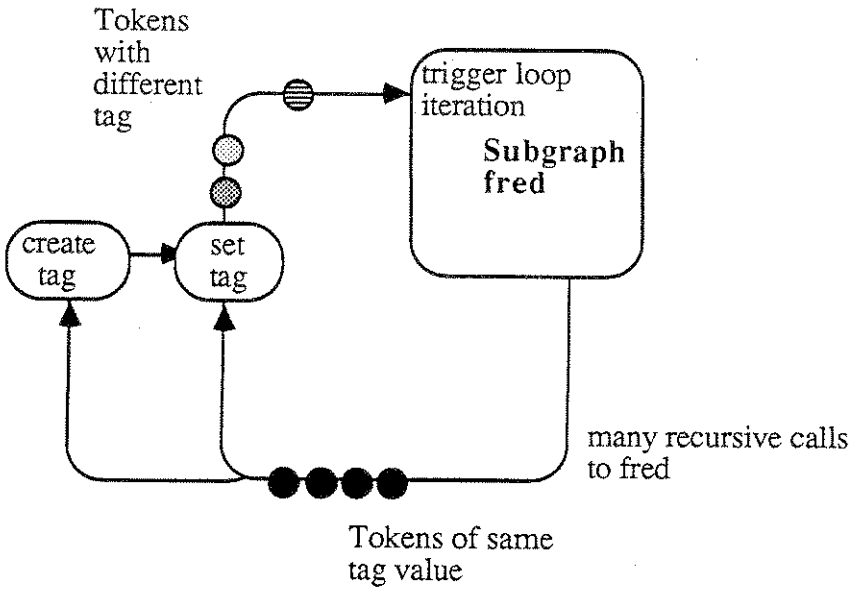


Figure 7 - Tail recursive dynamic loop.

7. TRANSMITTED TOKEN STREAMS

The stream concept for data-flow systems was originally proposed by Weng at the Massachusetts Institute of Technology [24]. Streams are ordered sequences of tokens which allow program arcs to carry more complex data structures than simple variables. Streams can be used as an alternative for permanently stored structures. They are circulated around the multiprocessor being consumed and reproduced as required. Streams fit into the data-flow architecture because they preserve the single assignment and data dependency property of such systems. Multiple concurrent assignments are created only on multiple streams, which must then be merged and synchronised appropriately.

On the RMIT machine streams are variable length, and are delimited by *stream markers*. The markers permit streams of streams and elements of the streams need not be of the same type. Several stream constructor and manipulation functions are provided.

Most of the machine nodes described in section 3 can accept simple tokens or streams of tokens. If a stream is encountered the data values are consumed as appropriate for the node function, with stream markers being passed through transparently. Thus, if a stream is sent to diadic nodes both operands should use streams of the same size. Because the stream markers are passed through the node the stream is preserved. If a stream is presented at one input of a diadic node and a single token at the other input, then a new copy of the simple token is effectively created for each element of the stream. In this way the simple token is matched with all elements of the stream.

The provision of the static queued model of the RMIT hybrid architecture makes the implementation of streams much simpler than in a purely dynamic system. Since all elements of a stream have the same colour their order is preserved by the queueing mechanism. On a dynamic machine the order of the stream can only be maintained by issuing a new tag value to each element of the stream, and then inserting code which keeps the elements ordered by inspecting the tag value.

8. EMBEDDED STORAGE NODES AND THE OBJECT STORE

There are two main methods of storing data structures in the RMIT machine. The first uses a special storage node, and is used for storing single values or small amounts of data. In these nodes the data is effectively held in the matching unit of the processing element. The second uses a special structure storage unit.

8.1 Embedded Storage Nodes

In some applications it is necessary to update 'constants' used in a program graph. For example, the difference equations which represent a digital compensator need to use different constants during the execution of the program. It is possible to retain such information by circulating it through the graph, however, this scheme results in much more data being transmitted than necessary. A special storage node is provided to hold such constants. The constant is provided on demand, and can be changed at any time. The data is loaded into a storage node by sending it to one operand of the node. If a token is sent to the other operand then a copy of the last token saved is sent to the successor node. If no token has been written an exception is generated. Because the data is held in the matching store it is not possible to hold large structures or persistent structures in storage nodes.

8.2 Object Store

A distributed Object Store is provided for the storage of large structures or shared information. Several node functions are provided for allocating storage by example, de-allocating objects, reading and writing to objects or fields of objects and adjusting reference counts. When an objects is created a *name* is returned. This name may then be stored in another object or passed around a program graph.

The mode of access required is defined when the object is allocated. There are three access modes, *deferred, non-deferred and stream* . When a read request is made for a deferred object the read is suspended until the data becomes available. This property allows data dependencies to be observed for stored objects as well as data held in tokens. Only one write operation is allowed for these type of objects, thus preserving the single assignment rules of most data-flow languages. When a read request is made on a non-deferred object the read returns data whether or not data has already been stored in the object. A special error token indicates whether the object was empty. Data in non-deferred objects may be overwritten. The non-deferred mode of access provides a conventional von Neumann store, and is not normally used by data-flow langauges. However, it is used by algorithms which are prepared to perform their own synchronisation. The object store provides semaphores which can then be used for synchronisation of access to non-deferred objects. Streams may be stored in the object store and these are called *stored streams* rather than *transmitted streams*.. Certain stream primitives are provided for manipulation of stored streams which are similar to the primitives provided for transmitted streams. A stored structure stream may be removed from the object store and converted into a transmitted stream, and visa versa.

In general the objects stored in the object store are typed and may be *simple*, *vector* or *compound* data.

Objects names area usually qualified by the process number of the executing graph. Process numbers are assigned when a graph begins execution, and are released when the graph terminates. Objects are normally stored in a process's own address space and are removed when the graph terminates. Objects which must be maintained between program executions are called *persistent objects* , and may be stored in a special process space, process 0. It is possible to create an object in the process 0 space rather than in the process's own address space. Such objects may only be accessed by those processes which hold the object name. Files are implemented as persistant objects, thus there is no special file system required in the RMIT dataflow architecture.

9. EXCEPTIONS

When a node detects an error a section of code for handling the exception must be invoked. This information is passed as a special token to code which can either correct or report the condition. Exceptions are handled by emitting a special exception token from the node in error. This exception token is then propagated by nodes further down the graph until the exception is handled. The special token type is denoted by *?* or *don't-know*.

Exceptions fall into two classes. The first class is detected in the evaluation of node functions. With this class of exception a *?* token is propagated to the succeeding nodes. *?* tokens propagated in this manner retain the original reason for the exception and the destination at which the exception first occurred. The *?* token can not be used as a control token on conditional path nodes (Pass-if-Present, Pass-if- True, Pass-if-False, Switch). Any attempt to do so results in a *?* token being sent to a reserved exception node. The exception node is initialised by sending a destination token to one input; subsequent *?* tokens arriving on the other input are sent to that destination. The second class of error is caused by invalid destinations. A destination exception is caused if a node cannot determine where to send the output token. As no successor node exists for this class of exception, a *?* token is sent to the processing-element's exception node.

This mechanism is equivalent to the stack roll back that occurs on most conventional computer systems.

10. INPUT-OUTPUT

Input-output operations are performed by input-output nodes, which may be placed directly in a data-flow graph. The names of input and output nodes are reserved, and are permanently associated with particular devices and processing-elements. When an input is requested a response-destination is sent to one operand of the I/O node. A token is sent to the other operand which is used to trigger the input operation. The response destination must remain valid during the period of the input operations. The response indicates the status of the input operation. Depending on the nature of the device, the associated input node will eventually respond with valid data or an exception. If no response destination has been specified, an exception is sent to the processing-element's exception-node.

When an output operation is requested a response-destination is sent to one operand of the output node and the data which is sent to the other operand. The node responds with a copy of the original data or an exception. If the output operation fails and no response destination is specified, an exception is sent to the processing-element's exception-node.

11. LANGUAGES AND SIMULATORS

The language currently used for writing programs for the RMIT data-flow machine is called DL1 [18]. DL1 is a low level single assignment language which is translated into the native node set of the machine. A number of programs written in DL1 are available. These include various benchmark programs, a robot manipulator control program, a laser range finder control program and some graphics manipulation programs. Some of these are shown in a separate technical report [19]. The syntax of DL1 is similar to that of PASCAL. Programs and procedures are replaced by graphs and subgraphs. Variables need not be separately declared because they are only assigned once in the program. However, errors are generated for unreferenced or uninitialized variables.

A simulator and an interpreter are also available for the RMIT architecture. The interpreter executes the program graph but does not model the machine performance. The simulator models the machine to the functional level, providing statistics on the various sections of hardware. The simulator takes into account the various timing constraints expected in the multiprocessor emulator discussed in the next section. Statistics include the queue sizes, network waiting times, matching store overheads, token traffic, processor activity, etc. Some utility programs are available for combining the performance results into a number of tables and graphs.

Compilers for the ID language from Arvind's group at MIT [20], and SISAL from Lawrence Livermore Laboratories [21] are currently being developed. These languages will provide access to some standard data-flow benchmark programs. The mapping between these languages and the RMIT machine is not obvious because of the hybrid execution model. A compiler is also being developed for the parallel Prolog variant, Guarded Horn Clauses [23]. This compiler will also take advantage of the hybrid architecture, and will be described elsewhere.

12. HARDWARE STRUCTURE

The RMIT data-flow architecture can be supported by a number of processors and a communications network. Each processing element receives a section of the data-flow graph being executed. Tokens are sent via a high speed multi stage switch network. The simulation facility discussed in the last section allows reasonable modelling of the architecture but executes too slowly to allow testing of any *real* application programs. Consequently, a multiprocessor emulation facility is being constructed which executes the emulator program in a multiprocessor configuration. This improves the simulation performance in two respects. First, the emulator is written in machine code, Second, more than one processor executes node functions simultaneously. The configuration also allows a more accurate modeling of the timing constraints.

12.1 Communication Network

The current communications network is a multistage network built from 2x2 cross bar switches as shown in Figure 8. Consequently, a token takes $\log_2$ (number-of-elements) levels before it emerges from the network. The switch allows data packets arriving at the switch inputs to be switched to either one of the switch outputs; the switch resolves any contention for the same output channel. If there is no contention then it is possible to transmit data on both channels simultaneously. Each level of the switch is buffered, which allows subsequent token words to emerge from the network one word per clock cycle. The first word of a token is the *element* number. Each switch level examines the bottom bit of the address and rotates the word right. The switch either passes the word through, or moves it to the other output. Once the first word of the token has passed through a switch level, the path remains established until the end of the token is detected (by an extra bit). This allows tokens of any length to be transmitted. Each switch includes arbitration logic to resolve conflicts for use of the switch channel. The switches are built from asynchronous state machines, thus there is no central clock distribution required. The interconnection network is further described in [13]. The prototype RMIT machine has 16 processing elements. The connection pattern for such a switch is shown in Figure 9.

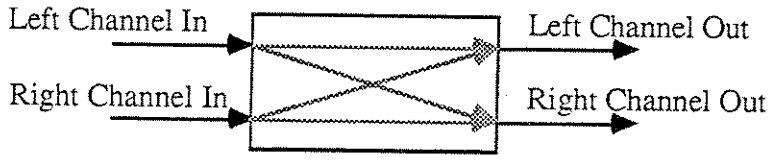


Figure 8 - A 2x2 switch element.

12.2 Processing-elements

The current RMIT emulation facility uses a processing element constructed from single or dual 68000 processors. Each processor executed a control program which interprets the nodes of the graph. The processing element may either be configured from a single 68000 boards, or two boards coupled together. If the single board configuration is used, then the control program performs the operand matching function for diadic nodes as well as the function execution. Consequently there is no concurrency in these two activities. If the dual board configuration is used then one 68000 performs the matching function whilst the other executes node functions. This arrangement allows a match operation to be performed an already matched operand pair is processed. It is advatageous when the time spent matching tokens is similar to the time spent executing functions. The 68000 configuration is in figure 10. The 68000 devices are currently being upgraded to 68020 chips and 68881 floating point coprocessors.

12.3 Fast Processing Elements

The 68000 based processing elements interpret data-flow node functions using an emulator program. Even though the program is coded in 68000 machine code the execution rate is many times slower than required for a high performance multiprocessor. Consequently, the 68000 based

processing elements will be replaced by high speed units, probably based on bit slice or discrete digital logic. These elements will be heavily pipelined (the 68000 elements cannot take advantage of any internal pipelining) to achieve a high node execution rate.

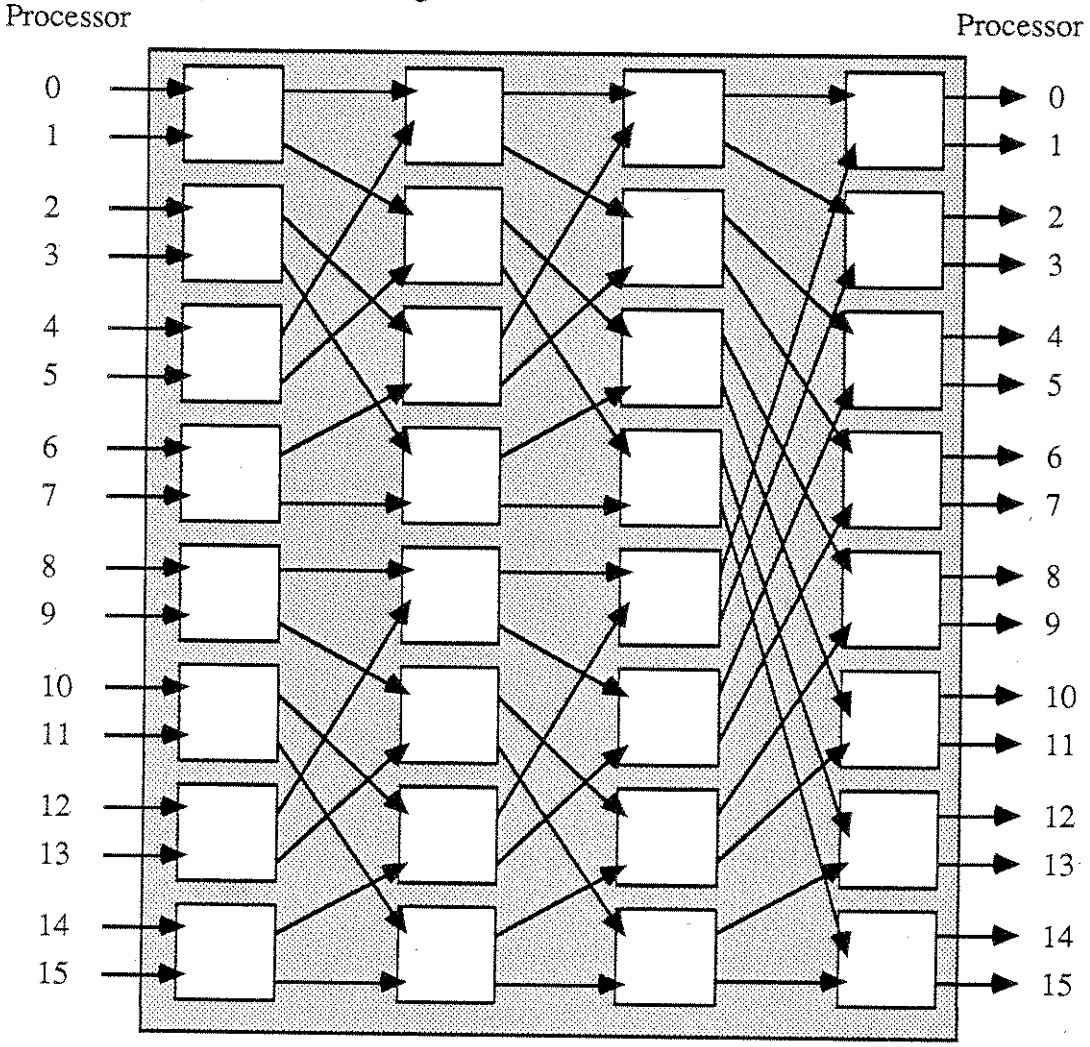


Figure 9 - Switch interconnection for 16 processing elements.

The matching unit will also be replaced by a hardware unit which implements a high speed token insertion and retrieval algorithm. The structure storage unit will also be implemented directly in hardware, and will incorporate the disk subsystem. These hardware changes should significantly increase the throughput of the elements.

13. SIMULATION RESULTS

In this section we present some siumlation results which indicate the expected performance of the multiprocessor emulator. Because the multiprocessor only interprets data-flow programs the absolute node execution rates are quite low, and should be ignored. However, the relative execution speeds indicate the changes in performance due to particular architectural features.

13.1 Hybrid Structure

In this section we examine the performance of a few programs under the hybrid architecture and compare these to the static and dynamic machine models. The architecture described in section is simulated taking into account the amount of network traffic, the varying execution times for individual nodes and the time taken to perform a match operation. The simulations were based on the execution times expected for the 68020 multiprocessor implementation of the machine with 128 processing elements. The floating point computation times are for a 68881 co-processor. The

matching store retrieval times are set for those expected performance from a hash table implementation.

A number of graphs are plotted by the simulator. Graph 1shows the number of tokens in the machine. The top trace shows the total number of tokens in the communication network and queues together with those stored in the matching store. Graph 2 shows the fraction of execution time devoted to the function evaluation, queue read time, queue write time and matching function. The queue read and write times indicate the amount of time spent transmitting tokens between processors. This graph clearly shows the amount of time spent on the matching process in relation to the other activities in the data-flow machine. Graph 3 shows the number of elements active at any time. The bottom trace shows the minimum activity level during the sampling period and the top trace shows the maximum activity level during the sampling period. The plot in the right top corner of the graphs shows processor activity plotted against time for each processor in the system. Each active processor is marked as a black horizontal line. If the processor is inactive then white space is shown. The processor number is held on the Y axis and time along the x axis. This plot is particularly useful for evaluating the workload distribution algorithm because "hot-spots" show as black areas on the graph.

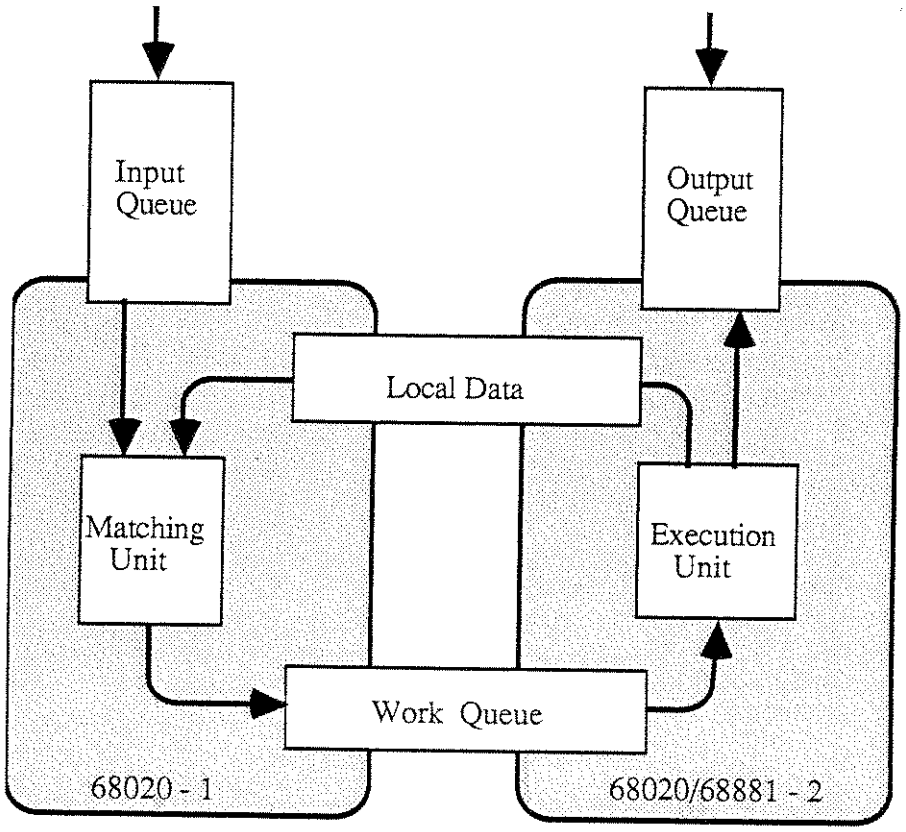


Figure 10 - Processing element structure.

The following programs have been simulated:

Program	Name
Fast Fourier Transform with 32 data sets	FFT
Fast Fourier Transform using dynamic tagging and 32 data sets	SFFT
Iterative trapezoidal integration using loop	ITR
Trapezoidal integration using single recursion - dynamic tagging	RTR
Iterative trapezoidal integration using double recursion	TR

The program FFT is a fast fourier transform program which is written as a flow-through graph. Data is introduced at the top of the graph and the results are extracted from the bottom. The program makes use of queuing on the graph arcs to distinguish different data sets, thus it is possible to push more than one data set through the graph. Consequently, providing sufficient datasets are entered to fill the graph, the amount of parallelism is determined by the static width of the graph multiplied by the depth of the graph.

The program SFFT implements the same algorithm as FFT, but uses shared subgraphs rather than one large graph. Consequently, the data must be tagged to separate different data sets which share the same code. Below we summarize the relative performance of these two programs:

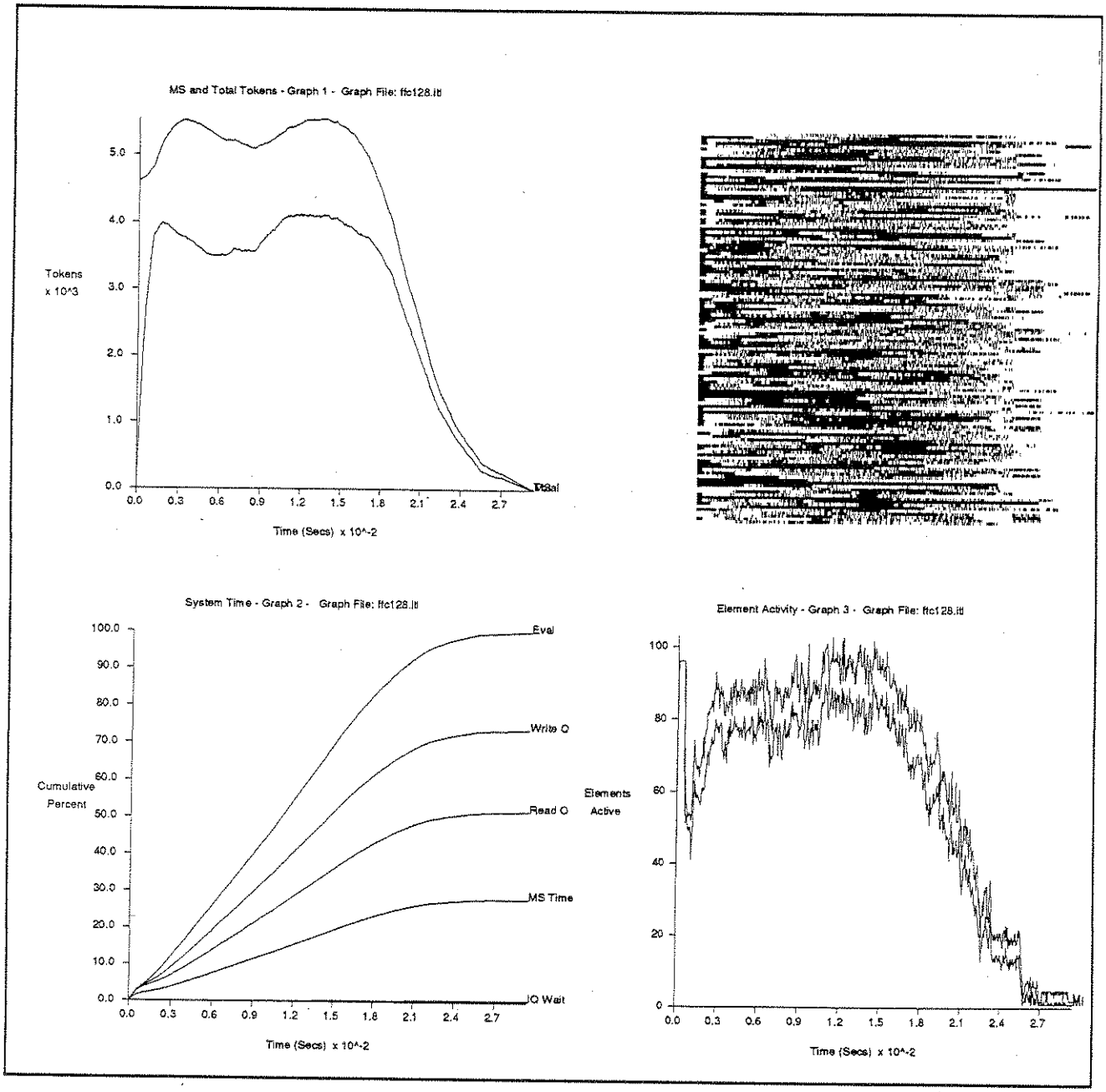
	FFT	SFFT
Total Execution Time in seconds	.03 secs	.05 secs
Time breakdown		
Time spent on function evaluation	27 %	17 %
Time spent writing tokens to network	22 %	23 %
Time spent reading tokens from network	23 %	24 %
Time spent matching tokens	28 %	35 %

The relative performance of these two programs demonstrate when the static queued data-flow model is appropriate. Because the tokens do not need to be tagged in FFT a simple matching process is used. The SFFT program uses shared sections of code, and thus the data must be tagged to distinguish the different instantiations of the code. The extra network traffic and more complex matching process, together with a larger graph (because of the inclusion of tagging operators) means that the program runs 66 % slower than FFT. The static model with queuing can offer superior performance for programs which are inherently flow-through in nature. It should be noted that these programs do not show the cost of resorting the data after the computation has completed. Also, the mix of two operand instruction to single operand instructions is not the same in the two programs. Because SFFT has more single operand instructions than FFT, fewer instruction require matching in SFFT, and thus the cost due to matching is deflated. The combination of these two factors means that the disparity between SFFT and FFT execution times should be even larger than shown.

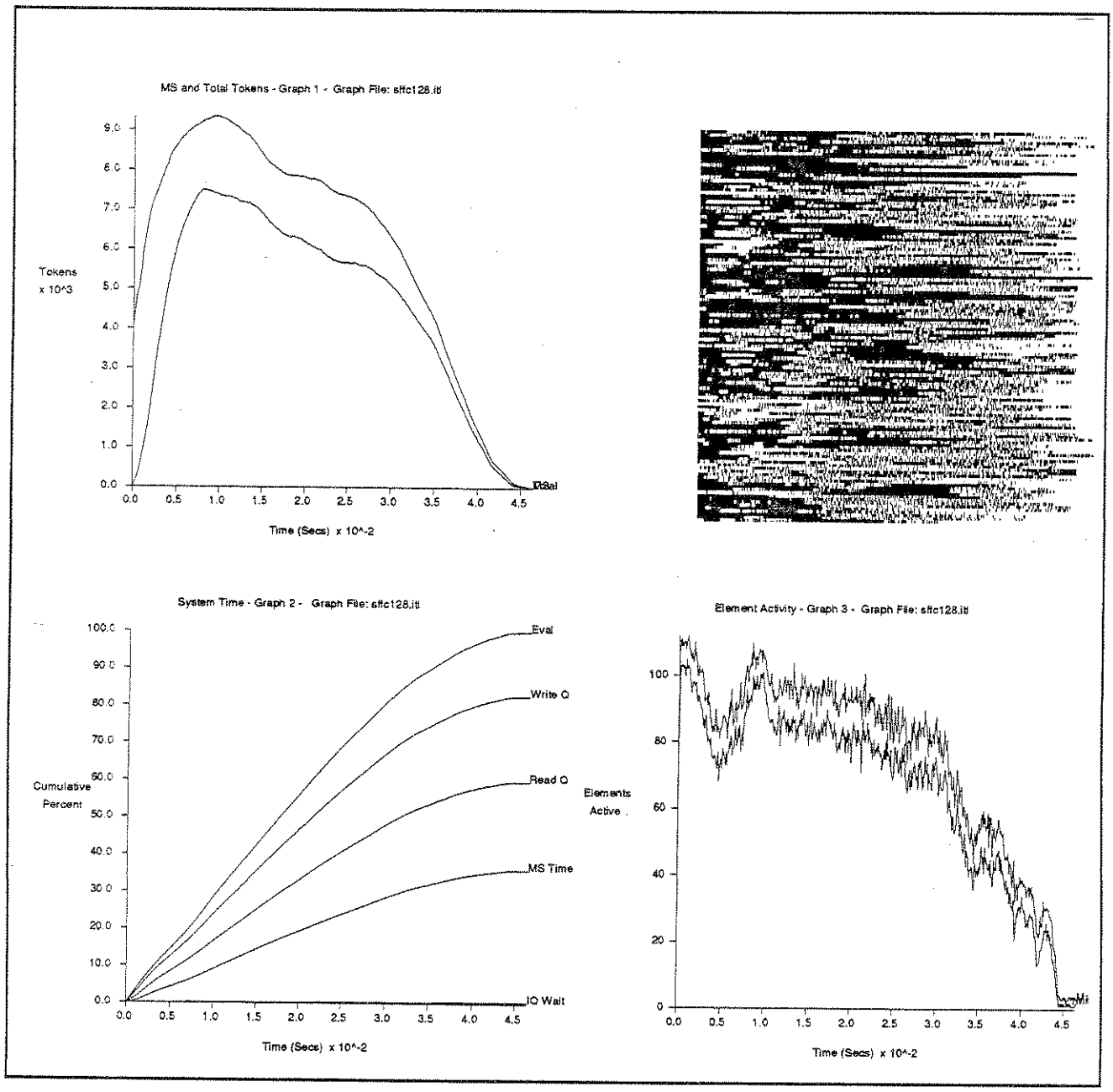
The relative performance of TR, RTR and ITR demonstrate when the tagged dynamic is appropriate. Their performance may be summarized by the following table:

	TR	ITR	RTR
Total Execution Time in seconds	.03	.14	.37
Time breakdown			
Time spent on function evaluation	20 %	32 %	23 %
Time spent writing tokens to network	25 %	25 %	25 %
Time spent reading tokens from network	26 %	26 %	26 %
Time spent matching tokens	29 %	17 %	26 %

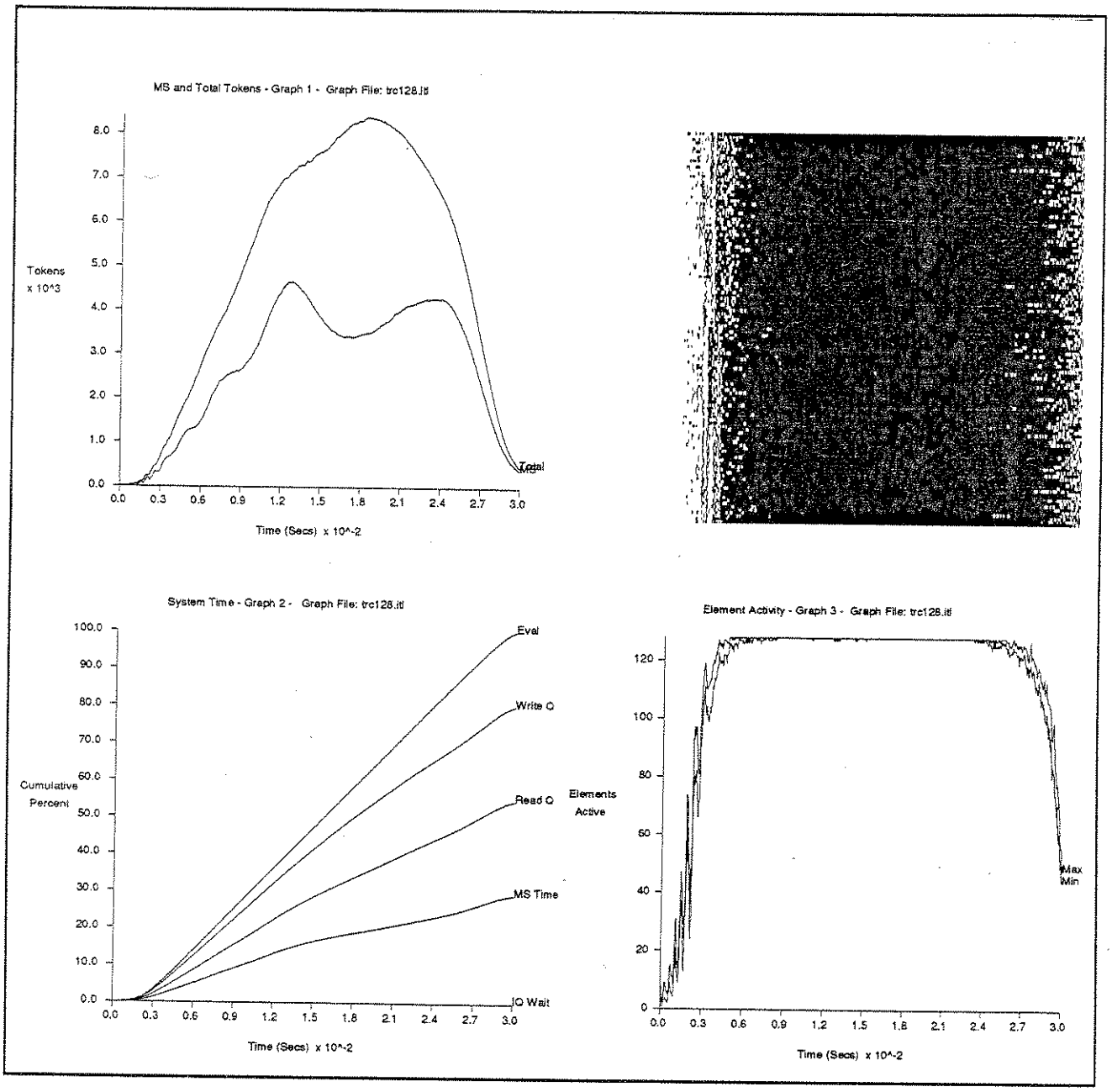
The ITR program implements a trapezoidal integration on a normal probability distribution function by iteratively moving from the start point to the end point of the integration. Because the algorithm is sequential there is very little parallelism. RTR is the same program coded using recursion instead of the loop in ITR. This program contains no more parallelism than ITR, but has the added cost of the tagging and recursion, and thus takes much longer to execute. TR, however, implements the integration by recursively dividing the interval in half until the interval converges to a single point. Whilst this program carries the tagging and recursion overheads as RTR, the algorithm is O(log n) and thus executes much faster than either ITR or RTR. Also, the amount of parallelism which can be exploited is very high. These programs demonstrate that the cost of tagging the data plus the recursion overheads can be absorbed if a good parallel algorithm is used. It is worth noting that ITR only needs 16 processors while TR needs eight time the number. The speedup, however, is less than five times.



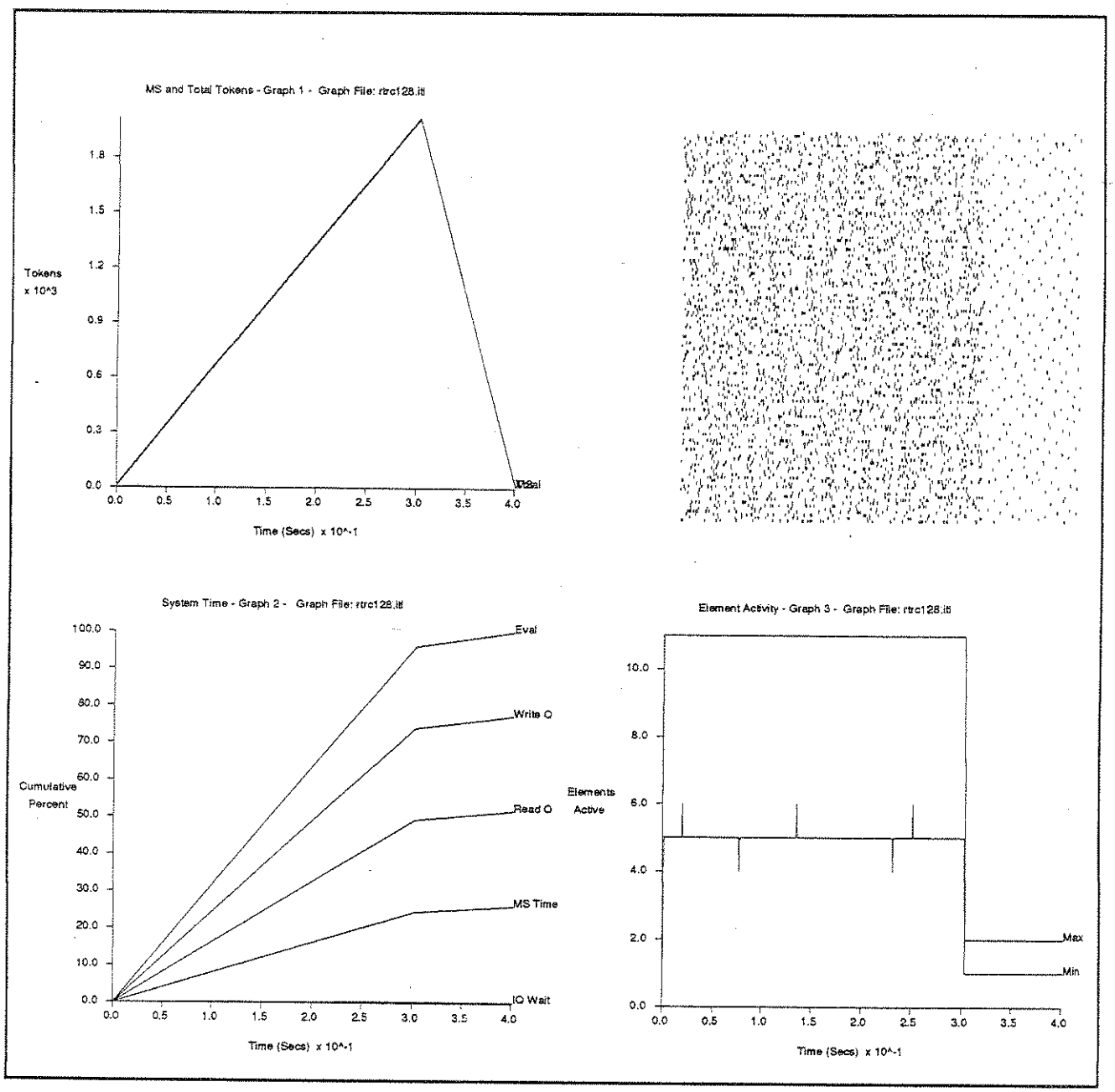
Fast Fourier Transform - FFT graphs.



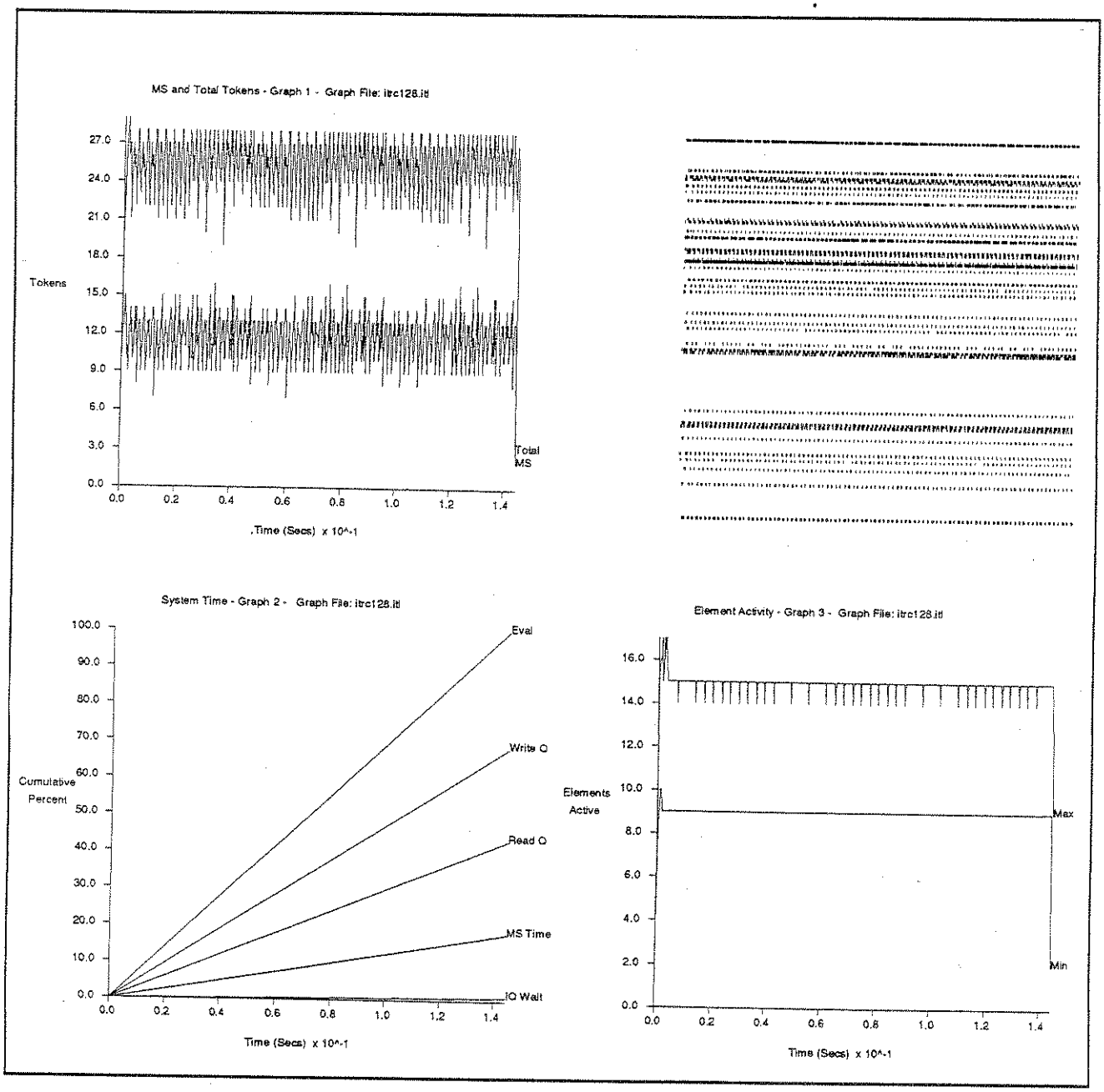
Tagged Fast Fourier Transform - SFFT Graphs.



Double recursive trapeziodal integration - TR Graphs.



Singly recursive trapeziodal integration - RTR Gaphs.



Iterative trapeziodal integration - ITR Graphs.

13.2 Limited Number of Processing Elements

It is important to consider the effect of insufficient resources in the multiprocessor on the performance of a given algorithm. A program may require more processing elements than available on the real machine. In this case is is desirable for the performance of the machine to degrade linearly with the loss of resources. A number of programs were studied under conditions of insufficient resources. They were:

Program	Name
Fast Fourier Transform with 32 data sets	FFT
Fast Fourier Transform using dynamic tagging and 32 data sets	SFFT
Iterative trapezoidal integration using double recursion	TR
Long trapeziodal Integration	TRLONG
6 Queens problem	QR6

Their performance is shown below in Figure 11. The dotted line on this plot shows ideal speedup. In this case the execution time of the programs is inversely proportional to the number of processors. In all cases programs deviate from ideal performance because they run out of concurrency. The difference between TR and TRLONG demonstrates the effect of startup and shutdown of algorithms. Linear speedup can only be achieved if the startup and shutdown time is insignificant with repect to the actual computation time. In TR the startup and shutdown times are quite large in relation to the total execution time. However, in TRLONG a much longer integration was performed, and thus the speedup curve is much closer to ideal. If the execution time was increased further the TRLONG curve would meet the ideal line. The 8 queens problem could not be simulated due to restrictions in the simulation environment.

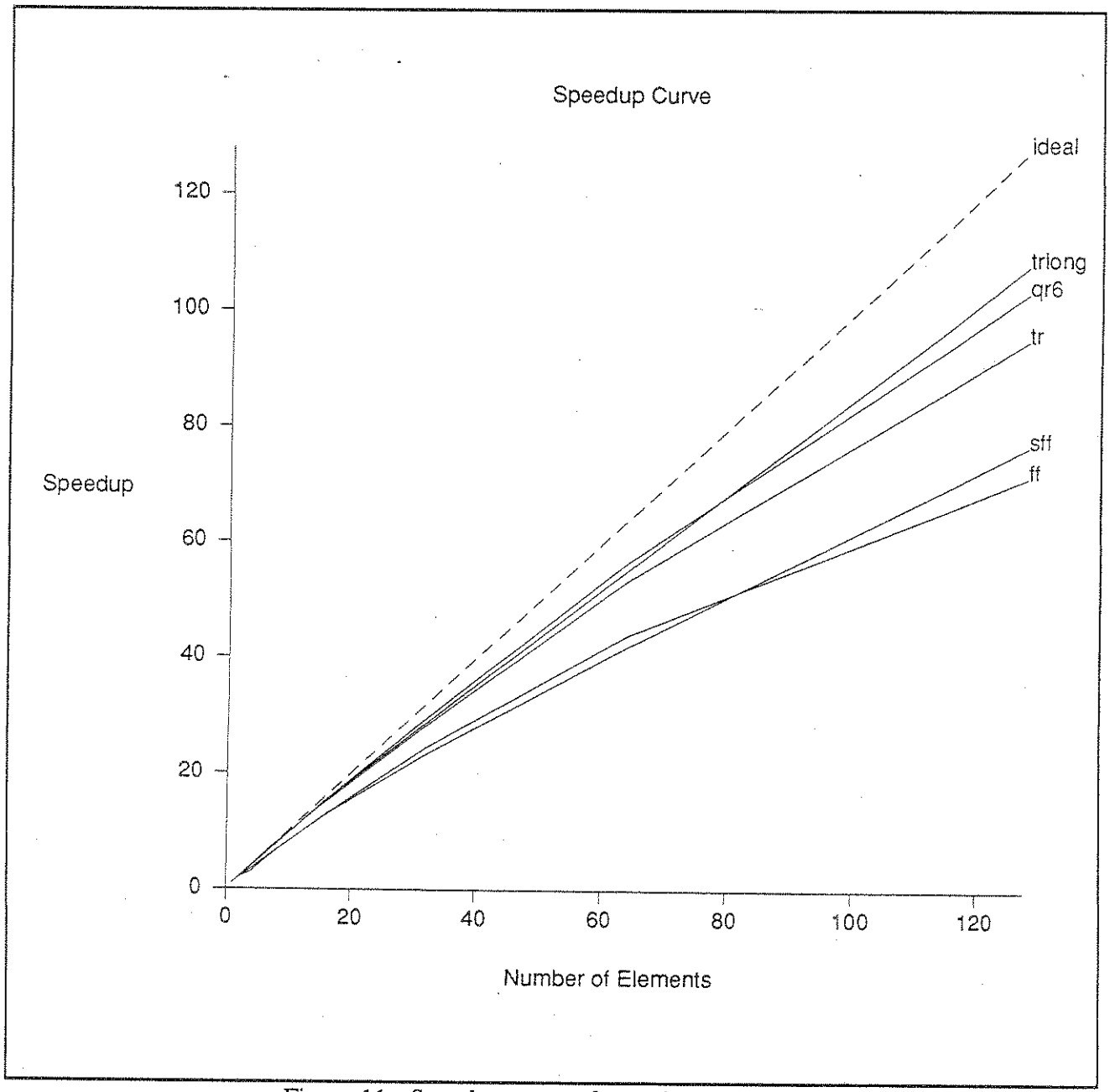


Figure 11 - Speedup curves for various programs

14. CONCLUSION

In this paper we have described the key architectural features of the RMIT data-flow machine. Some of these differ significantly from other data-flow architectures. The use of a hybrid static-dynamic execution model not only improves performance, as illustrated by the simulation studies, but simplifies many program graphs. The overall effect of the hybrid structure will not be fully appreciated until the ID, SISAL and Guarded Horn clause compilers are complete and real applications have been studied.

The use of variable length tokens allows the machine to support many data formats and representations. This will be particularly useful in the guarded horn clause implementation in which complex structures are required. It also allows variable length vectors for efficient vector processing.

The shared subgraph mechanism not only helps reduce graph size, but also provides an efficient mechanism for unfolding loops. Thus, normal loops are executed sequentially and tail recursive subgraphs allow multiple invocations of the loop body to execute concurrently. The cost of unfolded loops is the insertion of tagging code, the increased network traffic and a slower matching process.

The simple allocation strategy for loading graphs onto the processing elements has been shown to perform efficiently, and does not require complex and time consuming loader programs.

Storage nodes allow a graph to maintain small amounts of semi-permanent information without the cost of a structure store access. Large structures can, however, be placed in one of many structure storage units if only small sections of the object are being accessed. The structure store also provides a suitable repository for permament objects such as conventional files. The stream mechanism allows efficient processing of objects when the entire object is required.

The exception mechansim allows a graph to detect errors and possibly take corrective action. Exceptions which are not handled by a graph can be referred to the processing element and thus the monitor program.

In order to determine the effectiveness of the overall architecture, the project is undertaking a number of real application studies. These include:

- Using simulated annealing algorithms for optimal building layout.
- Robot trajectory planning algorithms
- Timetable compuation algorithms
- Some experimental expert systems
- High speed digital logic simulation
- Real time computer generated imagery

These studies will be the topic of future technical reports.

Acknowledgements

The authors wish to acknowledge the support of the project team. The Parallel Systems Architecture Project at RMIT is being supported by the Commonwealth Scientific and Industrial Scientific Organisation (CSIRO) under an Information Technology joint research project. The authors also wish to thank Mark Ross for reading a draft of this report.

References

[1] J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor", Proc 2nd Ann. Symp. Computer Architecture, New York, May 1975.

[2] J.B. Dennis, G.A. Broughton, and C.K.C. Leung, "Building Blocks for Data Flow Prototypes", Proc 7th Ann. Symp. Computer Architecture, LaBoule, france, May 1980.

[3] J.B. Dennis, G.R. Gao, and K.W. Todd, "Modeling the Weather with a Data Flow Supercomputer", IEEE Trans. Computers, Vol C-33, No 7, July 1984, pp 592-603.

[4] M Cornish, D.W. Hogan, and J.C. Jensen, "The Texas Instruments Distributed Data Processor", Proc. Louisiana Computer Exposition, Lafayette, La., March 1979, pp 189-193.

[5] A. Plas et al, "LAU System Architecture: A Parallel Data-driven Processor Based on Single Assignment", Proc. 1976 Int'l Conf. on Paralle Processing, pp 293-302.

[6] Arvind and R.A. Iannucci, "A Critique of Multiprocessing von Neumann Style", Proc 10th Ann. Int'l Symp. Computer Architecture, Stockholm, June 1983, pp 426-436.

[7] Arvind and K.P. Gostelow, "The U-Interpreter", Computer, Vol 15, No 2, Feb 1982, pp 42-50.

[8] J. Gurd and I. Watson, "Data Driven Systems for High Speed Parallel Computing - part 2: Hardware Design", Computer Design, July 1980, pp 97-106.

[9] A.L. Davis, "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine", Proc 5th Ann. Symp. Computer Architecture, New York, 1978, pp 210-215.

[10] V.P. Srini, "A Message-based Processor for a Data-flow System", Proc 1984 Int'l Workshop in High Level Computer Architecture, Los Angeles, May 1984, pp 1.10-1.19.

[11] G.K. Egan, "Data-flow: Its Application to Decentralised Control", Ph.D. Thesis, Department of Computer Science, University of Manchester, 1979.

[12] Arvind and D.E. Culler, "Data-flow Architectures", Laboratory for Computer Science, MIT technical report MIT/LCS/TM-294, 1986.

[13] G.K.Egan, C. Baharis, M Rawling, and A. Young, "RMIT Data-flow Project", IREE. Proc 2nd Australian Computer Engineering Conference, Sydney, May 1986.

[14] V.P.Srini, "An Architectural Comparison of Data Flow Systems", IEEE Computer. March 1986, pp 68-87.

[15] G.K.Egan, and C.P. Richardson, "Object Recognition Using a Data-Flow Computing System", Microprocessing and Microprogramming 7, North Holland, 1981.

[16] T. Shimada, K. Hiraki, K. Nishida and S. Sekigucki, "Evaluation of a prototype data-flow processor of the SIGMA-1 for scientific computations", Proceedings of 13th Annual International Synposium on Computer Architecture, pp 226 - 234.

[17] C.P. Richardson, "Object Recognition using a data-flow Machine: Algorithms for a Laser Range-finder", M.Sc. Dissertation, Department of Computer Science, University of Manchester, 1979.

[18] M. Rawling and C.P. Richardson, "The RMIT Data Flow Computer: DL1 User's Manual", Royal Melbourne Institute of Technology Technical Report, TR-112-059R,

1987.

[19] D. Abramson, G.K. Egan, M. Rawling and C Baharis, "The RMIT Data Flow Computer: Benchmarks", Royal Melbourne Institute of Technology Technical Report, TR-112-058R, 1987.

[20] R. Niknil, K. Pringali and Arvind, "Id Nouveau", Computation Structures Group Memo 265, Massachusetts Institute of Technology, Laboratory for Computer Science.

[21] McGraw et al, "SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual", Lawrence Livermore National Laboratories, M146.

[22] D. Abramson and G.K. Egan "The RMIT Data Flow Computer: A Hybrid Architecture", Royal Melbourne Institute of Technology Technical Report, TR-112-057R, 1987.

[23] Kazunori Ueda, "Guarded Horn Clauses", Doctor of Engineering Thesis, University of Tokyo, Graguate School, 1986.

[24] K.S. Weng, "Stream oriented Computation in Recursive Data-Flow Schemas", Technical Memo 68, Laboratory for Computer Science, Massachusetts Institute of Technology, Oct 1975.