The RMIT Data Flow Computer: A Hybrid Architecture

D. ABRAMSON* AND G. EGAN**
** Division of Information Technology, C.S.I.R.O., 55 Barry St, Carlton, Victoria 3053, Australia.*
*** Department of Electrical Engineering, Swinburne Institute of Technology, P.O. Box 218, Hawthorn, 3122, Australia*

This paper examines the two conventional models of data-flow machines, static and dynamic. The advantages and disadvantages of each scheme are described. A hybrid architecture is presented which gains the advantages of both. This architecture has been incorporated into a new data-flow machine built at the Royal Melbourne Institute of Technology. Some simulations are presented which illustrate the advantages of the hybrid approach.

Received November 1987, revised November 1988

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

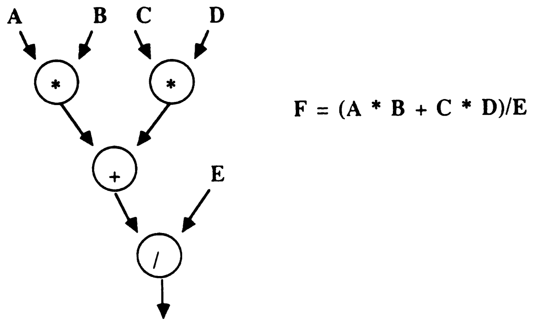


Figure 1. A data-flow graph.

a sample data-flow graph for an arithmetic expression and Fig. 2 shows 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 any computation are sent from the processor which holds the source node to the processor which holds the destination node. When the results arrive at the destination processor they wait 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.

There are currently two main classifications for data-flow architectures, *static* and *dynamic*. The static scheme was first proposed by Dennis,[7,8,9] and has been used by

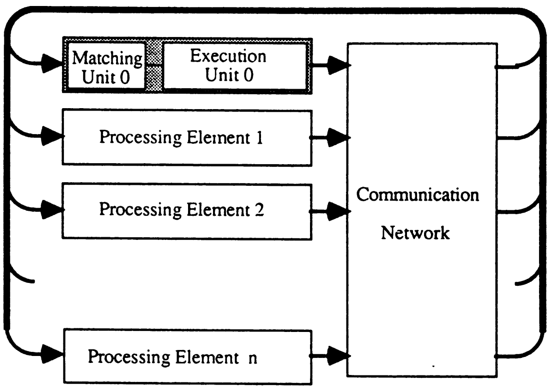


Figure 2. Hardware model for the data-flow machine.

various research architectures such as TI's DDP[5] and the LAU architecture.[13] The dynamic scheme is used in Arvind's research group at MIT,[3,4] at Manchester University,[12] the DDM architecture[6] and the EDFG system.[14] 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 Ref. 15. A data flow machine being built at Royal Melbourne Institute of Technology around an architecture originally devised by Egan[10] at Manchester University is a hybrid between the classic static and dynamic architectures. The hybrid attracts the advantages of both schemes and avoids most of their disadvantages.

This paper describes the classic architectures together with their strengths and weaknesses, and then describes the hybrid solution. Some simulation results are presented together with the future plans for the project.

2. DYNAMIC AND STATIC ARCHITECTURES

In the static data-flow architecture program instructions, or nodes, execute when all of the operands, or tokens, appear at their inputs (e.g. two input nodes require two tokens before the node executes). The ordering of the tokens on the arcs can be guaranteed by constructing the data-flow graph so that only one token can exist on

* To whom correspondence should be addressed.

any input at a time. Because a node can only have one input waiting there is no advantage in placing the node on more than one processor, as it cannot execute more than one computation at the same time. Thus, program graphs are statically allocated to the real processors, and the destinations are statically coded in the graph. Static architectures work well for processing "sets" of data which are streamed through the graph, because the data is output in the same order that it is entered. For any single "data set" the amount of parallelism is basically limited to the width of the program graph. If data values are fed back into the graph then the number of nodes active concurrently may exceed the graph width. More concurrency can be extracted when multiple data sets are introduced because a processor can start work on the next piece of data when it has completed a computation.

The hardware required by the static model is quite simple. Because an instruction can only have one outstanding operand, the matching unit only needs to retain which operand is waiting and the value. When the other operand arrives the match can occur. The main disadvantages of the static architectures are that it can be difficult to construct the graph to guarantee one-token per arc restriction. The one-token per arc requirement can be achieved by balancing the arc lengths in the program graph and inserting sequencing code in conditional expressions.[2] The amount of parallelism in a static architecture is severely restricted because the only way to achieve more concurrency than that dictated by the width of the program graph is to introduce multiple data sets. Another problem with those machines which statically allocate the graph to processing elements is that because nodes are permanently assigned to processors, a particular processor may attract an unequal share of the work load. There is no possibility of dynamically distributing the work load across the machine. This problem is not apparent on all static machines. The parallelism on these machines is further restricted because a sending node cannot transmit a result until the receiving node is ready to accept a new token. This has the effect of halving the number of active processors, because only alternate levels of the graph can be active at any time. A modified form of the static architecture has been used in the NEC dataflow chip. In this machine a small number of tokens may be queued on the inputs of nodes. The queues solve the problem of a sending node being unable to compute a result whilst a destination node is busy. However, because the queues are quite small, they can overflow and block computation as in the static scheme.

In the dynamic model every token holds a tag field, which determines the sequence number of the token. An instruction may have multiple tokens waiting for matching on an arc, and they are only removed and the instruction executed, when the token with the correct tag value arrives at the other arc. The addition of tags alone does not increase the concurrency in a data-flow graph as it is still limited to the width of the graph and the number of concurrent data sets. However, because a node can have more than one token waiting at a time, it may be sensible to allow more than one invocation of the node to execute at a time, each with a different set of data. Thus if two sets of tokens arrive for a node in a dynamic architecture and can match simultaneously then two processing elements can be occupied. In a static data-flow machine they would be forced to execute sequentially. This feature is called loop-unfolding, and can increase the amount of parallelism in a data-flow machine significantly above the width of the graph. In fact, loop unfolding may produce so much parallelism that it exceeds the number of real processors and much of the advantage is lost. Loop unfolding also helps to distribute the work load because a given node may execute on more than one processor. When a token is produced the destination processor is calculated dynamically, usually by applying a hashing function to the tag field. Nodes can only execute if there is a copy of the graph, or relevant part, held, at the destination processor. Thus it is necessary to duplicate the program graph across the processors or a subset of processors.

The addition of tags to tokens makes matching much more complex and time consuming. Thus, the cost of the large amounts of parallelism is a much more complex matching unit, which can increase the cost of each processing element over the static scheme. The amount of network traffic may also increase as the token need to carry tag information. A major disadvantage of the dynamic model is that data may be output in any order and must be resorted if ordering is important. This resorting may add a considerable overhead to the computation. Thus in cases when pipelining is the most important form of parallelism and there is little loop unfolding or feedback, as in many real world control problems, the dynamic architectures have the added complexities of tagging and untagging, the increased network traffic and the resorting of data. For example, each iteration of a loop in a dynamic machine must include tag generation code, even if the loop has a data dependency in it that forbids loop unfolding. This overhead is not present in the static model.

The architecture described in the next section allows efficient execution of pipelined data sets, without the disadvantaged of one-token per arc, and the high degree of parallelism obtained in dynamic schemes, without the necessity of always tagging data.

3. THE HYBRID ARCHITECTURE

3.1 Combining Static and Dynamic Architectures

The hybrid architecture allows more than one token per arc by having two different modes of operation. In the first, tokens may be queued on an arc for a particular node, and are executed in the order that they were placed on the queue. If there is a queue of tokens on one arc of a two input node, and a token arrives at the other input then the head of the queue is removed and the node executed. This arrangement is similar to the classic static architecture, but allows more than one token per arc by providing queues. The queues must be large enough for the probability of queue overflow to be small. Thus, nodes should never be blocked from transmitting results because a destination queue is full. In this mode tokens do not carry any tagging information. In the second mode, tokens may be tagged and "heaped" at a particular input until a token of a matching tag arrives at the other input. This is identical to the dynamic scheme. In this mode tokens must contain tag values which distinguish the different data sets.

Furthermore, tokens with the same tag value may be queued on an arc. Thus, it is possible for a computation to generate a new tag value and place it on many tokens. This provides a combination of a queued static model and the purely dynamic scheme. These combinations are shown in Fig. 3.

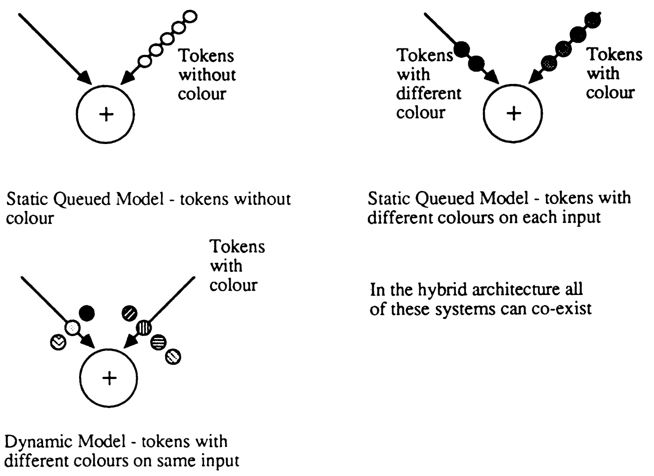

Tokens without colour
Static Queued Model - tokens without colour
Tokens with different colour
Tokens with colour
Static Queued Model - tokens with different colours on each input
Tokens with colour
In the hybrid architecture all of these systems can co-exist
Dynamic Model - tokens with different colours on same input

Figure 3. Hybrid architecture.

The allocation of the graph to the processing elements is done in two ways. It can either be allocated to processors with any given node appearing on only one processor, or the node can be copied and the copies allocated to multiple processors. If the machine is executing a section of code using the static-queued model then only one copy is required. In this case the nodes are allocated to processors using a uniform random probability distribution. Whilst this may not achieve optimal graph placement, it is much simpler than calculating optimal or even sub-optimal placements. This random placement has been shown to perform well, especially if the graph is large (see the activity plots later in this paper). If the dynamic tagged scheme is used it may be necessary to duplicate the graph on many processors to achieve the required concurrency. When a token is created as a result of a node firing, the destination address is calculated by hashing the tag field. This has the effect of randomly distributing the work to the processors and balancing the workload. The allocation scheme is specified by the programmer by the choice of control constructs when the graph is coded.

Providing both data-flow models achieves the advantages of both. If the programmer wishes to flow "sets" of data through the graph then no tags will be added to the tokens. There are a number of advantages with this approach. First, the cost of maintaining and manipulating tags is removed. Thus, loops which have data dependencies in them, which prevent loop unfolding, may be coded without tag generation logic. The queues preserve the loop ordering even if the data has some initial tag value when it enters the loop. Second, the network traffic is reduced because the tag need not be transmitted. Third, the matching function becomes relatively easy, and even a simple implementation of matching store yields predictable, scalable and acceptable performance. Fourth, unlike the dynamic model, the data does not need to be resorted when it emerges from the graph, and may be used directly. Because the architecture allows more than one token per arc it is not necessary to balance arc lengths in the graph as in the pure static case. However, some additional code may still be necessary to ensure correct sequencing of conditional expressions. Also no interlocks are necessary between the sending node and the destination node. The addition of queues allows a sending processor to compute results and send them even if the destination processor is busy and has a queue of pending tokens. A further advantage of the queued static model is that it makes the implementation of ordered token *streams* easier, because the elements of the stream remain in their intended order.[1,17] The implementation of streams on dynamic machines is much harder because the stream order can only be maintained by tagging each stream element with a unique value, and including code which examines the tag before the stream is processed. Unlike the static model, the hybrid architecture allows token tagging and loop unfolding if it is required.

3.2 An Implementation Strategy

The advantages of combining the two conventional models can be illustrated by considering the techniques necessary in order to build a matching unit. Two main tables are required, the hash table and the token table, as shown in Fig. 4. The token table is used to hold the tokens until their partners arrive. The hash table is used to point to various chains of tokens in the token table, and is large enough to hold one entry for each node of the graph held in the processor.

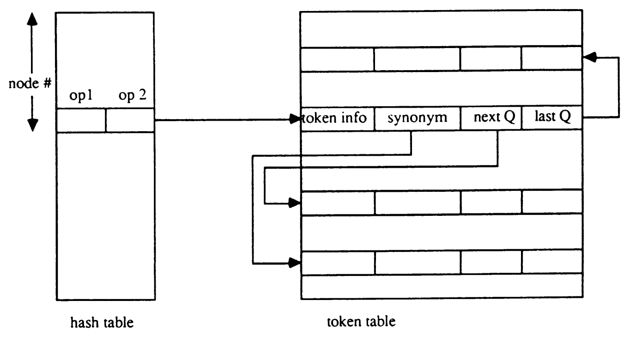


Figure 4. Tables required for the matching unit.

The matching unit interprets the hash table in two ways. If the incoming token does not contain a dynamic tag field then the requested node number is used as an index into the hash table. In this case the hash table entry points to the head of a queue of tokens for the particular node for each operand. If there is a chain for the opposite input of the node, then the head of the list is removed and the node is executed. If there is not a chain present, then the incoming token is placed on the end of a queue for the particular operand, by following the tail pointer in the token. In this way tokens may be matched with a fixed cost related to the time required to read the hash table and add or remove an entry from the token memory.

If the incoming token contains a tag field then the hash table cannot be directly accessed. In this case the tag is hashed against the node number to produce an index into the hash table. The head pointer is interpreted as a pointer to a chain of synonyms which all hash to the same entry. A separate queue of synonyms is searched for the required node and tag field. If a token with the required node and tag field is found then it is used as the head of a queue and is consequently removed. If a synonym is not found, then a new queue is started as in the static case. The performance of the retrieval and insertion operations using this scheme is governed by the effectiveness of the hashing function and the hash table occupancy. If the table is sparsely occupied and the hashing function yields a uniform distribution then the performance is likely to be acceptable. However, it is difficult to choose a hashing function which will give predictable performance for different token loads. Thus the dynamic scheme will almost certainly perform worse than the static scheme. These costs have been modelled in the simulations shown in Section 5.

4. CURRENT EXECUTION ENVIRONMENT

The architecture described above is being implemented and assessed by three different techniques. First, a discrete even based simulator has been available for some time and is used to produce some of the simulation results shown in the next section. The simulator models a multiprocessor configuration down to the functional unit level and provides statistics such as the queue sizes, waiting times, matching store overheads, token traffic, processor activity, etc. The major problem with the simulator is that it can only execute small problems because of the time stretch involved. The simulator currently executes about 1000 nodes per second on a small 68000 based workstation.

The second level of support is currently nearing completion and involves a multiprocessor hardware implementation of an interpreter for the architecture.[11] This system is constructed from a number of Motorola 68000 based microprocessor boards connected via a high speed multistage switch. The microprocessors include a modest amount of memory, hardware queues for holding pending tokens and a copy of the data-flow interpreter. The hardware can be configured in two ways. The first allows each logical processing element of the data-flow machine to be constructed from one 68000 board. In this case the emulator uses one 68000 to perform token matching and node function execution. The second configuration allows each logical processing element to be constructed from two 68000 boards, one performing the token matching functions and the other performing the node function execution. The second scheme is applicable when the matching times and the execution times are the same order. This scheme then provides a pipelined effect. If the function execution times dominate then the matching processor becomes idle and the first configuration gives better hardware utilization. The current hardware is further described in.[1] Based on a prototype processing element, each processor executes about 12000 node functions per second. The complete hardware will have 16 processing elements, giving a performance of about 200,000 node functions per second. The 68000 processors are currents being upgraded to 68020's with 68881 floating point co-processors.

The main advantage of the multiprocessor emulator over the simulator is that it allows the execution of larger programs. However, the node execution rate is still too slow to provide a high speed data-flow processor competitive with other von Neumann machines. Thus, the third level of support will be a hardware based processing element, matching store and structure store. This stage of the project is still under development.

5. SOME SIMULATION RESULTS

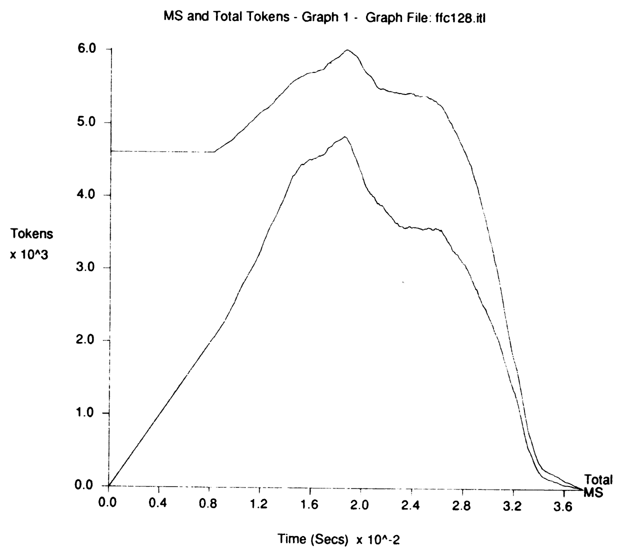
In this section we examine the performance of a few programs under the hybrid architecture and compare these to the static and dynamic machine models. The architecture is simulated taking into account the amount of network traffic, the varying execution times for individual nodes and the time taken to 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 the performance expected from a well designed hash table implementation.

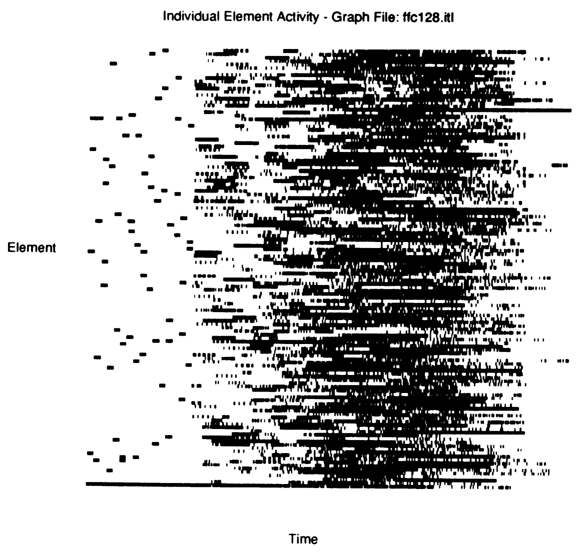
A number of performance graphs are generated by the simulator. Graph 1 shows the number of tokens in the machine. The top trace shows the total number of tokens in the communication network and queues together with those stored in the matching store. Graph 2 shows the fraction of execution time devoted 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 graph sets 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.

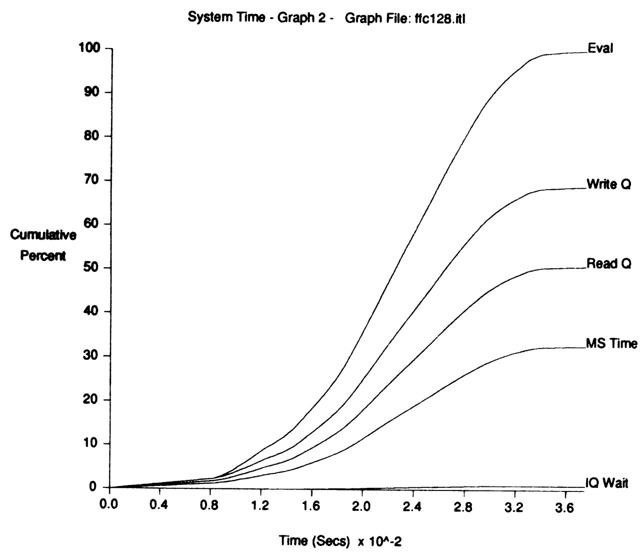
The simulation results presented are for the following programs:

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 simple 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







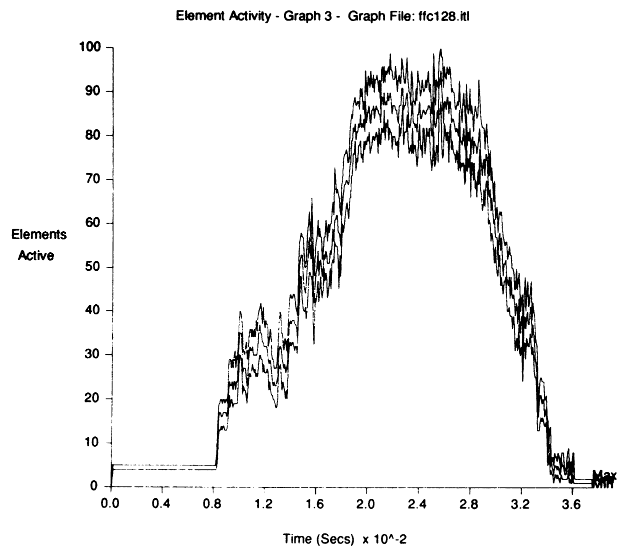
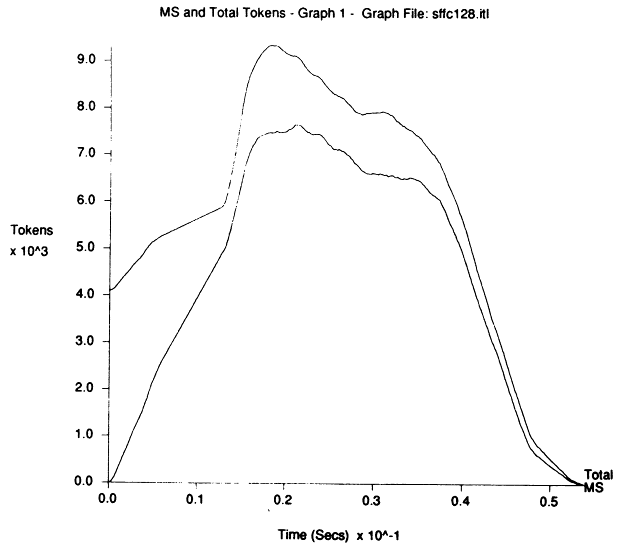
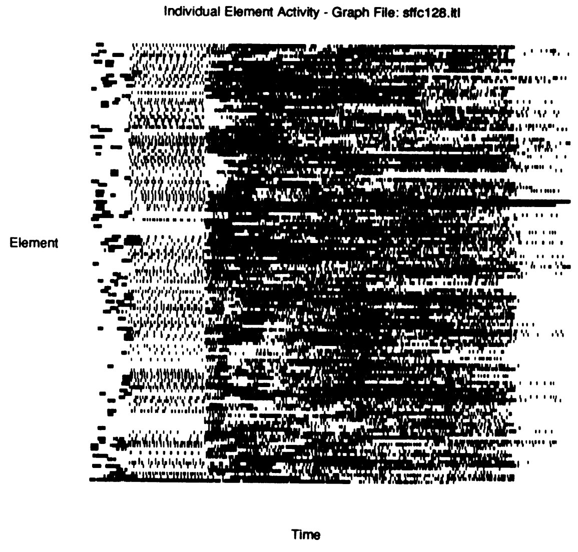
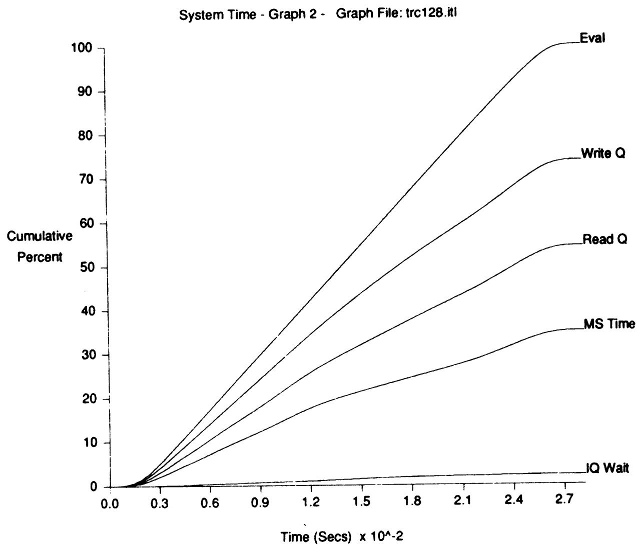


Figure 5. Fast Fourier Transform FFT Graphs.







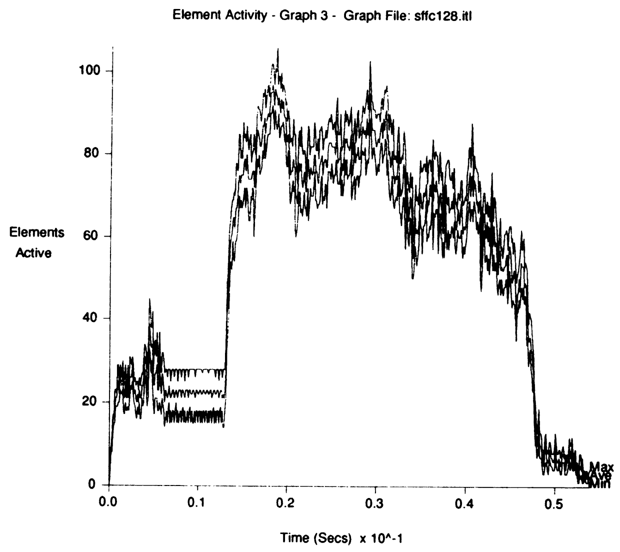


Figure 6. Tagged Fast Fourier Transform SFFT Graphs.

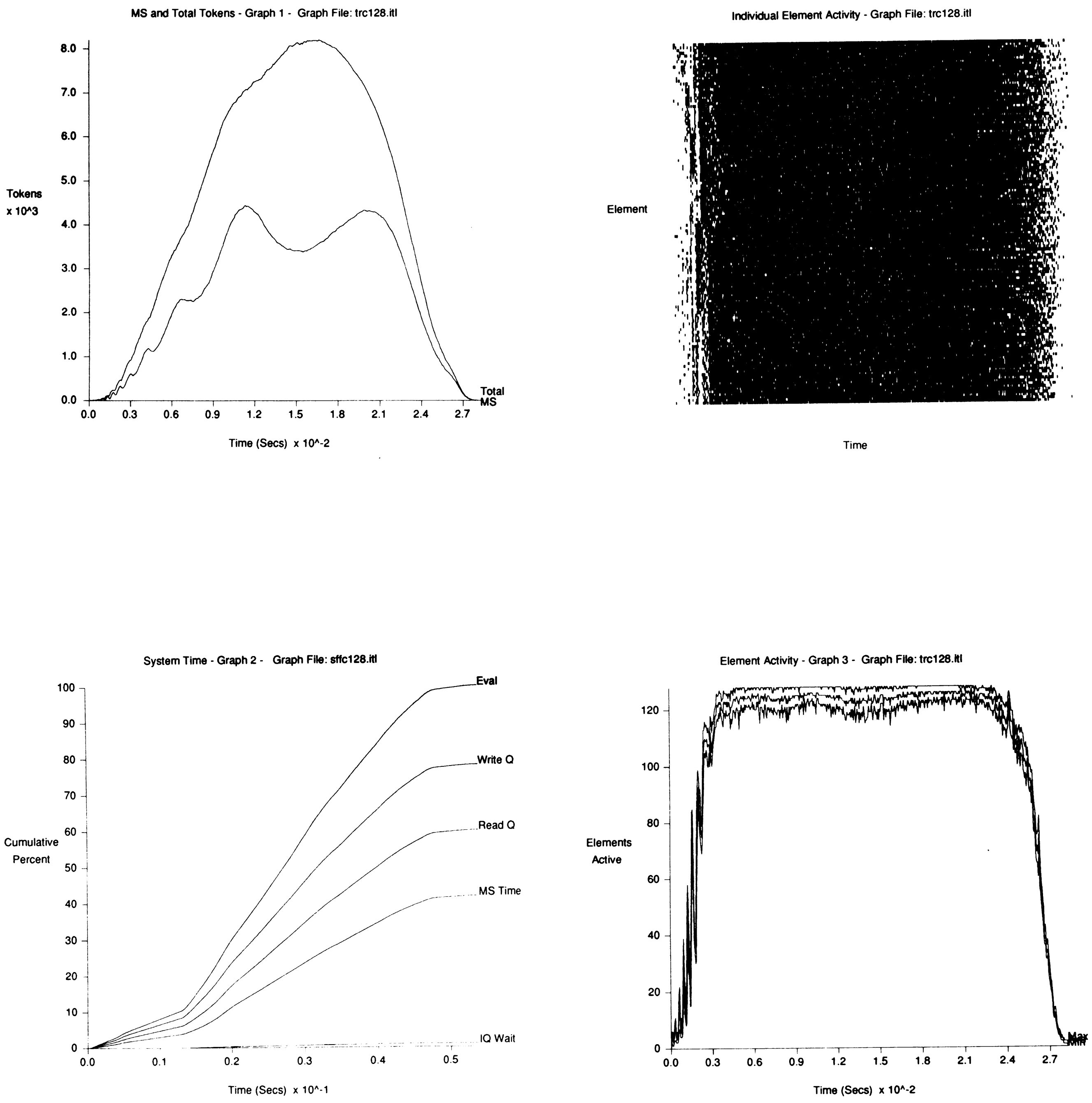
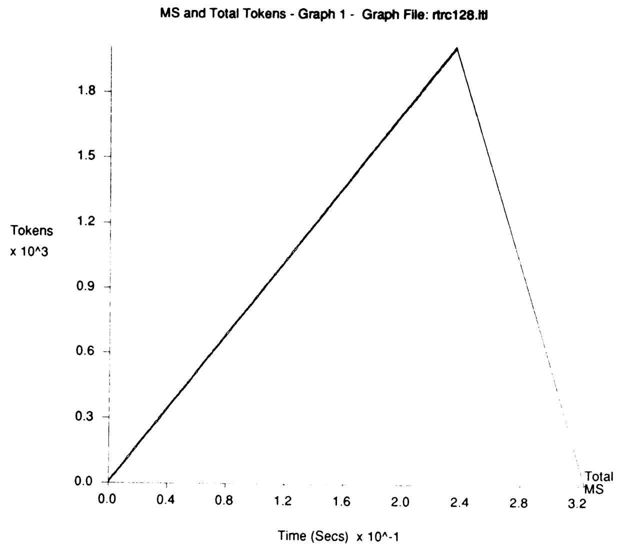
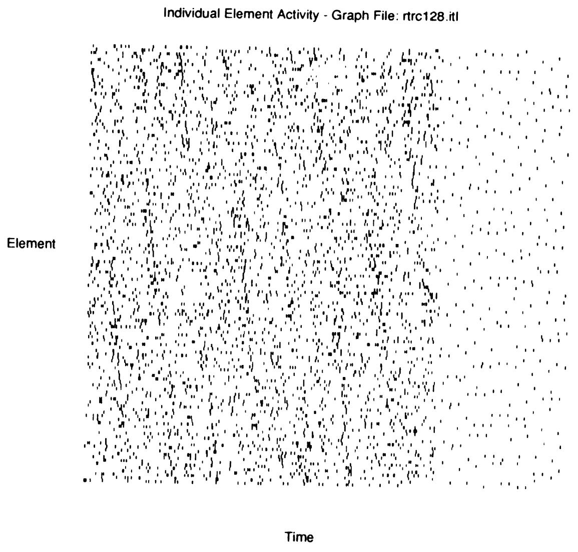
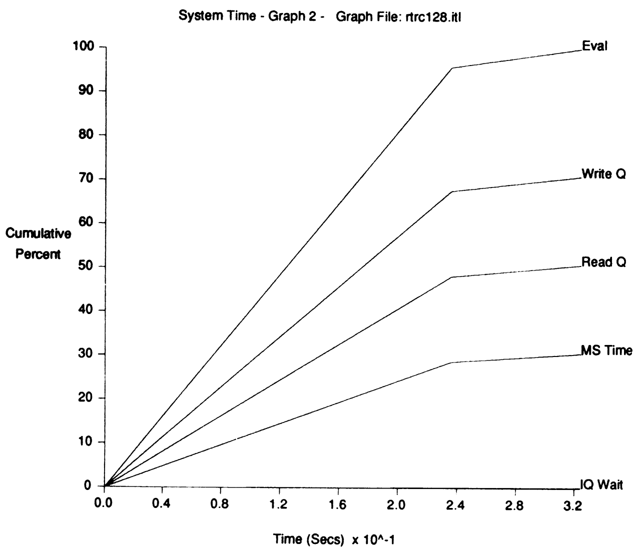


Figure 7. Double Recursive Trapezoidal Integration TR Graphs.







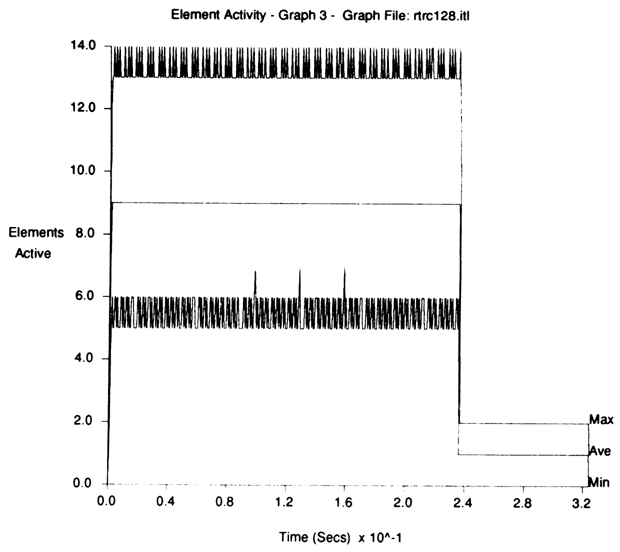
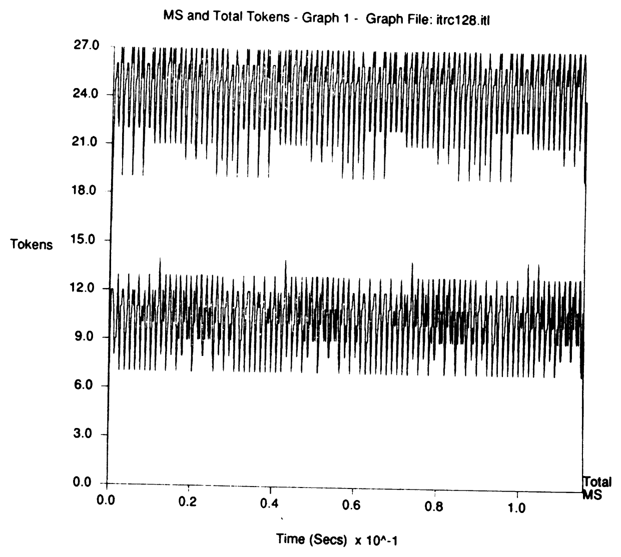
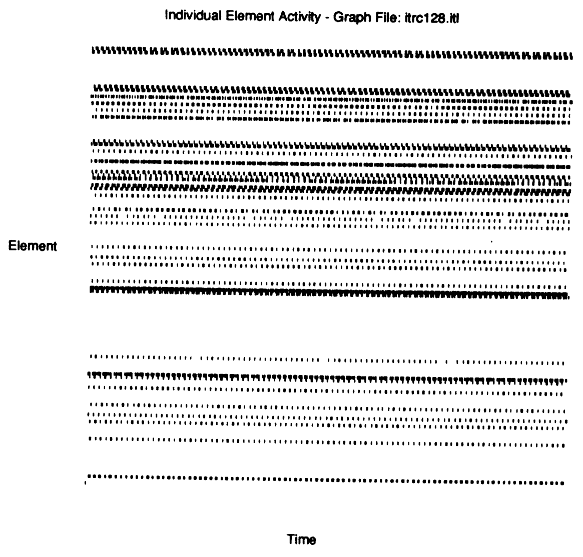
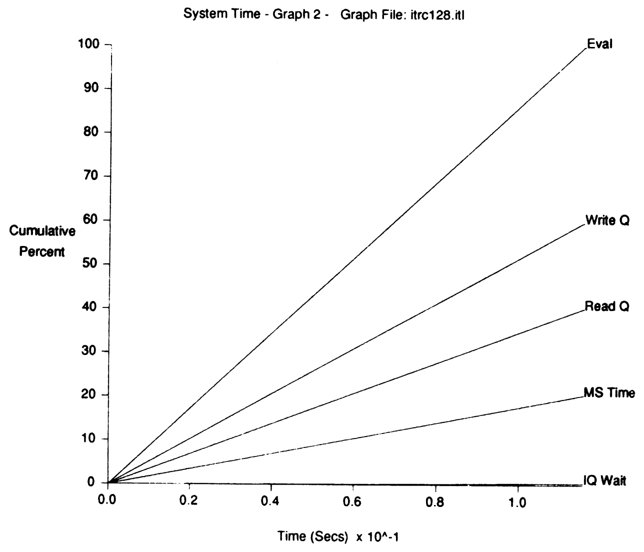


Figure 8. Single Recursive Trapezoidal Integration RTR Graphs.







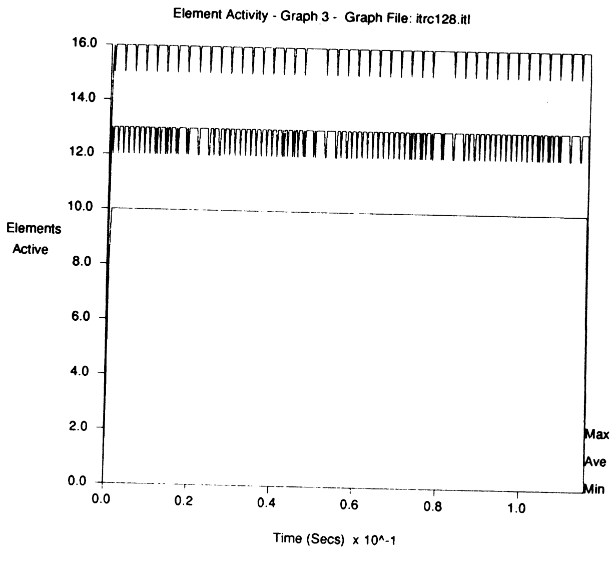


Figure 9. Iterative Trapezoidal Integration ITR Graphs.

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	0.037 secs	0.054 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 68% 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 mode is appropriate. Their performance may be summarized by the following table:

	TR	ITR	RTR
Total Execution Time in seconds	0.028	0.115	0.324
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 times that number. The speedup, however, is less than five times.

6. CONCLUSION

The architecture has been used to execute a number of other programs which require both static and dynamic models. These include object recognition programs using a laser range finder [16], a robot manipulator control, digital filter and logic simulation problems. The hybrid architecture is able to provide an efficient implementation because it allows both data-flow models to coexist.

The simulations illustrate that the hybrid architecture attracts the advantages of both conventional models. The emulation facility being constructed should allow much larger programs to be executed and traced, which will allow the scheme to be applied to some "real world" problems. 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 computation algorithms
- Some experimental expert systems
- High speed digital logic simulation
- Real time computer generated imagery

ACKNOWLEDGEMENTS

The authors wish to acknowledge the support of the project team, and in particular the help of Mr Mark Rawling who has provided much assistance with all aspects of this work. The Parallel Systems Architecture project is a joint project between the Commonwealth Scientific and Industrial Research Organisation (CSIRO) and the Royal Melbourne Institute of Technology (RMIT).

REFERENCES

1. D. A. Abramson, G. K. Egan, M Rawling and A. Young, The RMIT Data Flow machine: The Architecture,

Department of Communication Engineering Technical Report, TR112061R, Royal Melbourne Institute of Technology, Melbourne (1979).
2. Arvind and D. E. Culler, Data-flow Architectures, Laboratory for Computer Science, MIT technical report MIT/LCS/TM-294 (1986).
3. Arvind and R. A. Iannucci, A Critique of Multiprocessing Von Neumann Style, *Proc. 10th Ann. Int'l Symp. Computer Architecture, Stockholm,* pp. 426–436 (1983).
4. Arvind and K. P. Gostelow, The U-Interpreter, *Computer,* **15**, 2, pp. 42–50 (1982).
5. M. Cornish, D. W. Hogan and J. C. Jensen, The Texas Instruments Distributed Data Processor, *Proc. Louisiana Computer Exposition, Lafayette, La.,* pp. 189–193 (1979).
6. A. L. Davis, The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine, *Proc. 5th Ann. Symp. Computer Architecture, New York,* pp. 210–215 (1978).
7. J. B. Dennis and D. P. Misunas, A Preliminary Architecture for a Basic Data-Flow Processor, *Proc. 2nd Ann. Symp. Computer Architecture, New York* (1975).
8. 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* (1980).
9. J. B. Dennis, G. R. Gao and K. W. Todd, Modeling the Weather with a Data Flow Supercomputer, *IEEE Trans. Computers,* **C-33** 7, pp. 592–603 (1984).
10. G. K. Egan, Data-flow: Its Application to Decentralised Control, Ph.D. Thesis, Department of Computer Science, University of Manchester (1979).
11. G. K. Egan, C. Baharis, M. Rawling and A. Young, RMIT Data-flow Project, *IREE Proc. 2nd Australian Computer Engineering Conference, Sydney* (1986).
12. J. Gurd and I. Watson, Data Driven Systems for High Speed Parallel Computing – part 2L Hardware Design, *Computer Design,* pp. 97–106 (1980).
13. A. Plas *et al*, LAU System Architecture: A 'Parallel Data-driven Processor Based on Single Assignment, *Proc. 1976 Int'l Conf. on Parallel Processing,* pp. 293–302 (1976).
14. V. P. Srini, A Message-based Processor for a Data-flow System, *Proc. 1984 Int'l Workshop in High Level Computer Architecture, Los Angeles,* pp. 1.10–1.19 (1984).
15. V. P. Srini, An Architectural Comparison of Data Flow Systems, *IEEE Computer,* pp. 68–87 (1986).
16. G. K. Egan and C. P. Richardson, Object Recognition Using a Data-Flow Computing System, *Microprocessing and Microprogramming* **7**, North Holland (1981).
17. K. S. Weng, Stream oriented Computation in Recursive Data-Flow Schemas, Technical Memo 68, Laboratory for Computer Science, Massachusetts Institute of Technology (1975).

APPENDIX

Performance Graphs Generated by the Simulator

These graphs are Figs. 5, 6, 7, 8 and 9 in the previous pages.