NASA Technical Memorandum 89434

A Static Data Flow Simulation Study at Ames Research Center

Eric Barszcz and Lauri S. Howard

June 1987

National Aeronautics and
Space Administration

NASA Technical Memorandum 89434

A Static Data Flow Simulation Study at Ames Research Center

Eric Barszcz,
Lauri S. Howard, Ames Research Center, Moffett Field, California

June 1987

NASA
National Aeronautics and Space Administration

Ames Research Center
Moffett Field, California 94035

ABSTRACT

Demands in computational power, particularly in the area of computational fluid dynamics (CFD), have lead NASA Ames Research Center to study advanced computer architectures. One architecture being studied is the static data flow architecture based on research done by Jack B. Dennis at Massachusetts Institute of Technology. To improve understanding of this architecture, a static data flow simulator, written in Pascal, has been implemented for use on a Cray X-MP/48. A matrix multiply and a two-dimensional fast Fourier transform (FFT), two algorithms used in CFD work at Ames Research Center, have been run on the simulator. Execution times can vary by a factor of more than 2 depending on the partitioning method used to assign instructions to processing elements. Service time for matching tokens has proved to be a major bottleneck. Loop control and array address calculation overhead can double the execution time. The best sustained MFLOPS rates were less than 50% of the maximum capability of the machine.

INTRODUCTION

Demands in computational power at NASA Ames Research Center, particularly in the area of computational fluid dynamics (CFD), have lead Ames Research Center to study advanced computer architectures. One architecture studied is the static data flow architecture based on research done by Jack B. Dennis at Massachusetts Institute of Technology (MIT) (ref. 1). At Ames Research Center, a static data flow machine is being simulated and evaluated with respect to CFD.

The simulator is written in Pascal and executes on a Cray X-MP/48. The Cray permits larger problems than are typically simulated, allowing better evaluation of the architecture. The data flow machine presented in this paper is a modification of the architecture presented in reference 1. It incorporates a network connecting array memory (AM) modules and a separate network connecting processing elements (PE).

The paper contains: an overview of data flow, a description of the architecture simulated, information about the simulator itself, discussion on the generation and partitioning of code, justification of different service times used for token matching, results of simulation runs on a matrix multiply and two-dimensional fast Fourier transform (FFT), and a discussion of problem areas for these algorithms.

DATA FLOW

In a data flow architecture, instruction execution is determined by the presence of data, not a program counter. As soon as all data for an instruction is present, it is executed without regard for its position in the program.

A data flow program is represented by a data flow graph indicating the data dependencies. The graph is composed of two parts, nodes and directed arcs. Nodes represent instructions to be executed and arcs represent data dependencies between nodes. During execution of a data flow program, a node "fires," consuming input data and generating a result. Tokens carry copies of the result along output arcs to dependent nodes. A node is enabled or ready to fire when there are tokens on all input arcs. For a more complete description see reference 1.

There are two approaches to the implementation of instruction level data flow. One, static data flow, allows at most one token on an arc at any given instant (ref. 2). The other, dynamic tagged token data flow, allows multiple tokens on an arc (refs. 3-5). In this study, static data flow is implemented using instructions bound to processing elements at compile time. Acknowledge signals are sent from the consumer to the producer after an instruction has fired.

An instruction contains opcode, enable and reset counts, operands and destination addresses. The enable count records the number of outstanding operands. Space for operands is allocated inside the instruction. The acknowledge signals prevent operands being overwritten. The reset count indicates the number of destination addresses. The destination addresses contain locations of instructions that receive either a copy of the result or an acknowledge signal (ref. 1).

MACHINE ARCHITECTURE

Static data flow is implemented on a variety of machine architectures. In this study, the machine architecture has four basic elements: PE, AM modules, PE network and AM network. It is a modification of the architectures proposed in references 1 and 6. The simulator allows up to 1024 PE in the machine. A group of four processing elements share an AM module. Array Memory modules store array values and are used for I/O. Input values are placed in AM prior to execution of the data flow graph. Output values are extracted from AM after the simulation. The PE network provides communication among PE while the AM network does the same for AM modules. The networks are hybrid packet switched Omega networks (refs. 7 and 8). The PE and networks are assumed to have a 50-nsec cycle time. Figure 1 gives an overview of the machine architecture.

Processing Element

Each PE is composed of an update/matching unit, fetch unit, functional units, token generator unit, send and receive units, an instruction store, an enabled instructions queue, and a port to the local AM module (fig. 2).

The update/matching unit accepts tokens from the receive unit and the token generator unit. The instruction associated with the token's address is determined, and its enable and reset counts are fetched from the instruction store. If data is present, it is stored in the appropriate operand slot in the instruction at this time. The enable count is decremented and checked. When the enable count reaches zero, the instruction is enabled. The address of the instruction is placed in the enabled instructions queue, and the reset count is stored in place of the enable count.

The fetch unit removes addresses of enabled instructions from the enabled instructions queue. The unit then fetches the opcode and operands from instruction store. The opcode is partially decoded and a packet containing the opcode, operands and instruction address is passed to the appropriate functional unit or AM port.

The functional units are composed of two floating-point multipliers, a floating-point adder, and an arithmetic and logic unit (ALU). Each of the floating-point multipliers is assumed to be capable of 2.5 MFLOPS and the floating-point adder capable of 5 MFLOPS. (The Weitek chips WTL 1064 and WTL 1065 could be used as the multiplier and adder, respectively.) The ALU handles all other operations. All operations are done using 64-bit operands. Results and the associated instruction addresses are sent from the functional units to the token generator unit.

The AM port is the only link between the PE and AM. All AM requests are sent to the port from the fetch unit. When a request is satisfied, the result with its instruction address is sent to the token generator unit.

The token generator unit takes a result and fetches the associated destination address list. One token is created for each destination address. If the address is a local instruction, the token is passed to the update/matching unit. Otherwise, the token is passed to the send unit.

The send and receive units are the interface between PE and PE network. The send unit places tokens on the PE network and the receive unit removes them. When a token is removed from the network, it is passed to the update/matching unit.

Array Memory

Data flow does not have the classical concept of data stored in writeable memory. Instead, all variables and arrays are treated as values on tokens which are generated, never modified, and then consumed. The large amount of data on array tokens makes this difficult to implement. One approach is presented in reference 9. In this study, AM is used to store array values indefinitely.

Read and write nodes are the two node types that deal with the AM. Whenever a value is needed, the physical location is calculated and passed to one of these nodes. The read or write request is then sent to the local AM module.

All AM requests pass through the AM port. Each request has an opcode to describe the request, a physical location, and an associated instruction address. A write request also includes data. The high-order bits of the physical address are used for the AM module address. Each module can determine whether or not the address being referenced is local to the module.

Local requests are immediately satisfied while remote requests cause packets to be generated for the AM network. Remote writes are nonblocking; the associated instruction address is immediately sent to the token generator unit while the packet is waiting to be placed on the AM network. The writes do not cause read/write races because the AM modules use I-Structure Storage (ref. 3). Remote reads are blocking; a copy of the request is placed in a delay queue until a response returns from a remote AM module. When the response arrives, the request is removed from the delay queue. The data with its associated instruction address is then sent to the token generator unit.

The way AM is structured and connected has several advantages. First, only one copy of an array is needed. Second, the interface between PE and AM is very simple. There are only two requests the PE can make and they all pass through the AM port. Third, with multiple PE sharing an AM module, the AM network is smaller than the PE network and so has a smaller network delay. Fourth, having all remote AM requests handled on a separate network reduces the traffic on the PE network.

Networks

Both the PE network and the AM network are hybrid packet-switched omega networks. They are composed of 2- by 2-router nodes with 16-bit-wide data paths. Each input port can buffer a 96-bit packet. A packet header contains 32 bits of information broken into 10 bits for PE identification, 12 bits for instruction identification, 2 bits for operand port selection, and 8 bits to carry computation error codes (e.g., divide by zero). Packets, which consist of a header and data or just a header, are called data and acknowledge signals, respectively. A data signal contains a header with a nonzero port selection and 64 bits of data. An acknowledge signal is a header with a port selection of zero. Since the data paths are 16 bits wide, a packet is broken into slices. The first slice, called the destination slice, contains network routing information. The other slices of a packet follow the destination slice contiguously through the network. It is possible to move a slice between nodes every 50 nsec.

Conflicts can arise when packets travel through the network. A node conflict occurs when two packets in a node request the same output line simultaneously. Round robin selection is used to determine which packet advances. A line conflict occurs when a packet requests a line currently in use. It is resolved by making the packet wait until the line is idle. A block conflict occurs when a packet requests an input port already buffering another packet. Block conflicts are resolved when the destination slice of the buffered packet moves to another node. At this time, the blocked packet can start to advance. Each slice in the packet will occupy the location in the buffer held by the equivalent slice in the previous packet.

The method used to move packets through the network is similar to Virtual Cut-Through (ref. 7). The difference is that block conflicts are resolved using partial cut throughs. A packet can advance any time the line is not busy and the receiving port does not contain a destination slice for another packet. This has its greatest advantage when the network is lightly loaded. At best, it is about six times faster than regular packet switching for transmission of data signals. In the worst case, it is the same as normal packet switching.

SIMULATOR

The simulator is written in Pascal and executes on a Cray X-MP/48. It can simulate between 1 and 1024 PE in powers of 2. The data flow graph can be partitioned to use any number of PE between 1 and 1024. A typical simulation takes 30 to 760 Cray CPU sec. The ratio of simulated machine time to Cray time ranges from 1:25,000 to 1:750,000 depending on the number of PE simulated and the data flow graph.

Besides the ability to change the number of PE simulated, the following machine attributes can be varied:

1. Number of PE associated with each AM module.

2. Size of the local Array Memory modules.

3. Number and type of functional units.

4. Service time and maximum queue length for all units in the PE.

5. Cycle times for PE and networks.

6. Data path widths between nodes in the networks.

The simulator reads the machine attributes and checks for initial values to load into AM. It then loads the partitioned data flow graph into instruction store. A token is placed in the update/matching queue of PE zero. Execution of the data flow graph commences when its root node is enabled by the token.

During execution of the data flow graph, any of 30 machine opcodes can be executed. Real results are computed to check the correctness of the graph. The simulator does not handle computation errors; if one occurs, the simulation aborts. Three other opcodes are used solely by the AM modules for remote read/write requests.

All I/O is handled through AM. Input is achieved by placing values in AM before execution of the data flow graph. Results for output are placed in AM and extracted after the simulation is complete.

BENCHMARKS

Initially, ARC3D (ref. 10) and large eddy simulation (LES) (ref. 11) were considered as potential benchmarks. After the simulator was written, it was estimated one complete simulation of ARC3D would take 625 hr of Cray X-MP/48 time. At this point, it was noted that groups of benchmarking kernels already exist. Kernels representative of CFD work done at Ames Research Center have been assembled for the Numerical Aerodynamic Simulation (NAS) project (ref. 12). The data flow characteristics of the kernels were analyzed. Because of their high degree of parallelism, the matrix multiply, and two-dimensional FFT were chosen as benchmarks. These benchmarks can be expected to use a static data flow machine more efficiently than ARC3D and LES.

Two data flow graphs exist for each benchmark. A fully unrolled version of the matrix multiply calculates all inner products for the result matrix in parallel. A partially unrolled version calculates the inner products for a single column simultaneously. For the two-dimensional FFT, a fully unrolled version operates on all dual node pairs in all columns in parallel. A partially unrolled version operates on all dual node pairs in a single column simultaneously. Because of the symmetry in the two-dimensional FFT algorithm, machine code is generated to perform forward FFTs on columns only.

The large data flow graphs combined with human error make the use of code generators necessary. High degrees of parallelism allow the code generators to replicate and link many small sections of code. The generators also allow easy variation in problem size. Output from the generators is partitioned for use in the simulator.

Partitioning

The partitioning of instruction nodes among PE is an important and difficult problem. Utility of the machine is determined by the node assignments. Since optimal partitioning is difficult, an acceptable solution is an automatic partitioner that runs fast and produces execution times close to, or better than, a hand partitioning for the same data flow graph.

Hand partitioning of the benchmarks is done by looking at the overall structure of the data flow graph and assigning large blocks of replicated code to different PE. This is accomplished by the code generators assigning nodes as the machine code is created. In contrast, automated partitioners have no knowledge of the overall structure.

Two automated partitioners were developed at Ames Research Center. Each partitioner makes two passes through the data flow graph. On the first pass, a depth first traversal is performed to gather information about the bottom of the graph and propagate it upward. The second pass is a breadth first traversal using the information acquired in the first pass to make intelligent decisions about node assignments. The two partitioners assign nodes to PE at different times. The early binding partitioner assigns nodes as soon as possible while the late binding partitioner delays assignment as long as possible.

The automated partitioners are relatively fast and work fairly well. The late binding partitioner partitioned a 7000 node graph in 17 sec of Cray time. For the matrix multiply benchmark, the execution times of the automatically partitioned code (late binding) versus hand partitioned code averaged 6.9% slower for the partially unrolled version and 41.7% faster for the fully unrolled version.

SERVICE TIME

Each PE with its two multipliers and one adder has a potential floating point capability of 10 MFLOPS. However, if it is not possible to keep the functional units busy, an execution rate of 10 MFLOPS will not be achieved. Analysis done prior to simulation indicates the update/matching unit has the greatest influence on the megaflop rating. Taking into account the service time required for each token and the average number of destination addresses per instruction, the update/matching unit cannot supply enabled instructions fast enough to keep the functional units busy.

The floating point units can accomplish 1 addition and 2 multiplications every 400 nsec. Each arithmetic operation requires a pair of operands. Thus, at least one pair of operands must be supplied every 100 nsec to maintain full use of the floating point hardware. Since each token contains at most one operand, the update/matching unit must process at least one token every 50 nsec to maintain full use.

To achieve a service time of 1 cycle/token, the update/matching unit must be pipelined and the instructions stored in banks of 25-nsec RAM. The 25-nsec RAM is necessary since the update/matching unit and the fetch unit must access operands in the same 50-nsec clock cycle. In the first 25 nsec, the update/matching unit reads the enable and reset counts and stores any data. At the same time, the fetch unit reads an opcode. In the second 25 nsec, the updated enable count from a previous token is stored by the update/matching unit and the fetch unit reads an operand.

Service times closer to 2 or 4 cycles/token are more reasonable. The update/matching unit must still be pipelined, but slower and cheaper RAM can be used. As the service time increases, the average number of destination addresses per instruction becomes more important (demonstrated in tables 1 and 2 based on one PE).

The benchmarks average between 3 and 3.5 destination addresses per instruction. This limits the floating point capability to a maximum of 6.7 MFLOPS. Work has been done at MIT to reduce the number of acknowledge signals used in the data flow graphs (ref. 13). This lowers the average number of destination addresses per instruction, reducing the work load on the update/matching unit. However, as the number of acknowledge signals is lowered, the number of potentially enabled instructions decreases because of the loss of pipelining in the data flow graph. In any case, the floating-point units are not fully utilized.

SIMULATION RUNS

The matrix multiply is simulated by multiplying a 16 by 8 matrix by an 8 by 4 matrix. The two-dimensional FFT operates on a 16 by 16 array of values. Both benchmarks are run using partially unrolled and fully unrolled versions. Table 3 contains the number of nodes in each version. The benchmarks are simulated using 1 to 32 PE by powers of 2. For the matrix multiply, hand and automated partitioners are used. For the two-dimensional FFT, only automated partitioners are used. Each partitioning is run with update/matching service times of 1, 2, and 4.

RESULTS

Figure 3 gives the speedup curves for the partially unrolled matrix multiply when partitioned using hand and automated partitioners. Hand-partitioned code performs better until a large number of PE are simulated. The crossover is due to the hand partitioner partitioning across all PE and ignoring increased network delay. In contrast, the automated partitioners take network delay into consideration when partitioning. Code generated by the early binding partitioner averaged 13.6% slower than hand partitioned code. Code generated by the late binding partitioner averaged 6.9% slower than hand partitioned code.

Figure 4 gives the speedup curves for the fully unrolled matrix multiply when partitioned using hand and automated partitioners. Automatically partitioned code performed better than hand partitioned code. Early binding code averaged 34.7% faster than hand partitioned code. Late binding code averaged 41.7% faster than hand partitioned code (in one case, it was 56.0% faster).

In both versions of the matrix multiply, there is a kink in the graph for the early binding partitioner at eight PE. This is the first point multiple AM modules are simulated. Read requests are delayed in the second AM module for a significant amount of time while waiting for responses. No attempt is made to make data local to a particular AM module. Although the number of array accesses does not change, as more AM modules are simulated, the delay queue in each is shorter.

Figures 5 through 8 show the execution times for each version of the benchmarks simulated using update/matching unit service times of 1, 2, and 4. For small numbers of PE, the completion time was proportional to the service time of the update/matching unit. As the number of PE increases, the effect of service time is reduced as network delay becomes the dominant factor.

Two other problems that relate directly to CFD performance on a static data flow architecture are loop control and array address calculations. Both of these affect the execution time of the program. The fully unrolled version of the matrix multiply is about 3 times faster than the partially unrolled version for an update/matching unit service time of 4 clock cycles (figs. 5 and 6). This speedup is due to the reduction in loop control overhead and location calculations for array elements.

Loop Control

A time/space tradeoff is involved when a loop is unrolled. Loop unrolling decreases execution time but increases the data flow graph size. This tradeoff must be considered carefully. Loop control grows linearly with the information passed into the loop whereas loop unrolling often grows nonlinearly. This is demonstrated by the matrix multiply.

The matrix multiply is an $O(N^3)$ algorithm. For a square order 100 matrix multiply, there are approximately 2 million nodes in the fully unrolled data flow graph versus 49 nodes in the fully rolled version. Complete unrolling is prohibitive for problems of this size. For smaller problems such as the 16 by 8 times 8 by 4, unrolling is a practical approach to execution. In this case, the partially unrolled version and the fully unrolled version almost balance, 1334 versus 1366 nodes, respectively. However, the fully unrolled version has megaflops ratings 2 to 3.5 times higher than the partially unrolled version.

The node overhead for loop control can be calculated in the following manner. Let k be the number of items requiring a merge/switch pair for loop control (ref. 1), then the maximum number of overhead instructions is

$$3K + 2 \sum_{i=0}^{|\overline{\log_4 k - 1}|} 4^i + 1$$

where $3k$ represents the merge/switch pairs and fanout nodes. The summation represents an upper bound on the fanout of the conditional. Log_4 is used because there is a maximum fanout of 4. The conditional is represented by 1. Therefore, loop control is $O(k)$ since it can be shown that the summation reduced to approximately $k/3$ by noting that it can be rewritten as

$$k * \left\{ \sum_{i=0}^{|\overline{\log_4 k}|} (1/4)^i - 1 \right\}$$

Given the overhead for loop control, one wants to keep the ratio of productive work to overhead as high as possible. The fully rolled matrix multiply is a good example where the number of nodes involved in loop control overwhelms the desired computation. Loop control accounts for 65% of the nodes in the data flow graph. Only two nodes in the graph are floating-point instructions. The ratio of floating-point operations to overhead is 1:7.5.

Loops should be unrolled where feasible to reduce overhead (ref. 14). How loops are unrolled deserves careful consideration. In general, areas of heaviest computation are the first to be unrolled. But, it is unlikely they can be completely unrolled for real CFD problems. If loop invariant values did not need the control structure around them, the amount of loop overhead could be cut down. Results presented in reference 15 show the execution time decreasing by a third when loop invariants are held on tokens which are not consumed when the instruction fires.

Array Address Calculations

Another problem facing CFD applications is the software calculation of array element addresses. In many cases, the overhead for address calculations overshadows the computation in which the elements are used. If the computation is unrolled, the addresses are calculated at compile time, thereby producing a significant saving in execution time.

In the matrix multiply case, the fully expanded version calculates all locations at compile time. In the fully rolled version, array element locations are calculated at run time. Each location calculation involves the execution of four nodes. For the square order 100 case, the fully unrolled version executes a total of 2 million nodes. The fully rolled version would execute over 8 million nodes for location calculations alone.

Software calculations are eliminated if each memory module has an array map and addresses are calculated in hardware. This is possible because the location of arrays are known at compile time allowing the generation of one map which is replicated across memory modules. When a node asks for an element of a particular array the address is calculated in hardware inside the AM module.

CONCLUSION

Static data flow shows good speedup curves for the benchmarks over a limited number of PE. However, von Neumann multiprocessors have demonstrated similar speedups over the same numbers of PE. The expected gain from using a data flow architecture did not appear. The simulations proved sensitive to the partitioning method used and the service time for the update/matching unit. The late binding partitioner achieved an average speedup of 41.7% over hand partitioning for the fully unrolled version of the matrix multiply. For small numbers of PE, the update/matching service time caused a doubling of the completion time as it was doubled. As the number of PE increases, the network delay becomes the dominant factor. A service time of one cycle per token will be expensive to implement and does not guarantee full utilization of the 10 MFLOPS floating-point capability. The best sustained megaflops rates were less than 50% of peak speed and trailed off as the number of processors increased, similar to the observations for von Neumann machines.

Loop control overhead and array address calculation are important issues in the execution time of CFD applications. Both cause degradation of the machine's performance. The partially unrolled two-dimensional FFT had an average execution time 14% longer than the fully unrolled version. The partially unrolled matrix multiply had an average execution time 198% longer than the fully unrolled version. However, for real CFD problems it is not feasible to fully unroll loops because of the size of the data flow graph. Therefore, time/space tradeoffs must be carefully considered when coding and compiling real CFD applications.

The static data flow machine studied here does not enjoy a significant advantage over current von Neumann multiprocessors. Using VLSI technology, the update/matching unit and networks can be improved. The automatic partitioners show that good speedup can be achieved without human intervention. Using locality of data may further increase the speedup. Modified loop control will reduce the overhead associated with loop invariants. Hardware calculation of array addresses will decrease the execution time. As the hardware and software matures, static data flow may become a more attractive alternative in the future.

REFERENCES

1. Dennis, J. B.: Data Flow Supercomputers. Computer, vol. 13, no. 11, Nov. 1980, pp. 48-56.

2. Dennis, J. B.; Gao, Guang-Rong; and Todd, K. W.: A Data Flow Supercomputer. Computation Structures Group Memo 213, MIT Laboratory for Computer Science, Cambridge, MA, March 1982.

3. Arvind; and Culler, D. E.: Tagged Token Dataflow Architecture. TR 229, Mass. Inst. of Tech., MIT Laboratory for Computer Science, Cambridge, MA, July 1983.

4. Gurd, J.; and Watson, I.: Preliminary Evaluation of a Prototype Dataflow Computer. Information Processing 83, IFIP, Elsevier Science Publishers B.V., North-Holland Publishing Co. (Amsterdam), 1983, pp. 545-551.

5. Yuba, T.; Shimada, T.; Hiraki, K.; and Kashiwagi, H.: Sigma-1: A Dataflow Computer for Scientific Computation. Computer Physics Communications, vol. 37, Elsevier Science Publishers B. V., North-Holland Publishing Co. (Amsterdam), 1985, pp. 141-148.

6. Gostelow, K. P.; and Thomas, R. E.: Performance of a Simulated Dataflow Computer. IEEE Transaction on Computers. vol. c-29, no. 10, Oct. 1980, pp. 905-919.

7. Kermani, P.; and Kleinrock, L.: Virtual Cut-Through: A New Computer Communication Switching Technique. Computer Networks, vol. 3, Elsevier Science Publishers, North-Holland Publishing Co. (Amsterdam), 1979, pp. 267-286.

8. Siegel, H. J.: Interconnection Networks for Large-Scale Parallel Processing. Lexington Books, D.C. Heath and Company, 1985.

9. Dennis, J.; and Gao, Guang-Rong: Maximum Pipelining of Array Operations on Static Data Flow Machine. Computation Structures Group Memo 233-1, Mass. Inst. of Tech. Laboratory for Computer Science, Cambridge, MA, Sept. 1984.

10. Pulliam, T. H.: Solution Method in Computational Fluid Dynamics. Lecture Notes, NASA Ames Research Center, Jan. 1986.

11. Rogallo, R. S.: Numerical Experiments in Homogeneous Turbulence. NASA TM 81315, 1981.

12. Bailey, D. H.; and Barton, J. T.: The NAS Kernel Benchmark Program. NASA TM 86711, 1985.

13. Ackerman, W. B.: Program and Machine Structure for Incredible Fast Computation. Preliminary Draft, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, MA.

14. Ackerman, W. B.: Efficient Implementation of Applicative Languages. Ph.D. Thesis, Mass. Inst. of Tech., 1984.

15. Shimada, T.; Hiraki, K.; Nishida, K.; and Sekiguchi, S.: Evaluation of a Prototype Data Flow Processor of the Sigma-1 for Scientific Computations. Proc. 13th Int. Symp. on Comp. Arch., 1986, pp. 226-234.

TABLE 1.- THREE ADDRESSES/INSTRUCTION

SERVICE TIME	TIME TO ENABLE 1 INSTRUCTION	MFLOPS
4	600 ns.	1.67
2	300 ns.	3.33
1	150 ns.	6.67

TABLE 2.- THREE AND ONE-HALF ADDRESSES/INSTRUCTION

SERVICE TIME	TIME TO ENABLE 1 INSTRUCTION	MFLOPS
4	700 ns.	1.43
2	350 ns.	2.86
1	175 ns.	5.71

TABLE 3.- NODE AND FLOATING POINT OPERATION COUNTS

	MATRIX MULTIPLY		1/4 2D FFT	
	FULLY UNROLLED	PARTIALLY UNROLLED	FULLY UNROLLED	PARTIALLY UNROLLED
NO. NODES	1366	1334	7018	549
FL. PT. OPS.	960	240	5120	320

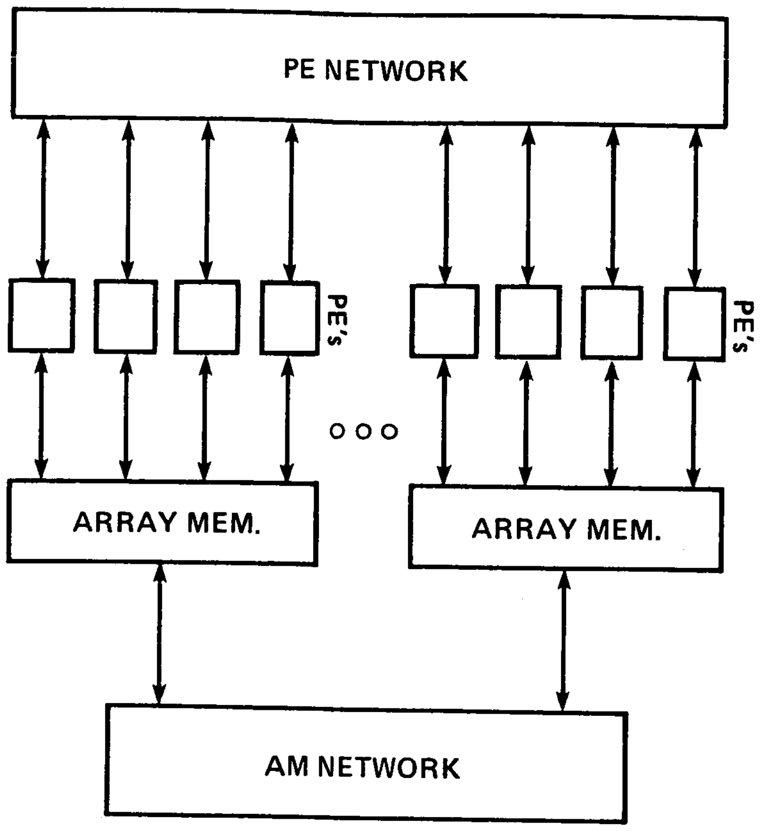

Figure 1.- Static data flow machine architecture.

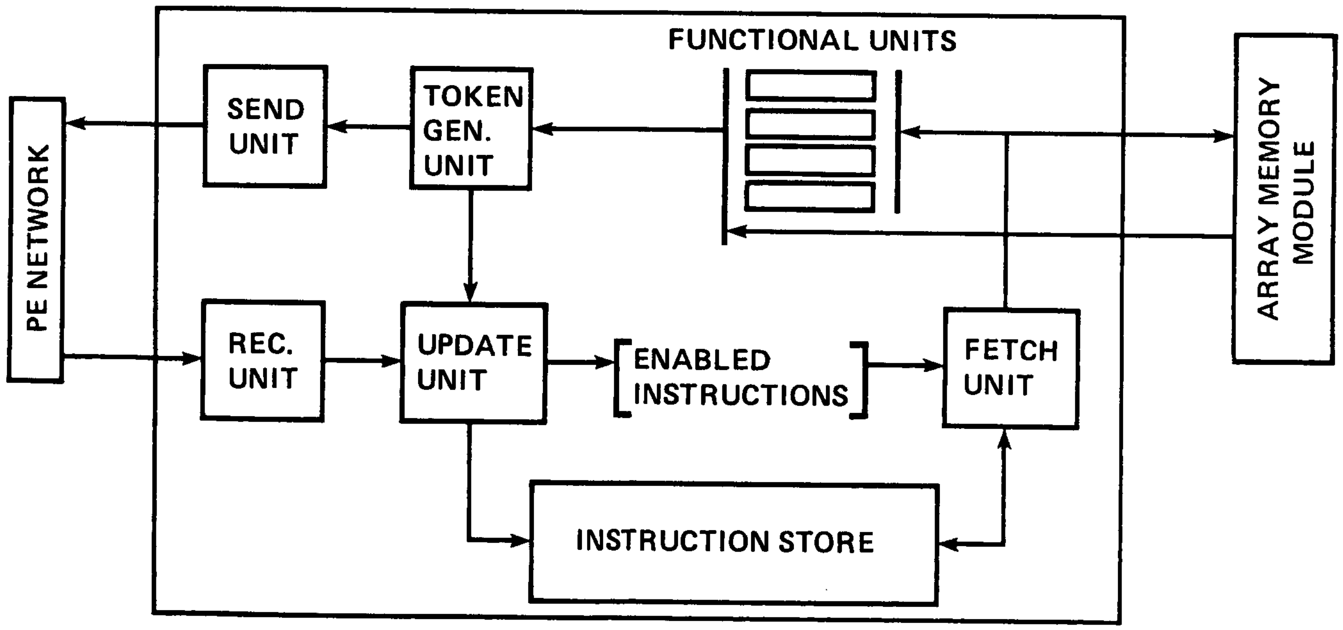

Figure 2.- Processing element block diagram.

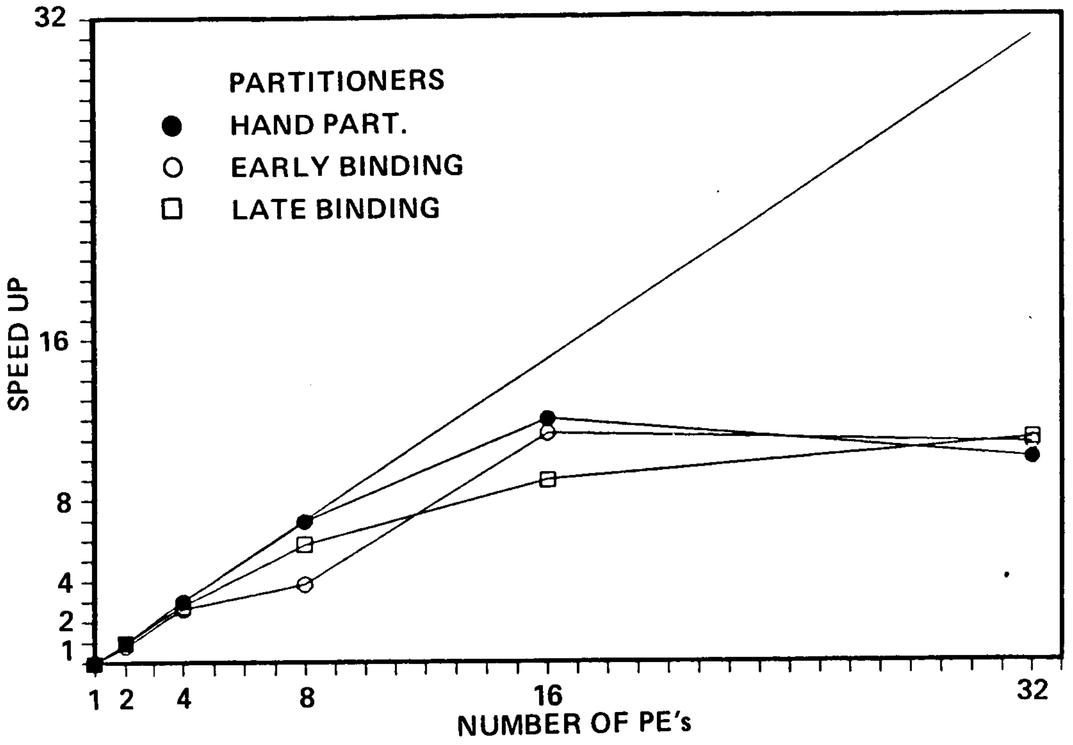

Figure 3.- Matrix multiply (partially unrolled, all partitioners, service time = 4).

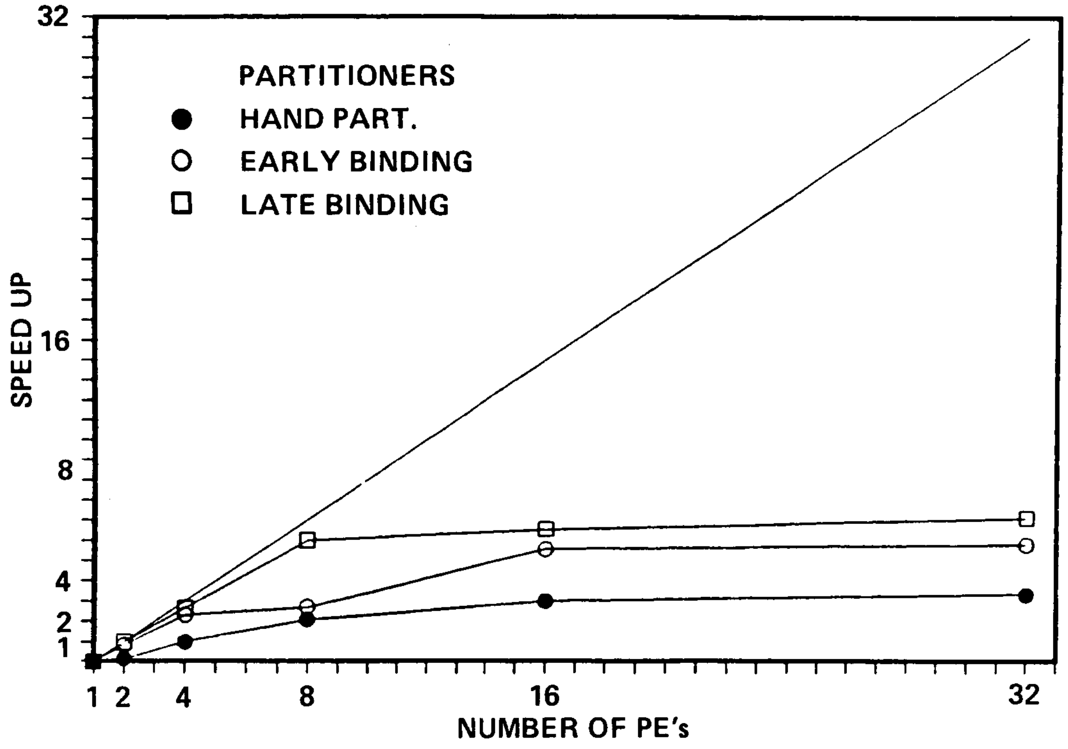

Figure 4.- Matrix multiply (fully unrolled, all partitioners, service time = 4).

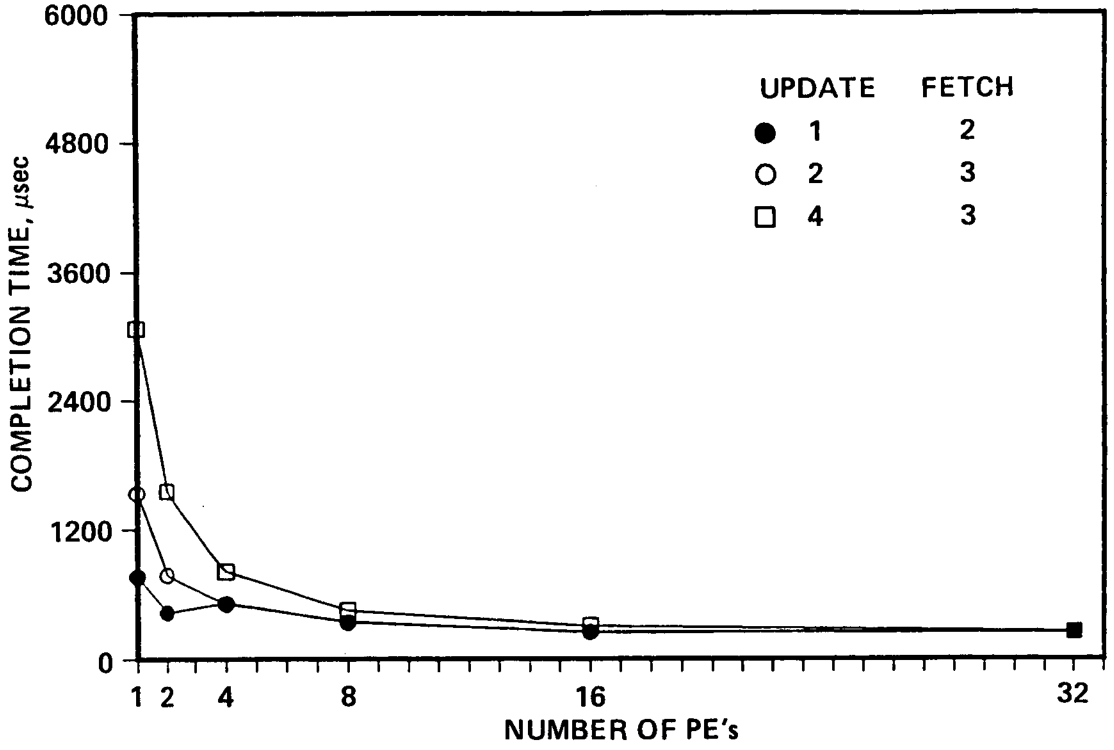

Figure 5.- Matrix multiply (partially unrolled, late binding partitioner, various service times).

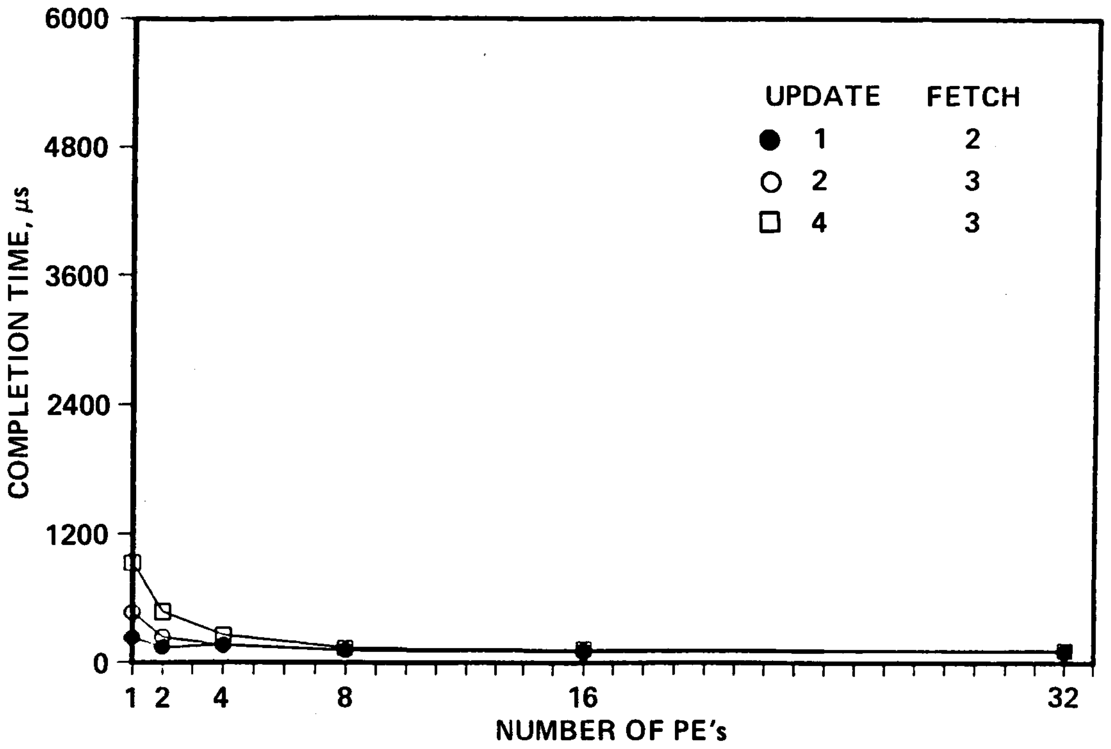

Figure 6.- Matrix multiply (fully unrolled, late binding partitioner, various service times).

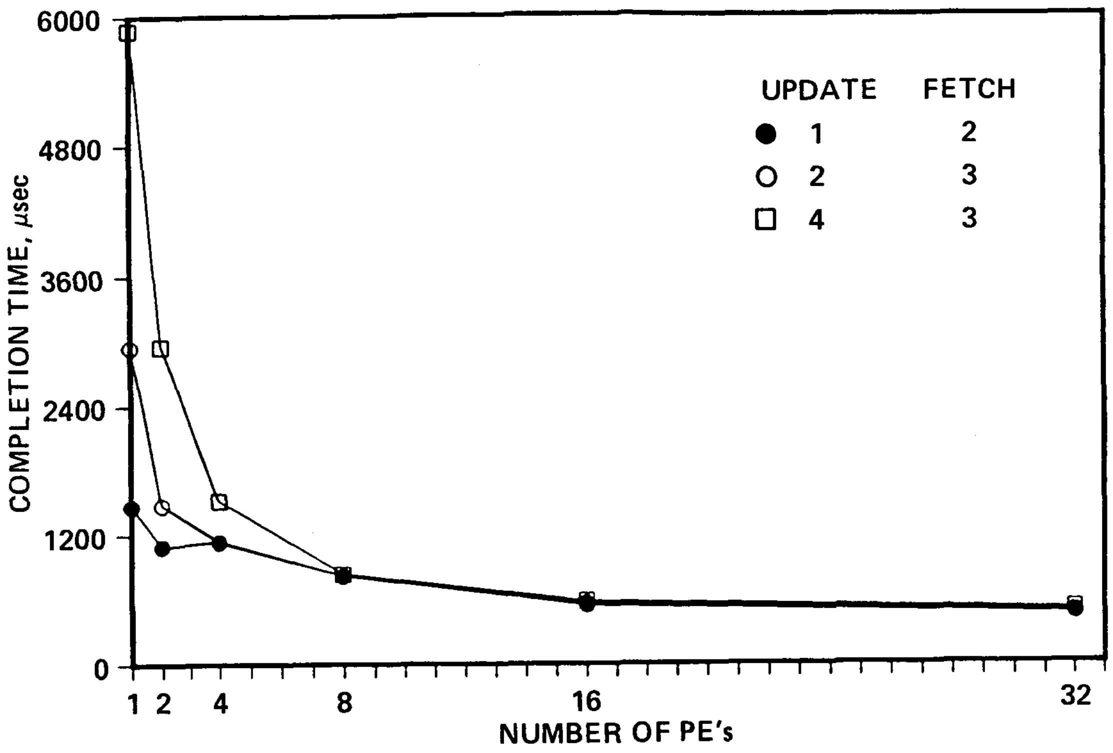

Figure 7.- Two dimensional FFT (partially unrolled, late binding partitioner, various service times).

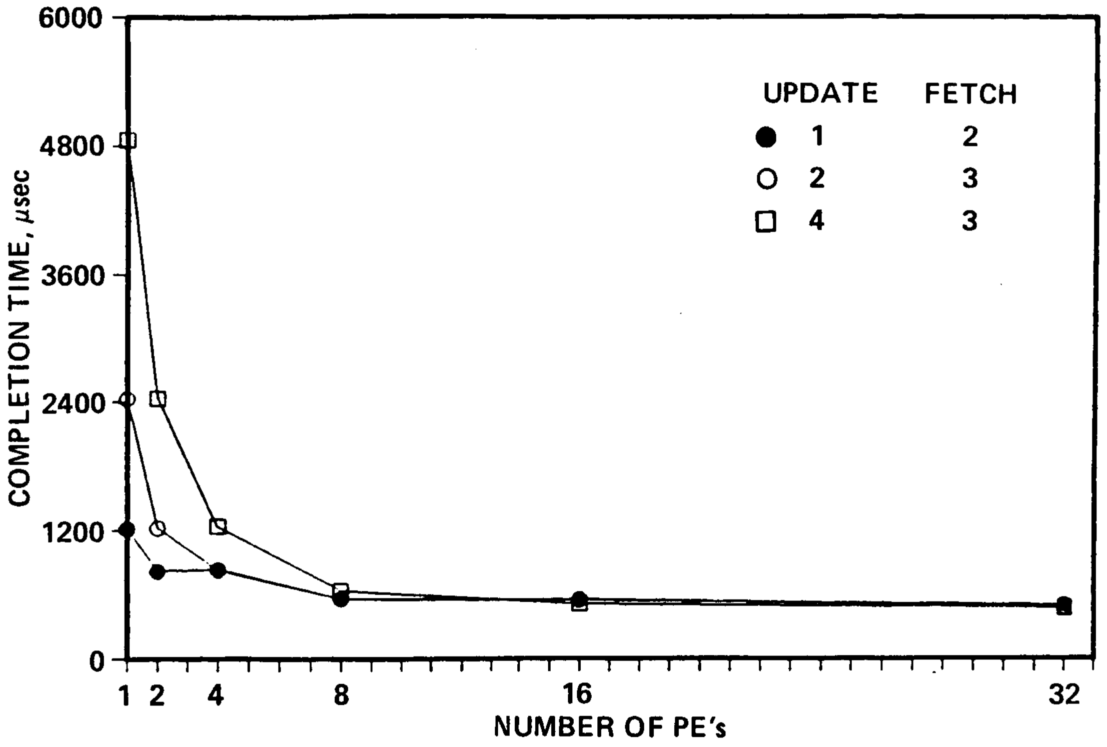

Figure 8.- Two dimensional FFT (fully unrolled, late binding partitioner, various service times).

Report Documentation Page

1. Report No. NASA TM-89434	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle A Static Data Flow Simulation Study at Ames Research Center	5. Report Date June 1987
	6. Performing Organization Code
7. Author(s) Eric Barszcz and Lauri S. Howard	8. Performing Organization Report No. A-87129
	10. Work Unit No. 505-01-00
9. Performing Organization Name and Address Ames Research Center Moffett Field, CA 94035	11. Contract or Grant No.
	13. Type of Report and Period Covered Technical Memorandum
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546	14. Sponsoring Agency Code

15. Supplementary Notes

Point of Contact: Eric Barszcz, Ames Research Center, M/S 233-14
Moffett Field, CA 94035 (415) 694-6014 or FTS 464-6014

16. Abstract

Demands in computational power, particularly in the area of computational fluid dynamics (CFD), have lead NASA Ames Research Center to study advanced computer architectures. One architecture being studied is the static data flow architecture based on research done by Jack B. Dennis at Massachusetts Institute of Technology. To improve understanding of this architecture, a static data flow simulator, written in Pascal, has been implemented for use on a Cray X-MP/48. A matrix multiply and a two-dimensional fast Fourier transform (FFT), two algorithms used in CFD work at Ames Research Center, have been run on the simulator. Execution times can vary by a factor of more than 2 depending on the partitioning method used to assign instructions to processing elements. Service time for matching tokens has proved to be a major bottleneck. Loop control and array address calculation overhead can double the execution time. The best sustained MFLOPS rates were less than 50% of the maximum capability of the machine.

17. Key Words (Suggested by Author(s)) Static data flow Computational fluid dynamics Loop control Array address calculation Simulator	18. Distribution Statement Unclassified-Unlimited Subject Category - 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	24	A02

NASA FORM 1626 OCT 86

For sale by the National Technical Information Service, Springfield, Virginia 22161