[14] J. R. McGraw and et al. SISAL: Streams and iteration in a single assignment language—language reference manual version 1.2. Technical Report M-146, Lawrence Livermore National Laboratory, 1985.

[15] J. M. Monti. Mds: A multiprocessor dataflow simulator. ACAPS Design Note 15, School of Computer Science, McGill University, Montreal, November 1990.

[16] J. M. Monti. Interprocessor communication supports for a multiprocessor dataflow machine. Master's thesis, School of Computer Science, McGill University, Montreal, March 1991.

[17] Krishnan Padmanabhan and Duncan H. Lawrie. Performance analysis of redundant-path networks for multiprocessor systems. *ACM Transactions on Computer Systems*, 3(2):117–144, May 1985.

[18] J. H. Patel. Processor-memory interconnections for multiprocessors. In *The Sixth Annual Symposium on Computer Architecture*, pages 168–177, New York, NY, April 1979.

[19] Daniel A. Reed and Dirk C. Grunwald. The performance of multicomputer interconnection networks. *IEEE Computer*, pages 63–73, June 1987.

[20] R. Tio. The A-code assembly language reference manual. ACAPS Design Note 02, School of Computer Science, McGill University, Montreal, July 1988.

Structures Group Memo 226, Laboratory for Computer Science, MIT, 1987.

[2] Arvind and R.S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[3] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[4] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. In *Proceedings of the Supercomputing '88 Conference*, pages 368–373, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

[5] D. M. Dias and J. R. Jump. Analysis and simulation of buffered delta networks. *IEEE Transactions on Computers*, C-30(4), April 1981.

[6] D. M. Dias and J. R. Jump. Packet switching interconnection networks for modular systems. *IEEE Computer*, 14(12), December 1981.

[7] Jack J. Dongarra. The LINPACK benchmark: An explanation. In C. D. Polychronopoulos, editor, *Proceedings of the 1987 Conference on Supercomputing, Athens, Greece*, pages 457–474, Berlin, June 1987. Springer-Verlag, LNCS-297.

[8] Tse-yun Feng. A survey of interconnection networks. *IEEE Computer*, pages 12–27, December 1981. Special issue on Interconnection Networks.

[9] J. T. Feo. An analysis of the computational and parallel complexity of the Livermore loops. *Parallel Computer*, 8(7):163–185, July 1988.

[10] G. R. Gao. A pipelined code mapping strategy for dataflow super computers. In *Proceedings of the Third International Conference on Supercomputing*, pages 209–215, May 1988.

[11] G. R. Gao, R. Tio, and H. H. J. Hum. Design of an efficient dataflow architecture without dataflow. In *Proceedings of the International Conference on Fifth-Generation Computers*, pages 861–868, Tokyo, Japan, December 1988.

[12] W.-K. Hong. IF1 parser for HDDG. ACAPS Design Note 01, School Of Computer Science, McGill University, Montreal, June 1988.

[13] C. P. Kruskal and M. Snir. The performance of multistage interconnection networks for multiprocessors. *IEEE Transactions on Computers*, C-32(12):1091–1098, December 1983.

structure memory operations.

The implementation of these mechanisms has been accomplished by the Interprocessor Communication Unit (ICU), yielding the McGill Multiprocessor Architecture. It is a modular and scalable system, that facilitates the inclusion of processors for increasing power, memory space and communications bandwidth. The main contribution of this implementation has been to keep the multiprocessor environment as transparent as can be, realizing interprocessor communications almost as if the PE boundaries were inexistent.

We have investigated the impact of long latency operations on the performance of multiprocessor dataflow programs. Our results are very encouraging. They have shown that the MMDA is a system that can effectively tolerate high latencies during interprocessor memory operations and that it can inexpensively support remote event synchronization. We have demonstrated that when the program contains enough fine-grain parallelism, it can absorb the effects of long latency operations, thereby featuring increasing execution speeds as additional processors were made available to the system. We have also shown that the performance can degrade in a considerable manner if the interconnection network becomes a bottleneck, thereby showing that it is a factor as critical as latency in the execution of an application. However, and overall, the MMDA performs remarquably well, achieving close to linear speedups, given a high bandwidth interconnection network, and a highly parallelizable application like some of the benchmark programs used in this work.

A brief survey of the related work that has been done in the field can be found in [16]. In that survey, we look at some of the advantages and drawbacks of our model in comparison with the mechanisms which are investigated in other dataflow projects, focusing on the types of interprocessor transfers and their implementation. It turns out that our approach is very competitive and in some cases more efficient.

7 Acknowledgments

I would like to express my gratitude to my supervisor Prof. G.R. Gao for his support and guidance in the development of this research. I also enjoyed the support from the other members of our research team— Advanced Computer Architecture and Program Structures Group at McGill University.

References

[1] Arvind and R. A. Iannucci. Two fundamental issues in multiprocessing. Computation

Dataflow programs are written in the forms of graphs, with a number of inputs, and outputs. Loops are coded by embedding cycles within the graph, that perform the body of the loop. All parallel operations within the loop are coded in different paths. When a long latency operation is executed within one path, finding other useful work is accomplished by switching to an alternate path. This process can be repeated as long as there are alternate paths available. Duplication of the loop body is one way of providing more paths to enlarge the "pool" of enabled instructions that can be fired at any given time. Intuitively, there should be at least enough paths to allow the first blocked path to become unblocked by receiving the acknowledgment from the network.

A problem arises, though, when the long latency operation is at a location where paths merge together. Such a situation is very likely to occur in some loops where the last instruction of the loop body to be executed is a Store. If the store is a remote operation, meaning that the result of the computation has to be stored in a remote PE, there are no other alternate paths to switch to (since its location is at the merging point within the graph). After "filling up" the graph, the processor has no other choice but to idle, which causes the performance to drop. We have tested this phenomena with our simulator, using the same benchmarks. The results show an increase of 10% to 15% in the total execution time of a program, when the store is a remote operation, in comparison with a local one. Unless some technique is found to optimize this degradation, it remains more effective to fetch data from a remote processor to ultimately store the result locally, than to do the opposite.

6 Conclusion

The dataflow concept has been attracting increasing attention as a radical alternative to the von Neumann architectures, emerging as an innovative model which offers simple yet powerful means of achieving highly parallel computations. The McGill Dataflow Architecture has been developed, to efficiently support this model of computation, based on the argument-fetching principle.

In a parallel system, communication between processors is one crucial problem to be solved, be it transfer of data or event synchronization. Any scalable multiprocessor must therefore address the fundamental problems of high latency memory operations, and the interprocessor synchronization of events. By providing the necessary supports for interprocessor communications to the MFDA, we have established a sound multiprocessor environment whose main features are transparency, flexibility, and an efficient use of the network and execution units of all the PEs. The key contribution of the approach is to separate the processing of remote signals from the main execution pipeline, and to provide the processor with flexible

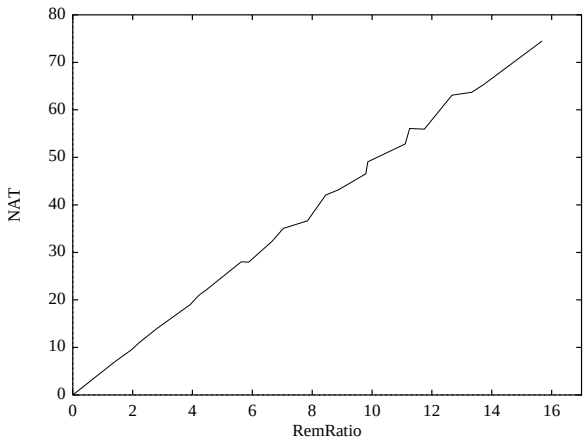


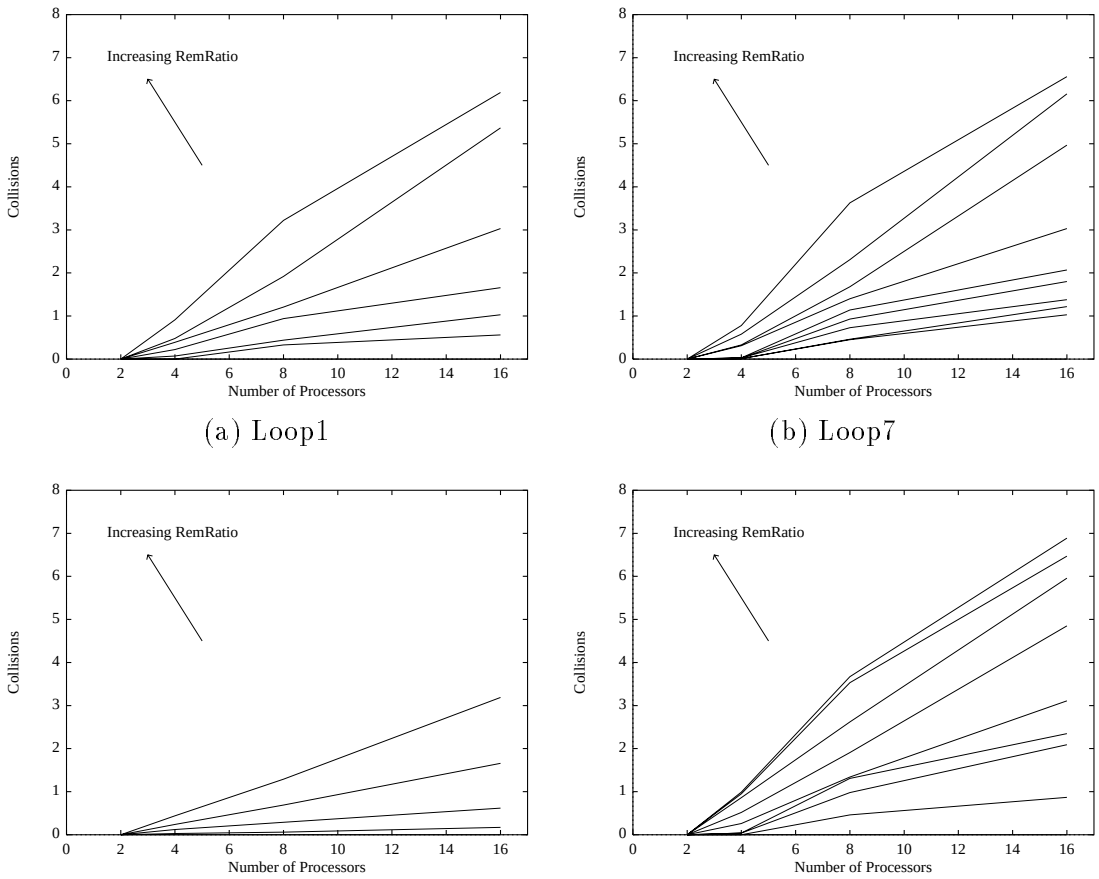
Figure 17: Network Average Throughput versus Remote-Operations Ratio

each loop body can be the basis of a good estimate of RemRatio. Finally, let us assume that the network has no MRT and thus execution can be performed under ideal circumstances. It becomes therefore possible to guess an *estimation of the network average throughput* (ENAT) required by the application, based on the estimated RemRatio. This is possible since each type of packet has a fixed known size. For instance, by plotting the values of NAT for which the network does not act as a bottleneck against the values of RemRatio, we obtain the graph shown in Figure 17. It can be observed that there is virtually a one-to-one correspondence which is independent of the type of application.

If such analysis can be done a-priori at compile time, a similar correspondence can be established to compute an ENAT. The compiler is therefore in a position to compare the ENAT against the MRT. This information can be a useful contribution to the process of estimating the optimal number of PEs required to execute the application program. For example, suppose the MRT for a given configuration is one fourth of the ENAT; this corresponds to a situation where the application overloads the network. A smaller configuration with a higher MRT can therefore constitute an interesting tradeoff to make a better usage of the network and obtain better performance.

Program Structure

Another observation relates to the way the position of remote memory operations within a graph can influence program performance. All our predictions, regarding the ability of the system to tolerate long latency operations, are based on the assumption that the PE can keep it's execution unit busy with other useful operations.



(a) Loop1 (b) Loop7

(c) Matmul (d) Saxpy

Figure 16: Global Collision Rate

can be improved by implementing various techniques such as more powerful switch technology, alternate paths or redundant paths through the network to decrease the number of collisions, and many others [8, 19, 5, 6, 13, 17].

A Look at Compiling Issues

An interesting point to make relates to the fact that the MRT is solely a characteristic of the network, and is independent of the application executing in the PE, as long as this latter does not generate memory hot spots (see Section 5.4.2). Let us assume that a compiler is given enough information about the network interconnecting the PEs, so that it has a good estimate of the MRT behavior in relation to the number of PEs, for well balanced applications. Let us also assume the compiler is able to evaluate the RemRatio of a given application by some analysis of the program structure. A simple static count of the number of Rloads within

congestion is measured by the *global collision rate* (CollRate) which reports the total number of collisions per cycle during which the network is in use. The CollRate variable is plotted in Figure 16 for each benchmark.

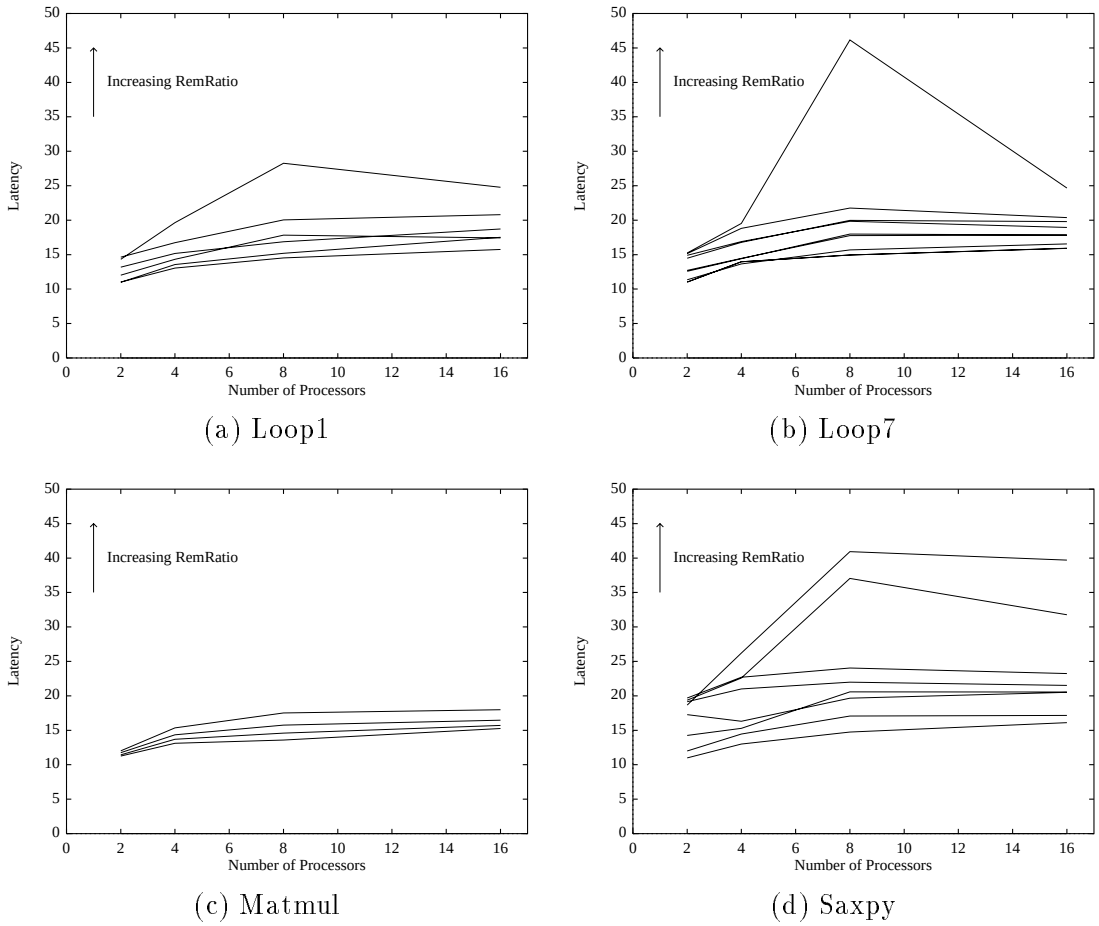
This measurement allows us to explain why the latency in a system of 8 PEs, for instance in the case of Loop7, is twice as long as the latency in a system of 16 PEs for the same RemRatio value, when in fact it is expected to be smaller. What we are actually measuring is the average amount of time that a packet is being delayed beyond the normal latency it is bound to incur. In order to have a monotonically increasing latency curve, the CollRate has to increase in the same proportion as the number of PEs, i.e. if there are x collisions in a system of n PEs, we expect to have $2x$ collisions in a system of $2n$ PEs to obtain a logarithmic increase of GlobLatency. As can be observed in Figure 16, CollRate does not increase proportionally to the system size for the two upper curves of Saxpy (RemRatio of 13.71% and 15.68%), the top curve of Loop7 (12.67%) and the top curve of Loop1 (13.33%).

Matmul's GlobLatency is a representative of a program which does not go beyond the network's MRT, and thus has a normal latency increase. Notice that for these programs, there is no relation between the latency of the memory requests going across the network, and the total execution time of the program. What we have observed is that if a program has enough parallelism, and if it does not require a bandwidth greater than the limit imposed by the network, then it can be expected to have linear speed-up.

5.5 Discussion

The simulations have shown that, under normal circumstances, long latency operations can be tolerated, and that they do not harm the performance of the program. We have observed that even for increased latencies, in larger systems, execution times decrease in the same proportion as the system size, and processor utilization remains almost constant. For these cases, virtually linear speed-up is achieved. For some other extreme cases, we have observed a loss of performance, but we have found that this loss is mainly due to an interconnection network overload, which has caused it to become a bottleneck to the computation.

Altogether, the key factor to achieve high performance is to program the applications in such a way so that: (1) the data partitioning does not require each processor to access the network beyond its MRT; and (2) the fine-grain parallelism inherent in the program is sufficiently exposed. These are reasonable assumptions to make, given an appropriate choice of network, with a high MRT, and an appropriate language support, such as dataflow languages, so as to allow the parallelism to be fully expressed. It turns out that the first assumption, about the network, is a difficult one to achieve. Network throughput, though,



(a) Loop1 (b) Loop7

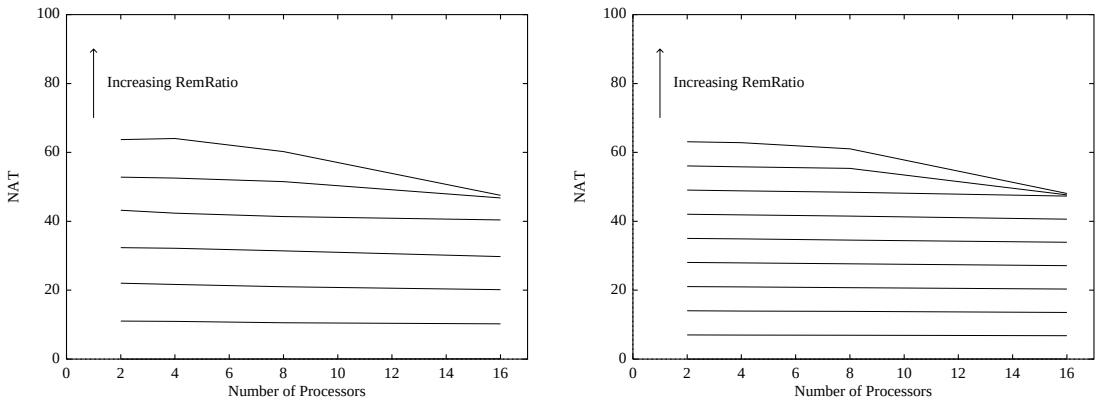
(c) Matmul (d) Saxpy

Figure 15: Global Average Latency

A different criteria, also often used, is the *delay*, defined as the period of time it takes for a message to go from an input port to an output port. The corresponding metric in the simulator is the *global average latency* (GlobLatency) which actually corresponds to two trips through the network.

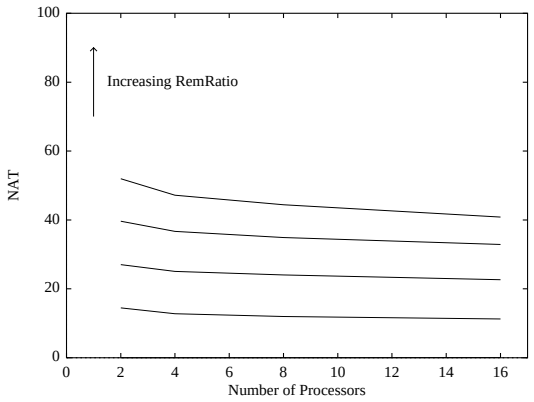
With the network we chose, the average latency of a packet is expected to increase as the logarithm of the number of processors. This assumption holds if the required network throughput is below the aforementioned MRT. Otherwise, the increase is considerably higher, as can be seen in Figure 15.

The anomalies of the top curves in Loop1, Loop7, and Saxpy, correspond to the simulations for which the network has been a bottleneck. The corresponding congestion causes the packets to stay longer in the switch buffers, thereby increasing the global average latency. This

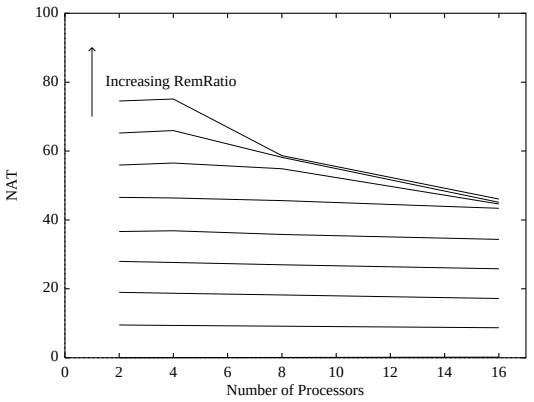


(a) Loop1

(b) Loop7



(c) Matmul



(d) Saxpy

Figure 14: Network Average Throughput

Their results show that even under these ideal conditions, the network features an upper bound on performance. There is a maximum rate of passing packets which is primarily a property of the network. We have observed the same phenomena with our simulations.

Therefore, an application consuming a bandwidth below the MRT corresponding to the system configuration can be expected to feature linear speed-up. Of course, this can only be achieved because the processor architecture can tolerate latency. Above the MRT, the number of collisions in the network will slow down the overall system and the performance will tend to decrease. This is exactly the kind of behavior we observed in Section 5.4.1.

5.4.3 Latency

In fact, the RemRatio indirectly measures the load that the PE imposes on the network. As will be discussed in the next subsection, there is a limit imposed by the interconnection network which depends on the size of the network. Once that limit is reached, the network becomes a bottleneck and the performance starts dropping. Below that limit, though, the program's execution time decreases in proportion with the number of PEs, thus yielding virtually linear speed-ups. This can be confirmed by observing the Figures 10, 11, and 13.

5.4.2 Network Throughput

The performance of a packet-switched interconnection network is usually characterized in terms of the *throughput*. Informally, it measures the amount of information that can be passed through it in a given period of time. The corresponding metric used by mds is the *network average throughput* (NAT). We have chosen a normalized metric because it gives us a measure of the amount of network bandwidth that the application has consumed in relation to its ideal performance.

Figure 14 shows the plots of NAT versus number of PEs, for all values of RemRatio. We can observe that, within the same system configuration, the throughput increases proportionally with RemRatio until it reaches a limit. We will call this limit the *maximum reachable throughput* (MRT). Below this MRT, the throughput is practically constant, independently of the size of the system, for each fixed value of RemRatio. The reason is that the load imposed by each PE to the network is a characteristic of the program. Since the programs do not change—only the size of the problem they are solving—the load on the network remains a constant. This is also valid because the destinations of all packets in our benchmark programs are equally distributed in the same way for all system sizes.

This observation confirms our earlier remarks about a limit imposed by the network, which is inversely proportional to its size. This is caused by the collision of packets within the switches of the network. Increasing the packet flow through the links automatically increases the probability of having a collision, which explains why the network can not reach its peak bandwidth. Furthermore, increasing the number of stages increases the probability for a packet to be blocked and thus reduces the overall throughput. The MRT therefore decreases as the network size increases, following a monotonically decreasing curve which is a characteristic of the network.

The same observation was made by Dias and Jump in a study of packet switched interconnection networks [6]. Their analysis is based on a network which is never "starving", i.e. packets are presented to the network at the maximum rate that the network can accept them. The destination addresses are uniformly distributed over the set of output port addresses.

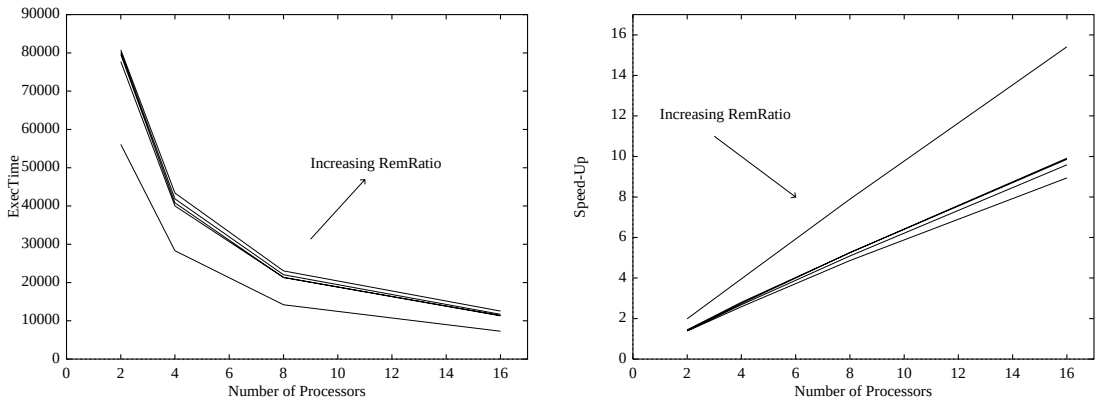
Benchmark	Remote-Operation Ratios (RemRatio in %) and Total Network Accesses (NetAcc in Ks)						
Loop1	0.00	2.22	4.44	6.66	8.88	11.11	13.33
	0	1.6	3.2	4.8	6.4	8.0	9.6
Loop7	0.00	2.82	4.22	7.04	8.45	11.26	12.67
	0	3.2	4.8	8.0	9.6	12.8	14.4
Matmul	0.00	4.41	6.62	8.82	13.23	15.44	17.64
	0	8.2	12.3	16.4	24.6	28.7	32.8
Saxpy	0.00	1.96	5.88	7.84	9.79	13.71	15.68
	0	0.64	1.92	2.56	3.2	4.48	5.12

Table 2: Remote-Operations Ratios Used to Test Each Benchmark

Both Loop1 and Loop7 have similar program structures and a maximum RemRatio neighboring the 13% of instructions. For these benchmarks, a virtually linear improvement in performance is observed for all curves, except for the extreme cases where the programs are importing more than 88.85% of their data, for Loop1, and more than 88.87%, for Loop7. Aside from these extreme cases, which are illustrated in the speed-up figures by the two lowest curves of each graph, we can observe that communication costs and long latency operations have practically no effect on the performance of the programs. This is a sign that the MMDA can effectively tolerate latency.

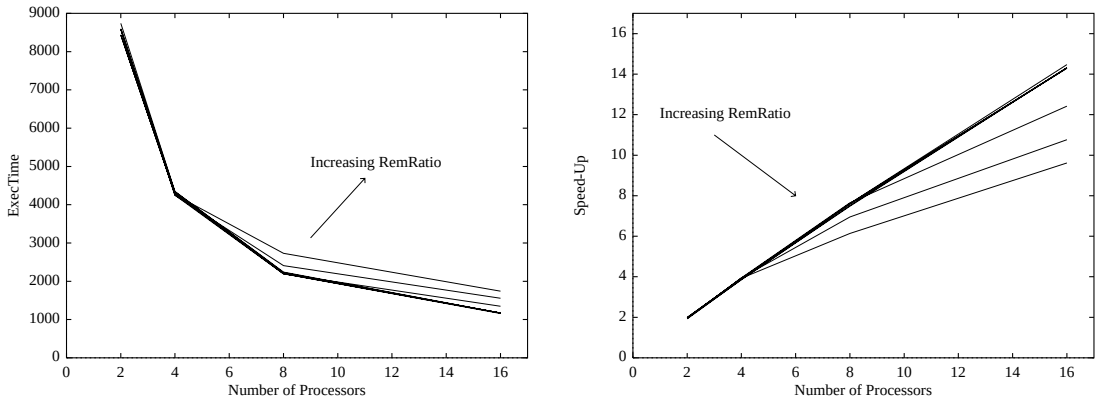
Saxpy has a very similar behavior: the performance of the program starts degrading once the RemRatio hits 11%. However, both Loop1 and Loop7 show performance losses in the 16 PEs configuration only, whereas Saxpy's degradation starts with the 8 PEs configuration, with a RemRatio value of 15.68%. Intuitively, the observations tend to lead us to the conclusion that a system of 8 PEs can execute an application without loss of performance, if the RemRatio is kept under 15%, whereas on a system of 16 PEs, that limit is 11%.

The results concerning Matmul show a distinct behavior than the other benchmarks. Execution time increases drastically as soon as the program starts fetching remote data—from 56.1K to 77.7K cycles, an increase of 41.8%—to later stabilize at that level. The RemRatio variation has little effect over the overall performance thereafter. This particular behavior stems from the way the program is structured: as opposed to the other benchmarks, which feature a single loop executing a portion of code, Matmul is programmed based on three nested loops, the innermost of which requires two loads at each iteration. The loss on performance is more precisely due to the fact that there are not enough instructions to execute while a particular path in the graph is blocked due to a long latency operation. This situation does not occur with the other benchmarks because of the more numerous paths available through the graph.



(a) Execution Time (b) Speed-Up

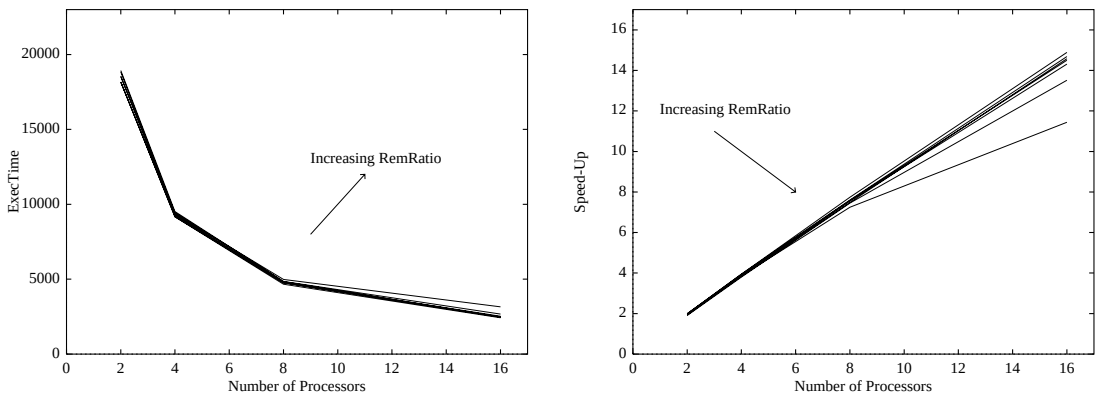
Figure 12: Performance of Matmul



(a) Execution Time (b) Speed-Up

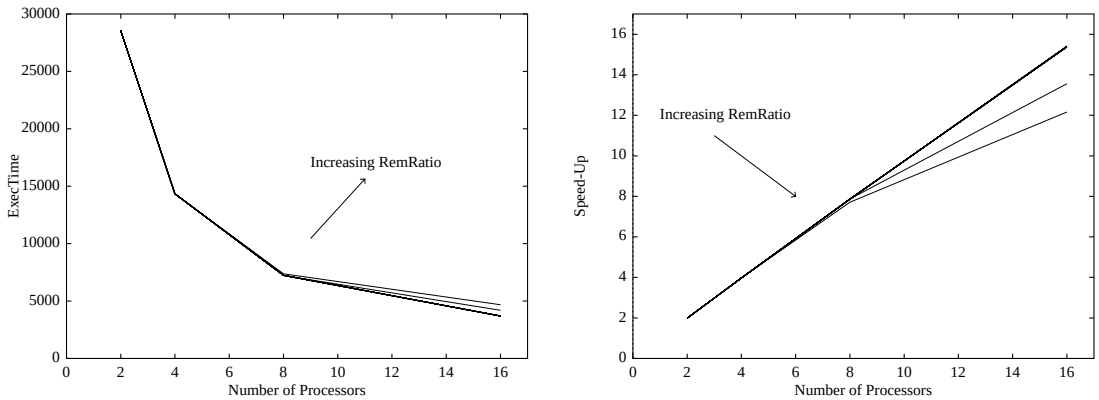
Figure 13: Performance of Saxpy

A more interesting set of results is obtained by gradually increasing the RemRatio from 0 to its maximal value (which is given in Table 1). When this maximal value is reached, all memory fetches are remote to the PE. The Figures 10 – 13 show the performance of each benchmark, with a RemRatio varying from its minimal value to its maximal value. Each figure illustrates the progression of the *total execution time* (ExecTime) as processors are added (a), and the corresponding speed-up curves (b). Each curve, within a specific graph, corresponds to a different value of RemRatio. Table 2 shows the different RemRatio values that have been used to test each of the benchmarks, and their corresponding NetAcc value. Each remote memory request translates into two network accesses, one by the PE issuing the request, and the second by the PE acknowledging the request.



(a) Execution Time (b) Speed-Up

Figure 10: Performance of Loop1



(a) Execution Time (b) Speed-Up

Figure 11: Performance of Loop7

size of the application, for each benchmark, is kept identical while the configurations of the system are increased from 2 to 16 PEs. In all cases, computations and memory loads have been balanced among all PEs.

5.4.1 Execution Time and Speed-Up

Almost linear speed-up is trivially achieved if the *remote-operations ratio* (RemRatio) for each processor is zero—meaning that all the array memory operations are accessing the local SM. The amount of work is therefore simply divided among the available processors, thereby dividing the execution time in the same proportion.

Benchmark	Size of Problem	Instructions Executed	Memory Operations	Execution Time	Processor Utilization
Loop1	4800	36008	13.33 %	36149	99.607 %
Loop7	3200	56808	12.67 %	56923	99.796 %
Matmul	32x32	92836	17.64 %	112078	82.831 %
Saxpy	3840	16328	15.68 %	16744	97.510 %

Table 1: Performance Results for a Single PE

for solving systems of linear algebraic equations such as the famous LINPACK [7]. *Matmul* is a matrix multiplication program which has been directly hand coded into A-code. Each processor is assigned the task of computing one portion of the result matrix. We have chosen the most commmon algorithm for this computation, based on three nested loops, even though it is not the most efficient one in terms of number of communications. The reason behind this choice is that we are interested in high levels of communication, to put to the test the capacity of the system to overcome the problems created by such loads.

For most of these benchmarks, the main loop has been unrolled twice to increase the parallelism in the program and thus reach a better processor utilization rate.

5.4 Simulation Results

In this section, we present the results we have obtained from testing several benchmark programs on *mds*. We have assumed a theoretical processor's clock speed of 25 MHz, yielding a clock cycle time of 40 nanoseconds. The simulation of this clock defines the unit of time within the simulated model. Assuming a 16 bits parallel capacity for ports and links within the network, the data rate is therefore set at 2 bytes per network cycle. Setting the network's clock at 50 MHz gives us a total peak bandwidth of 100 MBytes/second/port. The network's clock cycle being 20 nanoseconds, our PE-to-network clock ratio is therefore set at 2. The size of the buffers within each switch is also set at 2. All these are parameters that can easily be adjusted to simulate different system and network configurations.

The performance results obtained with a single processor are presented in Table 1. The problem size refers to the size of the Structure memory required by each of the PEs to run the programs. The high processor utilization is due to the unrolling of the loops. We want each PE to have enough parallel computations to keep its execution pipeline sufficiently busy in order to absorb the effects of long latency operations. During the second phase of the testing process we investigate performance of the benchmarks on the multiprocessor system. The

- The *Total Network Accesses* ($NetAcc$) is the cumulative number of remote requests issued by each PE:

$$NetAcc = \sum_{i=1}^{N} RemReq_i$$

- The *Normalized Average Throughput* (NAT) is the average number of bytes that passed through the network per unit of time, normalized over 100 (to have integer numbers). It is defined as:

$$NAT = \frac{(NbBytes \times 100)}{N \times ExecTime \times ClockRatio \times DataRate}$$

  where $NbBytes$ is the total number of bytes that passed through the network and $DataRate$ refers to the number of bytes that can go through a link per network cycle. The adjustement with $ClockRatio$ ($\frac{clock_{PE}}{clock_{network}}$) yields a data rate per processor cycle;

- The *Global Average Latency* ($GlobLatency$) is the average latency of each remote request over the N PEs:

$$GlobLatency = \frac{\sum_{i=1}^{N}(LocLatency_i \times Remreq_i)}{NetAcc}$$

- The *Global Collision Rate* ($CollRate$) is the ratio of the total number of collisions that occur within the network, over $BusyNet$, which is the number of cycles during which the network had at least one packet.

$$CollRate = \frac{\sum_{i=1}^{BusyNet} Collisions_i}{BusyNet}$$

5.3 The Benchmarks

The benchmark programs we have selected to test the simulator are mostly scientific applications, mainly because most scientific programs contain large portions of parallelizable code, often expressed in loops involving array operations. Hence, they correspond to applications which are both computationally intensive and well suited for parallel execution.

We have used *Loop1* and *Loop7* from the Livermore Loops [9] which are a collection of typical loops, extracted from widely used scientific applications. Their advantage, as benchmarks, is their big appetite for processor cycles. For each loop, we partitioned the memory so that all of the arrays were equally balanced among the processors. *Saxpy* is the kernel of the Lower/Upper Decomposition, a basic computation procedure widely used in linear algebra

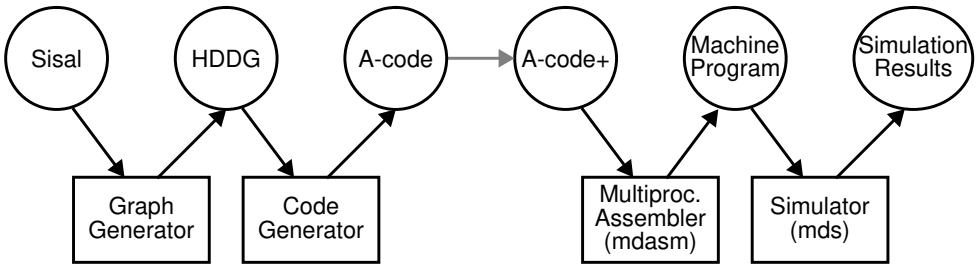


Figure 9: The Testbed Environment

(A-code+). A multiprocessor assembler, called *mdasm*, was developed to assemble multiple-processor A-code programs into an appropriate network-based load-image file, executable by *mds, the multiprocessor dataflow simulator*[7].

5.2 Performance metrics

Assume a multiprocessor system of N processors. The following are the most relevant metrics for each PE:

- The *Remote-Operations Ratio* ($RemRatio$) is the percentage of remote memory requests ($RemReq$) over the total number of instructions executed ($InstExec$);
- The *Processor Utilization* ($ProcUtil$) is the number of cycles during which the processor was busy over the execution time;
- The *Local Average Latency* ($LocLatency$) is the cumulated latency of each remote memory request, averaged over the total number of requests.

$$LocLatency = \frac{\sum_{i=1}^{Remreq} latency_i}{Remreq}$$

  Each latency comprises the time elapsed while going through the network both ways, and the time spent in the queues of the remote processor while it performs the operation.

The following are the metrics used to evaluate the performance of the overall system and the network:

- The *Total Execution Time* ($ExecTime$) reports the number of machine cycles consumed by the whole application. It is based on each individual PE's execution time:

$$ExecTime = \max_i \{ExecTime\ PE_i\} \ , \quad i = 1 \rightarrow N$$

[7] Altogether, the assembler and the simulator programs add up to about 12.5K lines of code

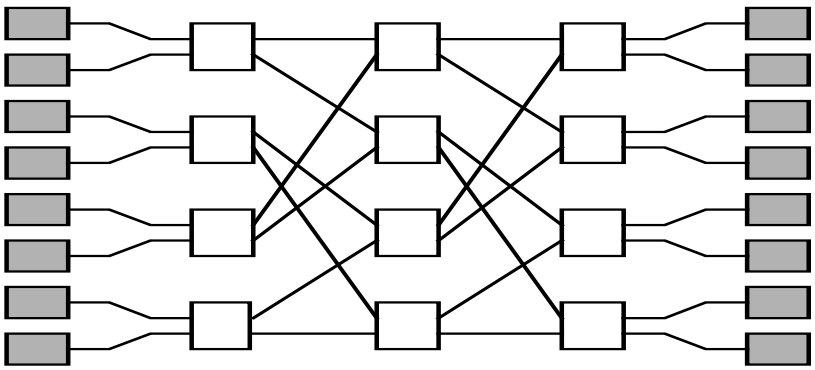


Figure 8: The Omega Network

The simulator has been designed to closely model the McGill Multiprocessor Dataflow Architecture, including an interconnection network, up to a maximal configuration of 16 PEs[6]. The interprocessor communication schemes, presented in the previous section, are an integral part of this simulator. Their efficiency has been evaluated by running the simulator on different system configurations, thereby yielding almost realistic measures of the performance of the overall system.

The 16 PE restriction is due to the actual implementation of the programming language used to write the simulator. Further releases should allow an increased number of virtual PEs.

5.1 The Testbed Environment

The framework of this research is focused on the architectural design of interprocessor communication schemes. The testbed suite consists of five software layers representing five different programs that are used to automatically generate and execute machine code on a simulation model of the MMDA (see Figure 9).

Starting from the SISAL front end, SISAL benchmark programs [14] are translated into a hierarchical data dependence graph (HDDG) [12]. A code generator translates the machine-independent HDDG forms into an intermediate representation of data flow graphs, which we have called A-code [20]. In A-code, the simple nodes of the HDDG form are mapped into instruction tuples. These uniprocessor based A-code programs are then manually modified into multiprocessor programs, written in an extended A-code language with multiprocessor flavors

[6] A detailed description of the program can be found in a companion report [15] where both the program parameters and the performance metrics are fully explained.

free available space in the output port buffer, incoming templates containing remote memory requests are not processed. Instead, the s-node$_{loc}$ address of the node is extracted from the template, and redirected back to the end of the buffer containing the fire signals. This will provide for some short delay during which the instruction containing the remote request will not be processed. This delay depends on the number of instructions awaiting for execution; ideally, it should neither be too long nor too short since either extremes would indicate that the pipeline is *starving* or that there is too much parallelism exposure. Meanwhile, the ICU can use the delay to free up some space in its output buffer by sending packets to the network.

4.2 A Multistage Omega IN for the MMDA

Pipelining multiple requests to a network is a feature that can be exploited by the MMDA system. Single-stage type of networks have not been considered because of their inability to effectively handle this feature. The crossbar type of IN would meet our requirements for fast response time and small delays, in the case of contention, because it provides direct paths from any input to any output. However, their complexity and growth expansion factor makes them a too expensive approach for a dataflow multiprocessor. None of these problems exist with multistage types of IN. Due to their flexibility and attractive features they are currently attracting much of the attention in the field.

We have chosen a multistage Omega type of network—a particular class of Delta networks [18]— to support the interprocessor communications within the MMDA [5]. It consists of $log_2 N$ stages of 2×2 buffered switches, with $N/2$ such switches per stage (see Figure 8).

Because high throughput is important, and because of the relatively small size of the messages, the packet-switched type has been preferred for the MMDA instead of the circuit-switched type. Also, since the requests in the MMDA are issued in a dynamic fashion, the network's operation mode is of the asynchronous type. Thanks to its decentralized type of control, messages can be propagated in a pipeline fashion, progressing through the network independently of each other.

5 Performance Evaluation

In this section, we report the results we have obtained from testing the simulator with various benchmarks, and we give a performance analysis based on those results.

[5] Notice, however, that the ICU has been designed independently of the network topology which implies that it can adapt to any network that guarantees direct point-to-point delivery of messages.

where PE is a processing element address, dm_add is a data memory address, sm_add is a Structure memory address, and s-node is a node address.

Upon reception of the packet, the target PE fetches a data value from location sm_add_{rem} in its own Structure memory, and replies with a message, structured in the following way:

$$<\text{ack-load, PE}_{loc}, \text{ s-node}_{loc}, \text{ dm_add}_{loc}, \text{ data_value}>$$

This message is also an acknowledgment that the fetching operation has been successfully performed. The ICU stores the data_value at location dm_add_{loc}, and uses the s-node_{loc} address to generate a done signal with an unconditional condition code[4] thereby completing the remote load operation.

Processing a Remote Store Request

In the case of a remote store, the data value is part of the remote request. Once the template is retrieved from the buffer, the ICU fetches a value from the local Data Memory address, contained in the template, and extracts the remote PE and s-node addresses from the global Structure memory address. It then proceeds to send a packet structured as follows:

$$<\text{remote-store, PE}_{rem}, \text{ sm_add}_{rem}, \text{ PE}_{loc}, \text{ s-node}_{loc}, \text{ data_value}>$$

where data_value is the data to be stored at address sm_add_{rem} in the Structure memory of PE_{rem}. This task is accomplished by the ICU in the remote PE, when it retrieves the packet from the buffer in its input port. PE_{rem} then replies with an acknowledgment message indicating that the operation has been processed. The packet is structured as follows:

$$<\text{ack-store, PE}_{loc}, \text{ s-node}_{loc} >$$

where s-node_{loc} is the address used by the ICU in PE_{loc} to generate the done signal.

4.1.3 A Retry Mechanism to Control Buffer Space

There are three sources of requests that are likely to generate messages to be processed by the ICU: (1) the Signal Processing Unit, redirecting ip-counts; (2) the Operand Fetch Unit, redirecting remote memory requests; and (3) the input port buffer, receiving requests from the network. All three types of requests can potentially be processed independently of each other. A common entity among them is the output port buffer, connecting the ICU to the network. A possibility of deadlock can occur if this buffer fills up with too many requests.

A retry mechanism has been implemented so that upon reaching a certain (low) degree of

[4] All memory operations such as Load and Store generate the unconditional code upon execution.

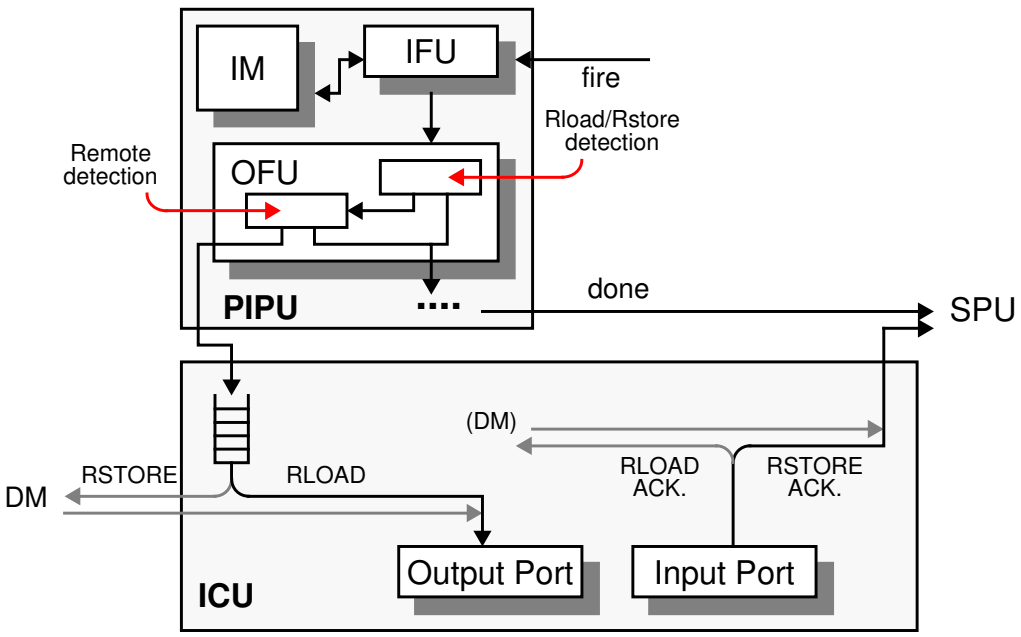


Figure 7: Sending a Memory Request to a Remote PE

To implement them, we require that the Operand Fetch Unit (OFU) of the pipeline be provided with a mechanism to analyze the operator code and detect the remote load and store instructions. Upon detection, the global Structure memory address is analyzed to extract the target PE address. If this latter points to the SM within the PE, then the operation is processed locally as a normal load/store operation. Otherwise, it is stored into a buffer which is repeatedly checked by the ICU, in search for requests. Requests stored in the buffer are processed in a FCFS basis. When a template is stored in the buffer, the ICU retrieves it, identifies the request and takes the appropriate actions to process it.

Processing a Remote Load Request

Figure 7 illustrates the process of sending a memory request to a remote PE. In the case of a remote load request, the address in the template points to a location in the global address space. From that global address, the ICU extracts the remote PE address and the s-node address within the remote PE.

To avoid a "parking" mechanism while waiting for the data, the ICU includes in the request all the information required to store the returning data into the local Data Memory. The packet being sent thus contains the following information:

$$<\text{remote-load}, \text{PE}_{rem}, \text{sm_add}_{rem}, \text{PE}_{loc}, \text{s-node}_{loc}, \text{dm_add}_{loc}>$$

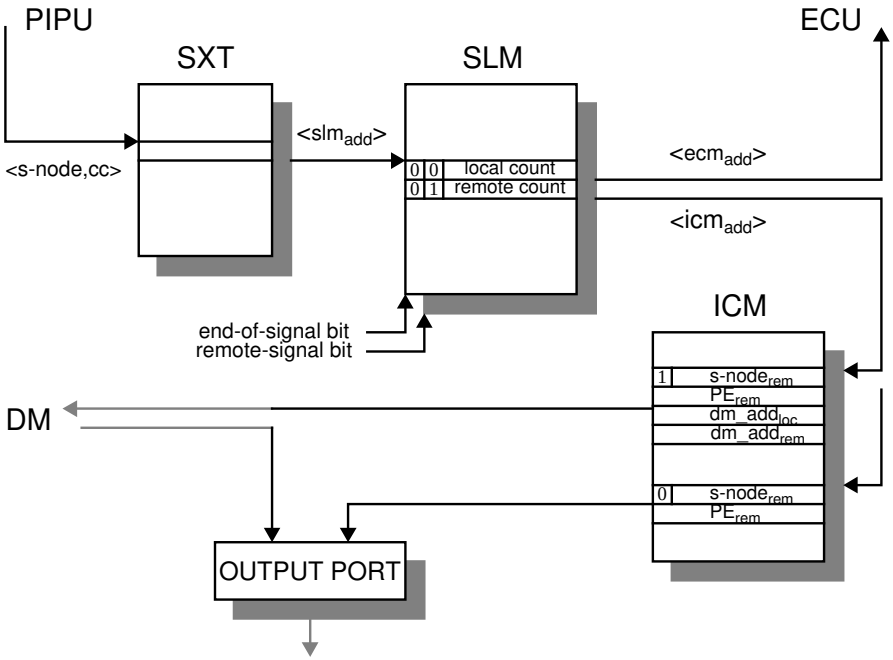


Figure 6: Sending a Signal to a Remote PE

Receiving an Ip-Count

The reception of an ip-count from the network is processed in the following way: the ICU retrieves the incoming packet from its input port buffer and identifies its type. If it is a remote-count, the node address is extracted from the packet and sent to the ECU to be processed as a local count. If it is a data-count, the data value and its address are first retrieved from the packet, and the address is used to store the value in the Data Memory before the count is sent to the ECU.

4.1.2 Remote Memory Requests

The remote load and remote store operations are used to transfer data values from a global Structure memory (SM) to a local Data memory (DM) and vice versa. To be used in a computation, array values have to be downloaded from SM, via the load instruction, and uploaded back, if necessary, using a store instruction. The operations are described as follows:

RLOAD	*global Struc. mem. addr.,*	*local Data mem. addr.*
RSTORE	*local Data mem. addr.,*	*global Struc. mem. addr.*

remote memory requests: which are array memory operations. Hence, a connection ICU-PIPU to independently process those instructions after the decoding phase. Upon completion of the request, the ICU has to generate an appropriate done signal and send it to the DISU.

4.1.1 Interprocessor Count Signals

There are two kinds of interprocessor count signals (also called *ip-counts*), depending on whether data has to be transmitted or not: the first kind is called *data-count* and the other *remote-count* signal.

Sending an Ip-Count

In the case of local count signals, for each node, the three destination lists containing the addresses of the target nodes, are stored in the SLM. The higher order bit of the address is reserved to indicate the end-of-list condition. However, processing ip-counts requires more information, namely the remote node specifications. When a local node signals a remote node, the address stored in the SLM has to be recognized as remote by the signal count dispatching mechanism of the SPU, which then redirects it towards the ICU. Detection is achieved by reserving a second bit, besides the end-of-list condition bit (see Figure 6). The remote specifications are stored within the ICU, in a new memory module called *Interprocessor Communications Memory*(ICM). When the ICU receives an ip-count, it uses it to access the ICM and retrieve all the necessary information to form the required packet before it is sent to the network.

Using the address contained in the ip-count (icm_{add}), the ICU fetches from the ICM the address of the remote node (s-node_{rem}) that has to be signaled and a remote PE address (PE_{rem}). A type-of-ip-count bit is extracted from that field that indicates if the ip-count is a data-count or a remote-count signal. For a data-count signal, the remote address (dm_add_{rem}) needs to be sent along so that the value can be stored in the remote Data Memory upon arrival of the packet. The local address of the value (dm_add_{loc})—used to fetch the value from local memory—and dm_add_{rem} are also stored in the ICM. The following are the packet structures of the ip-count signals:

$$<\text{data-count}, \text{PE}_{rem}, \text{s-node}_{rem}, \text{dm_add}_{rem}, \text{data_value}>$$
$$<\text{remote-count}, \text{PE}_{rem}, \text{s-node}_{rem} >$$

The packet header is actually the remote PE address. Once the packet is formed and properly identified with the type of message it is conveying, it is put on the buffer of the ICU output port which connects to the network.

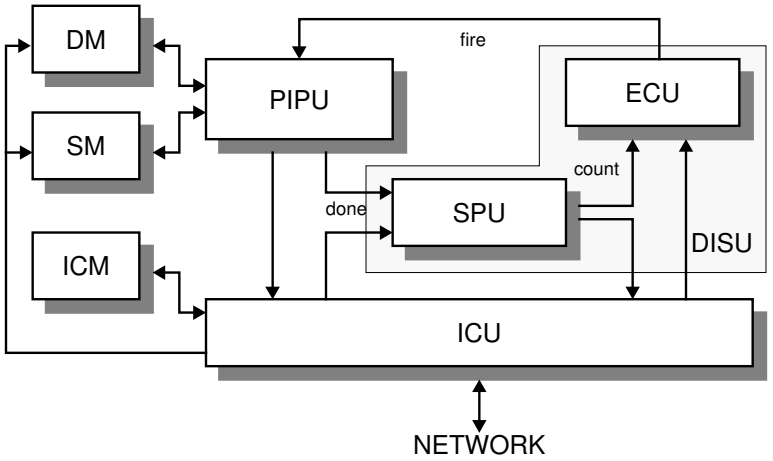


Figure 5: The MDFA and the Interprocessor Communication Unit

the effective address being a global address, it refers to an absolute position in the aggregate memory. The same principle applies to the remote Store operation.

Both operations require an acknowledgment from the remote processor, confirming the successful completion of the operation. Receipt of this acknowledgment triggers the corresponding done signal towards the scheduling unit.

4 The McGill Multiprocessor Dataflow Architecture

To support the communication schemes previously presented, each PE requires a mechanism for handling the communications with the network. This mechanism is implemented through the Interprocessor Communication Unit (ICU).

4.1 The Interprocessor Communication Unit

The Interprocessor Communication Unit (ICU) serves as an interface between the processing element and the interconnection network. Figure 5 shows a schematic block diagram of the MDFA including the ICU. The unit is responsible for all the communications between the local processor and the network. There are two kinds of communications:

interprocessor count signals: used to synchronize the scheduling of actors residing in different PEs. The ICU must be able to send/receive count signals to/from the network.

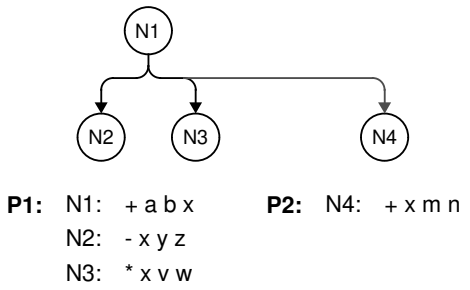


Figure 4: A Data Flow Graph with a Remote Target Node

to its successors, the instruction within that node gets executed and produces a result value (x), which gets stored in the Data memory of P1. The data transfer itself only starts upon execution of the instruction, and is in fact independent of the instruction itself (N1). Thus, the only constraint is to complete the transfer before N4 starts executing. This can be accomplished by sending the data and the signal together from P1, and within P2, by storing the data before signaling N4. The transfer process is thereby completely separated from the execution pipeline which executes instructions transparently of whether their results have to be sent to a remote processor or not.

3.2 A Data Communication Scheme

We have assumed so far that the source and target PE can be identified at compile-time for every data transfer. If this was the case, the above mechanism would suffice for processing distributed data flow program graphs. However, there are situations where the source and target PEs are not known at compile-time. The following is an example:

$$A[\,i\,] = B[\,C[\,i\,]\,]$$

where the array A has to gather the elements of the array B which is scattered among the memories of the PEs. The consumer PE executing the code does not know where to fetch the data until C[i] is evaluated.

Another problem comes from the fact that each data value transfer requires the synchronization of two nodes. Such a requirement is very costly when transferring large blocks of data (i.e. processor P2 consuming a large array produced by P1).

To palliate to these problems, we introduce two array operations: RLOAD and RSTORE, to perform respectively the remote loading and remote storing of a data value *from the global Structure memory to a local Data memory* (and vice-versa). When a remote Load is executed,

mines if can be set *enabled*. It then chooses an enabled instruction and sends the corrsponding *fire* signal to the PIPU. Since there may be more than one enabled instruction, the EC unit uses a scheduling mechanism to determine the order in which the instructions are fired. Such a scheduling mechanism should be "fair" [10] to ensure that the machine does not repeatedly fire a group of instructions without giving attention to other enabled instructions.

3 Interprocessor Communication Schemes for the McGill Dataflow Architecture Model

In this section, we propose an efficient interprocessor synchronization method to allow two data-dependent nodes residing in different processors to synchronize their execution. This synchronization involves sending an initial signal indicating that the source node has been executed, and possibly a data value, as the result of the execution, to be consumed by the receiving node. The memory latency issue is also addressed by providing the system with split-phase remote memory operations to access a global memory[3]. The objectives of the proposed scheme are the following:

1. Provide communication to a multiprocessing environment as transparent as possible, as if there were no boundaries between the processing elements;
2. Reduce the network traffic due to interprocessor data transferal and remote signaling. Ideally, it translates to reducing each data-transfer to two trips through the network: the first carrying the signal and the datum, the second carrying only a signal; and
3. Make an efficient use of the execution pipeline by avoiding the creation of "bubbles", and eliminate the use of unnecessary instructions ,e.g. duplication of data [16].

3.1 An Interprocessor Synchronization Mechanism

The key factor in achieving high performance is the network traffic. Minimizing the traffic can only be achieved by coupling the initial signal and the result value in the same packet, leaving to the remote processor the task of sending the acknowledge signal back.

In the graph shown in Figure 4, node N1 produces a data value consumed by N2, N3, and N4 which resides on a remote processor P2. Notice that before node N1 sends a signal

[3]All throughout this report, "split-phase" refers to an operation that does not stall the execution pipeline and only involves data communication, unlike the I-structures related split-phase operations which can enter in a waiting state if the data is not available.

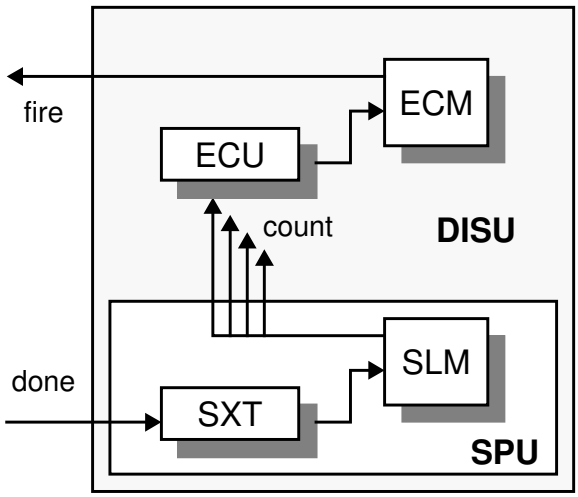


Figure 3: Structure of the DISU

the Signal List Translation Table (SXT).

The *Enable Count Memory* of the EC Unit holds *count* and *reset* status values for each node. It handles each count signal by decrementing the count value for the indicated node and testing for zero. If the count becomes zero, an enable flag for the node is set and the count value is set to the reset value. This unit continuously monitors all the enable flags and issues fire signals for those nodes who become enabled.

The SXT, SLM, and ECM storage modules account for the memories of the PE which keep all the information regarding the proper sequencing of the nodes within the program graph.

Instruction Execution and Scheduling Process

The execution phase of an instruction in the PIPU begins when a firing signal for that instruction is sent to the PIPU. The *fire* signal contains the address of an instruction, which the PIPU will retrieve (from IM) and execute in a conventional pipelined manner. When the execution phase ends, a *done* signal is generated and sent to the DISU together with a condition code indicating the result status of the operation.

The DISU performs the scheduling function by processing the done signal from the PIPU. The condition code determines the appropriate signal lists to be retrieved (from SXT and SLM) and a *count* signal is sent to the EC Unit for each address in those signal lists.

The EC Unit then analyzes the status of the actor addressed by the count signal, and deter-

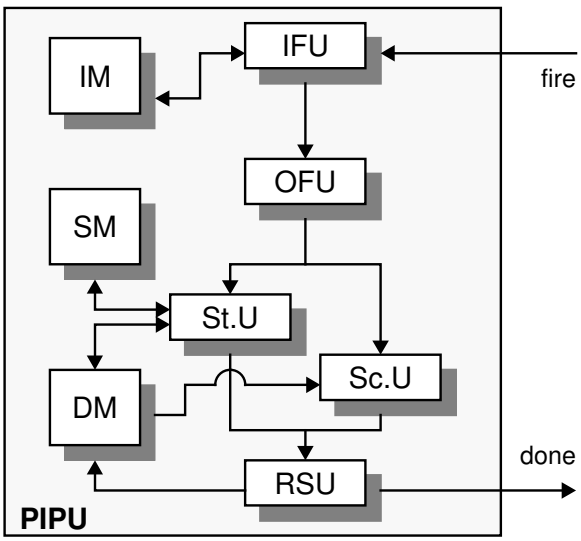


Figure 2: Structure of the PIPU

The Pipelined Instruction Processing Unit

The organization of the PIPU is shown in Figure 2. It consists of four major pipeline stages used for (1) instruction fetch and decode (IFU), (2) operand effective address calculation and fetch (OFU), (3) instruction execution (EU), and (4) result store (RSU).

The Execution Unit consists of a scalar operation unit (Sc.U) and a structure operation unit (St.U). The former performs arithmetic and logic functions (such as basic fixed and floating point arithmetic), as well as scalar memory operations. The latter performs data-structure oriented memory operations, such as array accesses.

This part shows three memory units, namely the Instruction Memory (IM), the Data Memory (DM) which mainly holds scalar operands, and a Structure Memory (SM) which holds arrays and data-structures.

The Dataflow Instruction Scheduling Unit

The DISU consists of a *Signal Processing Unit* (SPU) and an *Enable Controller Unit* (ECU) as shown in Figure 3. The signal graph of a program is represented in the DISU by the signal lists of each of the nodes of the program, stored in the *Signal List Memory* (SLM) of the SPU[2]. These signal lists hold the addresses of the nodes that have to be signaled. These addresses—also called *count signals*—are fetched in an indirect addressing fashion through

[2]For each node, there are three signal lists corresponding to the three possible values of the condition code.

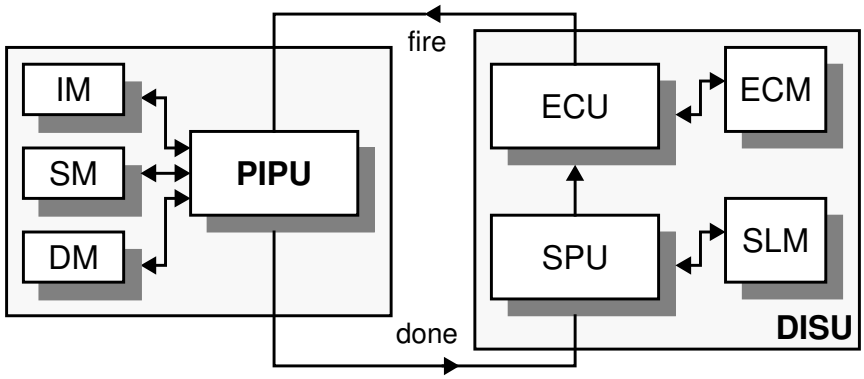


Figure 1: The Argument Fetching Architecture

Finally, Section 6 presents the conclusion of this work in addition to suggested areas for future research.

2 The McGill Dataflow Architecture

Under the argument-fetching principle, data values and control signals are separated. An instruction operates on "values", stored in a regular Data memory, and gets executed by an execution unit, in the same way than in a conventional processor. Upon execution, the unit generates a "signal" which ultimately informs all the successor actors that the instruction has executed. The main characteristic of this model is that data never "flows" through the units of a PE. Instead, signals, which hold the sequencing information among the instructions, are the only entities that move around the circular structure of the PE (see Figure 1).

The McGill Dataflow Architecture (MDFA) consists of two major sections specifically designed to perform the instruction execution and scheduling functions: (1) the *Pipelined Instruction Processing Unit* (PIPU) which is an instruction processor that uses conventional techniques to achieve fast pipelined operation; and (2) The *Dataflow Instruction Scheduling Unit* (DISU) which contains all the information defining the data-dependencies connecting the nodes of the dataflow program. The latter is responsible for identifying and "firing" those nodes that are available for execution. The architecture has several dedicated memories for code, data, and scheduling information.

The addresses of enabled instructions are sent to the PIPU via the *fire* link. Similarly, the addresses of the executed intructions are sent to the DISU via the *done* link, together with a *condition code*, used for sending conditional signals.

Architecture (MMDA). In this multiprocessor system, all PEs are sequentially ordered within the network so each Structure memory (azrray memory) within each PE is actually a portion of an aggregate global memory. It can been seen as a shared global memory, physically distributed, that spans its global address space. All interactions among PEs are therefore accomplished by means of asynchronous message-passing techniques.

An efficient interprocessor synchronization and communication mechanism is also proposed, for sending and receiving signals and data through a packet-switching interconnection network. We have focused our attention on the following objectives:

1. The development of efficient interprocessor mechanisms to allow an MDFA to efficiently interact with other similar machines through an interconnection network;
2. The design of an Interprocessor Communication Unit for the MDFA;
3. The design and implementation of a multiprocessor testbed for MDFA involving (1) design of an assembler in order to accept the multiprocessor features added to the base language of the MDFA: A-code; and (2) construction of a process-oriented simulator to allow us to conduct experimental work on the performance of multiprocessor dataflow programs;
4. The analysis of simulation results to determine (1) the impacts of long latency memory operations on program performance, and (2) the factors that can help minimize such impacts.

The testbed for our simulations thus consists of Mdasm, a Multiprocessor Dataflow Assembler, which is an extension of its uniprocessor version [20] and Mds, the Multiprocessor Dataflow Simulator [15].

Organization of the Report

The details of the basic architecture are described in Section 2 to familiarize the reader with the many modules and their interactions.

In Section 3, we introduce the new abstract model to support interprocessor synchronization and communication schemes. The fourth section introduces the Interprocessor Communication Unit, which is designed to provide a gateway to the network and to support the mechanisms described in Section 2.

Section 5 describes the simulation testbed and analyses the performance results obtained from simulation runs. It provides a deeper insight on the behavior of the argument-fetching approach executing in a multiprocessor environment.

environment, the coordination and synchronization of concurrent activities, executing on different processing elements, is implemented based on a *data-driven* mechanism. Therefore, no conventional process interrupt, busy waiting, or context switching mechanisms are needed.

The McGill Dataflow Architecture project (MDFA) was inpired from these ideas and has been under development in the past years. It is based on the principle of the *argument-fetching dataflow model* [4, 11] which will be described further in the report together with the machine architecture. We will assume, however, that the reader is familiar with the basics of dataflow computing.

Objective of the Research

In 1987, Arvind and Iannucci write:

> "The two most important characteristics of the dataflow processor are split-phase memory operations and the ability to put aside computations without blocking the processor" [1][1].

These are some of the main reasons why dataflow architectures—static or dynamic—show great potential in multiprocessor applications: they can support the implementation of interprocessor data communications and instruction synchronization without the overhead of context switching and program interrupts. By connecting together many dataflow processing elements, we can drastically increase the level of concurrency exploitable in a program in a scalable fashion.

However, whether that increase can be achieved or not, and whether great speed-ups can be obtained or not, depends on many issues, some of which are addressed by this research:

- Interprocessor communication schemes;
- Interprocessor synchronization mechanisms;
- Memory organization; and
- Interconnection network characteristics;

In this report, we propose a multiprocessor dataflow machine, in which each processor is constructed based on the argument-fetching dataflow principle, and connected to all other PEs through an interconnection network. We have called it the McGill Multiprocessor Dataflow

[1] A split phase memory operation has the property of allowing an arbitrary number of instructions to execute between the time when the memory request is issued and the time when the associated response is received. Such lapse of time is called *latency*.

Abstract

The dataflow model of computation offers a powerful alternative to the von Neumann based model for exploiting the fine-grain parallelism inherent in scientific computations. A distinctive feature of dataflow computers is the absence of the conventional program counter. Instead, instruction execution is solely determined by the availability of data which provides ample instruction level fine-grain parallelism. Thus, dataflow architectures show great potential in multiprocessor applications.

We propose an inter-processor communication mechanism to support the implementation of a multiprocessor MDFA system, based on a distributed memory organization. An efficient inter-processor synchronization and communication support is presented, for sending and receiving data through an interconnection network. An Interprocessor Communication Unit (ICU) has been designed to implement the above mechanism in the MDFA. A simulation testbed has been implemented to study the performance of the multiprocessor. It includes an assembler, with multiprocessor extensions, and a multiprocessor simulator. An analysis based on the simulations results is presented, focusing on the impact of long latency operations on program performance.

1 Introduction

Rapid advances in computer architecture and device technology have made it possible to build massively parallel computers. Commercial and research efforts are currently underway to develop parallel computers with performance far beyond what is achievable today.

However, traditional von Neumann systems are inadequate to meet the technological challenges of such an ever increasing demand for computing power. This stems from the nature of their architecture, where instruction scheduling is based on a sequential control mechanism, yielding the so-called von Neumann *bottleneck* [3]. This shortcoming is exacerbated in those multiprocessor systems which are built from conventional processors. Long memory communication latencies, unavoidable in parallel machines, considerably degrade their performance. Furthermore, conventional processors have failed to provide inexpensive synchronization mechanisms for task switching, also frequent in a parallel machine [1, 2].

The design of a parallel system must be based on a sound model of parallel computation, from its programming model down to its architecture. This conviction has recently lead to the introduction of novel forms of computer architecture, in an attempt to eliminate the crucial von Neumann bottleneck. One promising unconventional approach to high performance computer systems is based on the dataflow model of computation. Under this model, instruction scheduling is solely determined by the availability of the data on which the instruction operates, thus eliminating the need for the sequential program counter. In a multiprocessor

Contents

1 Introduction **3**

2 The McGill Dataflow Architecture **6**

3 Interprocessor Communication Schemes for the McGill Dataflow Architecture Model **9**

3.1 An Interprocessor Synchronization Mechanism 9

3.2 A Data Communication Scheme 10

4 The McGill Multiprocessor Dataflow Architecture **11**

4.1 The Interprocessor Communication Unit 11

4.1.1 Interprocessor Count Signals 12

4.1.2 Remote Memory Requests 13

4.1.3 A Retry Mechanism to Control Buffer Space 15

4.2 A Multistage Omega IN for the MMDA 16

5 Performance Evaluation **16**

5.1 The Testbed Environment 17

5.2 Performance metrics 18

5.3 The Benchmarks 19

5.4 Simulation Results 20

5.4.1 Execution Time and Speed-Up 21

5.4.2 Network Throughput 24

5.4.3 Latency 25

5.5 Discussion 27

6 Conclusion **30**

7 Acknowledgments **31**

Bibliography **33**

McGill University
School of Computer Science

Interprocessor Communications and the McGill Multiprocessor Dataflow Architecture

ACAPS Technical Memo 25
July 3, 1991

Jean-Marc Monti

Advanced Computer Architecture and Program Structures Group

3480 University St • Montreal • Canada • H3A 2A7