INTERPROCESSOR COMMUNICATION SUPPORTS FOR A MULTIPROCESSOR DATAFLOW MACHINE

by
Jean-Marc MONTI

School of Computer Science
McGill University, Montréal

May 1991

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

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. Under this model, a program is expressed in the form of a graph, where the data values are carried by tokens, moving on the arcs of the graph. 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. A highly pipelined static dataflow architecture has recently been proposed, based on the *argument fetching* principle, yielding the McGill Dataflow Architecture (MDFA).

In this thesis, an inter-processor communication mechanism is proposed. With this mechanism, a multiprocessor MDFA system can be constructed, 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.

Résumé

Le modèle de calcul Flux-de-données constitue une puissante alternative face aux modèles classés von Neumann, car il permet d'exploiter le parallélisme "finement granulé" (*fine-grain*) inhérent à la plupart des calculs scientifiques. Avec ce modèle, un programme est exprimé sous la forme d'un graphe dans lequel les valeurs sont convoyées par des jetons qui se déplacent sur les arcs du graphe. Un des traits distinctifs des ordinateurs flux-de-données est l'absence du compteur d'instructions conventionel. En contrepartie, l'exécution des instructions est uniquement déterminée par la disponibilité des données, ce qui conduit à un très grand parallélisme au niveau des instructions, finement granulé. L'Architecture Flux-de-Données de McGill (AFDM) repose sur cette idée, plus particulièrement, elle est une architecture de processeur de type flux-de-données statique, basée sur le principe "apporte-arguments" (*argument-fetching*). L'AFDM est l'architecture cible de cette étude.

Dans cette thèse, nous proposons un mécanisme de communication interprocesseur. Avec ce mécanisme, un multiprocesseur de type AFDM peut être construit, basé sur une organisation de mémoire physiquement distribuée. Un support efficace pour les synchronisations et les communications de type interprocesseur est présenté, de façon à permettre l'envoi et la réception de données à travers un réseau d'interconnections. Une Unité de Communication Interprocesseur (UCI) a été conçue pour implémenter les mécanismes en question dans l'AFDM. Un processus de simulation a été mis en place pour étudier la performance du multiprocesseur Ce processus comprend un assembleur, adapté au traitement des programmes multiprocesseurs, et un simulateur de systèmes à processeurs multiples. Notre analyse des résultats, obtenus à partir de plusieurs simulations, vise à évaluer l'impact des opérations de longue latence sur la performance des programmes.

Acknowledgments

A number of people have contributed to the miraculous completion of this thesis. First of all, I wish to express my sincere gratitude to my thesis advisor, Guang R. Gao, for his generous cooperation, support, and guidance in the development of this research.

A special thanks to the members of our research team—Advanced Computer Architecture and Program Structures Group at McGill University. In particular, I wish to thank my colleagues and friends, Russell Olsen and Philip Wong, for making my stay in the laboratory a most enjoyable experience. Their valuable discussions and suggestions have been a great contribution to this work. Also many thanks to the system staff of our School of Computer Science for being so patient and understanding with all of my numerous requests. I am also very grateful to David Samuel, our "experienced proof-reader", whose contribution has helped to improve the quality of this thesis.

Finally, I owe my greatest thanks to my parents, and my sisters, whose endless love and faith have given me the strength to accomplish this step. Without their support, none of this would have been possible. To them, I dedicate this work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Rapid advances in computer architecture and device technology, such as VLSI technology, have made it possible to build massively parallel computers integrating the function of hundreds or thousands of hardware units. Commercial and research efforts are currently underway to develop parallel computers with performance far beyond what is achievable today. Some examples are the SIGMA-1 [39], the EM-4 [44], and the Monsoon [55].

In this thesis, we are interested in the type of multiprocessors generally classified as MIMD—multiple instruction multiple data stream—machines. Compared to their counterpart, the SIMD machines, they have the advantage of performing well over a broader range of applications due to their more general design. MIMD machines typically consist of a collection of von Neumann type processors, executing their own instructions, and usually communicating via a common memory, or by messages being sent over a network. They mostly rely upon sequential conventional languages (like C, Pascal or Fortran), extended with parallel primitives, to program all their applications.

However, there is concern as to whether or not they can keep up with the ever increasing demand for computing power. There exists a mismatch between the amount of parallelism available in many large scale scientific computations and the amount of concurrency which can be efficiently supported by such multiprocessors.

Altogether, it becomes increasingly clear that traditional von Neumann systems are inadequate to meet such technological challenges. 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* [12]. This shortcoming is exacerbated in those multiprocessor systems which are build 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 [9, 11].

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 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.

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

The dataflow model of computation arose from the development of *data flow program graphs* in order to model program behavior. An abstract model was first introduced by Jorge Rodriguez in 1966 [59], who showed that programs could be represented in the form of a directed graph. Data flow program graphs were later formalized by Dennis in a seminal paper which introduces an effective representation of programs in the form of a block structured language [14].

A data flow program graph basically consists of a collection of nodes, connected by arcs. Data values are carried by *tokens*, moving on the arcs of the graph, and the nodes are *actors*, who execute instructions, i.e. apply an operator to the values carried by the input tokens connected to the node. The arcs of the graph thus denote the data dependencies between the operations. An actor becomes activated for execution

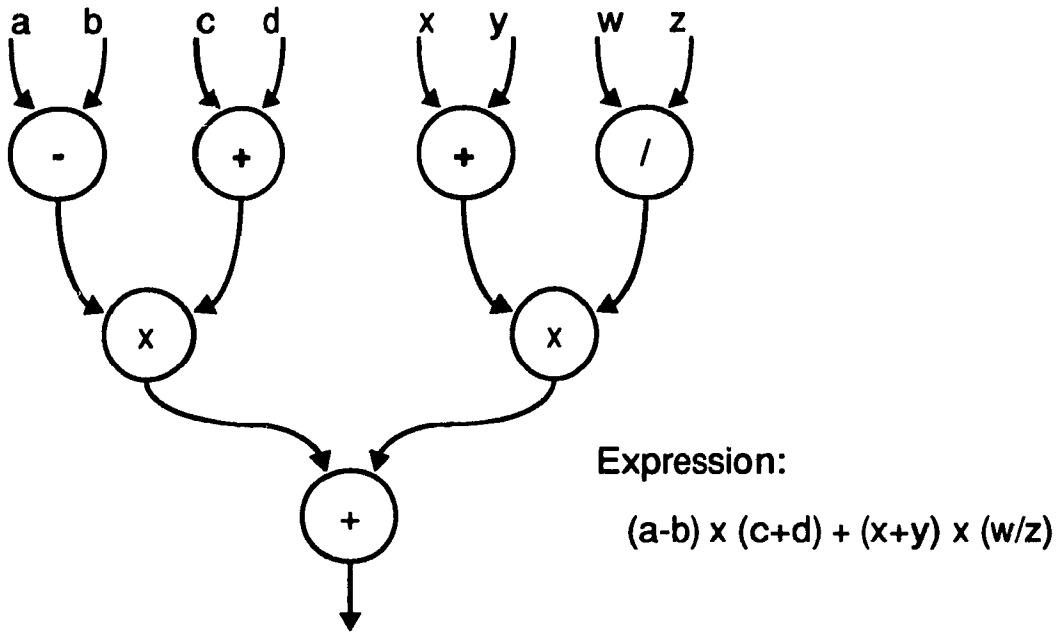


Figure 1.1: An Example of a Data Flow Program Graph

when all its input arcs have a token carrying a data value. The sequential program flow of control, no longer relevant, is thus replaced by a *data driven* flow of control, a major characteristic of dataflow computers. An example of a data flow program graph is shown in Figure 1.1.

By imposing restrictions on the structure of the graph, *parallelism* and *determinacy* are revealed as the two major properties of the dataflow approach. The former is due to the fact that different nodes in the graph can potentially execute in parallel unless there is an explicit data dependence between them; the latter states that the results of the execution of the graph do not depend on the order in which the nodes get executed (see Figure 1.1).

Dataflow research has further evolved, yielding two implementations of the abstract dataflow model which have been under active development ever since: the *static* and *dynamic* dataflow architectures (also called *tagged-token* architectures). The static architectures allow at most one token per arc. In the tagged-token architectures, the tokens are allocated dynamically and include a *tag* specifying their logical position on the arcs. A good presentation and comparison of these models can be found in an early paper by Arvind and Culler [4] and in a more recent publication

by Dennis [18].

1.2 The Static Approach

Under the static model, storage allocation for all operands is assigned at compile-time. The simplicity of the model allows for an efficient instruction execution mechanism as well as for a clean architectural design. However, sophisticated compiling techniques become necessary in order to handle features that support function activation and recursion. The overlapped execution of loop iterations can be accomplished by a pipelined execution of the code or by running multiple copies of the loop body.

The first static dataflow computer was proposed by Dennis and Misunas in 1974 [22]. In this model, the execution of an instruction is based on the data availability concept, and the resulting value is delivered as indicated by the destination field specified in the instruction. Premature rescheduling of the instruction is prevented by the presence of acknowledgment signals. This work helped set up the basis for later projects, namely the LAU Project at Toulouse, France [57], the Texas Instrument Dataflow Project [43], the Hughes Dataflow Machine [35], the MIT Static Dataflow Machine Project [19] and the NEC Dataflow Machines [61].

Another recent project has been under development at McGill University, namely the McGill Dataflow Architecture project [20, 30]. It is based on the principle of the *argument fetching dataflow model.* The main characteristic of this model is that data values and control signals are separated. An instruction fetches its own arguments from a Data memory, just like in conventional processors, so the data values are confined within the execution pipeline, leaving control signals as the only entities that travel around the structure of the processing element. A full description of this machine is given in Chapter 2 of this thesis.

1.3 The Dynamic Approach

The tagged-token approach eliminates the one-token-per-arc constraint and thus provides a higher potential for parallelism than the static model, at the expense of complexity in the design.

Tagged-token architectures generally provide explicit support for recursive function application and overlapped execution of successive loop iterations. Storage for values—computed by function activation—is allocated during program execution. Tokens carry a tag which denotes their context within the program. Conceptually, tokens are kept in a common *pool* of storage waiting for their counterparts to arrive before enabling an instruction.

In the early proposals, this ability to support several simultaneous activations of one instruction is supported by a special mechanism called *token-matching*. It is an unique feature by which pairs of tokens referring to the same instruction are "matched" and their values sent to an execution unit. Implementation of this mechanism with a matching store, however, turned out to be a relatively costly associative mechanism.

The first proposals for tagged-token dataflow architectures included the Manchester Prototype Dataflow Computer [38] developed at Manchester University, in England, and the MIT Tagged-Token Project from the Massachusetts Institute of Technology [8]. Both projects independently discovered the idea of explicitly labeling the tokens.

The Manchester prototype was the first dataflow machine to be built, providing a waiting-matching store of 16K token capacity [38]. At Arvind's group, the first step in the Dataflow Project was the U-Interpreter, developed in 1977 [7], which is an abstract model for interpreting dataflow programs. Based on this work, research further evolved towards what we know today as the Tagged-Token Dataflow Architecture [3, 6].

An alternative and more efficient implementation was later adopted at MIT, yielding the Monsoon Computer [5, 55]. It is based on the idea of frame activation by which frames of memory are allocated at every function call and every loop iteration, and the base addresses of these frames play the role of "tags".

Other proposals for dynamic architectures have been developed since, like the Epsilon Dataflow Processor, at the Sandia National Laboratory [37], the SIGMA-1 Machine, built by researchers at Japan's Electrotechnical Laboratory [39], and the EM-4 Project, from the same Laboratory [44], a dataflow computer planned to have 1024 processing elements. The most powerful dataflow machine that has been built to date, though, is the SIGMA-1 Machine; it consists of 128 processors and 128 I-structure stores interconnected by 32 local networks and one global two-stage Omega

network. and has demonstrated a performance of 170 MFLOPS on a small integration problem but the ideal peak performance sums to 427 MFLOPS [40].

1.4 Objectives of the Thesis

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" [9][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 in the 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 project:

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

In this thesis, 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

[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*.

Multiprocessor Dataflow Architecture (MMDA). In this multiprocessor system, all PEs being sequentially ordered within the network, each Structure memory (array 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 message-passing techniques.

To study the performance of the MMDA, interprocessor communication supports have been developed on the grounds of the distributed memory organization approach. An efficient interprocessor synchronization and communication mechanism is proposed in this thesis, for sending and receiving signals and data through a packet-switching interconnection network. The main objectives of the project have been:

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 [62, 50] and Mds, the Multiprocessor Dataflow Simulator.

1.5 Structure of the Thesis

Chapter 2 of this thesis is dedicated to the McGill Dataflow Architecture. It explains the concept of the argument fetching principle as an alternative to the conventional

argument flow static dataflow models. It also gives an overall description of the dataflow machine that has been proposed, based on this new principle, from the abstract model down to the architectural design. This is done in order to lay the proper ground work for the following chapters.

In chapter 3, we discuss some alternative interprocessor synchronization and communication schemes that have been proposed thus far within the framework of our research group, and introduce a new and efficient abstract model for both inter-PE synchronization and inter-PE communication. The multiprocessor machine that is proposed makes use of a globally addressable distributed memory organization and an asynchronous message passing synchronization mechanism.

The fourth chapter describes the modifications made to the original design in order to support the mechanisms described in chapter 3. It introduces the Interprocessor Communication Unit, which is designed to provide a gateway to the network. The interconnection network on which we have chosen to run our simulations is a packet-switching Omega type of network.

Chapter 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. The results demonstrate the MMDA's ability to effectively tolerate the latency of long memory operations with no significant impact on the performance of the programs. They also show that the overall throughput of the interconnection network is more critical than the latency factor. Given today's technology, we can build the type of networks needed to mask those long latency operations up to a certain degree.

Chapter 6 is an overview of the related work in the dataflow field, concentrating on the multiprocessor aspects of some proposed dataflow architectures. A discussion of the different techniques used to support interprocessor synchronization and communications provides with a survey of some of the major current research interests.

Finally, chapter 7 presents the conclusion of this work in addition to suggested areas for future research.

Chapter 2

The Static Dataflow Model and The McGill Dataflow Architecture

In recent years, the dataflow concept has attracted increasing attention as a radical alternative to the von Neumann model. It emerged as an innovative model which offers simple yet powerful means of achieving highly parallel computations. In this chapter, the concern is towards the static dataflow model of computation.

Indeed, in dataflow programs, there is no notion of a single point of control, nothing corresponding to the program counter of a conventional sequential computer. Computations are described in terms of locally controlled events, each of which corresponds to the "firing of an actor" in the data flow graph[1]. In a dataflow machine, many actors may be ready to fire simultaneously, resulting in an asynchronous data-driven concurrent execution. The first section of this chapter will focus on the basic data flow graph model, and the rules determining the execution of a program graph.

Many advances have been achieved in the dataflow field since Dennis and Misunas first proposed their static dataflow model [22]. A static dataflow processing element, based on the argument-flow principle, is briefly described in the second section.

Despite the parallel nature of the dataflow model, there have been serious doubts

[1]Firing an actor stands for executing an instruction who has previously been enabled due to the availability of its input operands. See Section 2.1.1.

that dataflow processor architectures can compete with the efficiency of their conventional counterparts. One major concern is the amount of data "flowing" in the processor. In order to exploit fine-grain parallelism, the model appears to require an excessive volume of data traffic when compared to a conventional processor architecture—a criticism common to many proposed dataflow architectures.

The McGill Dataflow Architecture, described in the third section, is based on the argument-fetching principle. This new approach brings a solution to the excessive data traffic problem encountered in previous proposals. This section gives an in-depth description of the new model, the architecture design, as well as the execution and scheduling mechanisms.

2.1 The Static Dataflow Model

2.1.1 The Basic Graph Model

The *static data flow graph model* constitutes the basic model governing the execution of a program graph on a static dataflow computer. The program graph is represented by a directed graph. The nodes in the graph represent instructions whereas the edges represent data dependencies between instructions. In dataflow terminology, the nodes are called *actors* and the edges *arcs*. Each actor in the graph has an associated ordered set of *input arcs* and *output arcs*.

When an instruction is executed, the tokens lying on the input arcs of the actor are "consumed" and new tokens, representing the result value of the instruction are "produced". Similarly, when an actor produces a data value to be transmitted to a successor actor, it "places a token" on the arc connecting both actors.

The execution of a graph can be modeled by a sequence of *configurations*, each describing the different states of the computation. A configuration is defined as an assignment of tokens on the arcs of the graph. There can by many such sequences of configurations but the ultimate result is unique, a property known as *determinacy*. The transitions between configurations are governed by *firing rules* which determine the conditions required to *fire* an actor.

With the exception of some special actors for implementing conditional graphs and iterative computations, the firing rules for the static dataflow model are defined as follows:

Firing Rules:

1. An actor becomes enabled iff all of its input arcs have one token and all of its output arcs are empty.
2. An enabled actor may fire, and once fired, all tokens on its input arcs are removed, and a token is placed on each of its output arcs.

Hence, firing an actor corresponds to applying the corresponding operation to the values carried by the tokens on its input arcs. Upon completion, a token carrying the result of the computation is placed on each of the output arcs. Note that once an actor has been executed, it cannot become enabled again until the tokens on its output arcs, carrying previous result values, have been consumed by its successor actors.

With the assumption that tokens are graphically represented by dots on the arcs of the graph, Figure 2.1, illustrates a possible sequence of configurations modeling the execution of the expression:

$$(a+b)\times(c-d)$$

As illustrated in Figure 2.1, data flow graphs can exhibit two kinds of fine-grain parallelism: the first one, called *spatial* parallelism, is exploited when two nodes are potentially executable concurrently; the second, called *temporal* parallelism, is exploited when independent waves of computation are pipelined through the graph. The nodes in the first stage of the graph being simultaneously enabled, they express the first kind. Upon execution of the node in the second stage, they can be rescheduled for execution, illustrating thus the temporal parallelism. A detailed study of this kind of parallelism, also called *pipelining* can be found in [30].

2.1.2 Dataflow Languages

The static data flow graph model exhibits many interesting properties among which: (1) the only dependencies among actors in a program graph are data dependencies,

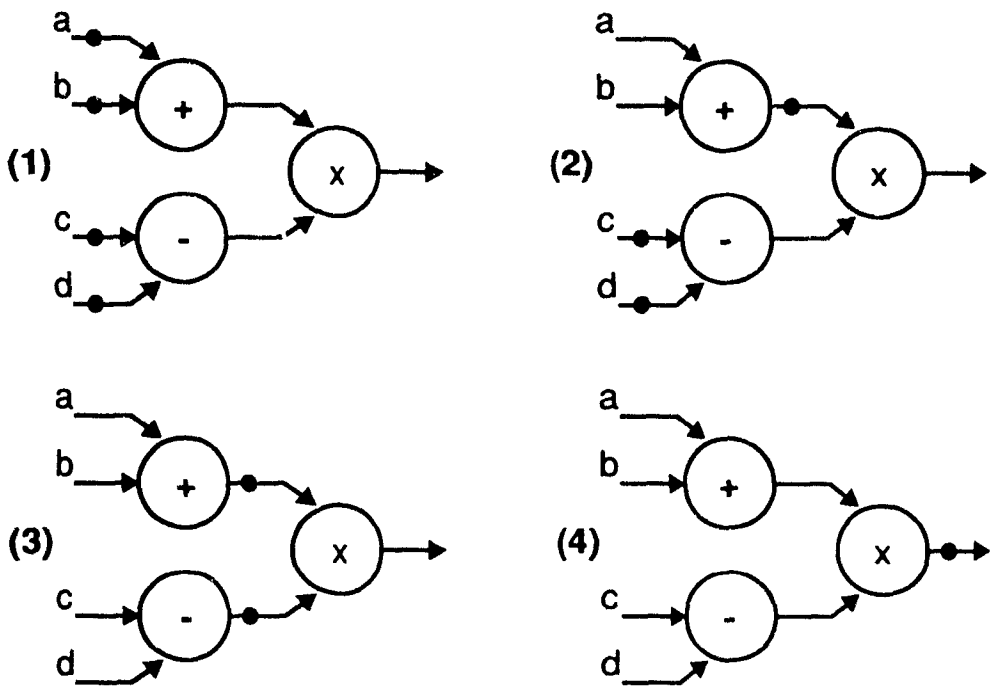


Figure 2.1: Sample Execution of a Data Flow Graph

(2) the execution of an actor is *side-effect free*, and (3) the determinate property which ensures that input/output behaviors are functional. Due to those features, the static data flow graph model, is an excellent model for mapping functional or applicative languages [42].

These languages are based on the functional programming style where the *single assignment* rule and the *referential transparency* also allow the parallelism to be fully and naturally expressed. There have been a number of functional languages, called *dataflow languages* [1], specifically designed for exploiting the dataflow model of computation.

Examples of dataflow languages are Val [2, 48], SISAL [49] and Id [53]. More information on dataflow languages can be found in [13] and [1].

2.2 A Static Dataflow Architecture

A typical static dataflow processing element (PE) based on the argument-flow model (also called *dataflow circular pipeline*) is described in [15]. This section's main concern is to briefly describe that architecture as a representative of early static dataflow proposals. It then covers the implementation details required to support the static graph model, and the criticisms made towards this approach—which is based on the argument flow principle.

2.2.1 Implementing the Basic Model

In the static data flow graph model, an arc can carry at most one token, so an actor can't become enabled unless all its output arcs are empty. This rule can be implemented by introducing a second form of arc, called *acknowledgment signal arc.* The role of this acknowledgment signal arc is to inform all the predecessors of an actor that the successor actor has consumed all the tokens residing on its output arcs, and thus are ready to accept the next set of data.

As a consequence, each actor has an associated set of input arcs, among which some are *data arcs* and some are *acknowledgment signal arcs*, and a set of output arcs with the same characteristics. Figure 2.2 shows the *signal graph* corresponding to the example shown in Figure 2.1. Implementations of the model have to provide mechanisms to support this signal graph.

The condition for firing an instruction in the static dataflow machine now requires that a *signal token* be placed on each of the input signal arcs, a normal *data token* be placed on the input data arcs. Upon firing, an actor places a data token—carrying the result value—on each of its output data arcs, and a signal token on each of its output signal arcs. The outcome of the firing of an actor is therefore the production of a datum for each of its successors, and also an acknowledgment for each of its predecessors, so that they can fire again.

2.2.2 The Architecture

Figure 2.3 shows a schematic representation of the architecture. Dataflow instruction templates are stored in the *activity store.* The *queue* holds the addresses of those

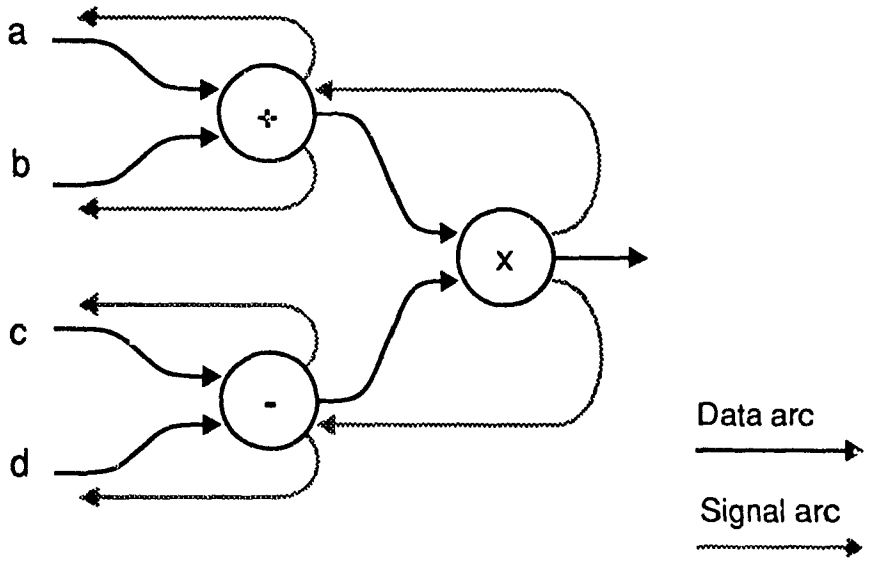


Figure 2.2: A Signal Graph

instructions which are enabled and ready to be executed.

The role of the *fetch unit* is to continuously pick the address of an instruction from the queue, fetch the corresponding instruction template from the activity store, and deliver it to the *operation unit* where the instruction gets executed. Each destination address will carry a copy of the result value to the *update unit* of the appropriate PE

The role of the update unit is to store those result values in the templates of the target instructions and determine whether or not they become enabled. All communications between the different units within a PE are achieved via data packets traveling in a circular fashion, which has caused the PE to be called *circular pipeline* [15].

2.2.3 The Argument-Flow Principle

In traditional proposals for a dataflow processor, instructions loaded into the processor represent, more or less directly, the actors of a data flow program graph. Such an instruction typically has two spaces for receiving operand values called *operand receivers*, a *signal-needed* field, a *signal-reset* field, and a field that holds a *destination list*—indicating the target instructions to which result values or signals are to be sent.

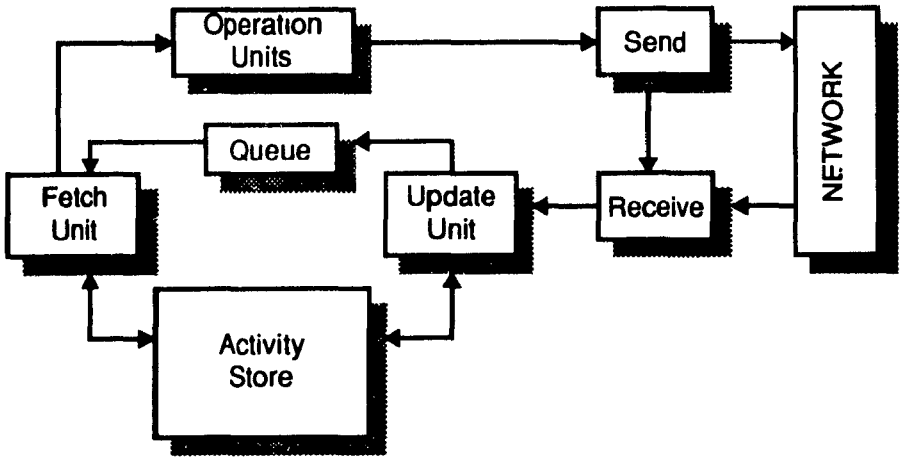


Figure 2.3: An Argument-Flow Dataflow Processing Element

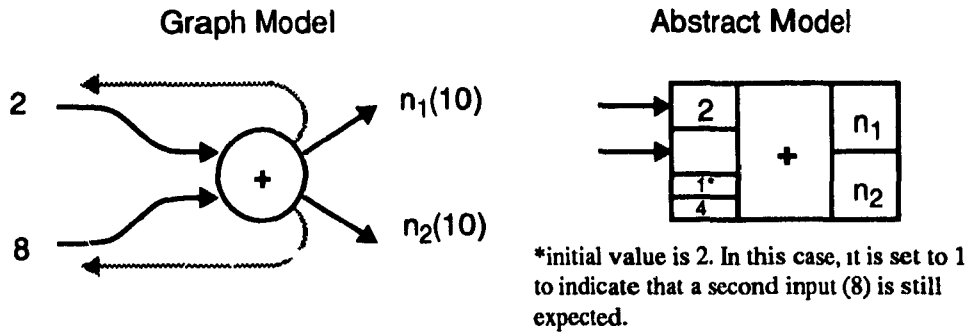


Figure 2.4: The Abstract Model of the Argument-Flow Approach

The signal-needed field contains an integer indicating the number of signals still to be received before enabling the instruction. The signal-reset field contains the value required to reset the signal-needed field when it reaches zero.

The arrival of a value corresponds to placing a token on the input arc of an actor. Therefore, the delivery of a result value to the operand receiver of an instruction template accomplishes two purposes: it signals the target instruction that an input is available; and it transmits the data value itself from an instruction to one of its successors. We call these architectures *argument-flow* dataflow architectures. Figure 2.4 illustrates the abstract dataflow model of the argument-flow approach.

The arrival of a signal (an *acknowledge signal*) from a successor instruction means

that a result token has been removed from the corresponding output arc. The arrival is processed by decrementing the signal-needed field by one. If the value reaches zero, the instruction is enabled and the field is reset to the value stored in signal-reset field.

An issue in this processor architecture is that more data movement is involved than seems necessary. Two cases in point are: the destination list passes through the instruction fetch unit and the operation unit, although the information in the list is not acted upon by either unit; and result values are copied and stored in duplicate whenever there is more than one target instruction. One of the consequences of this issue is that unnecessarily high traffic may be generated causing the performance of the machine to decrease.

These inefficiencies arise from the decision to keep data information (values) and control information (addresses) bound together in packets as they traverse the circular structure of the processor. The alternative approach has lead to the decision of separating the data and signaling roles of the information packets [20]. Once this has been done, it becomes evident that it is better for an instruction to fetch its own arguments from a Data memory than for its predecessor instruction(s) to store result value(s) in the operand fields of several target instructions.

These considerations have led us to the *argument-fetch architecture* [20] which can overcome those weaknesses. Based on this principle, the McGill Dataflow Architecture has been developed.

2.3 The McGill Dataflow Architecture

This section of the chapter presents an in depth description of the McGill Dataflow Architecture based on the argument-fetching principle. Under this new principle, the data and the control signals are separated. An instruction operates on values, stored in a regular Data memory, and an execution unit executes it, the same way a conventional processor would do. The instruction template now adopts the conventional three address format since all values are stored in memory. Upon completion of the execution, it generates a signal which ultimately informs all the related 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, thereby annihilating the concept of tokens.

The McGill Dataflow Architecture (MDFA) consists of two major sections specifically designed to perform the instruction execution and scheduling functions. The *Dataflow Instruction Scheduling Unit* (DISU) contains all the information defining the data-dependencies connecting the nodes of the dataflow program. It is responsible for identifying and "firing" those nodes that are available for execution. The mechanism used to select instructions for execution is based on the *signal graph* of the program. The *Pipelined Instruction Processing Unit* (PIPU) is an instruction processor that uses conventional techniques to achieve fast pipelined operation.

The operational semantics of the model are best described by first presenting an apercu of the architecture of an abstract machine.

2.3.1 The Abstract Model and The Program Tuple

A data flow program graph G for the MDFA is represented by a *program tuple* (P, S), where P is a set of instructions and S is a *signal flow graph*, represented by sets of signal addresses. Formally,

$$G ::= \langle P, S \rangle$$

Figure 2.5 shows a dataflow program tuple with its corresponding P-code and S-code entries. Each actor in the data flow program graph has an entry in both the P (P-code) and S (S-code) sections of the program tuple. The instructions in P-code, called *p-instructions* contain no information about the sequence of execution. Instead, the sequencing information is given separately by the signal flow graph called S-code.

Each p-instruction in P-code is a three address instruction similar to that in a conventional architecture. The instruction templates are stored in the PIPU Instruction Memory and are executed by the PIPU when it receives a signal from the DISU.

The signal graph S-code of the program tuple determines the sequencing of the instructions within the graph. S-code contains a set of s-nodes, each containing a list of actors to notify when its corresponding p-instruction has been executed. Each target actor has an *enable count* indicating the number of signals to be received before the actor can fire, and a *reset count* to reset the enable count value once it has fired. There is a one-to-one correspondence between each s-node address and its corresponding p-instruction address.

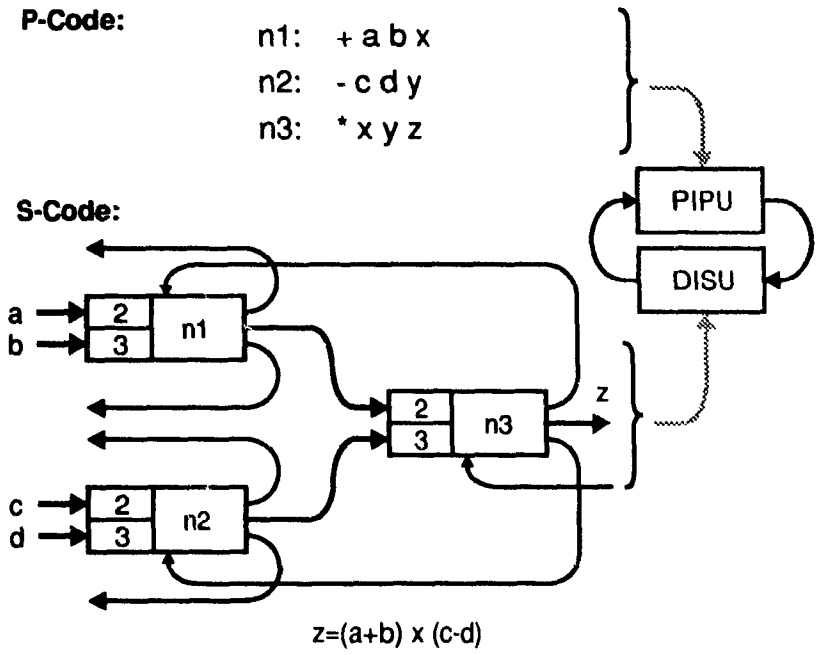


Figure 2.5: A Dataflow Program Tuple

2.3.2 The Argument Fetching Architecture

The following is a description of the architectural aspects of the MDFA, focusing on the single PE. The description of the multiprocessor aspects of the machine will be covered in Chapter 4 of this thesis. None of the modules necessary to connect the PE to the network will thus be part of this coverage.

The addresses of enabled instructions are sent to the PIPU via the *fire* link. Upon execution, the PIPU sends back to the DISU the address of the instruction—that has just been processed—through the *done* link, together with a *condition code* used in sending conditional signals. Figure 2.6 illustrates the circular structure of the architecture and shows the many units integrating each of its parts.

The Pipelined Instruction Processing Unit

The organization of the PIPU is shown in Figure 2.7. 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, and (4) result store (RSU).

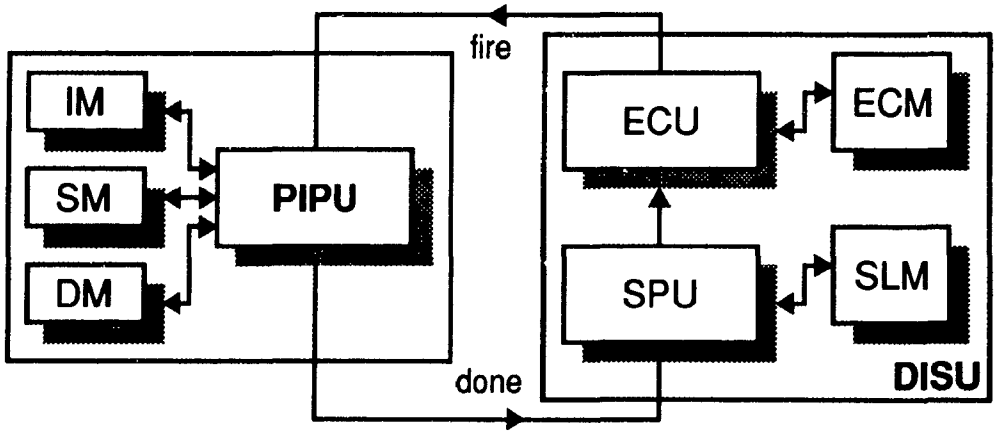


Figure 2.6: The Argument Fetching Architecture

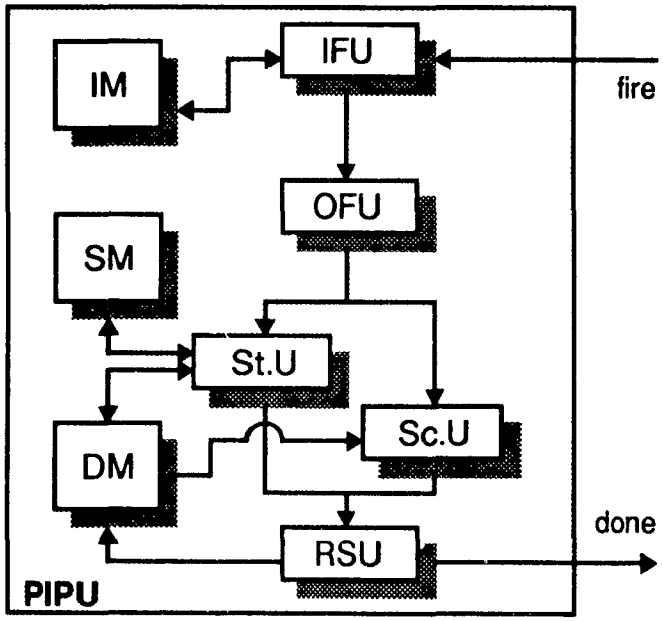


Figure 2.7: Structure of the PIPU

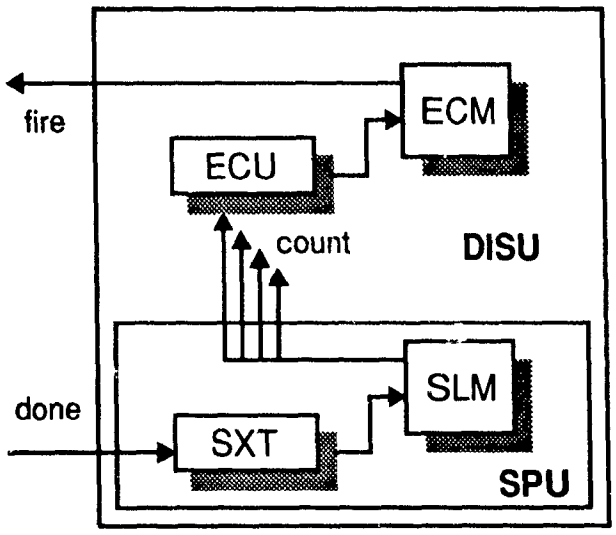


Figure 2.8: Structure of the DISU

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.

As can be seen, this architecture has several dedicated memories for code, data, and scheduling information. 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 2.8. 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. For each node, there are three signal lists corresponding to the three possible values of the condition code. These signal lists hold the addresses of the nodes that have to be signaled. We call those addresses *count signals*.

Since there is a one to one correspondence between each s-node and each p-instruction, and not all nodes have the same fan-out, a Signal Translation Table

(SXT) is required to store the addresses of the signal lists in the SLM. The fetching of the count signals is therefore done in an indirect addressing fashion.

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.

2.3.3 Instruction Execution and Scheduling Process

This section explains how the units can cooperate and communicate in order to implement the abstract model of the MDFA. The operational semantics of a data flow program graph is described by the firing rules of the actors in the graph. In the argument-fetching architecture, the firing rules are implemented jointly by the PIPU and DISU, where the PIPU performs the actual execution of an operation and the DISU performs the scheduling of the operation. These two phases are called the *execution phase* and the *scheduling phase.*

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 a p-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. A done signal has the following format:

<done-signal> ::= <address> <condition-code>

where <address> is a pointer to the signal list counterpart of the "done" p-instruction, and <condition-code> is either T, F or U (true, false or unconditional). When a

done signal is received, the condition code determines the corresponding 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 rules of signaling are:

- The addresses in the unconditional signal list are always signaled;
- If the condition code is T (F), the addresses in the true (false) list are also signaled.

The EC unit retrieves the status information of the node specified by the count signal and decrements its enable count field in the ECM. The instruction is then identified as *enabled* when this count reaches zero. At that time, the *count* field of the enabled s-instruction is reset to the value determined by the *reset* field. Finally, the EC unit chooses an enabled instruction and sends a *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" [29] to ensure that the machine does not repeatedly fire a group of instructions without giving attention to other enabled instructions.

In one implementation, the enable flag bits in the EC unit are organized as a two dimensional array (see [21]). The selection of instructions for execution is done by checking each row in turn and sending the contents of any non-zero row to a column encoder. No row is considered more than once unless all other rows have been examined. A column encoder scans each flag bit in the selected row in turn, issuing a fire command for each bit that is on.

2.4 Summary

This chapter has described the static dataflow model of computation and the McGill Dataflow Architecture. Under the basic data flow graph model the execution of a graph is modeled by a sequence of configurations, each of which describes a state of the computation. The transitions between configurations are governed by firing rules which determine the conditions required to fire an actor (an thus perform the execution of an instruction).

A typical static dataflow computer based on the argument-flow principle has been briefly described. Under that principle, the tokens carry both the data values of the

operation to be performed and the addresses of the destination nodes where the result value has to be forwarded. A common criticism to this approach is the unnecessary high traffic of data within the processing element.

The argument fetching architecture originated the idea of the separation of data and scheduling information as two distinct entities. As a direct consequence, the data-driven instruction scheduling mechanism is separated from the instruction processing unit.

The McGill Dataflow Architecture has been proposed to fully support this model. Its architecture consists of an execution pipeline and a scheduling unit. Signals (addresses) travel from one unit to the other in a circular fashion indicating which instructions to execute and which actor to enable.

Its main features are:

- *No data-dependent hazards in the pipeline.* The dataflow model guarantees that there can never be a data conflict between any pair of simultaneously enabled instructions. The principle that no instruction is initiated unless it has no data conflict with other instructions in the pipeline is honored. The immediate result is that data-dependent hazards are eradicated.

- *Elimination of pipeline gaps due to operand matching.* In MDFA, the "matching" of the operands is not performed in the critical instruction execution pipeline, thus eliminating possible pipeline gaps due to such operand matching; and

- *Token duplication is avoided.* In MDFA, a result value of an instruction never needs to be duplicated (copied) and routed to the input "arcs" of destination nodes. It is stored in the Data memory thereby allowing other subsequent instructions to directly fetch it when needed as an argument.

In the conventional dataflow models the transferal of the data is combined with the synchronization of the instructions so that the term "communication" includes both actions. In the MDFA, those actions are no longer integrated so we can differentiate between "data communication" and "event synchronization". However, synchronization is in itself a form of communication. Therefore, to avoid any confusion throughout the remaining chapters of this thesis, the term communication will be used interchangeably when referring to both actions together, or when referring to the specific communication of data.

Chapter 3

Interprocessor Communication Schemes for the McGill Dataflow Architecture Model

Many researchers agree that in a parallel system, one crucial problem to be solved is the *communication* between processors, be it transfer of data or event synchronization. Indeed, interprocessor communications constitute a major factor affecting the performance of a program. Minimizing the costs involve effective communication methods, an efficient implementation scheme to directly support the model, and a careful use of the methods, aiming at reducing the number of data transfers and synchronization during program execution. There are thus three aspects to this approach. The communication methods are strongly determined by the model of computation being used, the implementation schemes are provided by an efficient architectural design, and their use is dictated by compiler decisions regarding the partitioning and mapping of the programs onto the PEs. This chapter addresses the first of these aspects, namely it proposes a communication and synchronization method for the McGill Multiprocessor Dataflow Architecture. The implementation scheme is described in details in Chapter 4. A thorough study of the compiler aspect, however, is beyond the scope of this thesis, although the subject is briefly addressed in Chapter 5 which covers the performance results of the simulations.

In a pertinent paper published in 1985, Gajski and Peir provide a refinement of that crucial processor communication problem [28]. Their analysis provides a classification scheme, based upon what they consider to be the main factors affecting parallel

computer performance. They contend that multiprocessor performance depends on the ability of a system to handle *control*, *partitioning*, *scheduling*, *synchronization*, and *memory access*. The dataflow model of computation provides a good basis for seeking elegant solutions to these issues.

For instance, in a data flow program graph, parallelism can be fully expressed at the instruction level, thereby providing a flexible data-driven type of distributed control model. Also, the synchronous scheduling of instructions being naturally embedded in the dataflow model, little software cost is incurred during program execution. As for the partitioning problem, it comprises detecting the parallelism in a program, and assigning tasks to PEs so that execution speed is maximized. This task is facilitated by the use of dataflow languages, which are more appropriate for expressing highly parallel problems. The data dependencies are directly represented in the program structure, which eases the analysis done by the compiler to identify them [16].

The problems of *memory latency* and *cost of synchronization* have been closely addressed by Arvind and Iannucci in a paper published in 1987, where they are known as the *two fundamental issues in multiprocessing* [9]. They state that dataflow architectures are propitious for multiprocessor applications because they can provide solutions to those fundamental issues. Most von Neumann style processors are likely to "idle" during long memory references, unavoidable in parallel machines. It is also difficult to provide inexpensive task switching mechanisms, often required during waits for synchronization events. The dataflow model of computation, however, can effectively tolerate long latency operations, provided there is enough fine-grain parallelism available to hide them. It can also handle the synchronization requirements of a parallel system.

Hence, by efficiently addressing these fundamental issues, it appears that a multiprocessor can be *scalable*, i.e. achieve proportionally higher performance gains by increasing the number of PEs. It is among the objectives of this thesis to investigate this fact. In this chapter, 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 in this chapter by providing the system with split-phase remote memory

operations to access a global memory[1]. The objectives of the proposed scheme are the following:

1. Increase the flexibility of the mechanism;
2. Provide a transparent multiprocessing environment;
3. Reduce the network traffic; and
4. Make an efficient use of the execution pipeline.

There have been a few previous attempts to provide the McGill Dataflow Architecture with such a scheme. The first section of this chapter will describe these attempts and provides a brief discussion on their lack of efficiency. The following section will provide the solution that we propose in this thesis.

3.1 Some Alternative Implementations

Previous dataflow architectures are based on the argument-flow model, where the source instruction generates the necessary tokens that must be routed to all target instructions. This approach lacks of flexibility, stemming from the fact that the data arcs and signal arcs always appear in pairs. It also incurs the overhead of having to duplicate all instructions producing data to multiple PEs. However, these shortcomings set aside, the model is well suited for a multiprocessor environment Indeed, interprocessor data routing is automatically embedded in the model by assigning a processor address (tag) to each destination address. In the argument fetching approach, those mechanisms are not implicitly part of the model so they have to be developed.

A few attempts to provide interprocessor synchronization schemes to the design have been proposed thus far, disregarding the more simple problem of implementing remote memory operations with the data-structure/array memory. The first approach uses specific instructions to send and receive data through an interconnection network.

[1]All throughout this report, "split-phase" refers to an operation that does not stall the execution pipeline and only involves data communication Nothing like the I-structures related split-phase operations which can enter in a waiting state if the data is not available.

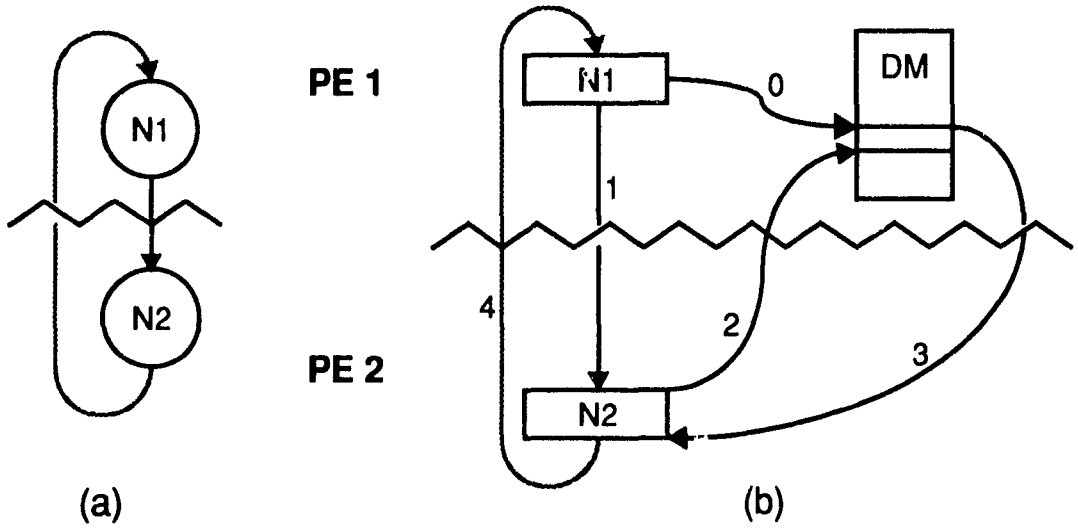


Figure 3.1: Interprocessor Transfer of Data

The second approach directly extends the argument-fetching principle to implement communications between multiple processors. In this section, both approaches are analyzed to stress their major deficiencies, in order to justify the approach proposed in this thesis.

The implementation schemes of a synchronization method are best described within the context of an example. All throughout this chapter, we will consider a scenario where a node producing a data value has to synchronize with a remote node which consumes it (see Figure 3.1 (a)). under the data-driven model, the transfer actually consists of the following steps: (0) preliminary step, an actor N1 residing on processor P1 executes its instruction and produces a result value which gets stored in the Data Memory; (1) the actor N1 signals its successor actor N2 residing in P2 that it has produced a result which can be consumed; (2) N2 proceeds by sending a request to fetch the datum from P1, (3) P1 fetches the data from its own memory and sends it to P2; (4) upon execution of N2, a signal is sent back to N1. The overall transfer involves 4 trips through the network as illustrated in Figure 3.1 (b).

3.1.1 Explicit Send/Receive Instructions

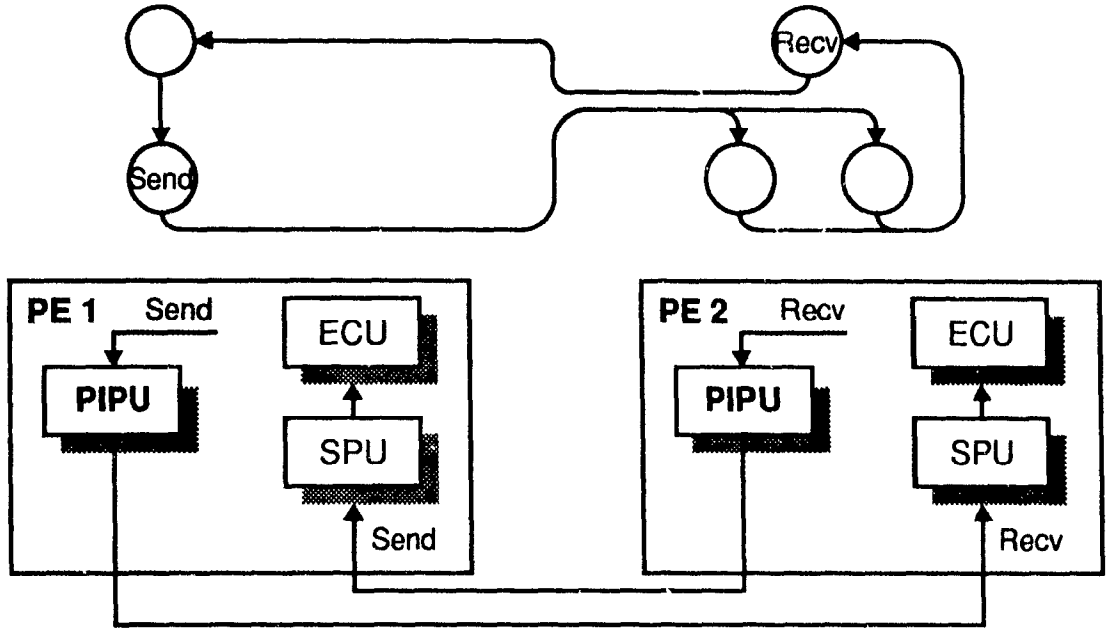


Figure 3.2: The Send/Receive Link

In this proposal, the instruction set is extended by introducing a *send* instruction and a *receive* instruction to support interprocessor data routing. A detailed description of this implementation can be found in [33]. Basically, each send instruction is coupled with a receive instruction in a remote PE, thereby implementing a bidirectional link between both PEs (see Figure 3.2). Following are the formats of the Send and Receive instructions:

SEND *local Data mem. addr., remote specs. addr.*
RECV *remote specs. addr.*

where *remote specs. addr.* is a Data Memory address containing a set of remote specifications, and *local Data mem. addr.* is a data value's address.

Upon creation of this data value, the send instruction is signaled and enabled. When it is executed, it is redirected to the I/O unit, which fetches the remote specifications of the target PE and the data value from the arguments of the instruction, and creates a data packet of the form:

$<PE_{rem}$, s-node$_{rem}$, data_address$_{rem}$, data_value>

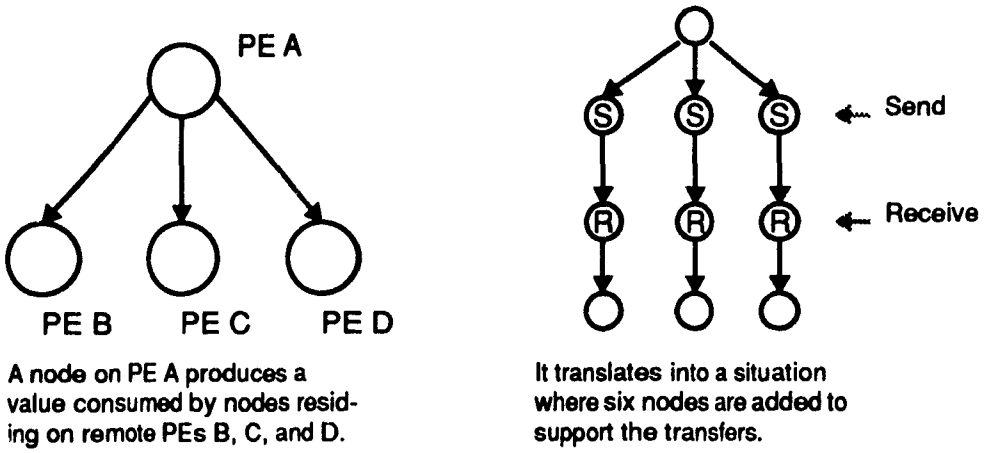


Figure 3.3: Instruction Duplication to Send Data to Remote PEs

where PE_{rem} is the address of the remote PE, s-$node_{rem}$ is the address of the node corresponding to the remote receive, and $data\_address_{rem}$ is the address in the remote PE where data_value has to be stored. When the I/O unit of the remote PE receives the packet, it stores the value at the proper address and generates a done signal for the instruction specified by the s-$node_{rem}$.

This done signal then follows the normal path, i.e. it retrieves a signal list and sends a count signal for each of the target nodes. When these target nodes execute, they will signal back the Receive instruction causing it to become enabled. A Receive instruction also gets redirected to the I/O Unit which fetches, from Data Memory, the remote specifications needed to initiate the following packet:

$$<PE_{orig}, \text{s-node}_{orig}>$$

where PE_{orig} is the address of the PE which originated the sending. When the packet arrives at PE_{orig}, the I/O unit originates a done signal for the s-$node_{orig}$ instruction, which is the address of the node corresponding to the Send instruction.

The appealing aspect of this solution is that only two trips through the network are needed to accomplish a transfer. Steps 1 and 3 are embedded in the send instruction thus eliminating the need for step 2.

The criticism of this approach is similar to the one made against the argument flow model: the need to duplicate instructions for multiple remote targets and a lack

of flexibility. Figure 3.3 illustrates the need to duplicate instructions when a node produces a data value for nodes residing in remote PEs. For each transfer, a pair of Send/Receive instructions has to be added to the graph. The lack of flexibility stems from the tight rules governing the use of the Send/Receive operations. It makes it impossible for a node to synchronize with a remote successor without transferring data. Also note that the overhead of adding special instructions for each transfer ultimately contributes to the congestion of the execution pipeline. Furthermore, the send instruction has to wait for all the successors of the corresponding receive instruction to complete before this latter is executed, thereby sending the acknowledgment signal back through the network. In the best case, the receive instruction has one successor; this still involves going twice through the execution pipeline. This can involve a considerable delay if the remote processor has an important pool of enabled instructions to execute—a situation likely to appear.

3.1.2 External Addressing Mode

A slightly more efficient solution to achieve interprocessor synchronization has been proposed in [32, 34]. This method behaves as if there were no boundaries between the PEs at all; instructions on different processors are able to interact with each other directly in an interprocessor argument-fetching fashion. Remote data is specified by a special addressing mode in the input operands of an instruction (refer to [62] for a complete description of the instruction set). Since the data values are *fetched* rather than *sent*, data from one PE can be accessed by several other PEs without adding extra instructions to duplicate it.

When instruction fires, the addressing mode of the operands is analyzed in the Operand Fetch Unit. If the extended addressing mode is detected, it causes the instruction to be "parked" away while the I/O unit sends the corresponding request(s) to the remote processor(s). When the data values arrive from the remote PEs, the blocked instruction is released and rejoins the execution pipeline to proceed with its execution. Note that the blocked instruction does not stall the execution pipeline by impeding other instructions from getting executed. Instead, it creates a "bubble" in the pipeline. As long as there are enough enabled instructions, the processor can be kept usefully busy while tolerating the delay due to the remote fetching

Implementing this method involves a complex I/O unit consisting of an associative memory where instruction templates are stored while waiting for the data values to

return from the network. Upon arrival, a search is conducted on the p-instruction address to locate the template and update it with the datum. This mechanism is further complicated by the fact that both input operands can potentially refer to a remote PE so the I/O unit has to make sure the instruction template is complete before unblocking it.

External addressing for the operand of instructions is paired with a similar mechanism where external count signals can be detected in the scheduling unit and sent to remote nodes. A done signal is an s-node address which ultimately retrieves from the SLM a list of count signals which are also s-node addresses (of the nodes that have to be signaled). In this particular proposal, external addressing is implemented by reserving one bit in the address specifying if it refers to a local or a remote s-node (This method is better explained in the next section).

By offering a separate scheme for data and signals we have obtained a higher degree of flexibility compared to the first proposal. Although this is a more transparent solution (no explicit instructions), and although it has provided answers to the criticisms of the previous scheme, the implementation requirements may be expensive and the simplicity of the MDFA design is lost. Indeed, the introduction of an associative memory implementing the "parking store", together with the mechanisms to unblock an instruction, upon arrival of the data, have added much complexity to the overall approach. Also note that, considering the scenario previously exposed at the beginning of this chapter, no savings are achieved in terms of network traffic and the basic four trips are still required: the initial signal has to be sent to P2, the remote node then sends a request for data back to P1, the data value is sent to P2, and the remote node sends an acknowledgment signal back to P1.

Finally, this solution can only be applied to situations where the compiler can determine, at compile-time, the location where the remote data is stored. The next section will show that there are situations when this information is not available until run-time.

3.2 The Proposed Solution

The aforementioned implementations are at opposite extremes of the spectrum of solutions. One follows the argument flow approach while the other follows the argument fetching; one blends signals and data together while the other considers them

as separate entities. This project's first goal was to find an efficient solution which avoided the deficiencies of both implementations. Given the architecture described in the previous chapter, it translated into the following objectives:

1. Increase the flexibility of the mechanisms by treating data and signals as independent entities, while maintaining the perspective of a clean design with a low level of complexity;

2. Provide communication to a multiprocessing environment as transparent as possible, as if there where no boundaries between the processing elements;

3. 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

4. Make an efficient use of the execution pipeline, i.e. avoid the creation of "bubbles", and eliminate the use of unnecessary instructions;

3.2.1 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 the remote processor the task of sending the acknowledge signal back. This method may appear as if it does not directly comply with the argument-fetching principle. However, it is important to realize that the decision to separate data values from synchronization signals in the argument-fetching model, is mainly due to the fact that within a single processor, memories can be put very close to the execution unit. Hence the data/signal separation avoids the overhead of copying and sending multiple tokens around the processor. In other words, the data does not need to travel outside the boundaries of the Execution Unit. This does not hold if the target node resides in a different PE, thus invalidating the main reason for separating data and signals.

In the graph shown in Figure 3.4, node N1 produces a data value consumed by N2, N3, and N4 which resides on a remote processor P2. Here, it is important to realize that before node N1 sends a signal to its successors, the instruction within that node

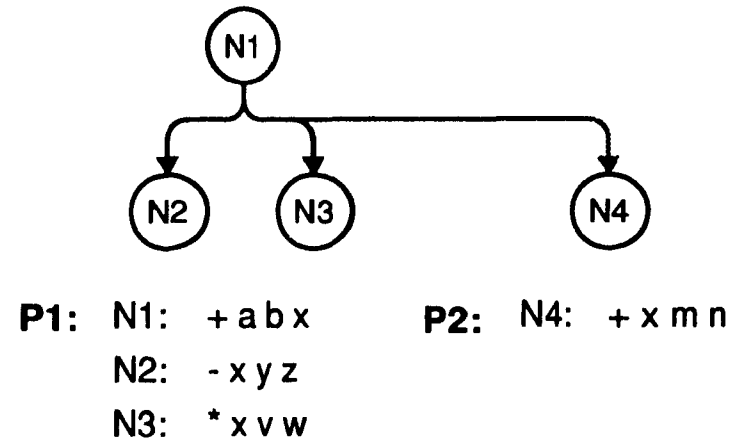


Figure 3.4: A Data Flow Graph with a Remote Target Node

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). The only constraint is that is has to be completed before N4 starts executing. This can be accomplished by sending the data and the signal together from P1, and within P2, by making the transfer before signaling N4. The figure graphically shows that it is simpler to differentiate a remote target from a local target by using different kinds of signals instead of using special instructions. Clearly, the transfer process is best initiated outside the execution pipeline, within the ISU.

Hence, in our new scheme, there are three different kinds of count signals:

- A local count, where N1 signals N2;
- A remote count, where N1 signals N4;
- A remote count paired with a data value;

We use an external addressing mode for count signals—similar to the one described in Section 3.1.2— to distinguish between the local and remote signals. An extension bit, indicating whether or not the signal has to be coupled with data can be added to the addressing mode. When the extension bit is set, the mechanism adopts an argument-flow approach where the data is sent to the target instruction rather than fetched.

The transfer process is thereby completely separated from the execution pipeline. This unit executes instructions transparently of whether their results have to be sent to a remote processor or not.

3.2.2 A Data Communication Scheme

We have assumed so far that the source PE and the target PE can be identified at compile-time for every data transfer. If this was the case, the above mechanism would suffice for the processing of distributed data flow program graphs. The producer PE would initiate each transfer and synchronize with the corresponding consumer PE.

However, this is not the case and there are situations where the source and target PEs are not known at compile-time. The following is an example of such situations:

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

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 transfer of data requires the synchronization of two nodes. Such a requirement is very costly when transferring large blocks of data, specially when that transfer can be done without disturbing the remote execution unit.

To palliate to these problems, we introduce two array operations to the instruction set design: RLOAD and RSTORE. They perform the remote loading (and remote storing) of a data value *from the global Structure memory to a local Data memory* (and vice-versa). Since they operate in the same way their local counterpart do, direct, indirect, and extended addressing modes are also supported (see [62]).

When a remote Load is executed, the effective address being a global address, it refers to an absolute position in the aggregate memory. For the sake of simplicity, we are assuming that all PEs have the same array memory size, and that each PE knows this size. Splitting the absolute address into a PE address, and a relative address within the PE is a trivial task. The same principle applies to the remote Store operation.

The above example is handled by evaluating first the C element, storing its value in Data memory, and issuing a remote Load instruction from that location. Should the remote Load instruction encounter a run-time computed address which turns out to be local, it is processed as a normal local load without going to the network. This latter instruction now becomes obsolete since the RLOAD can process both local and remote loadings.

Note that both operations are split-phase memory operations because of the long latency they require to execute, meaning that there is a lapse of time between the initiation of the instruction and its completion. However, there is no need to "park" an instruction template in the local processor while the operation is carried out, since it is performed in the same way a local Structure memory operation is performed. The request is sent to the network along with all the information needed to bring back the information requested.

Also, both operations require an acknowledgment from the remote processor, confirming the successful completion of the operation, without which the instruction cannot assume it has finished executing. Receipt of this acknowledgment then triggers the corresponding done signal towards the scheduling unit, again as any other local instruction would do, in accordance with the second objective of the implementation scheme (see beginning of the chapter).

3.3 Summary

In any multiprocessor system where the shared memory is physically distributed, interprocessor communication is an important factor in order to achieve high level performance. Synchronization, a special form of communication, and memory latency, are the two most fundamental issues in multiprocessing. The dataflow model of computation appears to be well suited to efficiently address these issues. Under the argument fetching principle, the mechanisms to synchronize nodes residing in separate PEs and to provide remote memory operations are not implicitly embedded in the model.

In this chapter we have proposed an effective communication method to provide interprocessor synchronization between two nodes and remote Structure memory operations. It provides the necessary supports for interprocessor communications to

the McGill Multiprocessor Dataflow Architecture. The multiprocessing environment is transparent, flexible, and makes an efficient use of the network and the execution units of all the PEs.

Basically, the method can be described as follows: a data transfer between a producer/consumer pair of actors is initiated by the done signal generated by the producer actor upon execution. It is a two-step operation: a special signal carrying the result value is sent through the network to the remote PE where the data is locally stored and the actor is signaled. Upon execution, the remote actor sends back the signal confirming that it has consumed the data. Simple synchronization can also be achieved by sending a normal signal to a remote node.

The key contribution of this solution is to separate the processing of remote signals from the main execution pipeline. It is implemented by introducing an external addressing mode in the signal lists of an actor. The execution pipeline is thereby completely separated from this matter, and there is no need to duplicate instructions in case of multiple remote target nodes.

The second component of this proposal consists of extending the instruction set with a remote load and a remote store operation. It allows the PE to perform those transfers where the source PE and the target PE cannot be determined during compilation phase. At the same time, it provides an inexpensive alternative to the transfer of large blocks of data, which incurs prohibitive synchronization costs when performed with the other methods. The reason lies in the fact that it becomes possible for a PE to fetch (or store) a block of remote data, one element at a time, without synchronizing with the target PE for every value. Only one synchronization signal is necessary to tell the consuming PE that the block of data is available.

Chapter 4

The McGill Multiprocessor Dataflow Architecture

In the design of a multiprocessor system, much of the efforts are dedicated to reducing the amount of time the processor has to wait, while memory operations are being performed. The dataflow approach is viewed as an extreme solution because of the ability of the processing element to execute other useful instructions while waiting. Another important factor that affects performance is the synchronization mechanism. Although dataflow architectures do not incur any software overhead for such operations, it is important that they be supported by an efficient hardware implementation. In this chapter, we describe the implementation details of the interprocessor communication supports, which were explained in Chapter 3 of this thesis.

The methods allow any two processors to interact with each other in an argument-flow fashion by means of messages conveying signals (possibly paired with data). Ultimately, the system we envision is a dataflow multiprocessor system, consisting of a number of MDFA processors, connected together by an interconnection network as shown in Figure 4.1. The overall system is modular and scalable, to facilitate the inclusion of more processors in order to increase power, memory space and communications bandwidth. It is also flexible enough to allow:

- a request to be issued without affecting the local PE's capacity to execute other instructions;

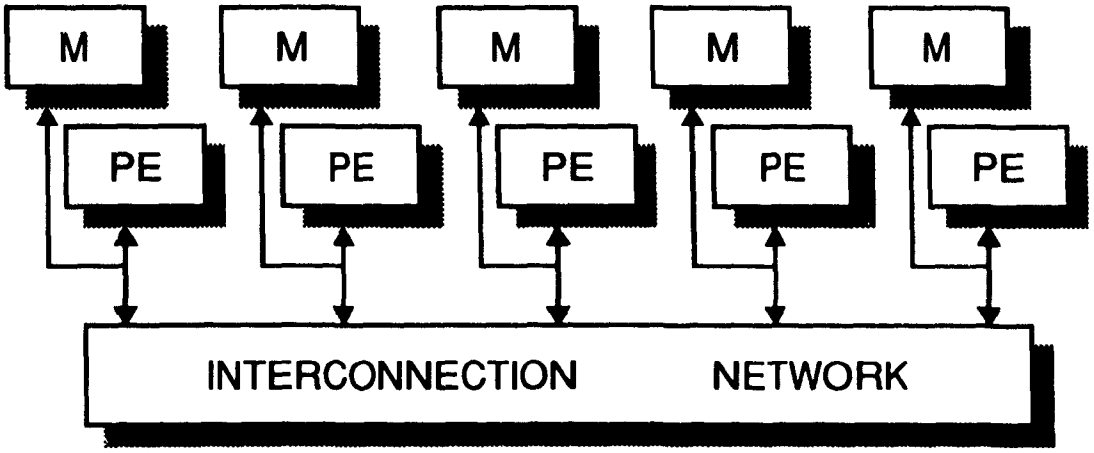


Figure 4.1: Structural Model of a Multiprocessor MDFA

- a request to be processed without affecting the remote PE's capacity to execute instructions;
- a node in a processor to signal multiple remote nodes;
- a PE to issue multiple independent requests to the network; and
- a PE to receive responses for those requests in a different order from that in which they were issued.

This chapter concentrates on the modifications that have to be done on the MDFA to support the interprocessor communication schemes. Each PE requires a mechanism for handling the communications with the network, which include remote memory operations and remote synchronization signals. This mechanism is implemented through the Interprocessor Communication Unit (ICU) which is described in the first section. Remote memory operations are introduced to allow any PE to interact with remote Structure memory (SM) modules.

Interactions among PEs are achieved through the sending and receiving of messages. The communication network is responsible for delivering each message from the sender PE to the receiver PE, based on the address within the message's header. Many routing network structures have been studied to implement MIMD machines; Section 4.2 of this chapter describes the characteristics of the packet switched interconnection network that has been chosen to support the Multiprocessor McGill Dataflow Architecture (MMDA).

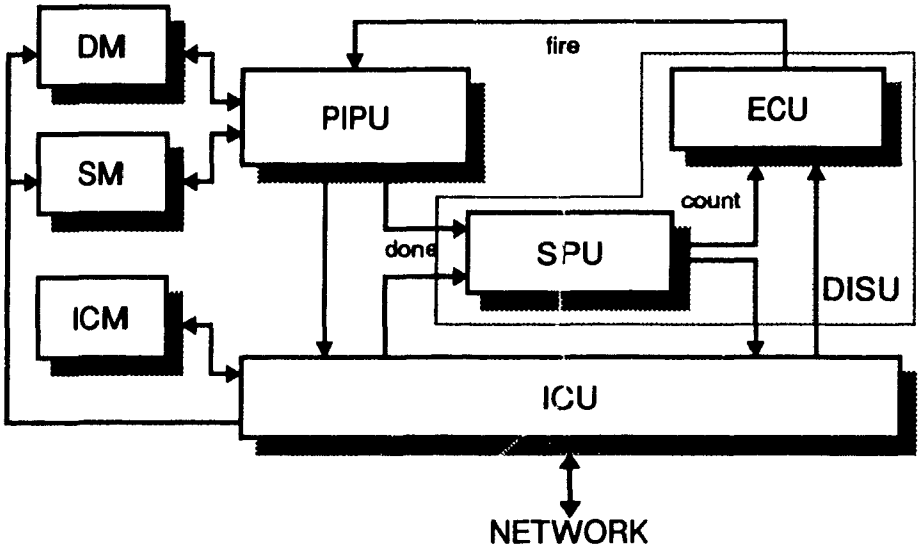


Figure 4.2: The MDFA and the Interprocessor Communication Unit

4.1 The Interprocessor Communication Unit

This section describes the Interprocessor Communication Unit (ICU) which serves as an interface between the processing element and the interconnection network. It is responsible for all the communications between the local processor and the network. There are two kinds of communications:

remote count signals: used to synchronize the scheduling of actors residing in different PEs. Hence, the ICU must be able to send (and receive) count signals to (from) the network. Part of the unit has therefore to connect with the DISU which manages the count signals within the PE.

remote memory requests: which are array memory operations. The consequence, at the design level, is a connection with the PIPU to extract those instructions after the decoding phase. Upon completion of the request, the ICU has to generate the appropriate done signal and send it to the DISU.

Figure 4.2 shows a schematic block diagram of the MDFA including the ICU. The connections between the ICU and the Data Memory and Array Memory will be explained in the following subsections, as we describe the unit in greater detail.

4.1.1 Interprocessor Signals

There are two kinds of count signals, depending on whether the count is a local signal or an interprocessor signal, in which case it is called *ip-count*. Furthermore, there are two kinds of ip-count signals, depending on whether data has to be transmitted or not: the first kind is called *data-count* and the other *remote-count* signal.

There is a direct correspondence between the data arc described in Section 2.2 of this thesis, and the data-count signal (which is paired with a data value). The same correspondence exists between the signal arc and the remote-count signal (with no data), with the exception that the remote-count can be used for purposes other than just to signal back a predecessor actor.

This section describes the steps involved in processing ip-signals. Since all the information required to send and receive those signals during execution time is fully expressed in the dataflow program, it can be gathered at compile-time. Target node addresses are stored in the SLM while the number of predecessor nodes is stored in the ECM for example. This information is then grouped into memory load-images, one for each memory module in the system and stored in a "load-image file". This is the kind of file that should be preloaded in the memory modules of the pro[illegible] element to execute the program.

Sending an Ip-Signal

In the case of local count signals, the compiler determines, for each node, the three destination lists corresponding to the possible values of the condition code: unconditional, true, and false. These 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.

For the ip-count signals, there is more to fetch than just the address of the target node. The remote specifications for the remote-count signal include the address of the remote PE and the address of the target node within that PE. A data-count signal needs yet more specifications, namely the address in the local Data memory from where to fetch the data value, and the address in the remote Data memory where it is to be stored after the transfer. All this information is available at compile-time so it is possible to store it into any of the existing memory modules or into a new module.

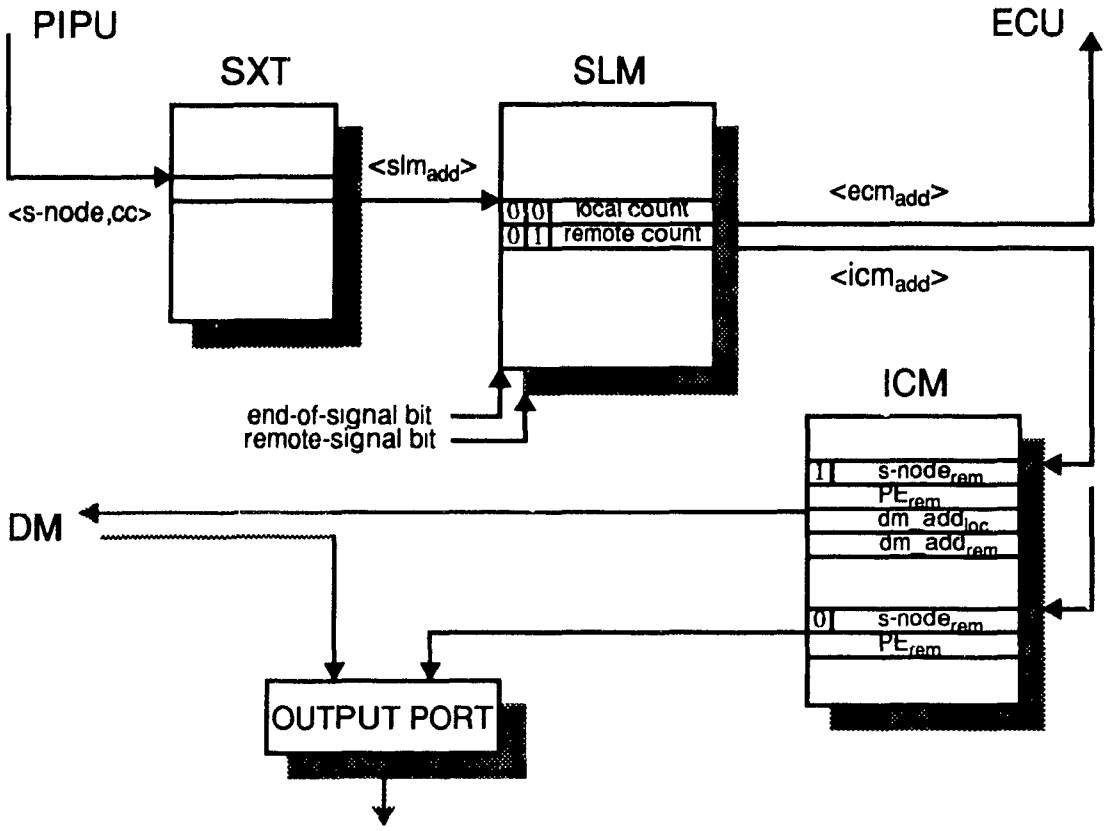


Figure 4.3: Sending a Signal to a Remote PE

We have chosen to store the specifications in a new memory module, within the ICU, which we have called *Interprocessor Communications Memory* (ICM). When a node has to signal a remote node, the remote specifications are fetched in an indirect fashion via a remote address stored in SLM which is recognizable by the signal count dispatching mechanism of the SPU. This can be achieve by reserving a second bit—besides the end-of-list condition bit—to indicate this condition. Thus, a remote address being detected in the SPU automatically gets redirected to the ICU where it is used to access the ICM. All the necessary information to form the required packet can be fetched from this memory before it is sent to the network.

When the ICU receives an ip-count signal from the SPU, the address contained in the signal (say icm_i) is used to access the ICM memory module. The first address that is fetched from the ICM is the address of the remote node (s-node_{rem}) that has to be signaled. The high order bit of that address is reserved to determine the type

of signal to be sent. Setting the bit indicates that it is a data-count signal, otherwise, it indicates a remote-count signal. Figure 4.3 illustrates the process of sending a data-count signal to the network.

Both types of ip-signals require the remote PE address and the remote node address (within that PE) to deliver the signal. For a data-count signal, the remote data address also has to be sent along, so that the value can be stored in the remote Data Memory upon arrival of the packet. Once the ICU detects the data-count signal, it proceeds to fetching from the ICM the next three words stored after icm_i (i.e. icm_i+1, icm_i+2, icm_i+3). They contain the remote PE address (PE_{rem}), the local address of the data value, and its remote Data Memory address (dm_add_{rem}). The local address is then used to fetch from the local Data Memory the data value that will travel along with the signal. The following is the packet structure of the data-count signal:

<data-count, PE_{rem}, s-$node_{rem}$, dm_add_{rem}, data_value>

If the type-of-ip-count bit is not set, the ICU knows that it is a remote-count signal. For this type of signal, only the remote PE address is necessary, besides the remote node address (within that remote PE). That PE address is stored in the ICM memory location contiguous to icm_i. After fetching it, the ICU sends a remote-count packet with the following structure:

<remote-count, PE_{rem}, s-$node_{rem}$ >

The packet header is actually the remote PE address[1]. 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.

Receiving an Ip-Signal

The reception of an ip-signal 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

[1]This will be explained in greater detail in Section 4.2.

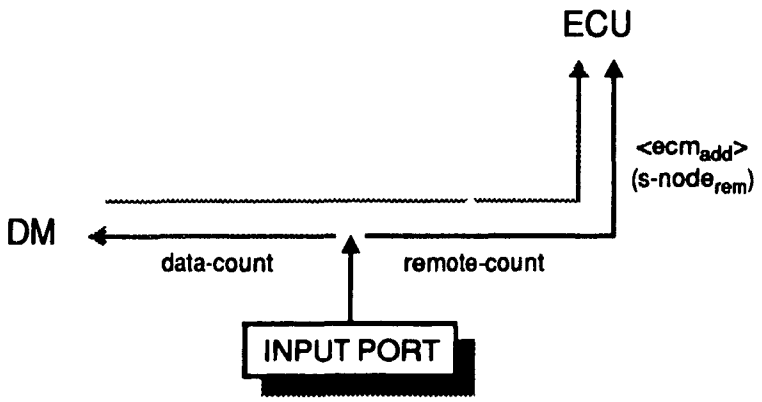


Figure 4.4: Receiving a Signal from a Remote PE

ECU to be processed as a local count. If it is a data-count (see Figure 4.4), 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.

Processing remote signals is therefore achieved independently of the processor's execution pipeline which is one of the main objectives that were set by this project. The DISU is also relieved from the burden since most of the action takes place within the ICU. The implications at this level are two: the homogeneity of the SLM is conserved, and the transparency of the remote signaling is maintained, since all the processing is confined within the ICU. Remote synchronization events are therefore processed in a simple and efficient manner, keeping the principles of modular design.

4.1.2 Processing RLOAD and RSTORE Operations

The remote load and remote store operations are used to transfer data values from the Structure memory (SM) to the Data memory (DM) and vice versa, when the SM is not local to the PE. The reader is reminded that no other operation has access to the SM. To be used in a computation, array values have to be downloaded from SM, using a load instruction, and then uploaded back, if necessary, using a store instruction. These remote memory operations are described as follows:

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

Their implementation is best described by giving a closer description of the first two stages of the execution pipeline beforehand.

The first stage is the Instruction Fetch Unit. It is responsible for decoding the fire signal which is an address in the Instruction Memory. With the information it retrieves from that address, it creates a template, containing the instruction number (n_i), the operator code (oc), and the addresses of the operands and result registers (add_{op1},add_{op2}, and add_{res}):

$$< n_i, \text{oc}, \text{add}_{op1}, \text{add}_{op2}, \text{add}_{res} >$$

The template is then forwarded to the Operand Fetch Unit which computes the effective operand's address and fetches the data values from DM. It is then forwarded to the next stage of the pipeline, the EU.

To implement the remote memory operations, we require that this stage of the pipeline be provided with a mechanism to analyze the operator code and detect the remote load and store instructions. Upon detection of the remote instructions, the template is redirected to a subsection where the "remote" address of the operand (first operand for a RLOAD, second for a RSTORE) is analyzed to extract the target PE address. If the address points to the local Structure memory (SM), then the template is just sent back to the second section and processed 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. When the memory operation is performed, the ICU generates a done signal directly to the DISU, as if the operation had been executed within the local execution pipeline. Figure 4.5 illustrates the way remote array operations are processed within the PIPU and the ICU.

Processing a Remote Load Request

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

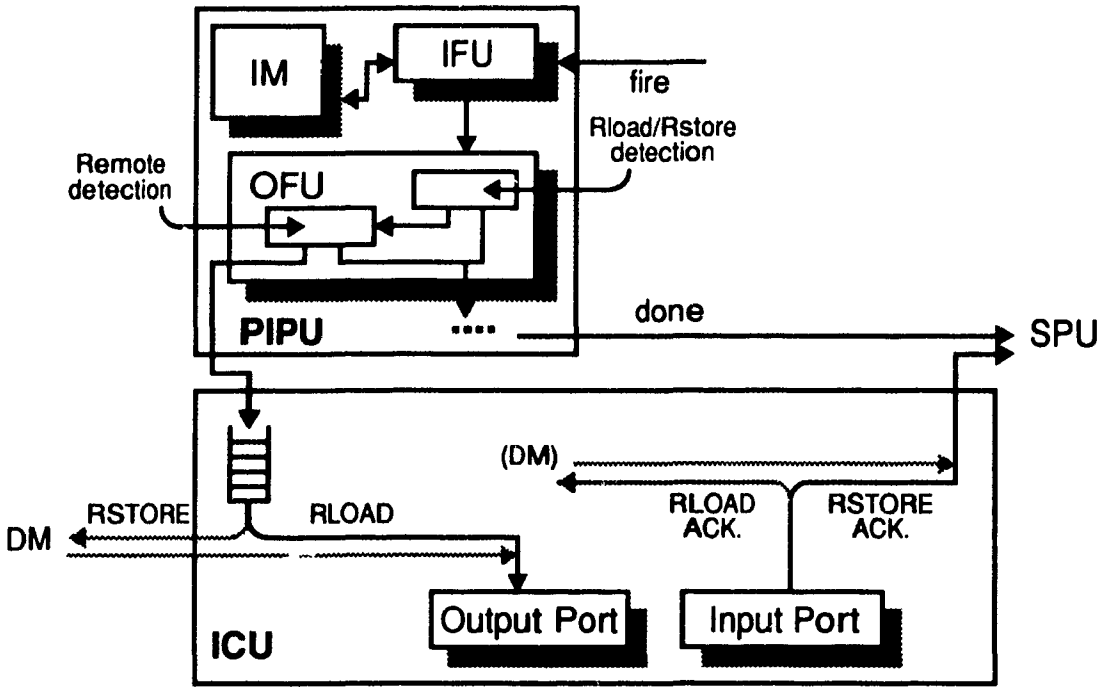


Figure 4.5: Sending a Memory Request to a Remote PE

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:

$$<\text{remote-load}, PE_{rem}, sm\_add_{rem}, PE_{loc}, s\text{-}node_{loc}, dm\_add_{loc}>$$

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 then replies with a message, structured in the following way:

$$<\text{ack-load}, PE_{loc}, s\text{-}node_{loc}, dm\_add_{loc}, data\_value>$$

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[2] 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:

<remote-store, PE$_{rem}$, sm_add$_{rem}$, PE$_{loc}$, s-node$_{loc}$, 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:

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

where s-node$_{loc}$ is the node address used by the ICU in the source PE (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. One of them is the Signal Processing Unit, within the DISU, redirecting ip-counts. Another one is the Operand Fetch Unit, within the PIPU, redirecting instruction templates which contain remote memory requests The third and last one is the input port buffer, receiving requests from the network which have to be processed by the ICU before sending back an acknowledgment.

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

Since all the modules generating the requests are independent, the lower level of design for the ICU should be organized such that all three types of requests can potentially be processed independently of each other. A common entity between the modules is the output port buffer, connecting the ICU to the network, through which all outgoing messages ultimately have to go. A possibility of deadlock can occur if this buffer fills up with too many requests.

Since neither acknowledgment messages nor interprocessor signals can be delayed nor blocked, little can be done about an overflow of such types of requests, besides providing ample buffer space for the input port buffer and the ip-count buffer—connecting the SPU with the ICU. It is possible, however, to delay the remote memory operation requests.

A retry mechanism has been implemented so that upon reaching a certain (low) degree of free available space in the output port buffer, incoming templates containing remote memory operation 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[3]. 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 The Interconnection Network

In order for our processors to cooperate during the execution of a particular application, we must provide them with some interconnection network. Such interconnection network must feature fast and reliable handling of interprocessor communication operations such as remote memory accesses and synchronization of events. This section describes the main characteristics of interconnection networks (IN) and gives some examples of well known network topologies which are available today. It then describes the model we have chosen for the MMDA.

[3]Fire signals are equivalent to s-node addresses as explained in Section 2.3.1.

4.2.1 Interconnection Network Characteristics

There are four design decisions involved in selecting the architecture of an IN: the operation mode, the overall control strategy, the switching method and the network topology (see [26]). The following are the alternatives for each characteristic:

- The *operation mode* can be synchronous, asynchronous or a combination of both. Synchronous systems are characterized by a central global clock signaling all the devices which operate in a lockstep fashion. Asynchronous communications are more suitable for systems in which connection requests are issued dynamically.
- The *switching methodology* to transfer data can be either circuit or packet-switching. With circuit-switching, once a processor is granted a path it keeps it for the duration of the communication. Under the packet-switching method, messages are broken into small packets which compete for paths. In general, circuit-switching is much more suitable for bulk data transmission whereas packet-switching is more efficient for short data messages.
- The *control strategy* can be centralized, where the IN is governed by a global controller, or decentralized, in which the requests are handled independently by the different devices in the IN.
- The *network topology* can be static or dynamic depending on whether the links in the IN are passive, i.e. dedicated, or if they can be reconfigured by acting on the switching elements.

Among these characteristics, the network topology aspect offers the wider range of alternatives.

Static Topologies

A static network topology is one that does not change after the machine is built. Those topologies are usually classified according to the dimensions required for the layout of the IN. For instance, the single bus is an example of a one-dimension static IN. Some well known two-dimensional topologies are the ring and the mesh. At the other extreme of the spectrum, there is the hypercube, a representative of the multi-dimension class (see Figure 4.6)

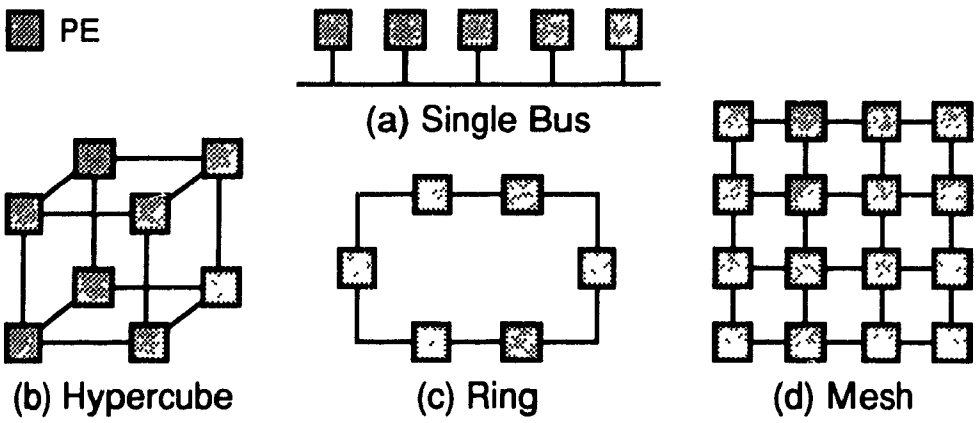


Figure 4.6: Examples of Static INs

Within bus based multiprocessors, individual processors, memory modules, and input/output devices are connected by one or more high-speed data buses, in such a way that all modules are accessible by all processors. The bus requester, driver, and receiver circuits on each device, handle the passing of addresses and data. Since the bus itself contains little or no active logic, it is the simplest of the interconnection architectures. Despite this advantage, though, as bus speed increases, each component attached to the bus must also increase its operating speed, raising the overall cost and complexity. Also, modules can be added to the bus at any time (as long as there is room on the backplane and control circuitry on each device), but the growth is limited: efficiency starts to decrease with additional processors, as each one must compete for the same fixed resource—bus bandwidth.

A different type of static IN, the hypercube, provides multi-step communication paths by connecting each processor to a subset of the other processors [60]. In a hypercube architecture of order N, each processor can communicate directly with N other processors, and indirectly with all other processors in a maximum of N"hops". Intermediate processors are used in a store-and-forward technique to convey the messages. Figure 4.6 (b) shows a hypercube of order three. One of its advantages is its ability to connect hundreds or thousands of individual processors. Thus, the processing power can be quite large, but it can only be reached at the expense of user-generated load balancing techniques, both at the program partitioning phase and at the data partitioning phase. A disadvantage is the overhead involved in the message passing operations that are required for interprocessor communications.

Dynamic Topologies

Dynamic topologies are divided into three classes: single-stage, multistage, and crossbar. These interconnection networks are usually designed so that they can be constructed of a single type of modular building block, the switching element, shown in Figure 4.7 (a). Switches are arranged into stages with data paths, called *links*, connecting the output terminal of a switch in one stage to the input terminal of a switch in the next stage. Input terminals of switches in the first stage are called *input ports* or *source nodes*, and output terminals in the last stage are called *output ports* or *sink nodes*.

A single-stage IN is composed of a stage of switching elements, which is based on a special connection pattern. The example shown in Figure 4.7 (b) is a shuffle-exchange network, based on the perfect-shuffle connection [46]. In a single-stage network, data may have to be passed through the switches several times before reaching the final destination; hence it is also called a *recirculating* network. The way the connections are configured determines the functional characteristics of the network. Most of these networks have a cost proportional to the number of processors they interconnect and their bandwidth increases as processors are added. However, a disadvantage with these networks is that requests cannot be pipelined, which forces a delay—until all the passes through the switches are completed—before more requests can be sent.

The crossbar interconnection is organized in a grid pattern allowing all possible connections between processors. It is shown in Figure 4.7 (d). Crossbar connection networks offer a possible solution to the limited-bandwidth problem due to their use of separate buses to connect each processor with each other. Contention appears if two processors attempt to access the same target module, in which case an arbiter temporarily delays one of the requesters. Theoretically, the crossbar interconnection has no upper limit to the addition of modules, but the number of grid connections unfortunately grows as N^2. This makes it an expensive and complex architecture for large-scale multiprocessing (a network interconnecting 128 PEs requires 16,384 connections, each of which might contain several wires)

Multistage networks provide many layers of switches, many of which are built from stages of the basic single-stage networks. The characteristics of the connection pattern are often used to classify these multistage networks among themselves. Figure 4.7 (c) shows an Omega network [47].

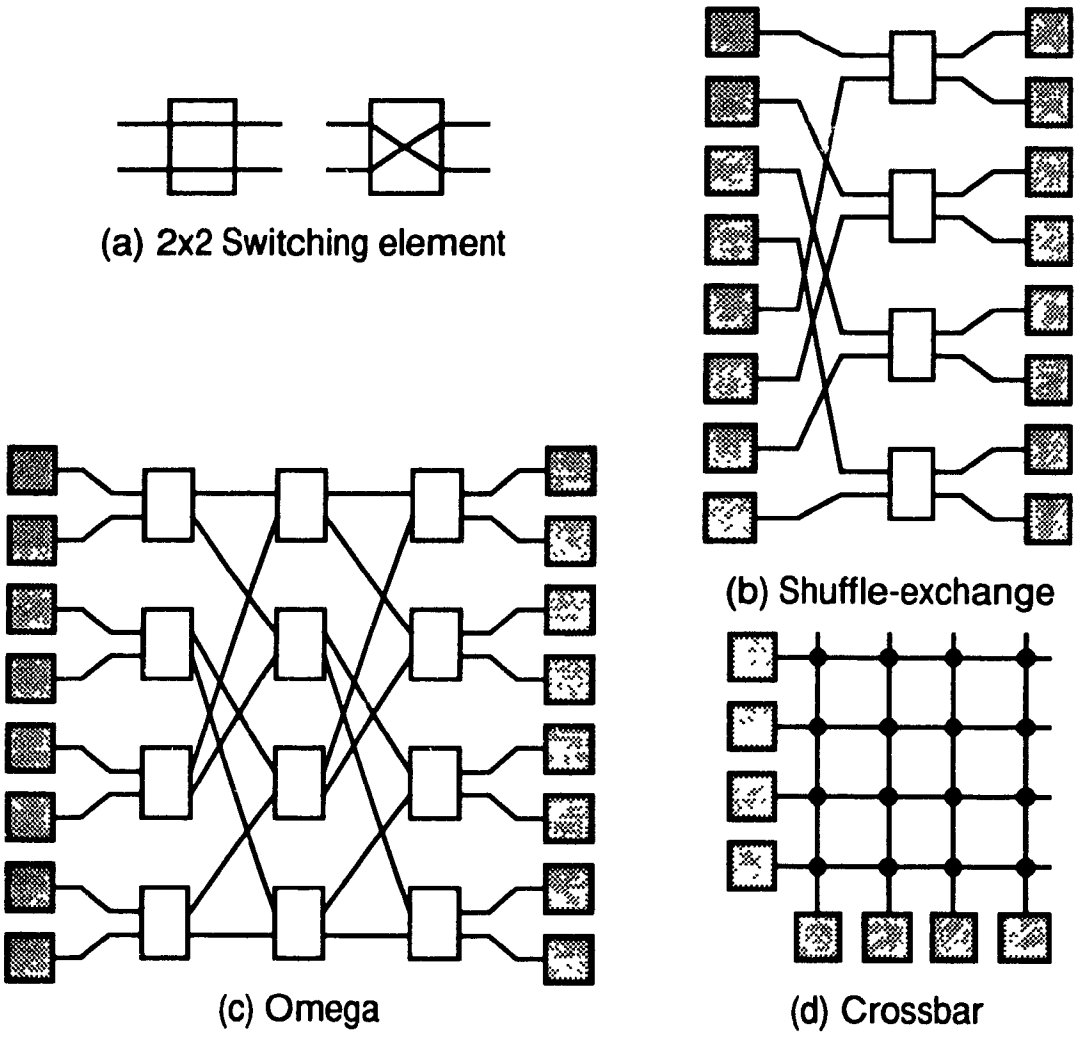


Figure 4.7: Examples of Dynamic INs

These networks combine the better aspects of all three classes of models: they allow direct connections between processors and support systems of hundreds or thousand of processors; the communication bandwidth capacity incrementally increases as processors are added; the complexity of the network for an N processor system only grows as $O(N log_i N)$, where i is the size of the switches (i×i); and the delays through the IN are proportional to $log_i N$.

A lot of research has been done on the performance of these INs, proposing a myriad of enhancements—like alternate/redundant paths, combining messages, buffered switches etc.—to increase the overall throughput. For more precisions, the reader is

referred to the references [26, 58, 23, 24, 45, 54].

4.2.2 A Multistage Omega IN for the MMDA

As we shall see in Chapter 5, the IN in a multiprocessor dataflow architecture has a significant impact on the performance of the entire system.

Pipelining multiple requests to the network is one of the features which can be exploited by the MMDA system. This is not effectively handled by single-stage networks which is the reason why they have not been considered. The crossbar type of IN seemed to meet the requirements for fast response time and small delays, in the case of contention, by providing direct paths from any input to any output. However, the complexity and growth expansion factor make it a too expensive approach for a dataflow multiprocessor. On the other hand, the multistage type of IN is currently attracting much of the attention in the field, mainly because of its flexibility and attractive features.

Delta networks [56] are constructed with switches of size $i \times i$, arranged in $log_i N$ stages of N/i switches. The connection patterns between stages allow any input port to send a message to any output port. The control of data movement through the network does not require a global controller. Instead, it is implemented locally at the switch level: the destination address digits in the message header are used to route the packet through the network–thus $log_i N$ digits in base i. At each stage, the switch uses a digit of the destination address to select which of its output ports to use.

If a message arrives at a switch and is directed to an output port which is already in use, a collision occurs, which causes the arriving message to block. A collision can also occur if two messages simultaneously arrive at a switch and are both directed to the same output port. The switch then has to use some arbitration strategy to pick which message gets blocked and which one goes through. If there is no collision, the switch is capable of passing them all to the next stage.

The links between the stages sometimes contain FIFO buffers of a predetermined length. All links, however, are homogeneous. The advantage is that the buffers can temporarily hold a message in case it is blocked at a switch, instead of disregarding it. In this latter case, the message has to be reissued.

Because high throughput is important, and because of the relatively small size of the messages, packet switched networks have been considered for this thesis instead of circuit switched networks. Also, since the requests in the MMDA are issued in a dynamic fashion, the network's operation mode should be of the asynchronous type.

Based on the characteristics laid above, we have chosen a multistage Omega type of network—a particular class of Delta network— to support the interprocessor communications within the MMDA[4]. We have thus considered a packet-switched, asynchronous Omega network, consisting of $log_2 N$ stages of 2×2 buffered switches, with $N/2$ such switches per stage (see Figure 4.7 (c)). Thanks to its decentralized type of control, messages can be propagated in a pipeline fashion, progressing through the network independently of each other.

4.3 Summary

This chapter proposes a modification to the McGill Dataflow Architecture to efficiently implement the interprocessor communication methods described earlier in the thesis. The Interprocessor Communication Unit has been added to the Architecture, yielding the McGill Multiprocessor Dataflow Architecture.

It effectively handles the ip-counts (which are remote count signals) that are redirected from the Signal Processing Unit. Its direct link to the Data Memory allows it to fetch the data values that have to be paired with the data-count signals, whereas normal remote-count signals are just redirected toward the IN. Incoming signals are processed in a similar way, with the data value being stored in memory, if necessary, and the signal being redirected to the Enable Count Unit.

A direct connection with the Structure memory allows it to process remote requests without interrupting the main execution unit in a straightforward manner. These requests are issued by the PIPU of the remote PE which has the ability to detect a remote memory address within a remote instruction template. When the address points to a the local PE, the instruction is kept in the execution pipeline and processed like a regular local memory operation. Otherwise, they are redirected

[4]Notice should be done, 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.

to the ICU which sends an appropriate packet to the network. The reply ultimately generates a done signal to the Signal Processing Unit, as if the instruction would have been executed locally.

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. The architectural design has kept its simplicity and modularity. It also provides a retry mechanism to redirect memory requests away from the ICU, in case the unit is caught into too much traffic, thereby eliminating a probable source of deadlock.

A multistage interconnection network is proposed to support the actual transfer of the messages from PE to PE. It is a packet-switching, asynchronous Omega network. It is best suited to the idea of a scalable multiprocessor, specially due to the delay growth which is proportional to $log_2 N$, and the complexity growth which is proportional to $N log_2 N$.

Chapter 5

Performance Evaluation

To investigate the impact of long latency memory operations on the performance of multiprocessor dataflow programs, we have developed a simulation testbed. It includes the implementation of an assembler for multiprocessor systems (*mdasm*), and a low level architecture simulator (*mds*) written in AT&T Concurrent C [36]. In this chapter, we report the results we have obtained from testing the simulator with various benchmarks, and we give a performance analysis based on those results.

The simulator has been designed to closely model the McGill Multiprocessor Dataflow Architecture, including an interconnection network, up to a maximal configuration of 16 processing elements[1]. A more detailed description of the program can be found in an associated technical report [51]. The interprocessor communication schemes, which we have developed and presented in the previous chapters, are an integral part of this simulator. Their efficiency has been evaluated by running various simulations of multiprocessor dataflow programs, executing on different system configurations. Better yet, we have been able to get accurate measures of the performance of the overall system, thereby providing a deeper insight on the behavior of multiprocessor dataflow programs.

The simulations which are presented in this chapter show that the MMDA is a system which can potentially (1) effectively tolerate high latencies during interprocessor memory operations, and (2) inexpensively support remote event synchronization.

[1]This limit is set by the actual implementation of the language at McGill, which restricts a Concurrent C program to run on one single machine. Future releases will allow programs to run in distributed mode, thereby increasing this limit.

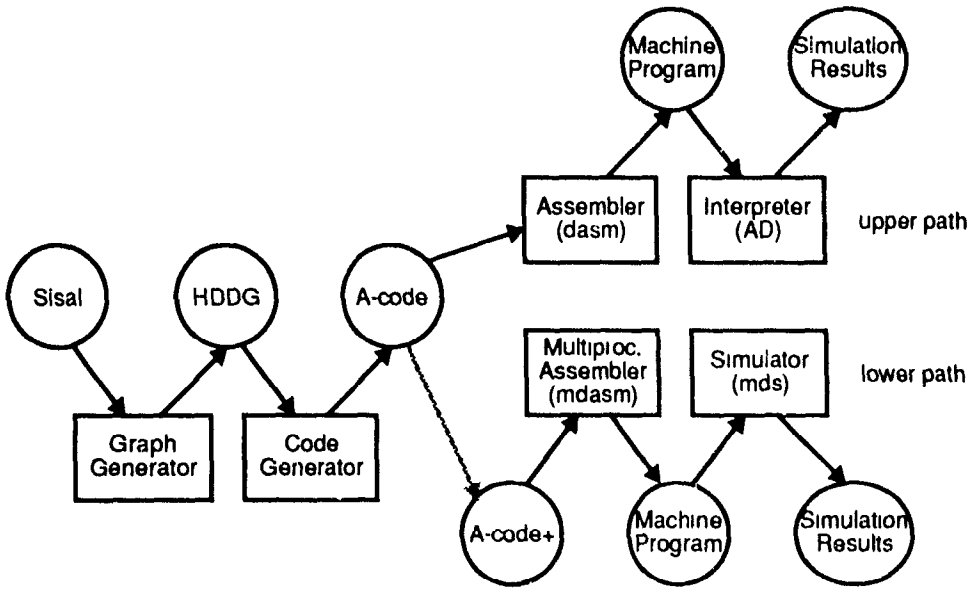


Figure 5.1: The Testbed Environment

It turns out that network throughput is a more critical factor than latency, in trying to achieve high performance.

5.1 The Testbed Environment

In this section, we give a description of the testbed environment that was used to simulate the multiprocessor system. The original testbed was developed for the MDFA and was therefore oriented towards the compilation and simulation of dataflow programs executing on a single processor machine [31]. This environment is shown in Figure 5.1 and represented by the upper path of the diagram. The work accomplished within the framework of this thesis involves the development of a new testbed, starting from the assembler down to the simulator. This new testbed is illustrated by the lower path in Figure 5.1.

The path along the testbed suite, for the uniprocessor architecture, 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 MDFA Starting from the SISAL front end, SISAL benchmark programs [49] are translated into a hierarchical data dependence graph (HDDG) [41]. A code generator translates the

machine-independent HDDG forms into an intermediate representation of data flow graphs, which we have called A-code [62]. In A-code, the simple nodes of the HDDG form are mapped into instruction tuples. An assembler (*dasm* [50]) is used to generate executable code for an interpreter/simulator (*AD* [63]) of the architecture.

The framework of this thesis is focused on the architectural design of interprocessor communication schemes. The simulations were prepared in the following way: SISAL benchmark programs were translated to A-code graphs, and then manually modified into multiprocessor programs, written in an extended A-code language with multiprocessor flavors (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*[2]. Appendix B shows an example of a A-code program.

5.2 Mds: a Multiprocessor Dataflow Simulator

Mds is a simulator for the MMDA written in AT&T Concurrent C which consists of two parts, a processor simulator and a network simulator. In the version we are currently using, it supports configurations of up to 16 processors interconnected by a packet-switched Omega network. It gathers information such as simulation times, network throughput, latency of requests, processor utilization and others. It was designed so that processing elements could be easily added, and to simplify the implementation of new communication network topologies.

The ICU, the two major units of the DISU, each of the sections of the PIPU, the memories, and the network are implemented by distinct self-contained code units which simulate the functionality of each hardware unit. The three major units of the PE (PIPU, DISU, ICU) and the network, operate under the same machine clock cycle. This machine cycle is the basic unit of time used by the simulator to measure all its results.

Assumptions regarding the execution time have been made in terms of the work that can be performed by each module within one unit of time. The PIPU can potentially receive one fire signal per clock cycle. All scalar operations and local Structure memory operations within the execution pipeline are assumed to have the

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

same processing delay, so each instruction going through the PIPU consumes the same number of units of time (the default value is 6). Within the DISU, the SPU can receive one done signal per cycle, retrieve the corresponding signal list and send it to the ECU. This latter can process n count signals per cycle, n being a machine characteristic and thus a parameterized value in the simulator (default value 4). The ICU can process one type of request per cycle, be it a request from the SPU (ip-count), a remote memory request coming from the PIPU, or an interprocessor request. These requests are serviced in a round robin fashion. Communications between the units are smoothed by means of FIFO buffers, where each element is paired with a time stamp determining the time unit at which it becomes available.

The network simulator can be set to operate at a faster rate than the processor by varying the ratio of $\frac{PE}{network}$ clock cycles. The movement of packets into and out of the network, and from stage to stage within the network, is assumed to take place at discrete, equally spaced points in time. The time between these points is the minimum delay experienced by a packet at a switch, based on the data rate supported by the link. Analysis based on this assumption can be used to approximate the behavior of an asynchronous network.

5.2.1 Simulator parameters

The simulator offers a wide number of parameters that allows it to perform simulations of various technological aspects, concerning both the PE's internal design, and the network characteristics. Parameters are also used to generate traces of various kinds of events.

A complete list of all available options can be found in Appendix C. The following are the ones most relevant to this chapter:

- **-S:** this parameter tells the simulator the amount of the Structure memory (SM) that each PE dedicates to the system's global shared memory. This value is the same for all PEs. The default is determined by the smallest Structure memory size, among the PEs.
- **-Nc:** this option determines the PE-to-network clock ratio (*ClockRatio*). It provides the possibility of experimenting with various network speeds[3]

[3]More flexibility can be reached by modifying the internal variable defining the data rate of the links.

- **-Nl:** this options sets the start-up latency for each packet sent on the network, i.e. the number of cycles necessary to form a packet. The default value is 2 processor cycles.
- **-Ns:** this parameter determines the switch buffer size within the network.

5.2.2 Performance metrics

The performance metrics used by the simulator are divided into two groups, the first of which reports on the performance of the processor and the second on the performance of the network. Each simulation run thus produces a result file for each individual PE, containing overall system results, local processor performance results, and network performance results. A sample result file is given in Appendix C.

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

- The *Processor Execution Time* (*ProcExecTime*) reports the number of machine cycles consumed by the PE.
- The *Remote Requests* (*RemReq*) is the number of remote memory operations (RLOAD and RSTORE) generated by the PE.
- The *Remote-Operations Ratio* (*RemRatio*) is the percentage of remote memory operations over the total number of instructions executed:

$$RemRatio = \frac{(RemReq \times 100)}{InstExec}$$

- The *Local Average Latency* (*LocLatency*) is the cumulated latency of each remote memory request, averaged over the total number of requests. 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 that are required to execute the whole application. It is defined as:

$$ExecTime = \max_{i} \{ProcExecTime_i\} \ , \quad i = 1 \rightarrow N$$

- 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 can be 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 *DataRate* refers to the data rate supported by the network links, i e. the number of bytes that can go through a link per network cycle The adjustement with ClockRatio yields a data rate per processor cycle.

- The *Global Average Latency* (*GlobLatency*) is the total of all the LocLatency metrics, averaged over N. Thus,

$$GlobLatency = \frac{\sum_{i=1}^{N} LocLatency_i}{N}$$

- 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.

$$CollRate = \frac{\sum Collisions}{BusyNet}$$

5.3 The Benchmarks

The benchmark programs we have selected to test the simulator are mostly scientific applications. Our choice is mainly due to the fact that 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.

The Livermore Loops [27] are a collection of typical loops, extracted from widely used scientific applications which were developed at Lawrence Livermore National Laboratory. Their advantage, as benchmarks, is their big appetite for processor cycles. The kernels capture the inner loop calculations which constitute the most computationally intensive portions of the applications from which they are extracted. We have used *Loop1* and *Loop7* from the collection. For Loop1, the main loop is unrolled twice to increase the parallelism of the program. This is enough to obtain a high rate of processor utilization.

Saxpy is the kernel of the Lower/Upper Decomposition, a basic computation procedure widely used in linear algebra for solving systems of linear algebraic equations such as the famous LINPACK [25]. The routine computes a vector as the sum of a scalar value times a vector plus a second vector. The benchmark has been modified to make it more computationally intensive, by adding the sum of a scalar value times a vector yielding the following expression: $a \times X[i] + b \times Y[i] + c$. The main loop is also unrolled twice to increase the parallelism in the program and thus reach a better processor utilization rate.

Matmul is a matrix multiplication program which has been directly hand coded into A-code. For each simulation, the result matrix is a (32×32). The first input matrix is of size (32×8) and the second is a (8×32) matrix. This is done to keep the problem size (and thus the duration of the simulations) at a reasonable level. 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 this load will create. Within each PE, the outer loop is partitioned into two independent code blocks, to increase the parallelism.

Each of these benchmark programs has been coded using a technique called *software pipelining* [31] to increase the amount of exposed parallelism. The source code for these benchmarks and a graph representation, can be found in Appendix A.

Benchmark	Size of Problem	InstExec	Memory Operations	ProcExecTime	ProcUtil
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 %

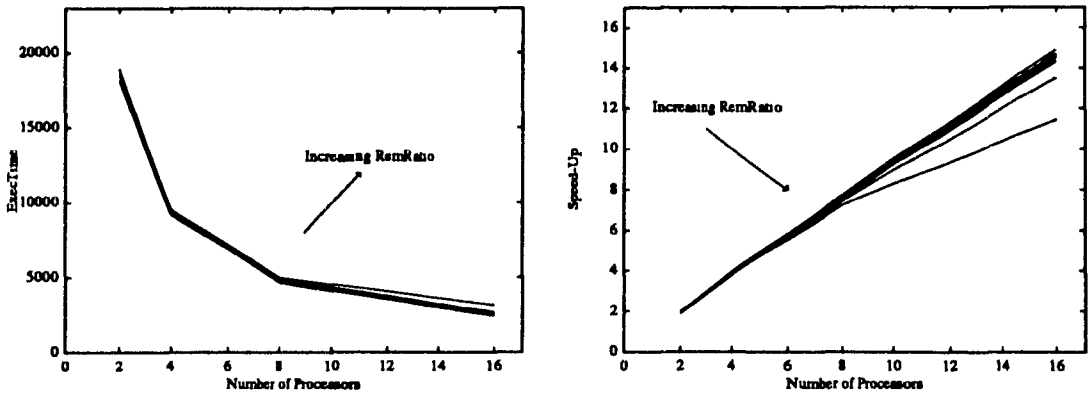
Table 5.1: Performance Results for a Single PE

5.4 Simulation Results

In this section we present the results we have obtained from testing several benchmark programs on mds. The simulator's default parameters have been set, using reasonable numbers for processor and network characteristics 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 starting point of the testing process is the execution of the benchmark programs on a single processor. The performance results are presented in Table 5 1 The problem size refers to the size of the Structure memory required by each of the PEs to run the programs. For instance, in Saxpy, there are 3 vectors of 1280 elements involved so the problem size is 3840. The percentage of array memory fetches, over the total number of instructions, is shown in the fourth column This number indicates the number of instructions that can potentially translate into long latency load operations in a multiple processor environment.

Notice that the programs feature a very high processor utilization, due to the unrolling of the loops, for the particular case of Loop1 and Saxpy. The reason for targeting such high percentage utilization is due to the fact that each PE has to have



(a) Execution Time (b) Speed-Up

Figure 5.2: Performance of Loop1

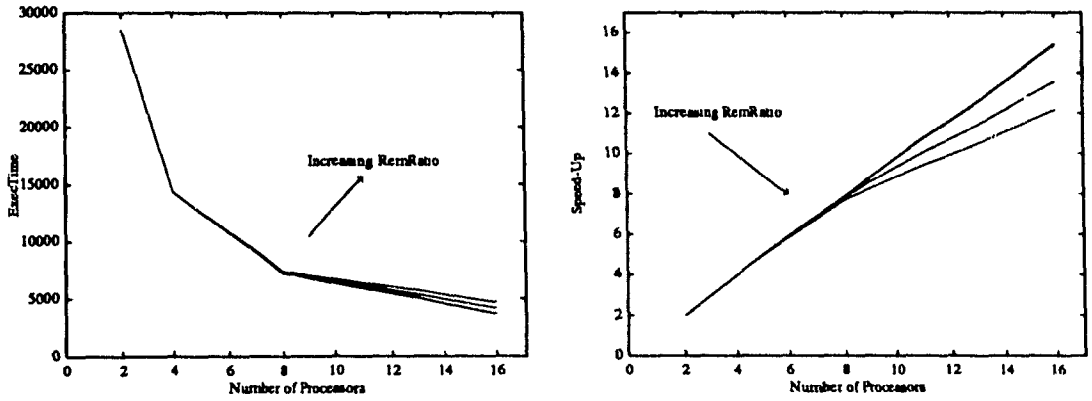
enough parallel computations to keep its execution pipeline sufficiently busy in order to absorb the effects of long latency operations.

The second phase of the testing process investigates the programs' performance on a multiprocessor system. The 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. In the following sections, we report on the execution times, speed-up curves, processor throughput, and memory latency, as we look at the performance of the MMDA.

5.4.1 Execution Time and Speed-Up

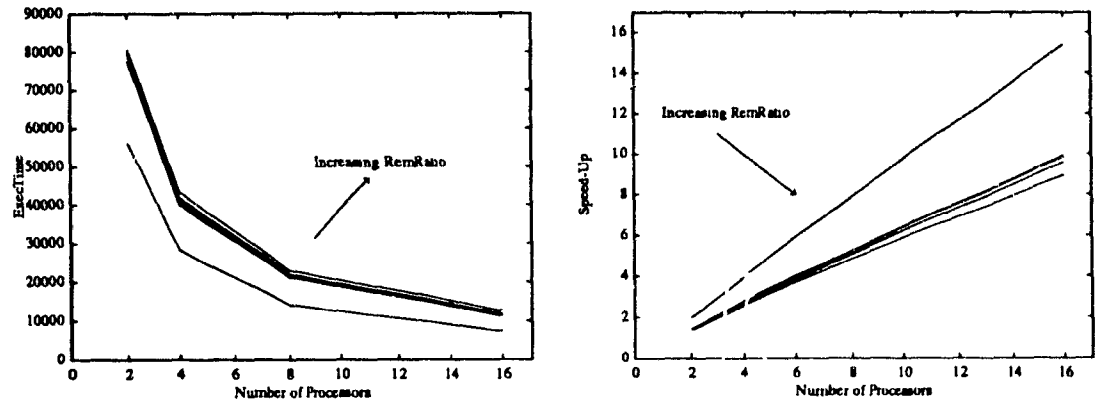
Linear speed-up is trivially achieved if the *remote-operations ratio* (RemRatio) for each processor is zero—meaning that all the array memory operations are accessing the local SM. A RemRatio of value zero translates into portions of code, running in each PE, which are completely independent of each other. The amount of work is therefore simply divided among the available processors, thereby dividing the execution time in the same proportion

A more interesting set of results is obtained by gradually incrementing the RemRatio from 0 to its maximal value (which is given in Table 5.1). When this maximal



(a) Execution Time (b) Speed-Up

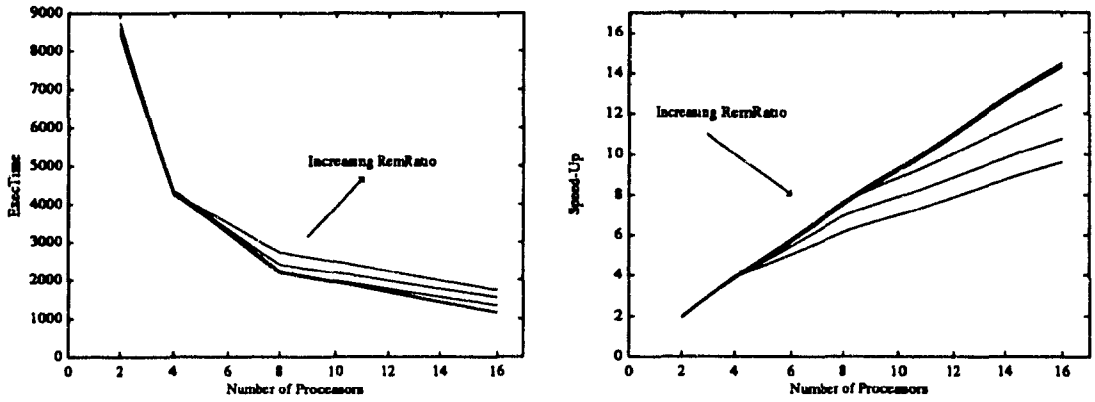
Figure 5.3: Performance of Loop7



(a) Execution Time (b) Speed-Up

Figure 5.4: Performance of Matmul

value is reached, all memory fetches are remote to the PE. The Figures 5.2– 5.5 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 the RemRatio The increment value of the RemRatio is not the same for each of the benchmarks This explains why some graphs have more curves than others. For Loop1, the spectrum is



(a) Execution Time (b) Speed Up

Figure 5.5: Performance of Saxpy

divided into 6 intervals yielding 7 performance curves. For Loop7 there are 10 curves, while there are 5 curves for Matmul and 9 for Saxpy. Table 5.2 shows the different RemRatio values that have been used to test each of the benchmarks, and their corresponding NetAcc. 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.

Both Loop1 and Loop7 have similar program structures and a maximum RemRatio neighboring the 13% of instructions, i.e. one out of 7.7 instructions is a remote load. 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 83.35% of their data, for Loop1, and more than 88.87%, for Loop7. These extreme cases are illustrated in the speed-up figures by the two lowest curves of each graph with RemRatios of 11.11% and 13.33% for Loop1, and 11.26% and 12.67% for Loop7. The corresponding loss of performance will be explained shortly (see Section 5.4.2). If we focus on the majority of the cases, 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 there is enough parallelism available to keep the processor busy with useful instructions while waiting for each of the remote loads, which translates into the fact that the MMDA can effectively tolerate latency.

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	1.41	2.82	4.22	5.63	7.04	8.45	9.86	11.26	12.67
	0	1.6	3.2	4.8	6.4	8.0	9 6	11.2	12.8	14.4
Matmul	0.00	4.41	8.82	13.23	17.64					
	0	8.2	16.4	24.6	32.8					
Saxpy	0.00	1.96	3.92	5.88	7.84	9.79	11.75	13.71	15.68	
	0	0.64	1.28	1.92	2.56	3.2	3.84	4.48	5.12	

Table 5.2: Remote-Operations Ratios Used to Test Each Benchmark

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 degrades as soon as it reaches the 8 PEs configuration, when the RemRatio reaches its maximum value of 15.68%. Intuitively, the observations lead us to a premature 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%.

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 5.2, 5.3, and 5 5.

The results concerning Matmul show a distinct behavior compared to the other benchmarks. With zero RemRatio, execution time is optimal, and speed-up curve is quasi-linear. However, the 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 almost does not affect 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 reason for chosing this algorithm

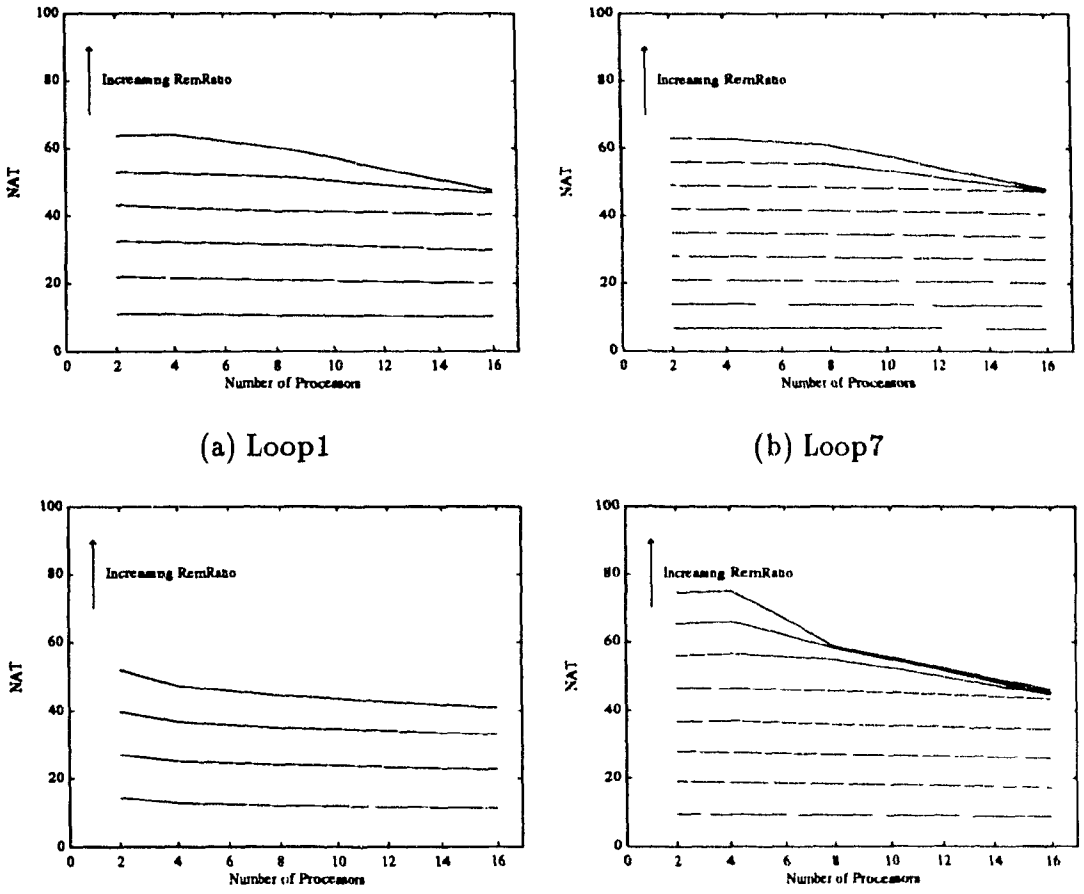
is to provide an example of a very computationally intensive program which does not expose enough parallelism to support remote data fetching. The loss on performance is more precisely due to the fact that there are not enough instructions to execute while a particular path in the graph is blocked due to a long latency operation. This situation does not occur with the other benchmarks because of the more numerous paths available through the graph. Appendix A shows a graphical representation of the benchmarks.

5.4.2 Network Throughput

The performance of a packet-switched interconnection network is characterized in terms of how much information can be passed through it in a given period of time. This criteria, known as *throughput*, can be informally defined as the number of packets the network can accept at the input port per unit of time. The corresponding metric used by mds to describe the network's performance 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 5.6 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,



(a) Loop1 (b) Loop7

(c) Matmul (d) Saxpy

Figure 5.6: Network Average Throughput

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 [24] 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. 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.

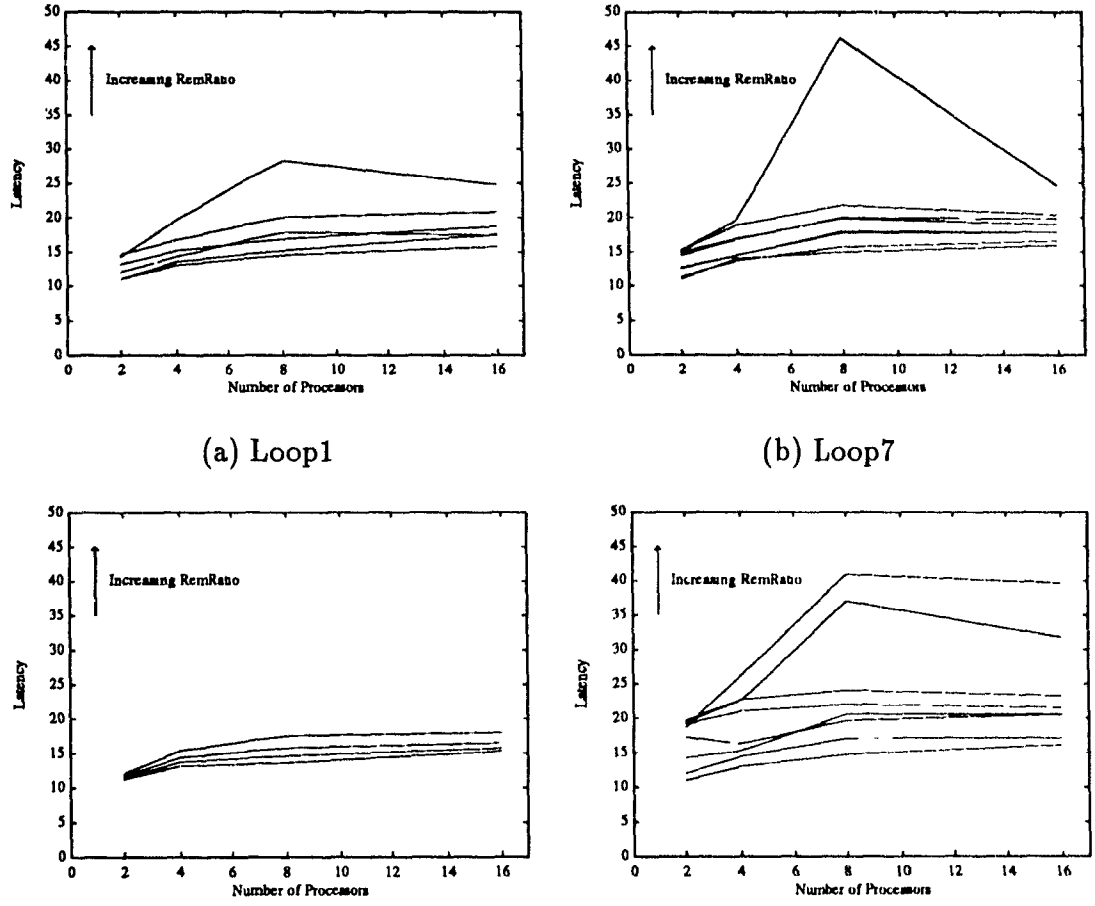
5.4.3 Latency

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.

The average latency of a packet is expected to increase as the logarithm of the number of processors in the system (see Section 4.2). 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 5.7. As before, each curve in the graph corresponds to a different value of RemRatio, from the bottom up.

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 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 5.8 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. In other words, 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 in the GlobLatency. As can be observed in Figure 5.8, CollRate does not increase proportionally to the system size for the two upper curves of Saxpy (RemRatio of



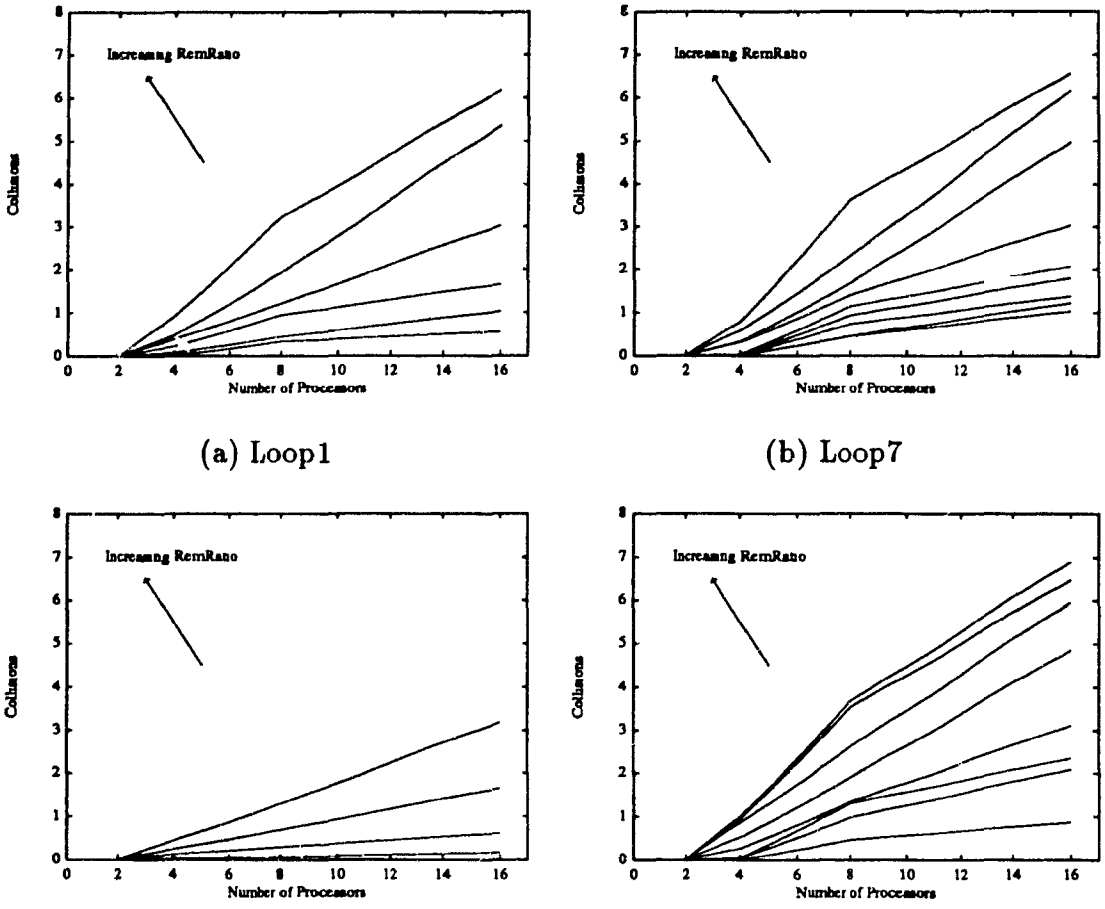
(a) Loop1 (b) Loop7

(c) Matmul (d) Saxpy

Figure 5.7: Global Average Latency

13.71% and 15.68%), the top curve of Loop7 (RemRatio of 12.67%) and the top curve of Loop1 (RemRatio of 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



(a) Loop1 (b) Loop7

(c) Matmul (d) Saxpy

Figure 5.8: Global Collision Rate

5.5 Summary and Discussion

5.5.1 Summary

The results of the simulations are very encouraging. Basically, they show that the MMDA can perform remarquably well, given a high bandwidth interconnection network.

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 ha're 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 h gh 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, 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 [26, 58, 23, 24, 45, 54].

5.5.2 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) Lets 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. Lets 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 each loop body can be the basis of a good estimate of RemRatio. Finally, lets assume that the network has no MRT and thus execution is 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

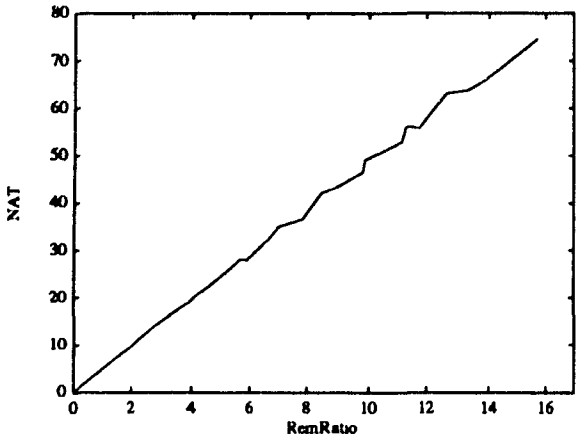


Figure 5.9: Network Average Throughput versus Remote-Operations Ratio

which the network does not act as a bottleneck against the values of RemRatio, we obtain the graph shown in Figure 5.9. It can be observed that there is 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 an 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.

5.5.3 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.

Dataflow programs are written in the forms of graphs, with a number of inputs, and outputs. Examples of such graphs can be found in Appendix A. 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.

Chapter 6

A Survey on Related Work

In this chapter, we present a brief survey of the related work that has been done in the field of dataflow computation, concentrating on the multiprocessor aspects of the architectures. Our concern is to compare our multiprocessor model with some of the other static or dynamic multiprocessor dataflow architectures that have been recently proposed. We are interested in analyzing the philosophy of varying approaches towards the multiple-PE environment, on the basis of the types of data transfers which are provided, and how they are implemented. In the first section of this chapter, we will look at some of the drawbacks and advantages of our model in comparison with the mechanisms which are investigated in other dataflow projects focusing on the types of interprocessor data transfers. We will then briefly survey the methods being used by some of the other architectures that effectively tolerate latency to implement those transfers

6.1 Interprocessor Data Transfers

Static dataflow architectures differ in many ways from the dynamic models. These differences have been stressed in the introductory chapter of this thesis. However, in our particular case, a closer look at the approaches towards multiprocessing reveals that there are some common aspects.

Essentially, there are two kinds of transfers that can take place between PEs that involve data values: DM-to-DM transfers, and SM-to-DM or vice-versa. The first type

of transfer takes place when a node produces a token and sends it to its successor. In the dynamic model, as in any argument-flow static model, the token carries the data value to be delivered. Upon execution, a remote destination token gets automatically redirected to the interconnection network by a Send unit. This is the basic mechanism as all tokens pass through this unit. The receiving PE then delivers the token to the Update unit which verifies whether it enables an instruction or not. This method was found inefficient for the processing of local tokens. However, it turns out being most effective for remote tokens because it minimizes network traffic Due to this particular feature, our static dataflow architecture ultimately adopted a similar mechanism for remote DM-to-DM transfers, represented by the data-count signal.

On the other hand, SM-to-DM data transfers are not so similar. To solve the problem of large data structures, the vast majority of dynamic architectures rely upon the concept of *I-structures* [10] or on some variant—at least those which derived from the Tagged-Token Data-Flow Machine, the Manchester Dataflow Computer having no explicit structure storage. An I-structure can be thought of as a storage space for data values which are governed by the single-assignment rule. I-structures usually reside in a separate memory called the *I-Structure Storage*, which is a globally addressable memory. There are some exceptions, though, like the Monsoon Architecture which implements them in the local memory. The usual operations that are allowed are: the allocation of storage, *I-fetch*, and *I-store*.

Our model also has a separate Structure memory, globally addressable by all PEs. Programs can use Rload and Rstore to transfer data values in and out of the SM. The major difference between both models resides in that there is no special hardware such as "presence bits" to support fine-grain synchronization Whereas the I-Structure Storage provides a mechanism by which the requesters are queued if the value has not been produced, in the model described in this thesis, the read-before-write situation has to be controlled by software means

Depending on the situation, the overhead generated by this type of control can be negligible, or become a serious drawback of the model The latter is illustrated by the situation where an array A is produced within a processor P1 in an order of increasing index, and consumed by processor P2 in the opposing order [30] In our model, this producer-consumer style of parallelism cannot be efficiently exploited. Dynamic models, on the other hand, generally allow the I-fetches in the consumer processor to fire and eventually accumulate while waiting for the data value to be produced. When this event happens, the release of the waiting fetches is automatic

and immediate so the situation is effectively handled. If the array is consumed and produced in the same order of increasing index and the consumer P2 can be detected at compile-time, then the producer-consumer pair can be software-pipelined. Our model can then choose between running code on the consumer PE to fetch data by means of the Rload instruction, or make use of a mechanism of FIFO buffers, to regulate the flowing of values between processors. In both cases, the overhead is minimal and there is no loss of parallelism exposure.

P-RISC is a hybrid von Neumann and dataflow architecture based on the dynamic dataflow model [52]. Memory is organized into frames, used to hold the set of operands associated with each instantiation of a code block within a program. This concept is also at the heart of the Monsoon architecture, but unlike Monsoon, it eliminates all forms of operand matching schemes, and uses a local memory to hold the frames. Shared data structures are supported by a global memory space, the Heap Memory, implemented as an I-structure memory. Load/store instructions are used to generate read/write messages to transfer values in and out of the Heap memory, and special I-read/I-write to take care of those producer-consumer situations. The mechanisms for sending and receiving data to this memory are very similar to the ones presented in this thesis. Both types of operations are extracted from the execution pipeline at the operand fetch level and are converted into a packet which gets sent to the global memory. The difference lies in that the write immediately generates a continuation token and inserts it in the the token queue, without waiting for an acknowledgment from the Heap memory. In our case, the done signal, equivalent of the continuation token, is only sent to the DISU after the acknowledge comes back. The reason is that the Heap provides a mechanism for deferring the read instructions on empty locations, whereas we have to rely on node synchronization mechanisms to insure that the read does not anticipate the write.

In his *New Static Dataflow Architecture*, extensively described in [17], Dennis introduces a novel multiprocessor based on the argument-fetching principle. In this architecture there is no notion of global memory. Each PE has a Memory Switch that acts as a local memory and holds both the code (instructions) and the data (operands) as well as the data structures Thus, there is no notion of SM-to-DM transfers, all transfers being done from one Switch memory to another Switch memory. They are implemented by means of send/receive instructions in a way similar to the method described in Section 3.1.1: upon computing a data value, the producer executes a send instruction which ultimately triggers a receive instruction on the remote PE which writes it to its local memory.

As pointed out by Dennis, values are transferred as they are created, no buffer is needed, and communication can proceed concurrently with computation All these are features which can be found in our model as well. However, our scheme provides more flexibility, by implementing the Rload/rstore operations. The reason is that these operations provide the possibility for a local processor to fetch data from a remote processor without affecting it's execution pipeline. As we mentioned earlier, these situations occur when the compiler is not able to detect the identity of the consumer PE beforehand. Such situation cannot be solved in an elegant way with a send/receive scheme. Furthermore, even when it can be identified, there can be cases where data values are not produced by code (input arrays, for example) For these cases there is a non negligible overhead incurred by the remote PE who has to execute code to send the data values to the consumer PE This overhead is avoided in our scheme because we provide a flexible mechanism that allows each of the PEs to fetch their values by directly accessing the remote PE's Structure memory. Finally, notice that communication and computation can only proceed concurrently if both the consumer and the producer generate the index in the same order Failure to do so implies that the consumer has to wait for the producer to finish, and send the values in the proper order In our scheme, again, the overhead of sending the values is avoided for the producer PE.

6.2 Avoiding Duplication of Instructions

In Section 3.1.1, we set forth the problem of instruction duplication as a drawback of one of our earlier schemes for implementing interprocessor communication of data The situation is illustrated in Figure 3.3 which is reproduced in Figure 6 1, where a node in PEA produces a data value to be consumed by three remote nodes Instruction duplication is usually not a problem within the argument-flow based architectures since the multiple destination list of a token generates automatic duplications of the result data value instead.

The MMDA is an argument-fetching architecture, but as we previously explained, the DM-to-DM remote transfers of data values are accomplished more efficiently if they are done in an argument-flow fashion. It turns out that our scheme for processing multiple remote target nodes avoids the duplication of instructions as well, which brings it close to the schemes upon which the pure argument-flow architectures are

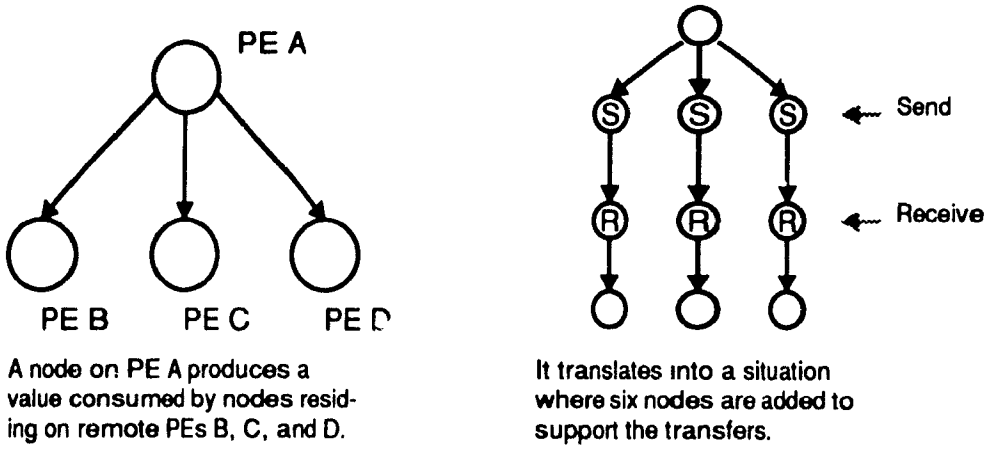


Figure 6.1: Instruction Duplication to Send Data to Remote PEs

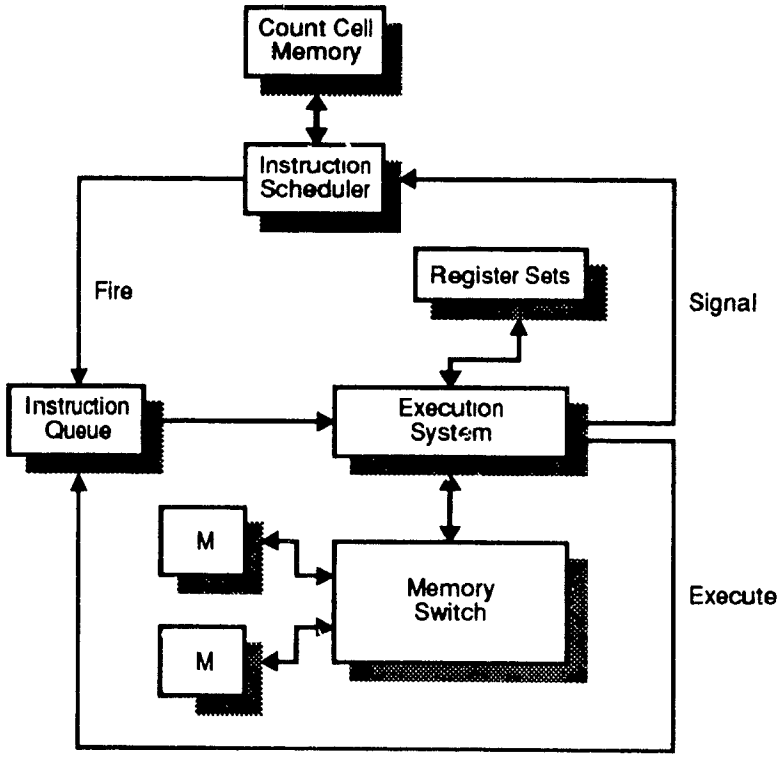


Figure 6.2: Organization of the New Static Dataflow Architecture

based. Data values are also duplicated—only when they leave the boundaries of the PE—instead of duplicating the instructions.

In Dennis' New Static Dataflow Architecture, *instructions* are made up of some arbitrary number of *sections* which can be of three kinds· execute sections, control sections, and send/receive sections (Figure 6.2 shows the major components of this architecture, without the interprocessor interface). Actually, each *instruction* corresponds to what other architectures have called a thread of instructions. Nevertheless, we will refer to them as instructions. Each instruction entering the Execution System gets exclusive usage of one of the register sets for all the duration of its activation

As pointed out earlier, data values are transferred from PE to PE by means of the send/receive section. Each send operation has to wait for an acknowledgment generated by the receive operation upon reception of the data value, so they interrupt the thread of execution within the instruction Since the execution system cannot afford to idle in the meanwhile due to the limited number of register sets, the send sections are always located at the end of an instruction. The consequences of this scheme is that there can only be one send section per instruction, and one instruction per data value being sent, thus a duplication of instructions which we have avoided in the scheme presented in this thesis.

Chapter 7

Conclusion

The design of a parallel system must be based on a sound model of parallel computation, from its programming model down to its architecture. 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.

Dataflow languages, based on the functional programming style, allow the parallelism, inherent in many scientific applications, to be fully and naturally expressed. The McGill Dataflow Architecture has been developed, to efficiently support this model of computation. It is based on the argument-fetching principle, which prones for the separation of data values and scheduling information, yielding a modular design where simplicity is a major characteristic. An execution pipeline free of data-dependent hazards and pipeline gaps due to operand matching, together with a virtually non existent data traffic, are among the attractive features of this architecture.

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. The former is directly attributable to the physical partitioning of the machine into independent processors, while the latter can be attributable to the logical partitioning of the application program. As the number of processors increases, the latency invariably increases too, the physical size of the machine being an insurmountable factor. Furthermore, the fine-grain parallelism, fully supported at the instruction level by the dataflow model, generally

leads to great synchronization overheads. The dataflow model of computation appears as an attractive model, well suited to efficiently address these issues

In this thesis, we have proposed an effective communication method to provide interprocessor synchronization between two nodes, and remote Structure memory operations. 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 structure memory operations.

The above mechanisms allow two processors to interact with each other in an argument-flow fashion by means of messages conveying signals, possibly paired with data values 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. Interactions among PEs are achieved through the sending and receiving of messages through a packed switching interconnection network. 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.

Within the framework of this thesis, we have developed some useful and flexible simulation tools that closely model the behavior of the multiprocessor system and its interactions with an interconnection network. 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 speed-ups, given a high bandwidth interconnection

network, and a highly parallelizable application like some of the benchmark programs used in this work.

We also believe that this work has been a valuable contribution in the sense that it has brought forward the problem of network contention, in a field where it is often disregarded. Most of the simulation results in other works estimate the effects of physically distributing a program, by assigning a cost to each interprocessor communication in terms of a fixed delay, thereby assuming a situation which is not always the case. With our simulations, we have achieved results which are closer to reality since they take into account this kind of network contention problem. For instance, our results show that in a system of 16 processors with an interconnection network of 1.6 gigabytes per second peak bandwidth, the network's average throughput adds up to about 720 megabytes per second, a figure that tends to decrease with larger systems. We believe that research should be done in order to find better suited networks for the dataflow model, and to find ways to include this network contention problem as an element of importance in the compiling of multiprocessor dataflow programs.

Appendix A

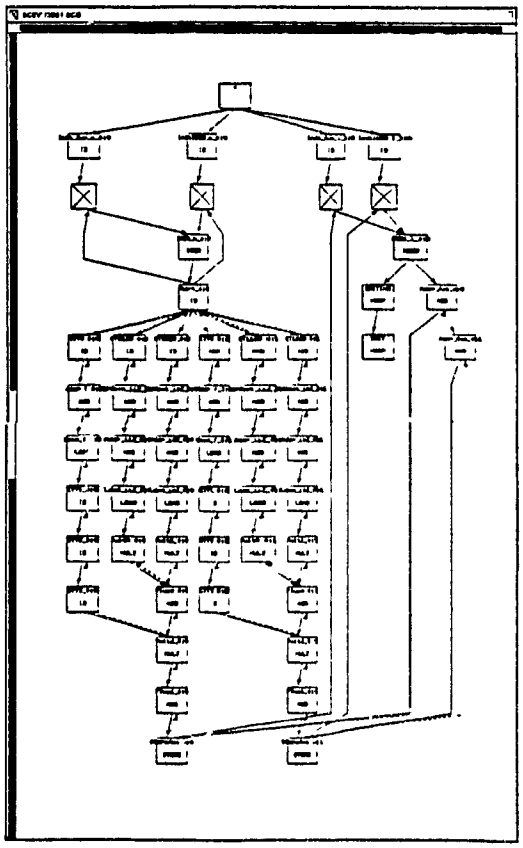
The Source Code of the Benchmark Programs

The Livermore Loops are 24 loops, widely recognized as performance benchmark programs, produced by the Lawrence Livermore National Laboratory. The Loops represent the type of computation kernels typically found in large-scale scientific computing. In this appendix, we present the SISAL source code for the loops that have been used for the performance evaluation of the MMDA The SISAL functions faithfully implement the computations of the loops, which originally were written in Fortran. The other benchmark programs are coded in Pascal. Each benchmark is also presented in the form of a graph.

Benchmark 1: Excerpt from a Hydrodynamics Code (Loop1)

This code fragment is the first Livermore Kernel and is excerpted from a hydrodynamics code. The values Q, R, T are scalar coefficients while Y and Z are one dimensional arrays. This loop returns an one-dimensional array of size n. Note that for the construction of a static array, the value of n should be known at compile time.

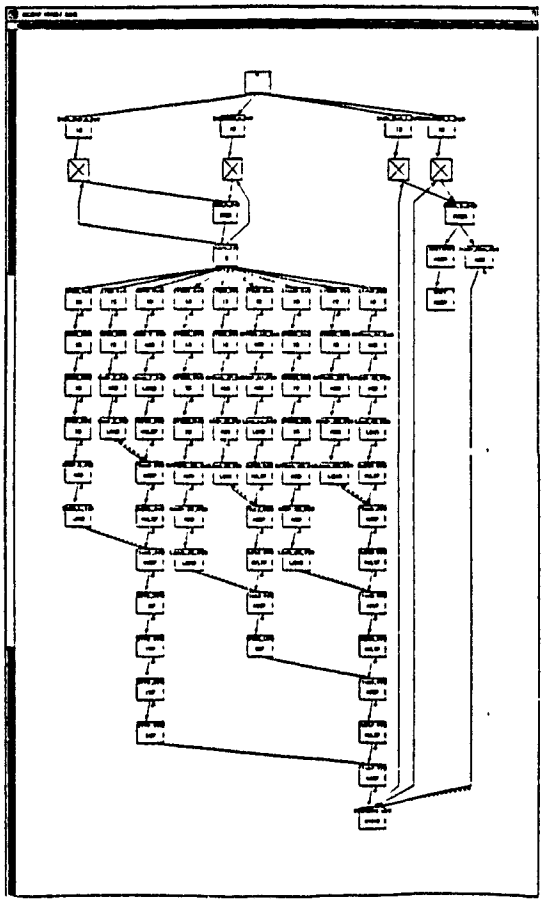
```
type OneDim = array[double];
 function Loop1 (n: integer; Q,R,T: integer,
                 Y,Z: OneDim; returns OneDim)
          for k in 1,n
          returns
                 Q + (Y[k] * (R * Z[k+10] + T * Z[k+11]))
          end for
end function
```



Benchmark 2: Equation of State Fragment (Loop7)

This code fragment is the seventh Livermore Kernel and returns a one-dimensional array of size n. R and T are coefficients, while U, Y, and Z are input arrays that are used for the construction of the returned array. The value of n should be known at compile time.

```
type OneDim = array[double];
function Loop7(n: integer; R,T:real; U,Y,Z: OneDim; returns OneDim)
        for k in 1,n
        returns array of U[k] + R * (Z[k] + R * Y[k])
                          + T * (U[k+3] + R * (U[k+2] + R * U[k+1])
                          + T * (U[k+6] + R * (U[k+5] + R * U[k+4])))
        end for
end function
```

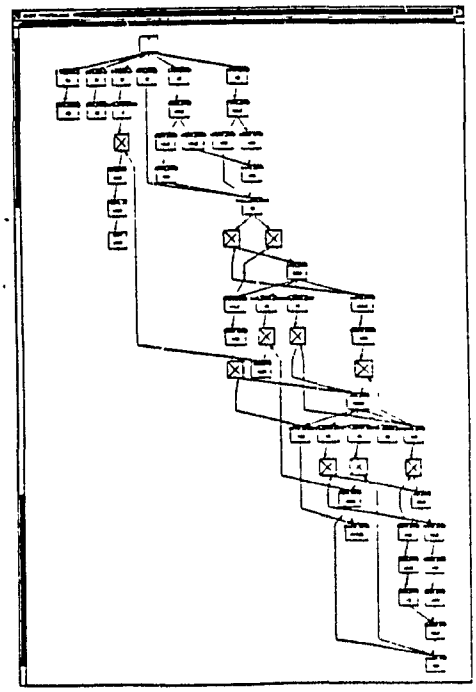


Benchmark 3: Matrix multiplication (Matmul)

The matrix multiplication is a function widely used in scientific applications. It is also a very common benchmark. It's inputs are two two-dimensional arrays A and B, of size $(n \times m)$ and $(m \times n)$ respectively. It returns a two-dimensional array C of size $(n \times n)$. Both n and m should be known at compile time.

```
type type1 : array [1..n,1..m] of real;
     type2 : array [1..m,1..n] of real;
     type3 : array [1..n,1..n] of real;

procedure Matmul (A: type1; B: type2; var C: type3);
var i, j, k : integer;
begin
for i := 1 to n do
    for j := 1 to n do
        for k := 1 to m do
    C[i,j] := C[i,j] + A[i,k] × B[k,j];
end;
```

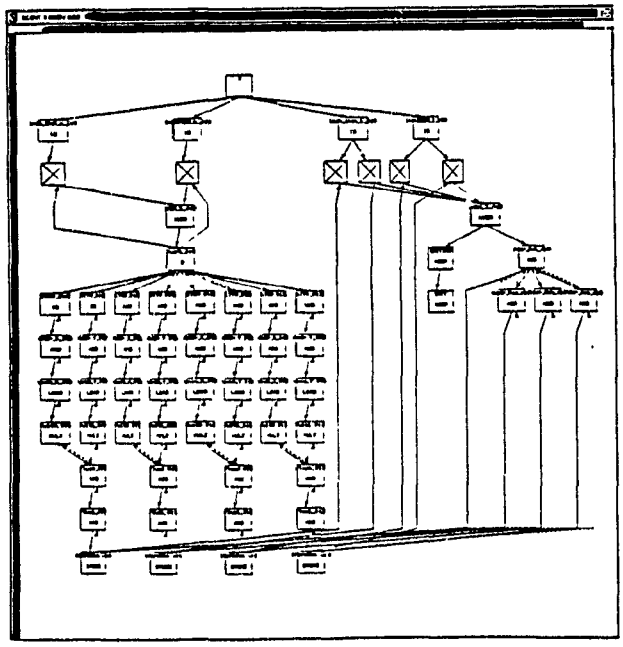


Benchmark 4: Fragment of Lower/Upper Decomposition (Saxpy)

Saxpy is a kernel of a program performing the Lower/Upper Decomposition. This is a computation widely used in linear algebra for solving systems of linear algebraic equations. The benchmark has been modified though, to make it more computationally intensive. X and Y are the input matrices of size n, and Z is the returned array, of the same size.

```
type arrtype : array [1..n] of real;

procedure Saxpy (X, Y: arrtype; a, b, c: real; var Z: arrtype);
var i : integer;
begin
for i := 1 to n do
    Z[i] := a × X[i] + b × Y[i] + c;
end;
```



Appendix B

An A-code sample program

The following is a fragment of the A-code version of the Loop7 program. The graph corresponding to this benchmark is shown in Appendix A. What we are showing here is called a *PE segment* and corresponds to the portion of code that runs on a particular processor— in this case, PE 0. The complete multiprocessor program consists of a list of these segments.

```
.PE loop7100

exit Res
ipsig EXITING~loop7101/EXIT
ipsig EXITING~loop7102/EXIT
ipsig EXITING~loop7103/EXIT
ipsig EXITING~loop7104/EXIT
ipsig EXITING~loop7105/EXIT
ipsig EXITING~loop7106/EXIT
ipsig EXITING~loop7107/EXIT

const LoopSize= 100 % Loop limit
const LoopLim =  99 % Loop limit - 1
const ArrSize1= 106 % Size array type1
const ArrSize2= 100 % Size array type2

const baseL_U =   0 % Base address U
const baseR_U = 406 % Base address U
const baseL_Y = 106 % Base address Y
const baseR_Y = 512 % Base address Y
const baseL_Z = 206 % Base address Z
const baseR_Z = 612 % Base address Z
const baseL_Res=306 % Base address Res
const baseR_Res=306 % Base address Res

const Inf =  -1

.segment DATA
dm   I1 init( 2.)
dm   I2 init( 3.)
.end

.segment STRUC
dm   U[ArrSize1] init((ArrSize1) 3.)
dm   Y[ArrSize2] init((ArrSize2) 4.)
dm   Z[ArrSize2] init((ArrSize2) 5.)
dm   Res[ArrSize2] init((ArrSize2) 0.)
.end


.seg PROG

.n EXITING
NOOP
count IGEN_1_0:0.f
count loop7101/EXITING loop7102/EXITING
count loop7103/EXITING loop7104/EXITING
count loop7105/EXITING loop7106/EXITING
count loop7107/EXITING
.end

.n EXIT
NOOP
count EXITING
.end

.n Init_Sup_k_0:0
ID LoopLim Sup_k_0:0
count INIT
.end

.n InitIGEN_k_0:0
ID Inf k_0:0
count INIT
.end

.n IGEN_k_0:0
IGEN k_0:0 Sup_k_0:0 k_0:0
count ( InitIGEN_k_0:0 Start_0:0 )
count ( Init_Sup_k_0:0 Start_0:0 )
.end

.n Start_0:0
ID k_0:0 R1_0:0
count IGEN_k_0:0.t
nocnt GTU0_0:0 GTZ0_0:0 GTY0_0:0
nocnt GT3U0_0:0 GT2U0_0:0
nocnt GT1U0_0:0 GT6U0_0:0
nocnt GT5U0_0:0 GT4U0_0:0
.end

%------ 1stbranch: U[k] ------%
.n GTU0_0:0
ID R1_0:0 R2_0:0
count Start_0:0
nocnt GTU1_0:0
.end
```

```
.n GTU1_0:0
ID R2_0:0 R3_0:0
count GTU0_0:0
nocnt GTU2_0:0
.end

.n GTU2_0:0
ID R3_0:0 R4_0:0
count GTU1_0:0
nocnt GTU3_0:0
.end

.n GTU3_0:0
ID R4_0:0 R5_0:0
count GTU2_0:0
nocnt Addr_U_0:0
.end

.n Addr_U_0:0
ADD baseR_U R5_0:0 add_U_0:0
count GTU3_0:0
nocnt Load_U_0:0
.end

.n Load_U_0:0
RLOAD add_U_0:0 val_U_0:0
count Addr_U_0:0
nocnt Plus1_0:0
.end
%------ End 1st branch: U[k] ------%

...

%------ 9th branch: R * U[k+4] ------%
.n GT4U0_0:0
ID R1_0:0 R38_0:0
count Start_0:0
nocnt Offset_4U_0:0
.end

.n Offset_4U_0:0
ADD R38_0:0 4 R39_0:0
count GT4U0_0:0
nocnt Addr_4U_0:0
.end

.n Addr_4U_0:0
ADD baseR_U R39_0:0 add_4U_0:0
count Offset_4U_0:0
nocnt Load_4U_0:0
.end

.n Load_4U_0:0
RLOAD add_4U_0:0 val_4U_0:0
count Addr_4U_0:0
nocnt Mult4_0:0
.end

.n Mult4_0:0
MULTF I1 val_4U_0:0 R40_0:0
count Load_4U_0:0
nocnt Plus4_0:0
.end

.n Plus4_0:0
ADDF val_5U_0:0 R40_0:0 R41_0:0
count Load_5U_0:0
count Mult4_0:0
nocnt Mult5_0:0
.end

.n Mult5_0:0
MULTF I1 R41_0:0 R42_0:0
count Plus4_0:0
nocnt Plus5_0:0
.end

.n Plus5_0:0
ADDF val_6U_0:0 R42_0:0 R43_0:0
count Load_6U_0:0
count Mult5_0:0
nocnt Mult6_0:0
.end

.n Mult6_0:0
MULTF I2 R43_0:0 R44_0:0
count Plus5_0:0
```

```
nocnt Plus6_0:0
.end

.n Plus6_0:0
ADDF R29a_0:0 R44_0:0 R45_0:0
count GT1U1_0:0
count Mult6_0:0
nocnt Mult7_0:0
.end

.n Mult7_0:0
MULTF I2 R45_0:0 R46_0:0
count Plus6_0:0
nocnt Plus7_0:0
.end

.n Plus7_0:0
ADDF R16_0:0 R46_0:0 R47_0:0
count GTY4_0:0
count Mult7_0:0
nocnt StoreRes_0:0
.end

.n StoreRes_0:0
STORE R47_0.0 add_Res_0:0
count Plus7_0:0
count Addr_Res_0:0
.end
%---- End 9th branch: R * U[k+4] ----%

.n Init_Sup_1_0:0
ID LoopLim Sup_1_0:0
count INIT
.end

.n InitIGEN_1_0:0
ID Inf 1_0:0
count INIT
.end

.n IGEN_1_0:0
IGEN 1_0:0 Sup_1_0:0 1_0:0
count ( InitIGEN_1_0:0 StoreRes_0:0 )
count ( Init_Sup_1_0:0 StoreRes_0:0 )
.end

.n Addr_Res_0:0
ADD 1_0:0 baseL_Res add_Res_0:0
count IGEN_1_0:0.t
nocnt StoreRes_0:0
.end

.end
.end % PE loop7100
```

Appendix C

The Mds Simulator

The mds simulator is written in Concurrent C, a process-oriented concurrent language. The language provides tools to create and handle *light-weight processes* which can communicate via *transactions*. Each major unit of the architecture has been simulated using this concept of process. The result is a modular program, flexible and easily modifiable, which can simulate systems of 1 to 16 processing elements. It can be used to test different configurations of processing element, and network topologies.

From each load image file that executes on the simulator, there is one file per PE that gets created, to store the results of the program as well as the performance metrics related to each specific PE. Most of the options are self explanatory as can be seen by this sample message obtained from running the simulator without any option:

```
        Mds: A Multiprocessor Dataflow Simulator (Release 1.1)
                (c) J.M. Monti  ACAPS-SOCS 05/90

Usage:  mds [ <flags> ] <ldifile>
flags:  -ns <num>: set number of stages in PIPU (def. 6)
        -nc <num>: set number of counts allowed per cycle (def. 4/ideal)
        -l <time>: starting time of run log (def. off/1)
        -t <time>: termination time of program (def. off/50000)

        -P <num> : set number of the PE generating the msgs (def. 0)
        -C       : show time counter
        -Lf      : generate log of firings (def. off)
        -Le      : generate log of enabled instructions (def. off)
```

```
-Lnu      : generate log of network utilization (def. off)
-Lnr      : generate log of network pending requests (def. off)
-Lnl      : generate log of network latencies (def. off)
-Nt       : print network traffic on stdout
-Nc <num>: PE to network clock ratio (def. 2)
-Nl <num>: network start-up latency (def. 2)
-Ns <num>: network switch buffer size (def. 2)
-S <size>: set Struc memory virtual size
```

In addition, there is the possibility of tracing some key events for all PEs or for only one PE, by using the L option combined with the P option. The traces are stored in *log files* which can be previewed using any graph software. If the P option is used, only the specified PE gets traced. All traces can be used at once, and are recorded at each unit of simulated time. The following is a list of each of the possible traces:

- Lf: PIPU utilization, i.e. whether or not the execution pipeline is busy;
- Le: size of the fire queue, which contains the enabled instructions;
- Lnu: network utilization, i.e. whether or not the network is used at each cycle;
- Lnr: number of pending network requests, at each cycle;
- Lnl: time of arrival and amount of latency, for each incoming packet;

The following is an example of a result file produced by a simulation:

```
SIMULATOR SETTING
=-=-=-=-=-=-=-=-=-=-=
                FIFO Scheduling;        1 PIPUs (P);
                6 stage(s) in PIPU;     4 count signals/cycle (C);
                Nb. procs 8;            Network: OMEGA (3 stages);
                PE/Ntwrk ratio: 2;      Start-up latency: 2;
                Network Switch size: 2 buffer(s).

SYSTEM STATISTICS PE#0:
-=-=-=-=-=-=-=-=-=-=-=-=-=
        File name: loop7.res
```

```
NETWORK STATS:
        Global Network utilization:
                        accesses:       6414    through stage 0
                        refused:        889
                        delayed:        0
                        delayed/cycle:  0 00    over 10542 cycles
                        total bytes:    64056
                        bytes/access:   9.99    on ave.
                        throughput:     27 65   MBytes/sec over total time
                        contention:     30.37 % over 10542 cycles
                        nb. coll :      9780
                        coll./cycle·    0.93    on ave.
        Local Network utilization:
                        accesses:       807     out / 807 in
                        refused:        210
                        total bytes:    8028
                        bytes/access:   9.95    on ave.
                        throughput:     27.72   MBytes/sec on ave.
        Remote requests:
                        total:          400
                        frequency:      18.10   over total time
                        pctge load:     5.63 %  over total inst.
        Latency (min/max/ave):
                        local:          2       8       2.06
                        network:        10      28      15.71
                        total (ave.):   17.77
PROCESSOR STATS:
        Total number of processor cycles:       7240
        Total idle cycles:                      132
        Processor usage:                        98.18 %
        Struc Memory Size:                      406 words

        Sequential Run time:                    42648
        Local Speedup:                          5.89
        Avg. population in enable
        memory and fire queue:                  12.28

        Count(s)        Done(s)         Fire(s)  /Sh.Fire(s)    Exec(s)
Total:  15810           7107            7108     /0             7108


        CountQ  Cnts/   DoneQ   FireQ   Fire    RemReqQ IntSigQ LocalQ
                Inst.                   Delay
Max:    4       11      4       23      22      1       7       2
```

Bibliography

[1] W. B. Ackerman. Data flow languages. *IEEE Computer*, 15(2):15–25, February 1982

[2] W. B. Ackerman and J. B. Dennis. VAL—a value-oriented algorithmic language. Technical Report 218, Laboratory for Computer Science, MIT, 1979.

[3] Arvind, S A Brobst, and G. K Maa Evaluation of the MIT tagged token dataflow project. Computation Structures Group Memo 281, Laboratory for Computer Science, MIT, 1988.

[4] Arvind and D E Culler Dataflow architectures. *Annual Reviews in Computer Science*, 1·225–253, 1986

[5] Arvind, M. L. Dertouzos, R S. Nikhil, and G. M. Papadopoulos. Project dataflow—the Monsoon architecture and the Id programming language. Computation Structures Group Memo 285, Laboratory for Computer Science, MIT, March 1988.

[6] Arvind and et al The tagged token dataflow architecture (preliminary version). Technical report, Laboratory for Computer Science, MIT, Cambridge, MA., August 1983

[7] Arvind and K P. Gostelow. The U-Interpreter *IEEE Computer*, 15(2):42–49, February 1982.

[8] Arvind and R A Iannucci. A critique of multiprocessing von Neumann style. In *Proceedings of the Tenth Annual International Symposium on Computer Architecture*, pages 426–436, 1983.

[9] Arvind and R A. Iannucci Two fundamental issues in multiprocessing. Computation Structures Group Memo 226, Laboratory for Computer Science, MIT, 1987.

[10] Arvind, R S Nikhil, and K K Pingali I-structures Data structures for parallel computing *ACM TOPLAS*, 11(4) 598–632, October 1989

[11] 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.

[12] 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

[13] A. L. Davis and R. M. Keller Data flow program graphs *IEEE Computer*, 15(2):26–41, February 1982

[14] J. B. Dennis First version of a data-flow procedure language. In *Proceedings of the Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 362-376. Springler-Verlag, 1974

[15] J. B. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11) 48–56, November 1980.

[16] J. B. Dennis. Dataflow computation: A case study. In Veljko Milutinović, editor, *Computer Architecture. Concepts and Systems*, pages 354-404. North-Holland, New York, 1988.

[17] J. B. Dennis. The evolution of 'static' data-flow computing. In J -L Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*. Prentice-Hall, 1990

[18] J. B Dennis Evolution of the static dataflow architecture In *Advanced Topics in Dataflow Computing* Prentice-Hall, 1991.

[19] J. B. Dennis, G. A. Boughton, and Leung C. K C Building blocks for data flow prototypes. In *Proceedings of the Seventh Symposium on Computer Architecture*, May 1980

[20] 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

[21] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. Technical Report TR-SOCS-88.06, School of Computer Science McGill University, Montreal, February 1988.

[22] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *The Second Annual Symposium on Computer Architecture*, pages 126–132, January 1975.

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

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

[25] 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.

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

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

[28] D. D. Gajski and J.-K. Peir. Essential issues in multiprocessor systems. *IEEE Computer*, 18(6):9–27, June 1985.

[29] 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.

[30] G. R. Gao. *A Code Mapping Scheme for Dataflow Software Pipelining.* Kluwer Academic Publishers, Boston, December 1990.

[31] G. R. Gao, H. H. J. Hum, and Y. B. Wong. Towards efficient fine-grain software pipelining. In *Proceedings of the ACM International Conference on Supercomputing*, Amsterdam, the Netherlands, June 1990.

[32] G. R. Gao and R. Tio. Instruction set architecture for an argument-fetching dataflow architecture. In *Proceedings of the Canadian Conference on Very Large-Scale Integration (CCVLSI-88)*, Halifax, October 1988.

[33] G. R. Gao and R. Tio. Instruction set definition for the argument-fetching dataflow machine. ACAPS Technical Memo 01, School of Computer Science, McGill University, Montreal, February 1988.

[34] 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.

[35] J. Gaudiot, R. Vedder, G. Tucker, D. Finn, and M. Campbell. A distributed VLSI architecture for efficient signal and data processing. *IEEE Transactions on Computers*, C-34(12):1072–1087, 1985.

[36] N.H Gehani and W.D. Roome. The Concurrent C project. Computer technology research laboratory technical reports, AT&T Bell Laboratories, Murray Hill, New Jersey, 1988. Collection of papers.

[37] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes. The Epsilon dataflow processor. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 36–45, Israel, June 1989.

[38] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[39] K. Hiraki, S. Sekiguchi, and T. Shimada. Efficient vector processing on a dataflow supercomputer SIGMA-1. In *Proceedings of IEEE Computer Society and ACM SIGARCH Supercomputing '88 Conference*, Orlando, FL, 1988.

[40] K Hiraki, S. Sekiguchi, and T. Shimada. Status report of SIGMA-1: a dataflow supercomputer. In *Advanced Topics in Dataflow Computing*. Prentice-Hall, 1991.

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

[42] P. Hudak. Conception, evolution, and application of functional programming languages. *Computing Surveys*, 21(3), September 1989.

[43] D. Johnson et al. Automatic partitioning of programs in multiprocessor systems. In *Proceedings of Compcon 80*, pages 175–78, 1980.

[44] Y. Kodama, S. Sakai, and Y. Yamaguchi. A prototype of a highly parallel dataflow machine EM-4 and its preliminary evaluation. In *Proceedings of Info-Japan 90*, October 1990.

[45] 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.

[46] V.P. Kumar and S.M. Reddy. Augmented shuffle-exchange multistage interconnection networks. *IEEE Computer*, pages 30–40, June 1987.

[47] D.H. Lawrie. Access and alignment of data in an array processor. *IEEE Transactions on Computers*, C-24(12):1145–1155, December 1975.

[48] J. R. McGraw. The VAL language: Description and analysis. *ACM TOPLAS*, 4(1):44–82, January 1982.

[49] 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.

[50] J.M. Monti. Dasm reference manual (version 1.4). ACAPS Design Note 17, School of Computer Science, McGill University, Montreal, November 1989.

[51] J.M. Monti. Mds: A multiprocessor dataflow simulator. ACAPS Design Note 15, School Of Computer Science, McGill University, Montreal, November 1989.

[52] R. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 262–272, Israel, 1989.

[53] R.S. Nikhil. The parallel programming language Id and its compilation for parallel machines. Computation Structures Group Memo 313, Laboratory for Computer Science, MIT, July 1990.

[54] 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.

[55] G. M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor*. PhD thesis, MIT, 1988.

[56] J.H. Patel. Processor-memory interconnections for multiprocessors. In *The 6th Annual Symposium on Computer Architecture*, pages 168–177, New York, N.Y., April 1979.

[57] A. Plas, D. Comte, O. Gelly, and J. C. Syre. LAU system architecture: A parallel data driven processor based on single assignment. In *Proceedings of the 1976 International Conference on Parallel Processing*, pages 293-302, 1976.

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

[59] J. E. Rodriguez. *A Graph Model for Parallel Computation.* PhD thesis, Laboratory for Computer Science, MIT, Cambridge, MA, September 1967. Also available as Technical Report 64, Project MAC, MIT, September, 1969.

[60] C. L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1), January 1985.

[61] T. Temma, S. Hasegawa, and S. Hanaki. Dataflow processor for image processing. *Proceedings of the 11th International Symposium on Mini and Microcomputers*, pages 52-56, 1980. Monterey, CA.

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

[63] Y. B. Wong. AD reference manual (version 1.4). ACAPS Design Note 16, School of Computer Science, McGill University, Montreal, October 1989.