MIT/LCS/TR-302

A MULTIPROCESSOR EMULATION FACILITY

Arvind
Michael L. Dertouzos
Robert A. Iannucci

This blank page was inserted to preserve pagination.

A Multiprocessor Emulation Facility

Laboratory for Computer Science
Technical Report 302
24 October 1983

Arvind

Michael L. Dertouzos

Robert A. Iannucci

This report is an excerpt from a Proposal submitted by the Laboratory for Computer Science of the Massachusetts Institute of Technology to the Advanced Research Projects Agency of the Department of Defense. Current funding is provided in part by the Defense Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under contract N00014-75-C-0661. The third author is supported by the International Business Machines Corporation.

A Multiprocessor Emulation Facility

Abstract

Interest in multiprocessor computer architectures has increased dramatically in the last ten years. However, it has become clear that, in order to effectively use multiprocessors in a general way, some fundamental changes in the model of computation are necessary. Moreover, experimentation in the field is hindered by low-performance simulation tools and high-cost hardware modeling schemes.

We present our approach to solving both of these problems: *Dataflow Architecture*, which provides an inherently parallel model of computation, and a *Multiprocessor Emulation Facility*, which is a general purpose tool for evaluating multiprocessor architectures at low cost. We also discuss our scheme for using the Emulation Facility to validate our claims about dataflow architecture.

Key words and phrases: computer architecture, dataflow, emulation, multiprocessor systems, packet communication, simulation

Table of Contents

1. Introduction and Objectives 1
2. Emulation 2
    2.1. The State-of-the-Art in Simulation and Emulation 3
    2.2. Goals of the Emulation Facility 5
    2.3. The Multiprocessor Emulation Facility 6
    2.4. Use of the Facility 10
    2.5. Benchmarking and Measurement 12
3. The Tagged-Token Dataflow Machine 13
    3.1. The Dataflow Alternative 14
4. Emulating the Dataflow Machine and Beyond 18
    4.1. The Overall Plan 18
    4.2. The First Experiment 19
    4.3. Expected Emulator Performance 20
    4.4. Summary 20

A Multiprocessor Emulation Facility

1. Introduction and Objectives

The nation is currently at a critical turning point concerning the formulation of appropriate research and development strategies in information technology for the following reasons:

1. The ongoing improvements of some 30% per year in the performance/price of the solid state circuits that make up computer processors and memories has created a new opportunity for aggregating thousands of processors at reasonable cost in architectures dedicated to the performance of a single task.

2. The research which has been performed to date on multiprocessor architectures has resulted in several promising technological avenues for achieving such architectures: The concepts of architectures that (1) are paired with associated programming languages; (2) are quasi-linearly scalable; and (3) are fault-tolerant, have reached a stage that calls for development-oriented experimentation.

3. The generic applications expected to become possible on these machines include machine vision, speech understanding, intelligent filing and retrieval systems, intelligent signal processing and computationally intensive simulations. These generic applications when further specialized into military or civilian uses will lead to qualitatively different and previously impossible computer uses that will induce revolutionary changes into every sector of our military and civilian structures.

4. In information technology, Japan has undertaken the challenge of leap-frogging the U.S. within ten years by pursuing ambitious long-term goals which involve super-computers. Our industries, by virtue of their shorter-term orientation, have not embarked on any similarly ambitious strategies. As a result, the geopolitical balance which will necessarily rest progressively more on information technology in the future than it has in the past, is in danger of shifting away from the U.S.

At the MIT Laboratory for Computer Science, we have been fortunate in identifying these reasons as very significant nearly four years ago when Japan announced its early plans for the Fifth Generation Computer Project (The Jipdec Plan). Prior to that time, we have had a strong scientific interest in multiprocessor architectures and associated languages for about 15 years, and have carried out specific research projects in the field (Dataflow and MuNet architectures).

Our plan for an *integrated project* in the super-computer field consists of two principal parts: (1) an *emulation facility* for multiprocessor architectures; and (2) a *dataflow architecture* which constitutes the first targeted use of this facility. These two parts are summarized below.

The *emulation facility* is a system consisting of 64 powerful computers and 64 8 x 8 network switches that interconnect these machines. These computers and switches can be programmed either through a high-level language (so as to reduce programming effort) or at the micro-code level (for increased performance) to emulate a variety of multiprocessor systems. Additional advanced

personal computers are used via a local network to access and control the centralized 64-processor and network aggregate. The emulation of such multiprocessor machines prior to their construction is deemed essential for the following reasons:

1. Analytical/theoretical or single-processor simulation techniques alone are not adequate for solving a variety of problems that must be addressed prior to implementation of a multiprocessor system, such as determining the size of buffer memories in each processor.

2. The alternative of implementing a proposed multiprocessor architecture directly with custom silicon chips is costly, time consuming and good for testing only one architecture at a time.

3. The emulation facility *fosters* a multiprocessor design orientation - in effect, the designers of multiprocessor systems become accustomed to "thinking parallel" precisely because their experimental base is the parallel structure of the emulation facility.

4. The availability of such a powerful emulation facility for use by members of the research community will stimulate the design of numerous architectures by top-level designers while simultaneously scrutinizing the potential success of these machines at a small fraction of the implementation costs.

The emulation facility is discussed in more detail in Section 2 of this paper.

The *dataflow architecture* that we propose to test on this facility is, in our view, the most promising potential multiprocessor architecture among the alternatives before us. It consists of a tightly-coupled network of processors and an associated programming language through which the dataflow machine is tailored to specific applications. Though internal architecture of a dataflow processor is different from that of the processors in the emulation facility, the overall architecture of the dataflow machine closely parallels the structure of our proposed emulation facility. The dataflow architecture is discussed in more detail in Section 3.

Our overall plan, which is discussed in Section 4, calls for a staged buildup of the emulation facility and for subsequent use of this facility by other members of the DARPA community via the ARPANET. Our plan also calls for simulation of the dataflow machine by an IBM 4341 computer loaned to us by IBM. The continuing involvement of IBM as our major industrial partner since the first day of this project is a key ingredient of our overall plan, for it will make possible a joint development effort of a VLSI version of this novel architecture when it has been adequately modified and verified through the emulation process.

2. Emulation

Recent advances in methodologies for designing custom VLSI circuits and the greatly reduced cost of their fabrication has made large parallel machines seem realizable in the near future. However, designs of large systems can be cast in hardware only after proper evaluation by

simulation, emulation, or the construction of a prototype of the system. The absence of extensive evaluation prior to construction may result in wasteful post-construction design changes.

Emulation is a powerful pre-construction evaluation technique which allows the system architect to abstract away from some of the low-level details of an implementation so that attention may be focused on the higher-level behavior of the system being designed. For the purposes of this paper, we define *emulation* as the process of reconfiguring a *base* computer system via programming, so that it behaves according to a *target* architectural specification. The programming may take the form of micro-coding, higher-level programming, or some mixture. To make the base machine behave according to the architecture of the target machine means that the base machine will accept the target machine-level programs and data structures, and will produce results identical to those of the target architecture, although not necessarily with the same level of performance. Unlike a *simulation* program, an emulator usually does not carry timing information explicitly.

2.1. The State-of-the-Art in Simulation and Emulation

Simulation basically requires as large a number of machine cycles as one can get, and some software for utilizing the cycles. The Cray-1 has dominated the high-speed computer market and can be taken as a representative of the class of machines that includes the Cyber 205, the Cray-xmp, and Hitachi and Fujitsu super-computers scheduled for delivery this year. All these machines provide substantial raw computing power but primarily in the context of numerical problem-solving. Their programming environments are strictly tailored to Fortran and assembly language, both of which are adequate for writing simulations. However, neither virtual memory nor Simula, which is generally considered to be the best language for writing simulation programs, is available on these machines. We do not consider this to be a serious drawback because our experience has shown that for a large simulation program a programmer has to write his own storage management routines. Fine grain control over what information resides in primary memory is essential for the efficiency of many simulation programs. A detailed simulation program for an earlier version of the dataflow machine was written by Gostelow and Thomas [7] in Simula on a PDP-10. The program would run out of address space in executing a matrix multiply program involving two 7x7 matrices. Problems 4 to 10 tomes larger could have been run if the simulation program had been written in Pascal or C. Of course, a much greater effort would have been required to write the same program in Pascal as opposed to Simula.

There are no more than 70 to 80 Cray class machines in the world because such machines are expensive (approximately $8 million to $12 million). These machines are generally not available to university researchers for remote program development. Even though the applicability of super-computers for simulation in undeniable, given the option of acquiring a Cray-1 or Cray-xmp versus putting together the less expensive proposed Multiprocessor Emulation Facility (MEF), we consider the MEF superior for the reasons outlined below.

As emulators, current super-computers are far from suitable because of their rigid internal structure. These machines are designed to exploit inner loop parallelism, and parallelism present in straight-line scalar code fragments. Their performance degrades rapidly if the control structure of

the program involves too many conditional branches. Emulating a multiprocessor architecture, by its very nature, involves testing the state of an emulated processor and its memory, and taking appropriate action. In addition, modeling of conflicts in interprocessor communication can not be done without explicitly "simulating" the passage of target-system time. The MEF, on the other hand, by providing physical interconnections that are very close to the target network, can emulate these conflicts substantially faster than a simulator.

One of the most successful simulation/emulation engines built to date is the 256 processor Yorktown Simulation Engine (YSE) [4, 14, 15] used by IBM to simulate proposed computer architectures prior to construction. Here, after the instruction set, performance goals, and technology for a new machine have been set, a block level simulation of the architecture is undertaken. In particular, the logical design of each subsystem is done by a different development group which checks the design of its subsystem by simulation. Gate level design is begun only after the logic simulation shows satisfactory results. Ultimately, the whole machine is simulated at the gate level, requiring enormous amounts of computing power. The YSE was designed specifically to speed up the gate level simulation, and is capable of simulating two million gates at three billion gate computations per second. At this rate, the YSE can simulate in eight hours a gate level system that would require one year of simulation on an IBM 370/168. Such enormous speeds are achieved because the YSE does not carry any timing information explicitly. The timing information is "compiled" into a YSE program which essentially behaves like a versatile 80 nanosecond gate machine. Since the YSE is *a gate level emulation machine*, it cannot be used for functional level or logic level simulation and hence would be useful to multiprocessor system designers only in later stages of logic design.

Recently, proposals have been made to use Control Data's Advanced Flexible Processors (AFP's) connected by a fast ring-bus into an emulation/simulation facility. As far as computation cycles go, there are few machines that can match the AFP's raw computing power. However, AFP's do not have *any* program development environment - a user is supposed to do program development on the AFP simulator that runs on Cyber machines. Programming an AFP can be best described as horizontal micro-coding which is at least an order of magnitude harder than assembly language programming. Accordingly, a 200 line program on AFP would be considered very large given the current programming aids. Under these circumstances, AFPs cannot be effectively used for simulation programs which often run to several tens of thousands of lines of code. The AFP may eventually become suitable as a building block for an emulation facility but the lack of programming and debugging aids and the need for an associated Cyber machine may pose difficult practical problems in such use.

One of the most interesting multiprocessor systems built to date is the BBN Butterfly machine [17]. It currently consists of 10 M68000 boards connected by a circuit switched network of butterfly (*i.e.*, FFT or shuffle exchange) topology. The machine can be extended to several hundred M68000s because the network is easily expanded, however larger systems would be more difficult because of the reliance on a single, central clock. Work is also underway to build a faster version of the switch in custom VLSI. The machine was designed for voice funnel applications, and two copies of the machine are successfully running that application. The program development for a

single M68000 can be done in C or in any of the ever growing programming aids for the M68000. Our main concern with the Butterfly machine is its lack of computing power. An M68000 processor, while adequate for many applications, is equivalent to only 1/10 of its computing power in emulating another processor architecture. Thus, even a 64-processor emulation facility based on the M68000 will be barely equivalent to a DEC 2060, and will not be sufficiently powerful to emulate large multiprocessor aggregates. Along these lines, it should be noted that one reason why Cm* [20, 21] did not turn out to be very useful was its lack of adequate computing power.

2.2. Goals of the Emulation Facility

An emulator should be easy to use and should provide early feedback to the designers - information which cannot be practically obtained in any other way. In what follows we summarize our goals for such an emulation facility; we propose its structure in more detail and show how it will be useful to researchers in the United States.

The existence of an emulation vehicle will help establish the feasibility of a concept at substantially lower-risk and cost than the actual construction of the target machine. Therefore it is imperative that the emulation facility provide its user with these benefits. This requirement necessitates, for example, an effective programming and debugging environment beyond raw computation power.

We believe that the proposed general-purpose emulation facility should meet the following goals:

- **High Capacity:** The facility should be composed of an adequately large number (64) of high-speed (200 nsec. micro-cycle) processors, each having a large virtual address space (2^{32}) and a large physical memory (1-8 megabytes with a bandwidth of 20 megabytes per second). The idea is to provide as close to state of the art basic machine performance as is practical, to reduce the performance gap between the emulator and the target architectures. The facility should not only be powerful by today's standards, but should also be useful to its users - only then can users be expected to put in the required effort for their architectural experiments.

  The facility that we propose in Section 2.3 is a 64 processor machine with 2 megabytes of memory per processor (*i.e.*, a total real storage size of 128 megabytes) and the capability of executing 320 million micro-instructions per second.

- **Reliability and Fault Tolerance:** The emulator machinery should be inherently reliable. Thus, dependence on exotic and unproven technologies should be avoided. The facility should be designed to tolerate and mask certain inevitable failures. This implies that the hardware must have at least minimal failure detection circuitry, and the rest of the system must be able to effectively make use of this information to correct aberrant behavior.

- **Reconfigurability and Partitionability:** The facility should be easily tailored to suit a wide variety of emulation tasks. This includes the necessary reprogramming of the processors and the possible reconfiguration of the interconnection network to suit the

task at hand. Also, the facility should be easily subdivided into several smaller, disjoint facilities so that more than one smaller architectural experiment can be run at one time.

- **Ease of Implementation:** Implementation of the emulation facility must be easier than implementation of the functioning target systems that the facility will emulate. This is an important goal that we discuss later in the context of implementing dataflow machines.

- **Ease of Use:** Perhaps the most important objective of an emulator, which encompasses the foregoing goals, is that it be easily used by architects of prospective architectures.

2.3. The Multiprocessor Emulation Facility

The proposed emulation facility is made up of two integrated parts - the *microprogrammable processor* and the *packet communication switch module.* The former is available commercially; the latter is not and will be developed by us. The facility will consist of 64 total processors, eight of which will be fully configured as software development stations (with display, keyboard, paging disk, and local network interface). The other 56 processors will be stripped down versions (with memory but without display/keyboard or disk). From the software point of view, all 64 machines will be programmed as if they were identical for emulation purposes. Each processor, fully configured or not, will have an integrated switch module. These switches will be interconnected under user control to a variety of inter-processor communication network configurations.

In more detail, the proposed processors and networks along with the reasons for our choices are as follows:

Processor: In addition to examining the raw power and cost of candidate processors, our selection criteria included the availability, and the human effort required to make the facility usable. We considered three classes of machines:

1. Commercially available Motorola M68000-based single board computers.

2. Our own design of an AMD2903 based multi-tasking micro-machine [12], and

3. Commercially available single user machines such as Three Rivers' PERQs, Xerox Altos and Dorados, DEC VAX 750s, and various Lisp Machines.

Machines of Class 1, though initially attractive, were rejected primarily because of inadequate computing power for emulation and the difficulty of integrating a communication network into already existing boards. There were also serious questions about the lack of error correcting memories in commercially available memory boards. Machines of Class 3 were originally not considered because of the associated expense. Our own design [12] was partially inspired by the machines of Class 3, and would have involved a major design effort and substantial risk only to produce something which is commercially available.

As a result of these considerations and after a considerable amount of research, we identified the

Symbolics 3600 as the clear choice for our emulation facility. It is a state-of-the-art machine which is user microprogrammable, and meets all the performance and capacity needs of the project. It also has the potential of expansion, particularly of its physical memory, which we feel is important for the future of the facility. Other important features include error-correcting main memory, parity checked registers and buses, 36 bit data paths (32 plus 4 tag bits), highly sophisticated LISP-based programming and debugging environment, availability of a high-speed floating point processor, local network interface, micro-coding and micro-tasking, and a pluggable bus/backplane.

The tag bit feature of this machine's main memory will provide a significant performance benefit for the dataflow emulation. Further, the micro-tasking processor will make possible the writing of the the dataflow emulator as a set of communicating processes - one per subsystem. This approach, in turn, will improve the fidelity of emulation without introducing a corresponding time penalty.

Network: The network that we propose is intended to make possible the attachment of a large number of microprogrammable processors. It will be made up of individual modules which will be integrated with the processors on a one-on-one basis. Each module will provide a direct memory access path to and from its processor's main storage as well as a status and control interface to that processor.

The design of the network is built around the concept of *packet switching*. In such a network, a processor formats information to be transmitted into data packets by the addition of some routing and control information (analogous to putting a letter into an addressed envelope) and hands it off to the network. The sending processor is then free to perform other work while the network goes about the task of delivering the packet to the proper destination. Such networks vary in their interconnection topology, but have the property that a packet may go through several links. Rather than reserving a complete path from source to destination for the entire duration of the packet's transit time called *circuit switching*, such networks allow packets to be *stored* at intermediate points until a path is available to the next intermediate point. The network then *forwards* the packet and frees the storage space used for the next packet.

The proposed design is optimized for, but not limited to, the binary n-cube topology. Referring to Figure 2-1, it can be seen that for a cube of dimensionality n, a node must act like the logical equivalent of an $(n+1) \times (n+1)$ crosspoint switch. One of the input/output pairs is connected to the processor associated with the node while the other n input/output pairs are connected to the nodes immediately adjacent in the cube. The Figure illustrates this with a 3-dimensional cube ($2^3 = 8$ processor nodes).

A simple algorithm exists for routing messages in the n-cube that can easily exploit the inherent redundancy for fault tolerance. While this algorithm is simple, it can be made more general by using a table-based routing scheme. By using such table, static reconfiguration and partitioning of the network can be facilitated. Depending on the table contents, the switch can implement a routing algorithm based on local, dynamic traffic information or it can implement a purely source-based (fixed path) algorithm. Specifically, a node may have many *alias* addresses, each denoting a unique path through the network. The message source may then predetermine the path a packet takes by suitable selection of an address. This flexibility is essential if the network is to be useful in

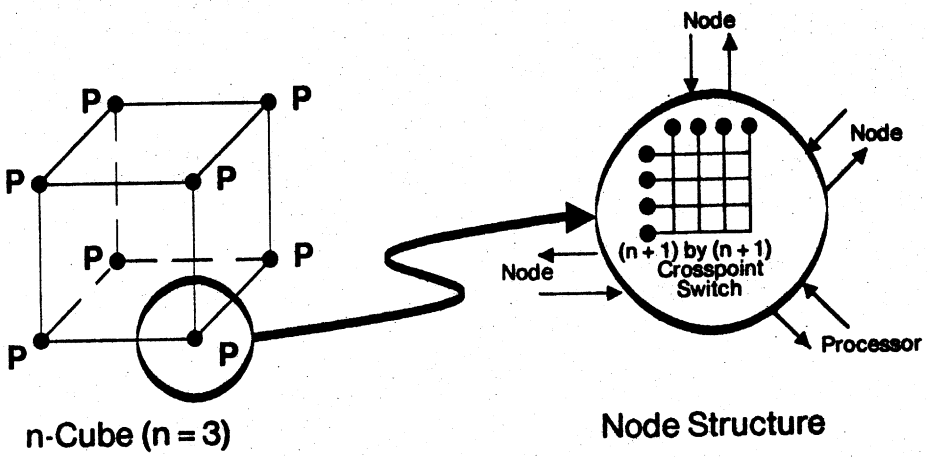

Figure 2-1: The Binary n-Cube and Node Structure

emulating several *target interconnections*.

The switch design, as mentioned, can be adapted to other topologies through physical reconfiguration by re-plugging of interconnection cables. Small, *fully-connected* networks (eight processors) can be constructed because each switch module provides seven bidirectional serial ports in addition to the direct connection to the local processor.

By treating inputs and outputs separately, a 7×7 butterfly switch can be configured. This topology gives up hardware-level redundancy in the network, but any of the traditional $n \times \log n$ networks with a processor per butterfly can be constructed. Routing tables would have to be set appropriately.

For our 8×8 switch, we can implement a 7 dimensional cube with $2^7 = 128$ processors. Due to the internal byte-oriented architecture of the design, packets can only *name* $2^8 = 256$ distinct destinations for the purpose of routing. This is a "soft" upper limit on the number of processors that the switch can interconnect - with processor intervention (*e.g.*, altering the routing address), an unlimited number of processors can be named. By altering the topology (*e.g.*, cube-connected cycles [16]), arbitrarily large networks can be composed.

The proposed switch implements extensive failure detection through analog and digital domain checks, and relies on the associated microprogrammable processor for error recovery operations. This approach reduces the complexity of the switch, while simultaneously raising the level of fault tolerance. The switch has also been systematically designed to *minimize* the chance for failure by relying only on robust techniques.

The overall structure of the switch is shown in Figure 2-2 (see [13] for more details). It is modular in character, and is organized around eight major control and data buses. Connected to the buses are eight input FIFOs, eight output buffers, and a Sequencer / Scheduler. Seven of the input

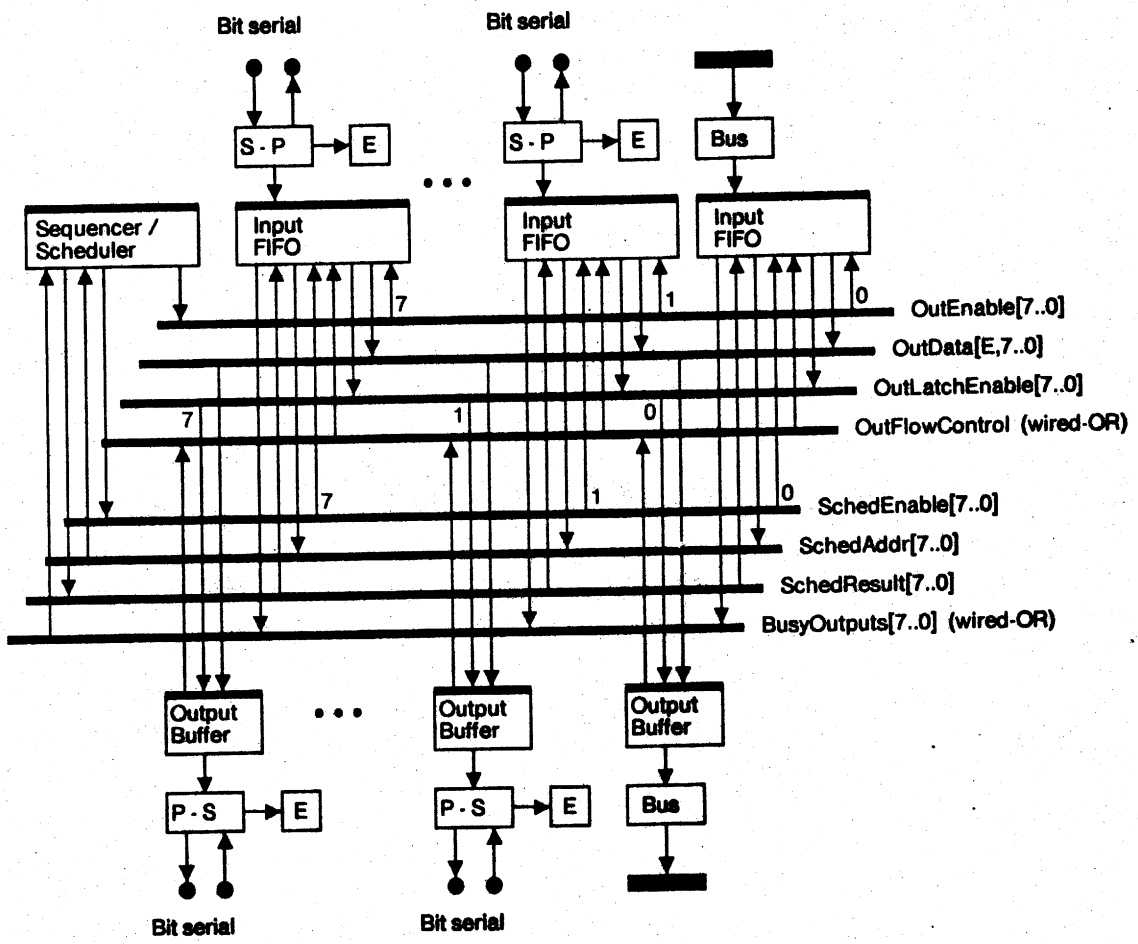

Figure 2-2: Basic Structure of the Packet Switch

FIFOs and seven of the output buffers are connected to serial - parallel conversion logic. The eighth input / output pair interfaces to the data bus of the attached microprogrammable processor. Internally, the switch is totally synchronous. Resynchronization of incoming data with the local clock is done by the serial-to-parallel conversion logic.

The Emulator as a Shared Memory Multiprocessor: Conceptually, an ideal way to build a large emulation facility would be to construct a machine in which a number of autonomous processors could cooperate using a single, contention free shared memory. Unfortunately, nobody knows how to build such a machine without incurring either very high cost, or severe performance degradation, or both.

Our emulator provides a clean model of communication which can easily be made to look like shared memory. The packet switch is designed for both high throughput *and* low latency. Thus, it would be possible to formulate a remote memory request into a network packet of small size (say, eight bytes) and to forward it to the appropriate remote processor. Each processor would be

prepared to process such incoming requests by dedicating a high priority micro-task to the job.

The exact performance of such a remote reference depends on locality in the interconnection network, traffic intensity, and routing conflicts. In the best case, a one-way message of eight data bytes from one processor to one of its nearest-neighbor processors without conflicts and in a low traffic situation will take approximately 2.7 μsec. from processor memory to processor memory. A micro-task switch can, in the best case, be done on a single micro-instruction basis (approximately 200 nsec.). Allowing 10 micro-instructions for processing the request and building the result packet, it should be possible to execute such a remote request in under 7.5 μsec. from the time the request is formulated until the result is received. Deviations from the best case are likely to increase this delay. Pipelining will improve the throughput, limited only by the raw bandwidth of the network which is four megabytes per second per link. This leads to a theoretical upper bound of 500,000 memory references per second with the practical upper limit being probably a factor of four lower than this theoretical bound.

2.4. Use of the Facility

Local Access: Use of the facility at MIT will rely on the development of the *target interpreter* (software for the processors and configuration tables for the network) on a Symbolics 3600. This is an interactive task that does not require the emulation facility, and is best done on one of the eight software development stations that are included in the facility.

Running an emulation can be done interactively if it is at all meaningful to do so. Since the eight development stations are integrated into the total system, they make natural emulation "consoles" from which to monitor the emulated system's behavior. The developed interpreter will be loaded onto some subset of the total facility and executed. Partitioning makes possible use of the facility by more than one user at the same time and assures that packets from two separate emulations will never mix in the network.

Remote Access: Once the emulator is established we envision several ways for making it useful to other U.S. researchers. One approach would be to duplicate the machine in whole or in part. This effort would only involve securing of the necessary hardware from the manufacturer and of the necessary software from us.

Alternatively, the target interpreter could be developed remotely on another Symbolics 3600, Symbolics LM-2, LMI Lambda, or MIT CADR and then transmitted via local area network or ARPANET to the emulation facility for processing. (See Figure 2-3.) The emulation results would be then communicated in reverse to the remote user.

Due to the flexibility of the Symbolics 3600 hardware, it is possible to keep complete memory core images of several target interpreters at the emulation facility site. This flexibility would allow usage of the emulation facility by researchers elsewhere in our Laboratory and throughout the United States.

The scenario for remote usage of the emulation facility is straightforward: A researcher designs, generates, and debugs code for the facility at the remote site, using another Symbolics 3600,

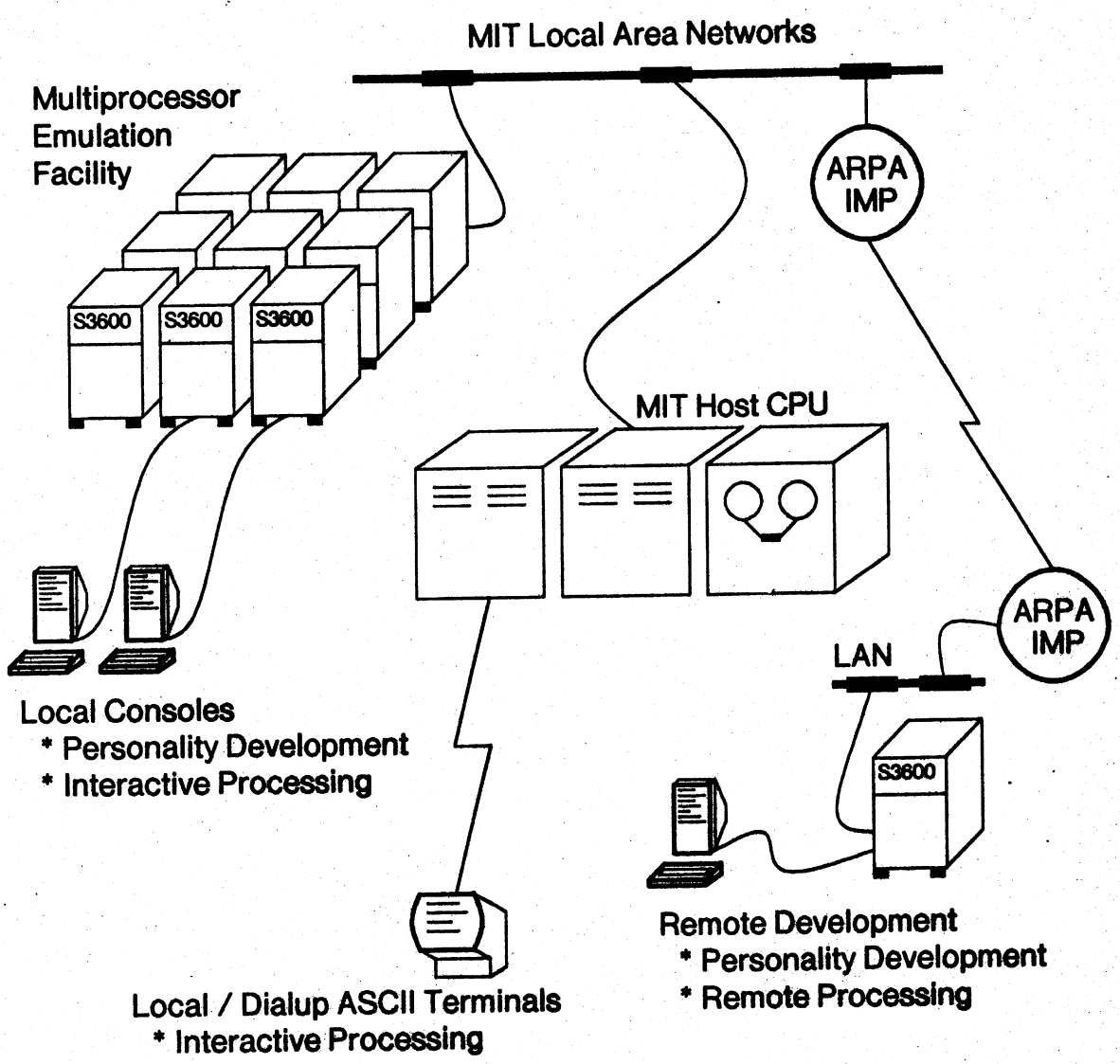

Figure 2-3: Use of the Multiprocessor Emulation Facility

Symbolics LM-2, LMI Lambda, or MIT CADR Lisp Machine. Since the dialects of LISP provided by these three machines are approximately compatible, this presents no translation hardship at a later date. When the researcher's program is completed, it is transmitted to the emulation facility maintainers (via the ARPANET) in the form of a compiled LISP program or a group of programs.

At the MIT emulation site, a "simple" 3600 machine core image is developed and stored. This simple core image will contain only the bare minimum to implement the LISP interpreter and compiler, together with the lower-level primitives of the emulation system. Upon reception of a researcher's system (*i.e.*, target interpreter), a new core image is built from this base and the user's program. This core image is then loaded into each of the processing elements of the system, and emulation is activated.

This solution to the problem of remote usage of the system was chosen for its natural simplicity and flexibility. It allows the researcher to develop programs in a leisurely way, on a foreign host. Remote users need only know the barest minimum about the facility itself - primarily that it runs standard Lisp Machine LISP with some added primitives but without any program development tools (such as an editor, disassembler, etc.).

Software Structure of the MEF: We propose to provide users of the emulation facility with all the necessary primitives for constructing a target interpreter. Each user will initially view the emulator as 64 separate but interconnected processors. It will then be up to each such user to piece together the supplied routines for remote memory reference or message-based communication in order to form the target system.

In developing the software structure of the MEF, our first goal will be to produce a LISP environment that can run without virtual memory. This is essential because initially 56 out of the 64 Symbolics 3600s would be without any disk or I/O subsystem. This LISP environment includes LISP primitives that provide (1) access to memory of other processors, and (2) capability of sending and receiving messages to/from other processes. In developing this environment, we will explore the possibility of providing a global uniform address space in which the eight higher order bits of an address will name the processor and the 24 lower order bits will be the local address within the so-named processor. Ideally, we would like to modify the addressing architecture of Symbolics 3600's so that a remote memory reference is automatically (*i.e.*, without invoking any primitive functions) processed by the local processor. The technical challenge of implementing such a scheme arises from the fact that remote memory references must be pipelined for the MEF to exhibit high performance emulation. Thus, not only does a remote reference have to be automatically generated, but the micro-task requesting such a reference must be automatically set aside in favor of a task that is ready to execute.

Beyond these factors, there are several questions regarding synchronization primitives, garbage collection and management of eight separate I/O subsystems that will be addressed in the course of developing the MEF's software subsystems.

Since Symbolics 3600s come with a sophisticated LISP programming environment, we do not plan to expend any effort in writing utility programs such as editors, formatters, local area network software, display management, single disk I/O subsystems, or LISP compilers.

2.5. Benchmarking and Measurement

An important measurement of the effectiveness of an emulation is the ratio of *emulated* machine cycles to *real* machine cycles. While it is difficult to quantify precisely this ratio we believe that the power and generality of the Symbolics 3600 micro-machine (*viz.*, tagged architecture, 32 bit barrel shifter / mask unit, powerful micro-tasking, high-speed memory) should keep the ratio between 100 (very simple target architectures) and 5000 (very complex target architectures). Thus, with a micro-cycle time of 200 nsec., we can expect an emulated machine cycle for reasonable applications (*i.e.*, one that implements a substantially different instruction set) to be in the neighborhood of 20 μsec. to 1 msec. For a 64-processor configuration, then, we can anticipate an upper bound of

approximately 3.2 million emulated cycles per second.

Arithmetic speed of the base machine is a less important measure than the emulated-to-real ratio, primarily because arithmetic is generally a small percentage of the overall emulator code - more time is spent in doing data movement and reformatting than in doing arithmetic for all but the most arithmetic-intensive target architectures. Fixed-point addition of 32 bit numbers is done in a single micro-cycle on the 3600; floating point arithmetic has been specified to run at approximately one million floating point operations per second (hardware assisted - not quite an order of magnitude slower if done in micro-code).

Memory size is an important measure, and has already been discussed. The facility has the potential to grow to 512 megabytes of real storage (64 processors with 8 megabytes per processor). Because of the switch design, it is also possible to add another 64 processors to this base with the attendant increase in overall capacity.

The overall efficiency of emulating other architectures of current interest (*e.g.*, the Connection Machine [11], Columbia University's Non-Von [19], University of Texas' TRAC [18], NYU Ultracomputer [8], MuNets [9] and the BBN Butterfly machine [17]) on the MEF is open to question. The interconnection schemes used in all these machines can be more or less directly emulated on the MEF (the only exception is the Ultracomputer which requires a fundamentally different switch). A lack of global clock type synchronization between processors may pose difficulty in modeling those architectures which make heavy use of a global clock. However two facts should be noted that (1) in all cases, the MEF would provide a far superior emulation/simulation vehicle (both in terms of performance as well as the ease of use) than what is available to researchers of these groups, and (2) applications to be run on these novel architectures can be directly implemented on the MEF. It should come as no surprise that the MEF may not easily out perform a machine which is a direct implementation of a specific architecture.

3. The Tagged-Token Dataflow Machine

The architecture of an effective multiple processor system, it has been argued [2], must address two basic issues. The first is the ability to tolerate memory latency, *i.e.*, the time between issuing a memory request and getting a response. The second is the ability to share data between processes without constraining parallelism. We believe that the traditional von Neumann model precludes a satisfactory solution to these problems.

The dataflow architecture on the other hand addresses these problems at a fundamental level - its model of computation. The goal of dataflow research is to solve these problems and to establish that linear computation speedup can be achieved with linear increase in hardware complexity.

3.1. The Dataflow Alternative

When one desires to build a machine capable of issuing multiple memory requests and of tolerating long latencies, the most troublesome aspect of the von Neumann architecture is the built-in sequentiality (*viz.*, the program counter). By eliminating the notion of control flow for program sequencing, we can circumvent this problem directly. One alternative to sequential control flow is *dataflow*, where the execution of instructions is triggered solely by the availability of the operands [5]. In order to explain the operation of a dataflow processor, we proceed next with a discussion of program, data and structure representations.

Program Representation: Dataflow compilers translate high-level programs into directed graphs; *vertices* in the graph correspond to machine instructions, and *edges* correspond to the data dependencies which exist between the instructions.

The implication is, quite simply, that instructions which depend on other instructions should be sequenced accordingly; but where no dependence (edge) exists, instructions can be executed in parallel. A simple example of this graphical translation is shown in Figure 3-1, compiled from the following ID program which integrates a function f from a to b over n intervals of size h by the trapezoidal rule:

```
(initial s ← (f(a)+f(b))/2;
        x ← a+h
 for i from 1 to n-1 do
    new x ← x+h;
    new s ← s+f(x)
 return s)*h
```

The graph shown is somewhat stylized; the box marked *f* represents the subgraph necessary for invoking function f (which is, itself, a graph). Instructions D, D^{-1}, L, and L^{-1} are included to provide proper entry, iteration, and exit by manipulating context-identifying information (discussed in the next section). The remainder of the operators are arithmetic, relational, and conditional instructions whose function should be self-evident. The graph generated by the compiler is reentrant and is shown in Figure 3-1.

Data Representation: It is the processor's task to propagate data values through the graph of Figure 3-1, triggering instructions when the operands are available. Data values are carried on logical entities called *tokens*; a token contains not only a data value but also the name of the instruction to which it belongs. Conceptually, tokens move about on the vertices of the graph. Instructions are enabled for execution when tokens are present on all input vertices. Upon execution, the instruction absorbs the input tokens, and produces an output token for the next instruction in the graph. A program is said to *terminate* when no enabled instructions are left.

Our execution model allows more than one token to be present on an arc; and, therefore, the next-instruction label also contains some dynamic, or context-sensitive information. In their full generality, these next-instruction labels or *activity names* contain four parts:

- **u**: The *context* field, which uniquely identifies the context in which a code block is invoked. The context itself is specified by an activity name, thus making the definition

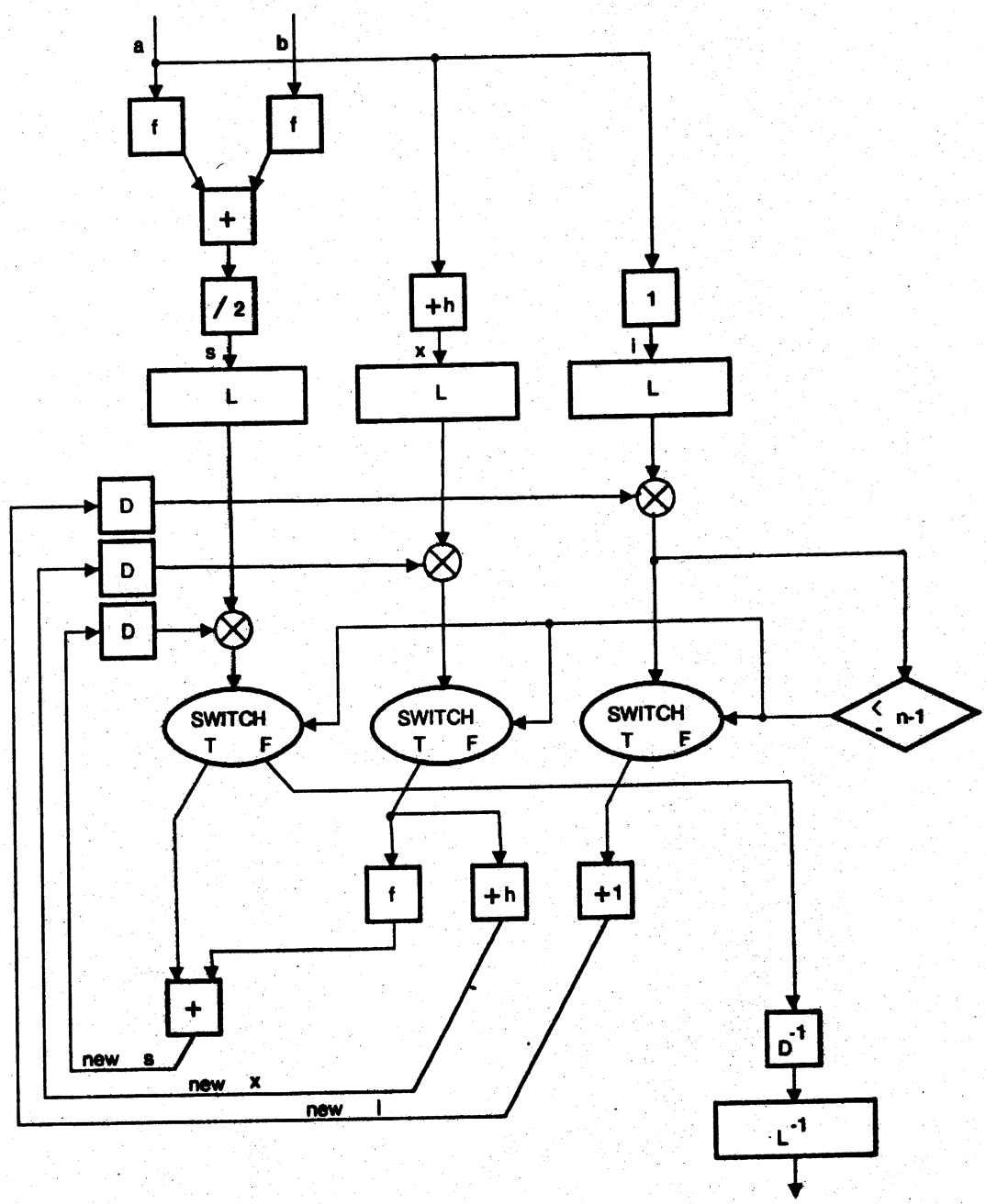

Figure 3-1: Compilation of the Loop Expression for the Trapezoidal Rule

recursive.

- c: The *code block* name. Each procedure and each loop has a unique code block name.
- s: The *statement* (instruction) number within the code block.
- i: The *initiation* number, which identifies the loop iteration in which this activity occurs.

This field is 1 if the activity occurs outside a loop.

Activity names, then, define an unbounded namespace. Names in this space are mapped dynamically into a finite namespace. The activity name plus some mapping information uniquely define the runtime *tag* and processing element (PE) number.

Since instructions may have more than one input operand, we also include two more pieces of information on each token: the *total* number of operands required by its target instruction (called *nt* - number of tokens), and an index value (called the *port*) which specifies the operand number associated with this token. Tokens of this type are called *normal* tokens, abbreviated as $d=0$.[1] (A more complete discussion of formats is given in [1].) The complete token, then, looks like this:

<d=0,PE,tag,nt,port,data>

Structure Representation: An important property for data structures of interest is that the mechanics of their creation and use should also be highly parallel. To achieve this, we associate with each memory cell in the machine special flags (called *presence* bits) which indicate the memory cell's status - written or unwritten. This gives us the ability to solve the *read-before-write* synchronization problem as follows: Assume that a memory module has just received a request to read a particular memory location and to forward the contents to instruction x. The memory module interrogates the presence bits associated with that location. If the bits indicate that the cell has already been written into, the contents are retrieved and forwarded to instruction x. If the bits indicate that the location is empty, the memory module puts the read request aside, and marks the empty location to indicate that a read request is outstanding.

This mechanism, when coupled with a processor which is able to issue multiple, overlapped memory requests and which can tolerate out-of-order responses, allows the uncoupling of memory latency from the performance of a multiprocessor. We call such a memory *I-structure storage* [3, 10].

Organization of the Dataflow Machine: Figure 3-2 is a block diagram of an abstract dataflow machine and its processing element. Assume that the program to be executed has been compiled into a directed graph, and an encoding of this graph is stored in the *program memory*. As tokens arrive at the machine's input, they are classified according to type. The $d=0$ tokens (above) which require partners ($nt \geq 2$) are routed to the the *waiting - matching* section.

Since each token carries the name of its target instruction, we can match up related tokens (*e.g.*, the two input operands for an addition) by comparing the tags that they carry. This is the function of the *waiting-matching* section. When a match is found, the pair is passed on to the *instruction fetch* unit. When a match is expected but not found, the token remains in the waiting-matching unit's associative memory until its partner arrives. The instruction fetch unit also directly receives $d=0$ tokens which require no partners ($nt=1$).

The instruction fetch unit looks up the operation code and other information associated with the

---

[1] The reason for this notation has been lost in historical obscurity.

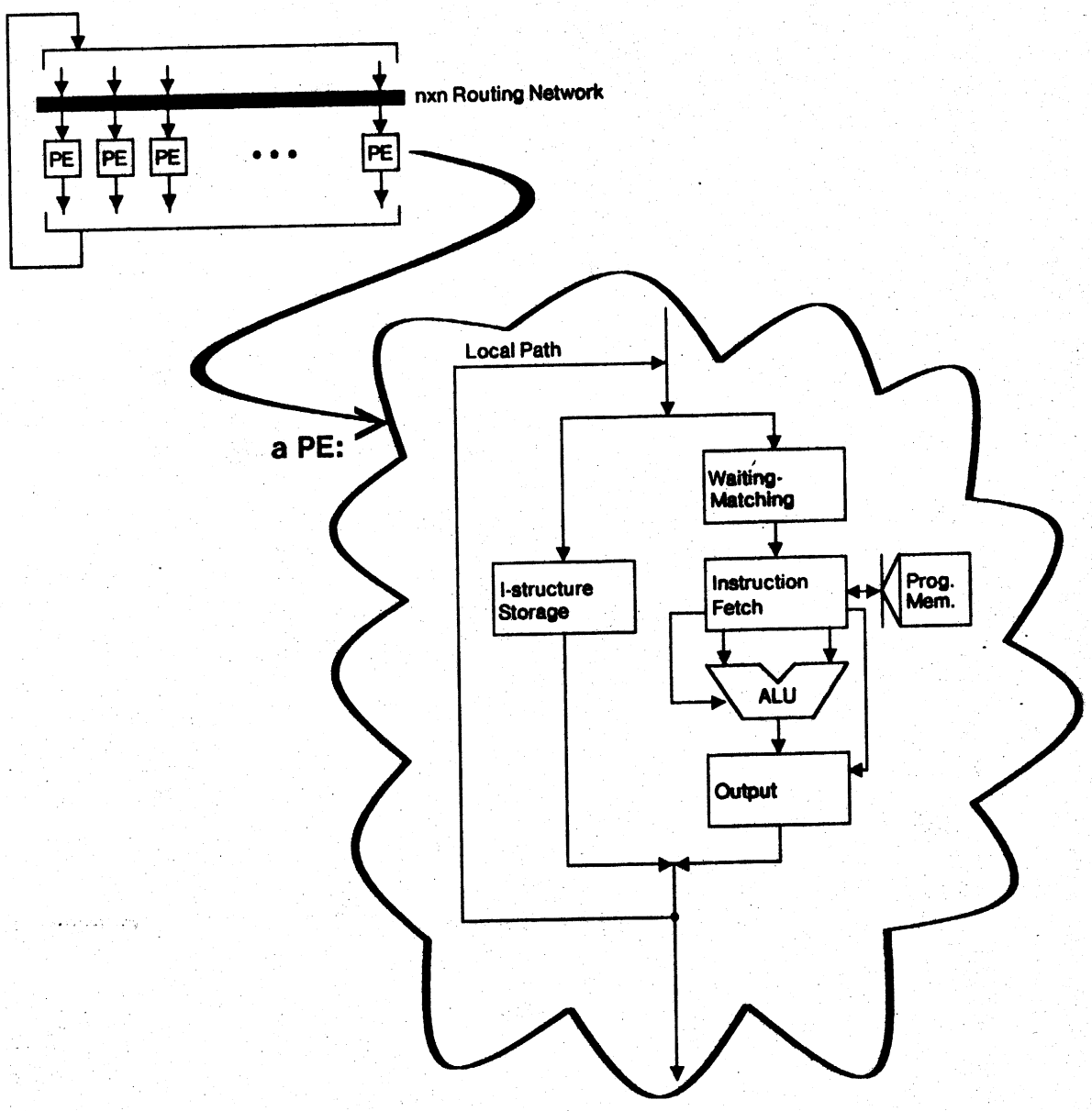

Figure 3-2: Organization of the Tagged-Token Dataflow Machine

token-carried names, and passes this enabled instruction on to the *ALU*. At this point, no other information is needed to carry out the operation save that which is in this enabled instruction packet.

The ALU output represents a datum which is ready to move off to its target instruction; but first, it has to be put in a token. We build this output token by computing a new tag, using the old tag along with information stored in the instruction itself. The *output* section handles these operations.

The output section also computes the PE number for the new token. A routing translation table turns this PE number into a network routing address.

The second major data path in the machines is for the processing of *fetch* and *store* requests in the I-structure storage unit. Since these memories collectively form a global address space in the dataflow multiprocessor, the I-structure storage unit will receive both local and nonlocal requests, and must be able to forward the results accordingly.

State of the Research: The previous paragraphs have briefly described the proposed solution that is indicated by many years of investigation into dataflow architectures. The details are sufficiently well understood that it is appropriate at this time to construct a prototype tagged-token dataflow machine for the purpose of experimentation and evaluation.

At present, we have a working compiler which translates programs written in our high-level dataflow language ID into directed graphs. We also have several application programs (such as a PDE Simulation - SIMPLE, and a MOS circuit simulation - MOSSIM) already coded in ID. In addition to these engineering and scientific applications we are interested in artificial intelligence applications that can be modeled as marker propagation on semantic networks (*e.g.*, NETL [6]).

4. Emulating the Dataflow Machine and Beyond

4.1. The Overall Plan

Our overall plan calls for: (1) constructing the emulation facility; (2) emulating our proposed dataflow architecture; (3) making the emulation facility available via ARPANET for the emulation of architectures proposed by other members of our Laboratory and of the DARPA community; (4) *simulating* the dataflow machine (discussed next) and finally (5) proceeding beyond the activities proposed here with VLSI implementation of the suitably redesigned machine.

We are currently writing a simulator that describes the internal behavior and timing of the dataflow machine at the subsystem level. Though internals of a subsystem (*e.g.*, waiting-matching section, ALU section in Figure 3-2) are not modeled in the simulator, data-dependent timing to process a token in any subsystem can be specified. The simulator will also model token traffic congestion and resolution of routing conflicts in the communication network. The simulation will run on IBM processors - an IBM 3081 at IBM Yorktown and, an IBM 4341 (loaned to us by IBM Endicott) located in our Laboratory. The simulator, when it is run in stand alone mode on an IBM system, will be a *simulated dataflow machine* in the sense that it will directly accept graphs generated by our ID compiler, and produce results like a real, albeit very slow, dataflow machine. In addition, it will also produce timing information which will be invaluable for understanding and evaluating the architecture.

The most critical parameters to be determined by the simulation experiments are: the size of buffers in the machine; size of the waiting-matching store to reduce the probability of overflow to an acceptable level; the size of the deferred read section in the I-structure storage; and the effect of the relative speeds of various subsystems and of the communication system on the overall

performance. We plan to execute *20 million dataflow instructions* per experiment, and our preliminary estimates show that it will take approximately *24 CPU hours on the IBM 4341* (stand alone) to conduct one experiment. These experiments are to be conducted in cooperation with IBM (Yorktown) whose personnel are involved in designing the experiments as well as modifying the architecture based on the outcome of experiments.

As discussed earlier, the emulation of the Tagged-Token machine on the proposed emulation facility will be done by modeling each subsystem as a micro-task on each Symbolics 3600. A conservative estimate shows that the emulated dataflow machine should run approximately 100 times faster than the simulated dataflow machine because of multiple processors and because no time-related information will be carried explicitly. A 100-fold speedup will allow us to study the dynamic behavior of programs for a longer duration. Without the emulated version of the dataflow machine, it is unlikely that end-users will get involved directly in writing code for the dataflow machine. All claims about the ease of programming of dataflow machines will be suspect if applications are not written by end-users.

It is likely that we may want to estimate the improvement in the overall performance of the dataflow machine by speeding up a subsystem (*e.g.*, waiting-matching section, I-structure controller), and such speeding up may be ruled out by the architecture of the emulation facility. Under such circumstances, we will have to rely on the simulator to study the desired effects. The simulator will also be crucial for fine tuning the emulated version of the dataflow machine. Thus, we think that emulated and simulated dataflow machines will complement each other and we consider both essential to establishing the viability and the programmability of the Tagged-Token dataflow machine.

Currently we are cooperating with Dr. Tilak Agerwala and his research group in Dr. Herb Schorr's organization at IBM Yorktown[2]. Our joint efforts are in the following areas: (1) simulation experiments including the development of programs to monitor the behavior of the dataflow machine; (2) paper designs of high performance versions of the Processing Element; and (3) compiler related work to decompose programs for mapping on multiple processors.

4.2. The First Experiment

The first experiment will be an emulation of the dataflow architecture. Because of the complexity of the target architecture, we have devised a three stage strategy. Stage one will be the implementation of the complete architecture on the emulator and will be written entirely in LISP. By doing so, we retain a significant degree of flexibility at a time when such flexibility is most crucial, *i.e.*, during the early testing and evaluation of the tagged-token machine.

One implication of this approach is that data would be stored in *simulated* memory, *i.e.*, in LISP arrays. Operations passed between the major blocks of the dataflow processing element will be represented by *flavor instances*, or at least *structures*, to give us the maximum coding simplicity and

---

[2]Dr. Schorr is Vice President, Systems in IBM's Research Division.

flexibility.

Since LISP-simulated storage will, in the long run, be too costly in terms of access time penalty for our purposes, the second stage would be to use the innate 3600 store. In addition, since most of the processors in the completed machine will not have virtual memory, this is the first stage in which we can run real computations. PE memory will be represented directly by *wired* pages, which the LISP code deposits into and reads from with the standard 3600 micro-coded primitives. We should be able to set aside wired, non-garbage-collected, untouched-by-LISP, *permanent* pages to hold I-structure memory, program memory, and the waiting-matching store. Most or all of the code at this stage, however, would still be in LISP.

The last stage of code migration would be the translation of some of the LISP code into micro-code. Before we come to this point, many test runs will be executed to determine any *hot spots* in the target interpreter code, that is, LISP code fragments which are executed often and which can be executed faster by micro-coding.

4.3. Expected Emulator Performance

Due to the complexity of the Tagged-Token architecture, we are estimating mean code paths on the order of 500 to 5000 micro-instructions per emulated machine cycle (including the overhead of those functions implemented in LISP rather than in micro-code). This implies that the facility will interpret dataflow graphs at a rate of approximately 64,000 to 640,000 instructions per second.

It should be clear that the proposed emulation facility will be substantially easier to construct than a VLSI version of the intended target systems (especially the Tagged-Token dataflow machine). "Cost" of construction, aside from the necessary financial support, involves designing, building, testing, and replicating the packet switch module along with the necessary micro-code routines for using the switch and for error recovery.

4.4. Summary

This paper has presented our plan for constructing a Multiprocessor Emulation Facility comprised of 64 user-microprogrammable processors interconnected with a high bandwidth, reconfigurable communications network. The Facility will be used by a number of projects, but our primary interest will be its use in studying the Tagged Token dataflow architecture. To that end, we expect to have a small piece of the Facility operational (*i.e.*, only a few processors hooked together by a preliminary network) by the middle of 1984. The full facility will be available to the MIT community late in 1985. By then, a full dataflow emulator will have been completed including the necessary microcoding of hot spots. The Facility and the dataflow emulator will be ready for demonstration and use outside MIT by the end of 1985.

References

1. Arvind, and R. A. Iannucci. Instruction Set Definition for a Tagged-Token Data Flow Machine. Memo 212, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., December, 1981. revised February, 1983

2. Arvind, and R. A. Iannucci. A Critique of Multiprocessing von Neumann Style. Proc. of the 10th International Symposium on Computer Architecture, June, 1983.

3. Arvind, and R. E. Thomas. I-Structures: An Efficient Data Type for Functional Languages. Tech. Rep. TM-178, Laboratory for Computer Science, MIT, Cambridge, Mass., September, 1980.

4. Denneau, M. M. The Yorktown Simulation Engine. 19th Design Automation Conference, Las Vegas, Nevada, Institute of Electrical and Electronics Engineers, Piscataway, N. J., 08854, 1982.

5. Dennis, J. B. First Version of a Data Flow Procedure Language. In *Lecture Notes in Computer Science, Volume 19: Programming Symposium: Proceedings, Colloque sur la Programmation,* B. Robinet, Ed., Springer-Verlag, 1974, pp. 362-376.

6. Fahlman, S. E. *NETL: A System for Representing and Using Real World Knowledge.* MIT Press, 1979.

7. Gostelow, K. P., and R. E. Thomas. Performance of a Simulated Dataflow Computer. *IEEE Transactions on Computers C-29,* 10 (October 1980), 905-919.

8. Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers C-32,* 2 (February 1983), 175-189.

9. Halstead, R. H. The Architecture of a Myriaprocessor. Proceedings of COMPCON 81, September, 1981, pp. 299-302.

10. Heller, S. K. An I-Structure Memory Controller. Master Th., Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., June, 1983.

11. Hillis, W. D. The Connection Machine: Computer Architectue for the New Wave. Tech. Rep. 646, Artificial Intelligence Laboratory, MIT, Cambridge, Mass., September, 1981.

12. Iannucci, R. A. Implementation Strategies for a Tagged-Token Data Flow Machine. Memo 218, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., June, 1982.

13. Iannucci, R. A. Packet Communication Switch for a Multiprocessor Computer Architecture Emulation Facility. Memo 220, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., October, 1982.

14. Kronstadt, E., and G. Pfister. Software Support for the Yorktown Simulation Engine. 19th Design Automation Conference, Las Vegas, Nevada, Institute of Electrical and Electronics Engineers, Piscataway, N. J., 08854, 1982.

15. Pfister G. F. The Yorktown Simulation Engine: Introduction. 19th Design Automation Conference, Las Vegas, Nevada, Institute of Electrical and Electronics Engineers, Piscataway, N. J., 08854, 1982.

16. Preparata, F. P., and J. Vuillemin. The Cube-Connected Cycles: a Versatile Network for Parallel Computation. *Comm. ACM 24*, 5 (May 1981), pp. 300-309.

17. Rettberg, R., C. Wyman, D. Hunt, M. Hoffman, P. Carvey, B. Hyde, W. Clark, and M. Kraley. Development of a Voice Funnel System: Design Report. Tech. Rep. 4098, Bolt Beranek and Newman Inc., August, 1979.

18. Sejnowski, M. C., *et. al.*. Overview of the Texas Reconfigurable Array Computer. Proc. AFIPS, Vol. 49, 1980, pp. 631-642.

19. Shaw, E. D. The Non-Von Supercomputer. Department of Computer Science, Columbia University, August, 1982.

20. Swan, R. J., S. H. Fuller, and D. P. Siewiorek. Cm* - A Modular Multiprocessor. Proceedings of the National Computer Conference, 1977.

21. Swan, R. J., A. Bechtolsheim, K-W. Lai, and and J. Ousterhout. The Implementation of the Cm* Multi-microprocessor. Proceedings of the National Computer Conference, 1977.

This blank page was inserted to preserve pagination.

CS-TR Scanning Project
Document Control Form

Date : 7/14/95

Report # LCS-TR-302

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 25 (32 IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:
☒ DOD Form (2) ☐ Funding Agent Form ☒ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number: UN#'ED

Ⓐ IMAGE MAP! (1-4) UN#'ED TITLE, ABSTRACT, i, UN#'ED 1ST PAGE
(5-25) PAGES #'ED 2-22
(26-29) SCANCONTROL, COVER, DOD (2)
(30-32) TRGT'S (3)

Ⓑ ABSTRACT HAS ~~tear~~ TAPE ACROSS TOP THIRD.

Scanning Agent Signoff:
Date Received: 7/14/95 Date Scanned: 7/18/95 Date Returned: 7/20/95

Scanning Agent Signature: Michael W. Cook

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: MIT/LCS/TR-302	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): A Multiprocessor Emulation Facility	5. TYPE OF REPORT & PERIOD COVERED: Interim Research 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Arvind, Michael L. Dertouzos, Robert A. Iannucci	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0661
9. PERFORMING ORGANIZATION NAME AND ADDRESS: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: DARPA/DOD/IPTO, 1400 Wilson Boulevard, Arlington, VA 22209	12. REPORT DATE: September 1983 13. NUMBER OF PAGES: 20
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Office of Naval Research/Dept of Navy, Information Systems Program, Arlington, VA 22217	15. SECURITY CLASS. (of this report): Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Architecture, dataflow, emulation multiprocessor systems, packet communication, simulation.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Interest in multiprocessor computer architectures has increased dramatically in the last ten years. However, it has become clear that in order to effectively use multiprocessors in a general way, some fundamental changes in the model of compuation are necessary. Moreover, experimentation in the field is hindered by low-performance simulation tools and high-cost hardware modeling schemes.

...continued

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Continued...

We present our approach to solving both of these problems: DATAFLOW ARCHITECTURE, which provides an inherently parallel model of computation, and a MULTIPROCESSOR EMULATION FACILITY, which is a general purpose tool for evaluating multiprocessors architectures at low cost. We also discuss our scheme for using the Emulation Facility to validate our claims about dataflow architecture.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 7/19/1995

M.I.T. Libraries
Document Services