LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Two Fundamental Issues in Multiprocessing: The Dataflow Solution

Computation Structures Group Memo 226-3
7 August 1985

Arvind

Robert A. Iannucci

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099. The second author is supported by the International Business Machines Corporation.

Source File = TFI.MSS.19, Last updated 7 August 1985 at 4:09pm

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Abstract

To exploit the parallelism inherent in algorithms, any multiprocessor system must address two very basic issues - long memory latencies and waits for synchronization events. It is argued on the basis of the evolution of high performance computers that the processor idle time induced by memory latency and synchronization waits cannot be reduced simultaneously in von Neumann style multiprocessors. Dataflow architectures are offered as an alternative because, given enough parallelism in a program, they can reduce both latency and synchronization costs.

Key words and phrases: caches, cache coherence, dataflow architectures, hazard resolution, I-structure storage, instruction pipelining, LOAD/STORE architectures, multiported memories, multiprocessors, von Neumann architecture.

This paper is a major revision of our previous work on the subject:

Version	Title
CSG Memo 226-1	*A Critique of Multiprocessing von Neumann Style* Presented at the 10th International Symposium on Computer Architecture, Stockholm, Sweden, June 14-17, 1983
CSG Memo 226-2	*Two Fundamental Issues in Multiprocessing: the Dataflow Solution* Reprinted as MIT/LCS/TM/241 September, 1983

Table of Contents

List of Figures

Two Fundamental Issues in Multiprocessing: The Dataflow Solution

1. Importance of Processor Architecture

It is generally believed that processor architecture is of little importance in designing parallel machines. To show the fallacy of this assumption, we will discuss two basic issues, namely, *memory and communication latency* and *synchronization*, that any architect of a scalable, general purpose parallel machine must confront. We believe these issues to be as limiting and as fundamental as those imposed by circuit technology (power consumption, heat dissipation, length and thickness of wires, packaging, etc.). We further believe that parallel machines of the next generation are more likely to hit these architectural limits than the technology-imposed limits. As such, they are more immediately relevant.

We are primarily interested in *general purpose parallel computers*, *i.e.*, computers that can exploit parallelism, when present, in any algorithm. Further, we want multiprocessors to be *scalable* in such a manner that adding hardware resources results in higher performance without requiring substantial rewriting of application programs. The benefits of such scalable systems are obvious; the pitfalls in designing them are subtle.

To understand the effect of latency and synchronization on performance, one also needs to understand the execution of programs on parallel machines. First of all, one needs to identify parallel subcomputations in a source program. This can be done with the help of a compiler, via programmer-provided annotations, or both. To exploit this parallelism, a parallel machine must provide run time support for the *creation* and *synchronization* of tasks corresponding to these subcomputations.

In the next few sections we present our framework for addressing the issues of latency and synchronization, and our formal statement of the problem. The framework is based on two abstract and fairly orthogonal views of multiprocessors. One view which deals with the gross hardware organization is embodied in the *structural model*; the other view which deals with the essential elements of parallel programming is embodied in the *operational model*. Our statement of two fundamental issues in multiprocessor design is based on these models.

1.1. The Structural Model

The model shown in Figure 1-1 will be used as the baseline for describing multiprocessor organizations in this paper. It abstracts away the physical packaging and network topology because, as shall become clear later, the design of these aspects of multiprocessors will not affect the main hypothesis of this paper. The *structural model* is made up of three parts:

- **Processing elements:** Modules which perform arithmetic and logical operations on data. Each processing element (PE) has a single *communication port* through which all data values are received. Processing elements interact with other processing elements by issuing and responding to *synchronizing signals*, *e.g.*, WAIT and SEND semaphores,

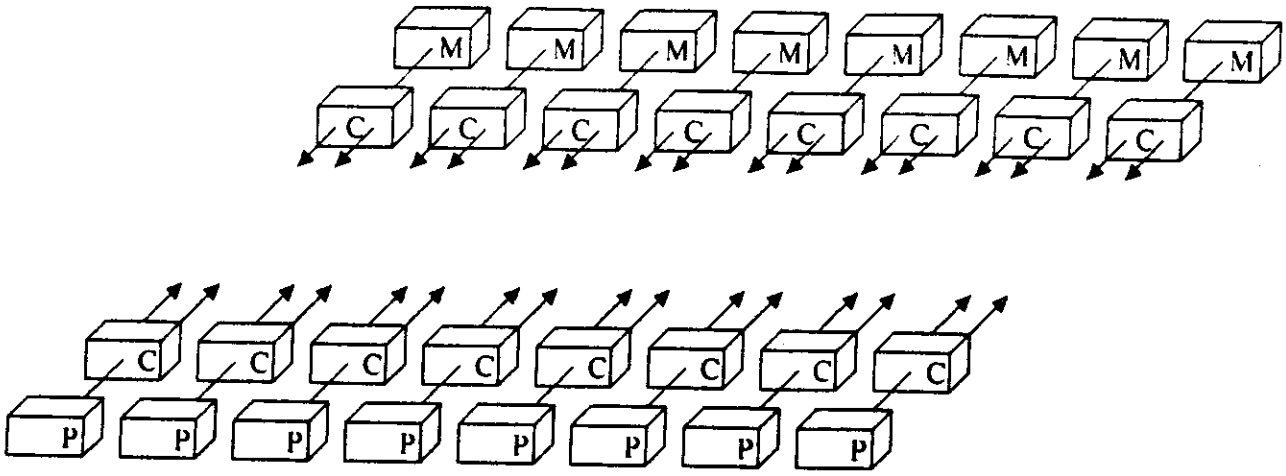

Figure 1-1: Structural Model of a Multiprocessor

interrupts, etc., and with memory elements by issuing LOAD and STORE instructions embellished as necessary with atomicity modifiers. Processing elements are characterized by the rate at which they can issue and respond to such signals, instructions, and data.

- **Memory elements:** Modules which store data. Each memory element has a single communication port. Memory elements respond to instructions issued by the processing elements by returning data through the communication port, and are characterized by their total capacity and the rate at which they respond to these instructions.

- **Communication elements:** Modules which transport data. Each nontrivial communication element has at least three communication ports. Communication elements neither originate nor receive synchronizing signals, instructions, or data; rather, they retransmit such information when received on one of the communications ports to one or more of the other communication ports. Communication elements are characterized by the rate of retransmission, the time taken per retransmission, and the constraints imposed by one retransmission on others, *e.g.*, blocking. The maximum amount of data which may be conveyed on a *communication port* per unit time is fixed.

A multiprocessor system may be composed by interconnecting each communication port of one module with exactly one other communication port of another module. Processors and memories may only be connected to communication elements. Communication elements may be connected to either processors, memories, or other communication elements. Bus oriented multiprocessors can also be represented in our structural model given that communication elements are capable of *broadcast* behavior. This aspect of the bus structure can play an important, albeit indirect, role in lowering memory latency as we shall see in section 2.3.1.

1.2. The Operational Model

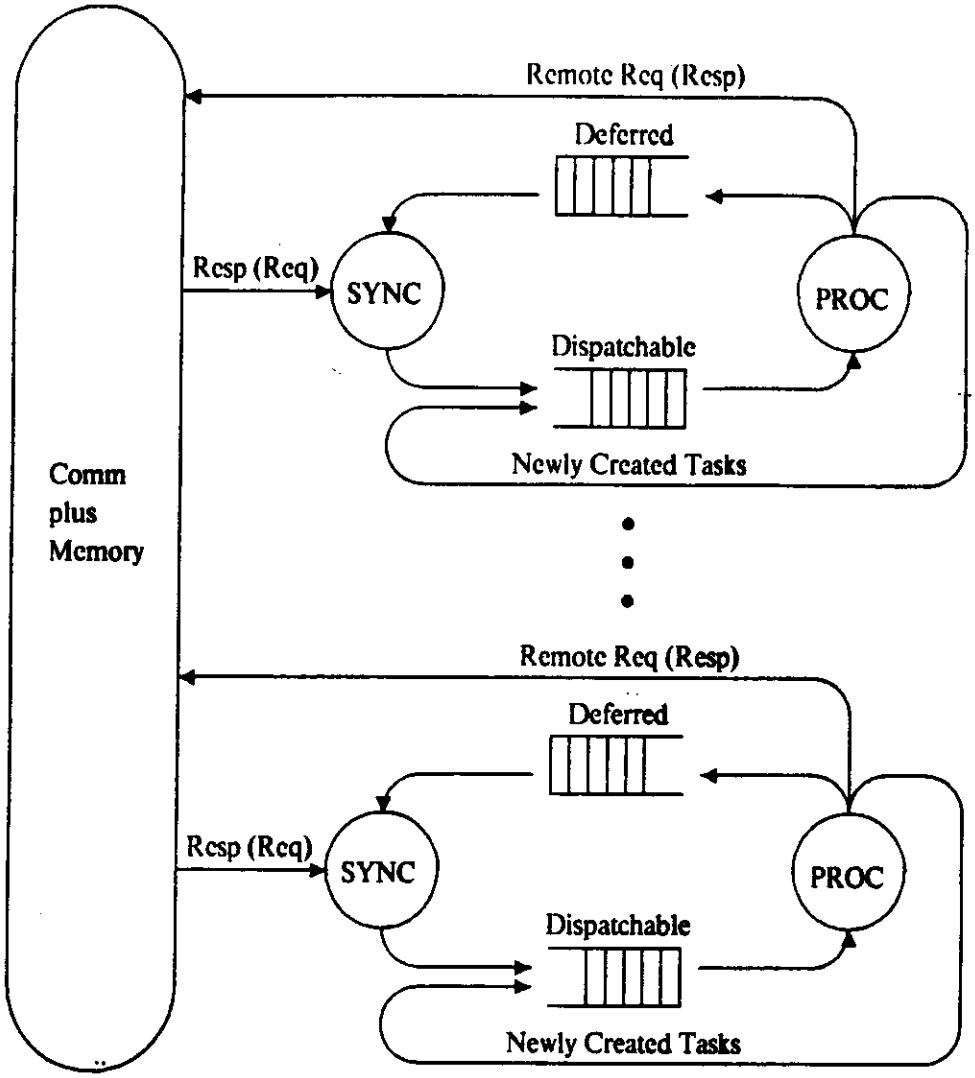

Figure 1-2: Operational Model of a Multiprocessor

To derive the benefits of parallel hardware, a program must be decomposed into basic units of computation which we shall call *computational tasks* or simply *tasks*. One may view these tasks as being units of work as small as machine instructions or as large as procedures comprised of thousands of instructions. Tasks have several interesting properties:

- They are the smallest unit of independently schedulable work on the machine.
- The set of legal primitive operations a task may perform must include one which is capable of spawning another task.
- Tasks communicate with one another by sending and receiving signals and/or data, *e.g.*, one task produces data which is consumed by another task.

- Each task is logically associated with a set of unique names, called its *context*, to reference task-held resources such as memory locations, registers, etc.

We have chosen to model the operational behavior of a multiprocessor as shown in Figure 1-2. Tasks ready for execution may be queued locally (on a per-PE basis) or globally. When selected, a task will occupy a PE until it can proceed no further because it must wait for a synchronization signal. Some hardware or software scheduling mechanism may also suspend a task and release the PE if, for example, the task makes a nonlocal reference which may take a long time to process. Notice that regardless of whether the PE is released or not, when a task makes a nonlocal request, it is *logically suspended*, and will wait until the result of the nonlocal reference has been returned. At that point, the task becomes dispatchable once again. One may view the components of the model as operating asynchronously with respect to one another. The queues shown need not adhere to any particular ordering discipline, *e.g.*, FIFO or LIFO. An event to enable or dispatch a task needs a *name*, such as that of a register or a memory location, and thus, the machinery implied by our operational model must capture the essence of managing task-generated names for both task creation and task synchronization. Hardware design usually dictates the number of names available for synchronization as well as the cost of their use.

1.3. The Two Fundamental Issues

We now discuss issues related to *latency* and *synchronization*, two universal characteristics of multiprocessor organizations.

Latency is the time which elapses between making a request and receiving the associated response. Of immediate interest is memory latency which, in a multiprocessor system, determines the time taken to execute an instruction involving a remote operand reference. A PE in a multiprocessor system faces larger latency than in a uniprocessor system because of the transit time in the communication network between PE's and the memories. When latency cannot be hidden via overlapped operations, a tangible performance penalty is incurred. We will count the cost associated with latency as the total *induced processor idle time* attributable to the latency. It includes arbitration time, time of flight through the network, and the time required to process the request.

The Operational Model implies that tasks need to communicate with each other. For example, a task may produce a datum which is needed by another task, or may request a resource currently in use. In either case, the sequencing, or *synchronization*, of an event in one task with an event in another is required. The cost associated with such synchronization is also the *induced processor idle time* attributable to synchronization event waiting. It is made up of a fixed time to perform the synchronization operation itself plus the variable time of waiting for the satisfaction of the constraint. If a PE immediately suspends the task causing a synchronization event wait, the cost per synchronization event is fixed.

The performance of a parallel machine is likely to hit an absolute ceiling if adding processors to the machine increases processor idle time due to increased latency and a greater need for synchronization. Thus, we assert that

It is necessary to simultaneously minimize the costs of latency and synchronization in order to build a scalable multiprocessor.

One is tempted to assume that the latency issue pertains solely to the hardware organization of the machine, and that the synchronization issue pertains exclusively to the software systems that run on the machine. However, this is not the case. It is likely that attempts to reduce the latency cost will increase the synchronization cost and *vice versa*. In the rest of this paper we will show that

- for multiprocessor organizations based on *von Neumann processors*, it is impossible to independently minimize both latency cost and synchronization cost, and
- for a class of multiprocessor organizations *including dataflow architectures*, it is possible to independently minimize both latency cost and synchronization cost assuming the program has sufficient parallelism.

In Section 2 we trace the evolution of high performance von Neumann computers to show how reductions in latency costs have been achieved. It is shown that the techniques used for reducing latency costs are either not applicable in the multiprocessor setting or come at the expense of increasing the synchronization cost. Section 3 presents the essential features of dataflow machines, with special emphasis on the MIT Tagged-Token Dataflow machine to show how such architectures can trade parallelism in programs for simultaneous reductions in latency and synchronization costs. In Section 4, we present our view of how multiprocessor architectures should evolve in the future.

Throughout the paper we make references to many planned and existing uniprocessor and multiprocessor architectures. Most, if not all, of these machines were designed to achieve goals other than those set forth here, that is, scaling and programming generality. We have taken the liberty to analyze these architectures using our criteria. In spite of our sometimes less than flattering evaluation, these machines may be enormously successful in meeting their own goals. Note that our intention is to better understand the limits of multiprocessor architectures, and not to make an absolute value judgment on any machine.

2. Multiprocessing based on von Neumann Processors

In this section we begin our study of the evolution of von Neumann architectures. We ask for the reader's indulgence as we begin rather slowly and at a basic level. The points we wish to make are most clearly seen within such a simple, uncluttered framework, and it is for this reason that we reiterate what most readers will view as elementary[1].

Figure 2-1 depicts the modern day view of the von Neumann computer model (*sans* I/O) as a mutation of von Neumann's original vision. His description [7] was of a processor-memory pair with most or all of the computation's *state* residing in the memory. The depicted view emphasizes the migration of state toward the processor side of the processor-memory interconnection. The reasons for this have relevance to both uniprocessor and multiprocessor architectures and are discussed below.

---

[1] We further ask that the reader set aside any *technology specific* prejudices, *e.g.*, the relative speeds of processors and memories.

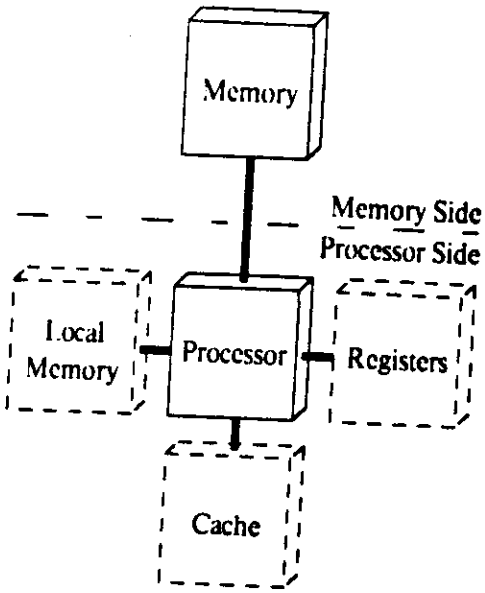

Figure 2-1: The von Neumann Processor (from Gajski and Peir [16])

The processor's sole purpose is to repeatedly carry out the following instruction interpretation cycle (see Figure 2-2):

1. Fetch an instruction from the memory.
2. Decode the instruction.
3. (Fetch operands from memory.)
4. Execute the decoded instruction using the fetched data.
5. (Store results in the memory.)
6. Determine the next instruction to be executed.

The steps shown in parentheses are optional for some instructions. In early machines the time taken to go through one iteration of this interpretation cycle was the cycle time of the machine. Different steps in the cycle took different amounts of time. Because memories were relatively slow compared to the processors, time to fetch an instruction and time to fetch and store operands completely dominated the cycle time. Speeding up the Arithmetic Logic Unit was of little use unless the memory access time could be reduced.

The earliest solution to speeding up the machine was to increase the *processor state, i.e.*, to provide fast storage on the processor side in the form of registers. Appearance of multiple "accumulators" reduced the number of operand fetches and stores, and index registers dramatically reduced the number of memory references by almost eliminating the need for self modifying code.

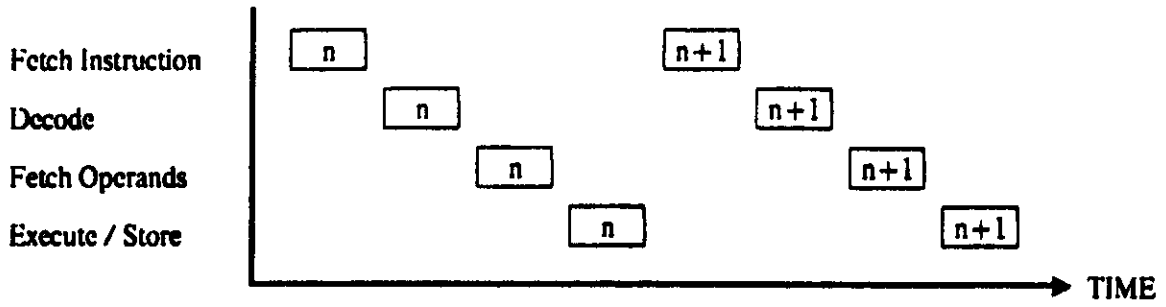

Figure 2-2: Traditional Interpretation of Instructions

A later technique involved reducing the number of instructions executed and, hence, the number of instructions fetched. This was accomplished by making the instructions themselves more complex (*i.e.*, defining a richer language). This technique was less successful than adding registers due to difficulty in designing complex control circuitry. Even though the cumulative effect of these two techniques was that programs executed much faster than before, the basic cycle time improved only as a function of improvements in circuit speeds, *i.e.*, technology. The enlarged processor state reduced the number of memory references, but it did not reduce the time lost during memory references and, consequently, did not contribute to an overall reduction in cycle time.

2.1. Pipelined von Neumann Processors

The most successful solution to hiding memory latency is the *pipelined execution* of instructions. The time taken by instruction fetch (and perhaps part of instruction decoding time) can be totally hidden if prefetching is done during the execution phase of the previous instruction (see Figure 2-3). The IBM STRETCH [6] and the Univac LARC [12] represent two of the earliest attempts at implementing this idea. Prefetching can reduce the cycle time of the machine by 20% to 30% depending upon the amount of time taken by the first two steps of the instruction cycle with respect to the complete cycle. However, the effective throughput of the machine cannot increase proportionately because overlapped execution is not possible with *all* instructions.

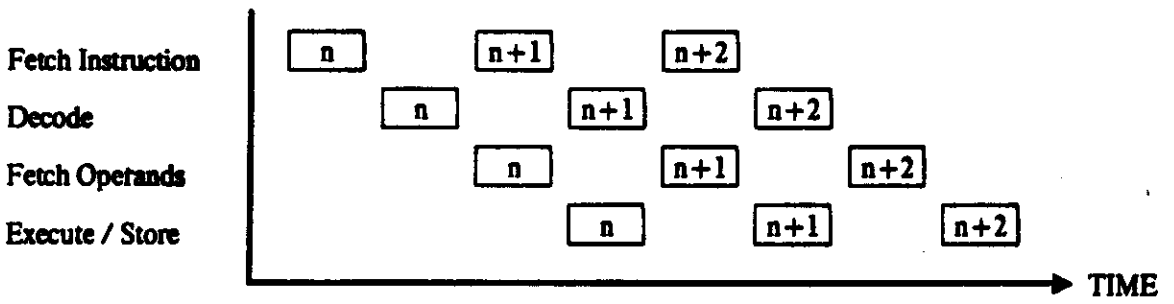

Figure 2-3: Overlapping of Instruction Fetch / Decode

Instruction prefetching works well when the execution of instruction n does not have any effect on either the choice of instructions to fetch (*e.g.*, as is the case in a BRANCH) or the content of the fetched instruction (*e.g.*, self-modifying code) for instructions n+1, n+2, ..., n+k. The latter case is usually handled by simply outlawing it. However, effective overlapped execution in the presence of BRANCH instructions has remained a problem. Techniques such as prefetching both BRANCH targets have not shown much performance/cost benefits. Lately, the concept of *delayed* BRANCH instructions from microprogramming has been incorporated, with success, in LOAD/STORE architectures (see section 2.2). The idea is to delay the effect of a BRANCH by one instruction. Thus, the instruction at n+1 following a BRANCH instruction at n is always executed regardless of which way the BRANCH at n goes. One can always follow a BRANCH instruction with a NO-OP instruction to get the old effect. However, experience has shown that 70% of the time a useful instruction can be put in that position.

If instructions and data are kept in separate memories (the so-called Harvard architecture, another idea borrowed from microprogramming), it is possible to overlap instruction prefetching with fetching of operands, too. It is also possible to reduce the instruction fetch time by providing a fast instruction buffer. The buffer may be automatically loaded with n instructions in the neighborhood of the referenced instruction (assuming some spatial locality in code references) if the referenced instruction is found to be missing (*e.g.*, the CDC 6600 [31]). To take advantage of either separate instruction memory or instruction buffers, it is necessary to also speed up the operand fetch and execute phases. The two most common techniques for doing this are:

- providing *operand* caches or buffers, and
- overlapping the operand fetch and execution phases (Figure 2-4).

If successful[2], these techniques can reduce the machine cycle time to one fourth or one fifth the cycle time of an unpipelined machine. However, overlapped execution of 4 to 5 instructions in the von Neumann framework presents some serious conceptual difficulties.

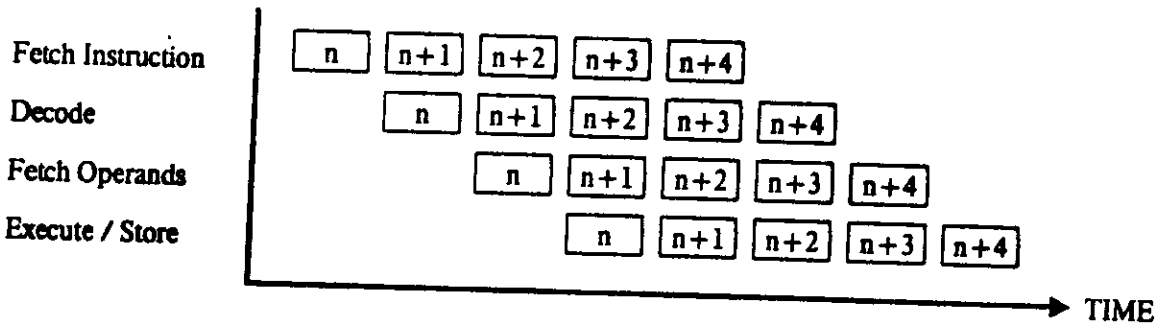

Figure 2-4: Totally Pipelined Execution

[2]Of course, it is likely that balancing the pipeline under these conditions may require further pipelining of the ALU.

2.2. Load/Store Architectures

Next, we discuss techniques used in machines built by Seymour Cray, *e.g.*, the CDC 6600 [31] and the Cray-1 [28], and more recently, by Reduced Instruction Set Computer (RISC) enthusiasts, *e.g.*, the IBM 801 [26], Berkeley's RISC [25], and Stanford MIPS [20], because of their success in pipelining von Neumann machines.

2.2.1. Difficulties in Instruction Pipelining

Designing a well-balanced pipeline requires that the time taken by various pipeline stages be more or less the same, and that the "things", *i.e.*, instructions, entering the pipe be independent of each other. Obviously, instructions of a program cannot be totally independent except for some special trivial cases. Instructions in a pipe are usually related in one of two ways: Instruction n produces data needed by instruction n+k, or only complete execution of instruction n determines the next instruction to be executed (the aforementioned delayed BRANCH problem).

Limitations on hardware resources can also cause instructions to interfere with one another. Consider the case when both instructions n and n+1 require an adder, but there is only one of these in the machine. Obviously, one of the instructions must be deferred until the other is complete. A pipelined machine must be able to temporarily prevent a new instruction from entering the pipeline when there is a possibility of interference with the machine resource requirements of instructions already in the pipe. Detecting and quickly resolving these possibilities of interferences, or *hazards* as they are commonly known, is very difficult with ordinary instruction sets, *e.g.*, IBM 370, VAX 11 or Motorola 68000, due to their complexity.

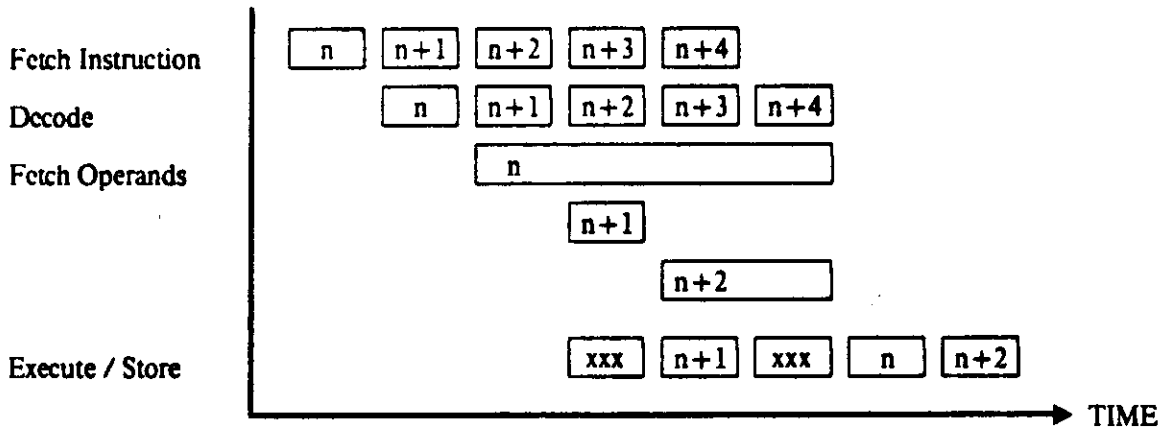

Figure 2-5: Variable Operand Fetch Time

A further complication in pipelining complex instructions is the variable amount of time taken in each stage of instruction processing (refer to Figure 2-5). Operand fetch in the VAX is one such example: determining the addressing mode for each operand requires a fair amount of decoding, and actual fetching can involve 0 to 2 memory references *per operand*. Considering all possible addressing mode combinations, an instruction may involve 0 to 6 memory references in addition to the instruction fetch itself. A pipeline design that can effectively tolerate such variations is close to impossible.

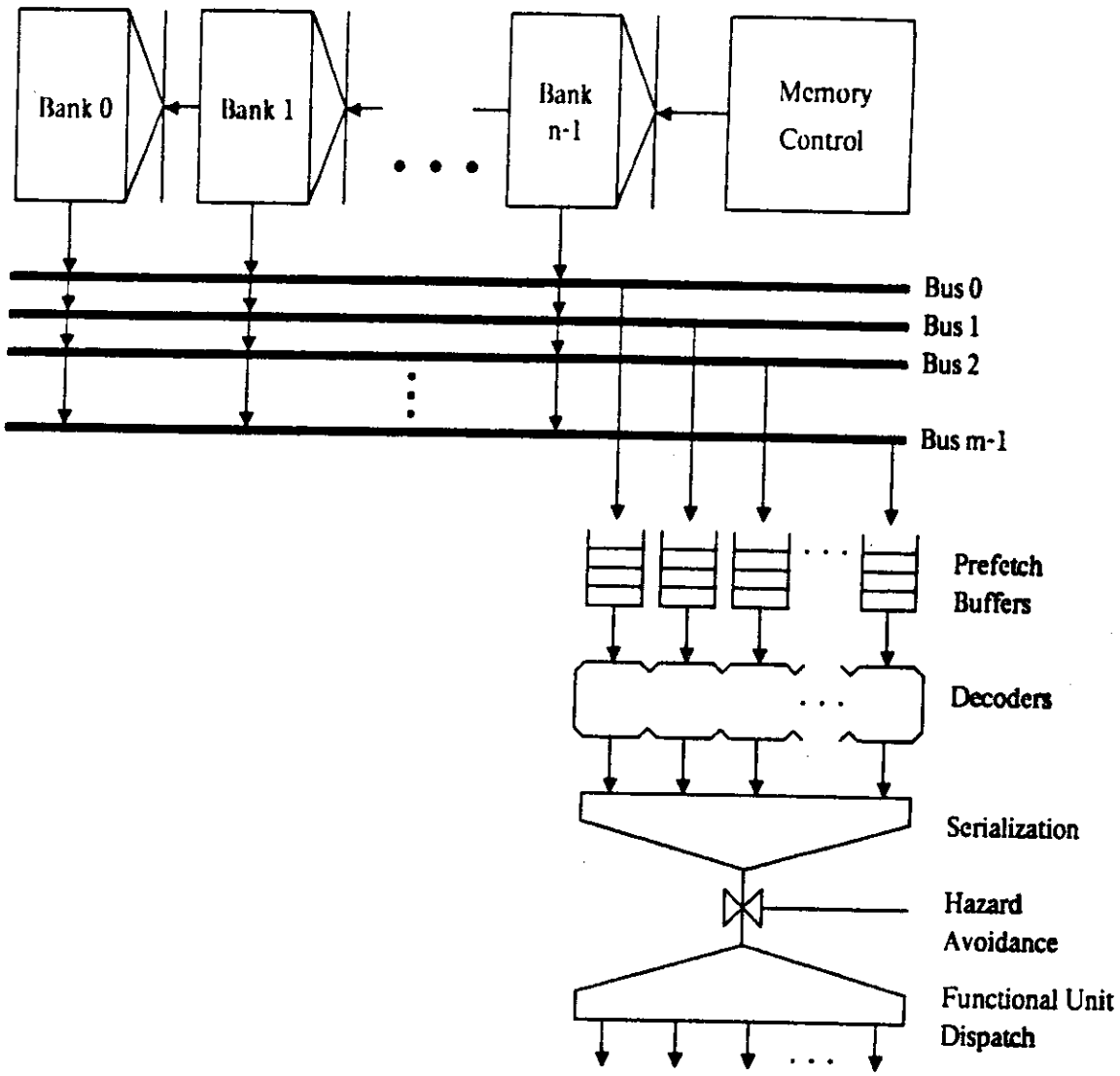

Figure 2-6: Hazard Avoidance at the Instruction Decode Stage

An idea of Seymour Cray (first seen in the CDC 6600) is to design an instruction set in which instructions that refer to memory are separable from those which do not at the instruction decode stage. Such is the case with LOAD/STORE architectures - the only memory reference instructions are those which move data unchanged between the memory and the registers. All other instructions are constrained to use the high speed registers [20, 26, 25]. Further, given that instructions in a pipeline cannot be independent of each other, the design of the pipeline is simpler if processing of an instruction can be stopped at only one stage of the pipeline. In other words, if an instruction gets past some fixed pipe stage, it should be able to run to completion without incurring or creating any previously unanticipated hazards. LOAD/STORE architectures allow for such implementations by using the time between instruction decoding and instruction dispatching for hazard detection and resolution (see Figure 2-6).

2.2.2. Hazard Resolution and Memory Latency

LOAD/STORE architectures are much better at tolerating latencies in memory accesses than other von Neumann architectures. In order to explain this point we will first discuss a simplified model which detects and avoids hazards in a LOAD/STORE architecture similar to the Cray-1. Assume there is a bit associated with every register to indicate that the contents of the register are undergoing a change. The bit corresponding to register R is set the moment we dispatch an instruction that wants to update R. Following this, instructions are allowed to enter the pipeline only if they don't need to reference or modify register R or other registers reserved in a similar way. Whenever a value is stored in R, the reservation on R is removed, and if an instruction is waiting on R, it is allowed to proceed. This simple scheme works only if we assume that registers whose values are needed by an instruction are read before the next instruction is dispatched, and that the ALU or the multiple functional units within the ALU are pipelined to accept inputs as fast as the decode stage can supply inputs[3]. The dispatching of an instruction can also be held up because it may require a bus for storing results in a clock cycle when the bus in needed by another instruction in the pipeline. Whenever BRANCH instructions are encountered, the pipeline is effectively held up until the branch target has been decided.

Notice what will happen when an instruction to load the contents of some memory location M into some register R is executed. Suppose that it takes k cycles to fetch something from the memory. It will be possible to execute several instructions during these k cycles as long as none of them refer to register R. In fact, this situation is hardly different from the one in which R is to be loaded from some functional unit (*e.g.*, the Floating Point multiplier) that takes several cycles to produce the result. These gaps in the pipeline can be further reduced if the compiler reorders instructions such that instructions consuming a datum are put as far as possible from instructions producing that datum. Thus, we notice that machines designed for high pipelining of instructions can hide large memory latencies provided there is local parallelism among instructions[4].

Some LOAD/STORE architectures have eliminated the need for reservation bits on registers by making the compiler responsible for not scheduling instructions until the time when the result is supposed to be available. The compiler performs this static hazard resolution by assuming deterministic time for each operation (*e.g.*, ADD, LOAD) and inserting NO-OP instructions wherever necessary. Because the instruction execution times are intimately built into the code, *any* change to the machine's structure (scaling, redesign) will at the very least require changes to the compiler and regeneration of the code. This is obviously contrary to our notion of generality, and hinders portability of software from one generation of machine to the next.

Current LOAD/STORE architectures assume that memory references either take a fixed amount of time (1 cycle in most RISC machines) or that they take a variable but predictable amount of time (as in the Cray-1). In RISC machines, this time is derived on the basis of a cache hit. If the operand is found to be missing from the cache, the pipeline stops. Equivalently, one can think of this as a situation where a clock cycle is *stretched* to the time required. This solution works because, in most of these machines, there can be either one or a very small number of memory references in progress

---

[3] Indeed, in the Cray-1, functional units can accept an input every clock cycle and registers are always read in one clock cycle after an instruction is dispatched from the Decoder.

[4] The ability to reorder two instructions essentially means that these instructions can be executed in parallel.

at any given time. For example, in the Cray-1, no more than 4 independent addresses can be generated in any memory cycle. If an address causing a bank conflict is generated, then the pipeline must be stopped. However, any conflict will be cleared in at most 3 cycles.

2.2.3. Summary

Good implementations of LOAD/STORE architectures can effectively pipeline many instructions at a time. Even though instructions are decoded in order, they may finish out of order. If there is sufficient parallelism in the source code and the compiler is good at reordering instructions, latencies of memory accesses can be hidden behind useful ALU work.

Latency cost can be reduced by introducing a cheap synchronization mechanism: reservation bits on processor registers. However, the number of *names* available for synchronization, *i.e.*, the size of the task's processor-bound context, is precisely the number of registers, and this restricts the amount of parallelism that can be exploited[5]. In order to understand this issue better, consider the case when the compiler decides to use register R to hold two different values at two different instructions say, n and n'. This will require n and n' to be executed sequentially while no such order may have been necessary in the source code. It would seem that more programmable registers in the architecture will provide more *names* for synchronization and, hence, a greater opportunity for tolerating latency. The obvious disadvantage of relying on this scheme is clear: the machine's ability to manipulate such names is bound tightly into the instruction set and thereby limits scalability.

It would be desirable if all the techniques developed for uniprocessors carried over directly to multiprocessor architecture. In fact, they do not. It was mentioned earlier that memory latency in a multiprocessor is going to be larger and less predictable than in a uniprocessor system, and additionally that multiprocessor systems must support some mechanism for the synchronization of computational tasks. The method of synchronization used for reducing latency is usually not used for synchronization at the programming level. In fact, we will show that low cost synchronization at the programming level calls for small, instead of large, processor state. In the following sections we discuss the methods of reducing the *induced processor idle time* due to latency and synchronization that have been either implemented or suggested for multiprocessor systems.

2.3. Latency Reduction Methods and their Cost

2.3.1. Caches

Let us assume that all memory modules in a multiprocessor form one global address space and that any processor can read any word in the global address space. This immediately brings up a number of problems:

- The time to fetch an operand may not be constant because some memories may be "closer" than others in the physical organization of the machine.

---

[5] It is interesting to observe in passing that all high performance machines seem to be based on general register architectures rather than on stack architectures. One possible explanation is that, given the same amount of fast storage, a stack machine has far fewer names available to uniquely identify synchronization events and is therefore poorer at trading synchronization cost for latency cost.

- No useful bound on the worst case time to fetch an operand may be possible at machine design time because of the scalability assumption.

- If a processor were to issue several (pipelined) memory requests to different remote memory modules, the responses may arrive out of order.

Some multiprocessors, *e.g.*, Cm* [15] have avoided rather than solved these problems by making the simplifying assumption that a memory request would not be issued until the previous response had been received. Not surprisingly, even running a program with tremendous parallelism, Cm* researchers discovered that the problem took longer to complete if more than 8 to 10 processors were used [15]. We think a likely reason is that *processor idle time* induced by the increase in memory latency could not be recovered by additional processing power.

LOAD/STORE architectures can solve the problems mentioned above to a limited degree. However, a general solution for accepting out of order memory responses requires a synchronization mechanism to match responses with the destination registers (*names* in the task's context) and the instructions waiting on that register. The Denelcor HEP [21] is one of the very few architectures which has tried to provide such mechanisms in the von Neumann framework. However, the architecture of the HEP is sufficiently different from von Neumann architectures as to warrant a separate discussion (see section 4).

The most popular way of *circumventing* the latency problem is to employ a local cache in a processor to keep the contents of the most recently used memory locations. This highly successful idea from uniprocessors suffers a great deal in the multiprocessor setting due to a problem called *cache coherence*. Censier and Feautrier [8] define the problem as follows: *"A memory scheme is coherent if the value returned on a* LOAD *instruction is always the value given by the latest* STORE *instruction with the same address."* In a multiprocessor context, it is easy to see that this may lead to difficulties.

Suppose we have a two-processor system tightly coupled through a single main memory. Each processor has its own cache, to which it has exclusive access. Suppose further that two tasks are running, one on each processor, and we know that the tasks are designed to communicate through one or more shared memory cells. In the absence of caches, this scheme can be made to work. However, if it so happens that the shared address is present in both caches, the individual processors can read and write the address and *never* see any changes caused by the other processor. Using a store-through design instead of a store-in design does not solve the problem either. What is logically required is a mechanism which, upon the occurrence of a STORE to location x, invalidates all other cached copies of location x wherever they may occur, and guarantees that subsequent LOADs will get the most recent (cached) value. This can incur significant overhead in terms of decreased memory bandwidth. All solutions to the cache coherence problem center around reducing the cost of detecting (or rather avoiding) the possibility of cache incoherence, and such solutions seem to work only for bus oriented machines. Some of these are discussed next.

There have been many proposals for solving the coherence problem by using a *logically* centralized directory for all cached data. Each entry reflects the state (*e.g.*, private, shared, etc.) of the associated cache quantum, and is responsible for guaranteeing that coherence is not violated. Implementations of this idea are generally intractable except possibly in the domain of bus oriented multiprocessors. Relying on the broadcasting capability of a bus, it is easy to see how all caches can

purge entry *x* if a processor attempts a STORE to *x* (the so-called *snoopy bus*).

In such a system at most one STORE operation can go on at a time in the whole system and, therefore, system performance is going to be a strong function of the bus's ability to handle the coherence-maintaining traffic. It is possible to improve upon this solution if some more information is kept with each cache entry. Suppose entries are marked "shared" or "non-shared". A processor can freely read shared entries but an attempt to STORE into a shared entry immediately causes that address to appear on the snoopy bus. That entry is then deleted from all the other caches and is marked "non-shared" in the processor that had attempted the STORE. Similar action takes place when the word to be written is missing from the cache. Of course, the main memory must be updated before purging the private copy from any cache. When the word to be read is missing from the cache, the snoopy bus may have to first reclaim the copy privately held by some other cache before giving it to the requesting cache. The status of such an entry will be marked as shared in both caches. The advantage of keeping shared/non-shared information with every cache entry is that the snoopy bus comes into action only on cache misses and STOREs to shared locations, as opposed to all LOADs and STOREs. Even if these solutions work satisfactorily, bus oriented multiprocessors are not of much interest to us because of their obvious limitations in scaling.

As far as we can tell there are no known solutions to cache coherence for non bussed machines. It would seem reasonable that one needs to make caches partially visible to the programmer by allowing him to mark data (actually addresses) as shared or not shared. In addition, instructions to flush an entry or a block of entries from a cache have to be provided. Cache management on such machines is possible only if the concept of shared data is well integrated in the high level language or the programming model. Though it may not be obvious, often a direct trade off exists between decreasing the parallelism and increasing the cachable or non shared data. Schemes have also been proposed to explicitly interlock a location for writing or to bypass the cache (and flush it if necessary) on a STORE: in either case, the performance goes down rapidly as the machine is scaled. Ironically, in solving the latency problem via multiple caches, we have introduced the synchronization problem of keeping caches coherent.

2.3.2. Pipelined Memory Systems

One can observe from the Cray-1 and other machines that there is an asymmetry between a heavily pipelined processor and a non-pipelined memory system. Memory systems continue to be slow relative to processors built with comparable technologies, and thus, are usually the performance limiting factor. Interleaving as a technique for reducing apparent access time is unsuitable as a general solution because of sensitivities to addressing patterns.

We have done some initial investigations over the past year into the architecture of pipelined memory systems aimed at solving this problem. If memory systems were designed to accept memory references in a pipelined manner with a large capacity to hold memory requests, stretching of the clock cycle as described in section 2.2.2 can be avoided. To exploit a pipelined memory in its full generality requires a mechanism such as reservation bits to provide fine-grained synchronization rather than relying on a rather complex compiler to predict machine configuration dependent data arrival times. The benefits, however, should be clear.

2.4. Synchronization Methods and their Cost

We turn from our very hardware-intensive view of multiprocessors to the issues of programming a multiprocessor. From this, we will reason about architectural implications of software systems on the underlying hardware. A general model of parallel programming must assume that tasks are created dynamically during a computation, and die after having produced and consumed data. Situations in parallel programming which require task synchronization include the following non-orthogonal basic operations:

1. *Forks* and *Joins*: The *join* operation forces a synchronization event indicating that two tasks which had been started earlier by some *forking* operation, have in fact completed.

2. *Producer-Consumer*: A task produces a data structure that is read by another task. If producer and consumer tasks are executed in parallel, synchronization is needed to avoid the *read-before-write* race.

3. *Mutual Exclusion*: Non-deterministic events which must be processed one at a time, *e.g.*, serialization in the use of a resource.

In order to understand the effect of hardware synchronization on software methodology, consider the case wherein the fixed cost of synchronization is high, say equivalent to the time taken by 10 ordinary instructions. Under such conditions it will not pay to exploit producer-consumer parallelism on an element-by-element basis. Rather, one would first produce n (>> 10) elements and then signal the consumer to start consuming. The same procedure would be repeated after producing the next n elements. This way, the cost of synchronization would be kept low by perhaps inducing some extra idle time on the processor on which the consumer task executes. The choice of n certainly depends on the machine and deeply affects how one would write code. If the elements are produced and consumed in somewhat irregular order, or if the data structure comprising the elements is nonuniform, it may be practically impossible to write code to exploit parallelism given certain types of synchronization mechanisms.

2.4.1. Global Scheduling on Synchronous machines

If a multiprocessor is totally synchronous, then it is at least conceptually possible to prepare a master plan in which instructions for every moment on every processor are specified. An analogy can be made between programming such a multiprocessor and coding a horizontally microprogrammed machine. Recently there have been advances in compiling for such machines which have caused several machine proposals to appear [14, 27].

While these machines are able to resolve run-time sharing conflicts by moving them to compile time and are usually able to plan memory references and control transfers in advance of the need (*e.g.*, the delayed BRANCH), these machines suffer from their special-purpose nature. Except in the simplest of cases, compilers require "hints" from the programmer or, in some cases, rely on luck (and hardware interlocks) in doing the code generation. Clearly, these machines are not well suited to real-time computations which involve nondeterministic situations or computations requiring dynamic resource (*e.g.*, memory) management.

We believe that this technique is effective in its currently-realized context - special purpose computation with a small number (4 to 8) of processors, but the technique is not sufficiently general as to allow significant scaling up. Software problems associated with this approach will be

overwhelming far before the hardware cost of latency and synchronization plays any significant role in scaling such machines.

2.4.2. Interrupts and Low Level Context Switching

Almost all von Neumann machines have the capability of accepting and handling interrupts. Not surprisingly, multiprocessors based on such machines permit the use of inter-processor interrupts as a means for signalling events. However, interrupts are rather expensive because, in general, the processor state needs to be saved. The state-saving may be forced by the hardware as a direct consequence of allowing the interrupt to occur, or it may occur implicitly, *i.e.*, under the control of the programmer, via a single very complex instruction or a suite of less complex ones. Independent of *how* the state-saving happens, the only important thing to note is that each interrupt will generate a significant amount of traffic across the processor - memory interface.

In the previous discussion we concluded that larger processor states were good in that they provided a means for reducing memory latency cost. In trying to solve the problem of low cost synchronization we have now come across an interaction which, we believe, is more than just coincidental. Specifically, in very fast von Neumann processors, the "obvious" synchronization mechanism (interrupts) will only work well when the amount of processor state which must be saved is *very small*. Said another way, reducing the cost of synchronization by making interrupts cheap would generally entail increasing the cost of memory latency.

Uniprocessors such as the Xerox Alto [32], the Xerox Dorado [22], and the Symbolics 3600 family [24] have used a technique which may be called *microcode-level context switching* to allow sharing of the CPU resource by the I/O device adapters. This is accomplished by duplicating programmer-visible registers, *i.e.*, the processor state. Thus, in one microinstruction the processor can be switched to a new task without causing any memory references to save the processor state. This dramatically reduces the cost of processing certain types of events that cause frequent interrupts. As far as we know, nobody has adapted the idea of keeping multiple contexts in a multiprocessor setting (with the possible exception of the HEP, to be discussed in section 4) although it should reduce synchronization cost over processors which can store only a single context. It may be worth thinking about adopting this scheme to reduce the latency cost of a nonlocal memory reference as well.

The limitations of this approach are obvious. High performance processors may have a small programmer visible state, *i.e.*, the number of registers, but a much larger implicit state, *i.e.*, caches. Low level task switching does not necessarily take care of the overhead of flushing caches[6]. Further, one can only have a small number of independent contexts without completely overshadowing the cost of ALU hardware[7]. This technique if employed in the large would reduce the synchronization cost only at the expense of latency cost.

---

[6] Multicontext caches and address translation buffers have been used to advantage in reducing task switching overhead, *e.g.*, the STO stack mechanism in the IBM 370/168.

[7] The Berkeley RISC idea of providing "register windows" to speed up procedure calls is very similar to multiple contexts.

2.4.3. Semaphores and the Ultracomputer

Next to interrupts, the most commonly supported feature for synchronization is some *atomic operation* to test and set the value of a memory location. A processor can signal another processor by writing into a location which the other processor keeps reading to sense a change. Even though, theoretically, it is possible to do such synchronization with ordinary read and write memory operations, the programming is much simpler with an atomic TEST-AND-SET instruction. TEST-AND-SET is powerful enough to implement all types of synchronization paradigms mentioned earlier. However, the synchronization cost of using such an instruction can be very high. Essentially, the processor that executes it goes into a *busy-wait* cycle. Not only does the processor get blocked, it also generates extra memory references at every instruction cycle until the TEST-AND-SET instruction is executed successfully. Implementations of TEST-AND-SET that permit non busy waiting imply context switching in the processor and thus are not necessarily cheap either.

It is possible to improve upon the TEST-AND-SET instruction in a multiprocessor setting as has been suggested by the NYU Ultracomputer group [13]. Their technique can be illustrated by the atomic FETCH-AND-ADD instruction (sometimes called REPLACE-ADD). The instruction requires an address and a value, and works as follows: suppose two processors, i and j, simultaneously execute FETCH-AND-ADD instructions with arguments (A,v_i) and (A,v_j) respectively. After one instruction cycle, the contents of A will become $(A)+v_i+v_j$. Processors i and j will receive, respectively, either (A) and $(A)+v_i$, or $(A)+v_j$ and (A) as results. Indeterminacy is a direct consequence of the race to update memory cell A. The implementation of FETCH-AND-ADD calls for a *combining* packet communication network which connects n processors to an n-port memory. If two packets collide, say FETCH-AND-ADD(A,x) and FETCH-AND-ADD(A,y), the switch extracts the values x and y, forms a new packet (FETCH-AND-ADD$(A,x+y)$), forwards it to the memory, and stores the value of x temporarily. When the memory returns the old value of location A, the switch returns two values ((A) and $(A)+x$). The main improvement is that some synchronization situations which would have taken O(n) time can be done in O(log n) time. It should be noted, however, that one memory reference may involve as many as $\log_2 n$ additions, and implies substantial hardware complexity. Further, the issue of processor latency has not been addressed at all. In the worst case, the complexity of hardware may actually increase the latency of going through the switch and thus completely wipe out the advantage of FETCH-AND-ADD over its "non combining" version.

The simulation results reported by NYU [13] show quasi-linear speedup on the Ultracomputer (a shared memory machine with ordinary von Neumann processors, employing FETCH-AND-ADD synchronization) for a large variety of scientific applications. We are not sure how to interpret these results without knowing many more details of their simulation model. Two possible interpretations are the following:

1. Parallel branches (*i.e.*, tasks) of a computation hardly share any data, thus the costly *mutual exclusion* synchronization is needed rarely in real applications.

2. The synchronization cost of using shared data can be acceptably brought down by judicious use of cachable/non cachable annotations in the source program.

If true, these interpretations would not invalidate the analysis presented in this paper; the losses due to latency and synchronization still impose fundamental limits. Rather, it would show that it possible to build larger high performance von Neumann multiprocessors than what is implied here.

2.5. Lessons Learned thus Far

In order to reduce memory latency cost, it is essential that a processor be capable of issuing multiple, overlapped memory requests. To effectively deal with this we must view the memory / communication subsystems as a logical pipeline. As latency increases, keeping the pipeline full implies that more memory references will have to be in the pipeline. We note that memory systems of current von Neumann architectures have very little capability for pipelining.

Even with pipelined memory systems, von Neumann processors must observe instruction sequencing constraints. Adding hardware support for synchronization is difficult because it interacts directly with instruction decoding and dispatching. The decoding becomes encumbered whenever a BRANCH instruction is encountered, while dispatching is suspended when the hardware synchronization mechanism detects a resource conflict between instructions being executed and the instruction about to be dispatched.

At the architectural level, the only "handle" the programmer has on memory reference synchronization is the number of separate synchronization points (registers) which he can name. Typically, the number of programmer visible registers is small, thereby limiting the number of outstanding memory references, regardless of the amount of parallelism in the source program. Providing more synchronizing registers is a superficially appealing idea. This provides a way of dealing with long latencies and out of order memory responses[8]. The difficulty arises when one wishes to share the processor across multiple tasks (*e.g.*, multiprogramming, interrupts) because context swapping incurs a heavy penalty.

3. The Dataflow Approach

In this section we present a machine structure which theoretically, given sufficient parallelism in the program, can show high performance in the presence of extremely large memory latencies and waits for synchronization events. Dataflow processors do not have any notion of a program counter and offer ultimate flexibility in issuing overlapped memory requests. The execution of instructions in dataflow computers is triggered solely by the availability of the operands. Dataflow architectures are sufficiently different from von Neumann architectures that, without a discussion of dataflow program representation, the instruction execution mechanism is difficult to understand and evaluate.

3.1. Dataflow graphs

A dataflow graph is a directed graph whose nodes correspond to dataflow machine instructions, and whose arcs correspond to the data dependencies between the instructions. The implication is, quite simply, that instructions which depend on other instructions should be sequenced accordingly; but where no dependence (arc) exists, instructions can be executed in parallel. Whatever may be the technique of actual implementation, it is often convenient to think in terms of a dataflow instruction sending operand values, or *tokens*, to instructions connected to it by outgoing arcs. An instruction is said to be ready to execute, or is *enabled*, when all the required input

---

[8] In essence this amounts to solving a latency problem by introducing a mechanism which, in turn, requires solution of a low level synchronization problem.

operands are present.

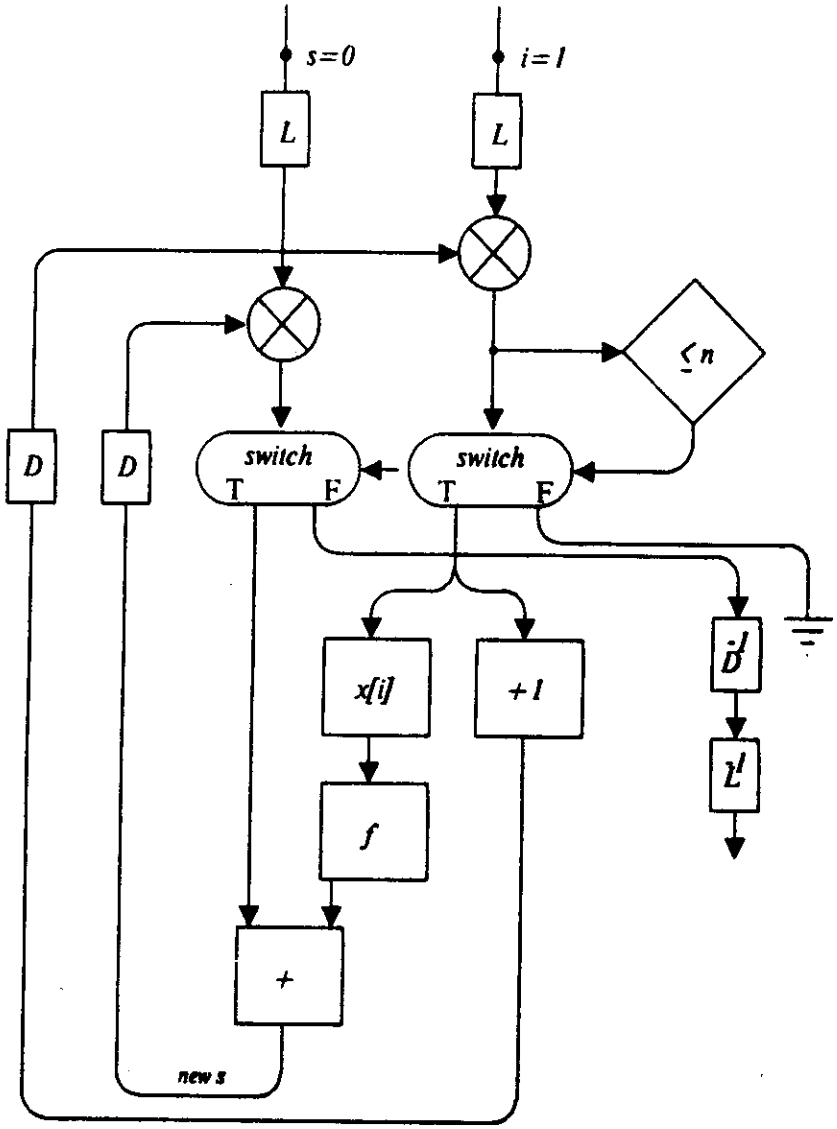

Figure 3-1: Compilation of the Loop Expression for $\Sigma f(x_i)$

An example of a dataflow program graph is shown in Figure 3-1. This graph was compiled from the following Id[9] [4] program which applies function *f* to each element of array *x* and sums up the results thus obtained:

```
(initial s ← 0
 for i from 1 to n do
    new s ← s + f(x[i])
 return s)
```

[9] Id is a high-level, functional language designed specially for dataflow machines.

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} all act as identity operators on the input values but are essential for manipulating context-identifying information (discussed later). For the time being, let us assume that the box containing *x[i]* somehow produces a token containing the value at the i^{th} selector in array *x*. The remainder of the operators are arithmetic, relational, and conditional instructions whose function should be self-evident.

A dataflow processor conceptually moves tokens along the arcs of the graph (duplicating them upon encountering a fork), looking for instructions which have become *enabled*. Upon execution of an enabled instruction, the input tokens are absorbed, and output tokens for the following instructions in the graph are produced. A program is said to *terminate* when no enabled instructions are left.

3.2. Static Allocation of Storage for Operands

Dataflow machines can be broadly classified on the basis of the method used for allocation of token storage. *Static* dataflow machines allocate storage for operands along with the nodes of the graph. An early version of Dennis' machine [11], which is the basis for all static dataflow machines, is shown in Figure 3-2. The program, along with the data, resides in the Activity Store, and a bit associated with each operand is used to indicate whether the operand is present or not. Tokens produced by the Operation Units carry destination pointers which are simply addresses in the Activity Store. Just before a token is stored, a check is made to see if its partner is available. If so, the instruction is ready for execution. A packet containing the opcode, the operands, and the list of destinations is formed and forwarded to a non-busy Operation Unit. After this, the corresponding operand slots are marked *empty*. The Operation Unit produces results and forms a token for each destination in the list; the Update Unit delivers these tokens to destination instructions, potentially enabling them for execution. Interestingly, correctness and determinacy in program behavior are independent of the time order in which enabled instructions are executed.

This simple instruction execution mechanism can exploit parallelism in two ways as discussed in [11]. First, the processor can be heavily pipelined because the operations of Operation Units, Activity Store and Update/Fetch Units can all be overlapped easily. Second, and more importantly, many processors can be connected together to work on different parts of a dataflow graph. As long as all the Activity Stores are part of one address space, the Output Unit can easily deduce the destination processor number for a token from the instruction address on the token.

The simplicity of connecting several dataflow processors stems directly from the fact that the dataflow processor does not treat internal and external tokens differently [30]. Though for static machines, the compiler must decide which part of the graph should be loaded on which processor, the decision can be straightforward for programs with massive parallelism. Notice that increased latencies in the communication system do not necessarily affect the performance provided there are sufficient concurrently enabled nodes in the graph to keep the pipelines between Activity Stores and Operation Units full.

The differences between the dataflow execution mechanism just described and the von Neumann machines discussed in section 2 are worth noting. There are no registers, not even program

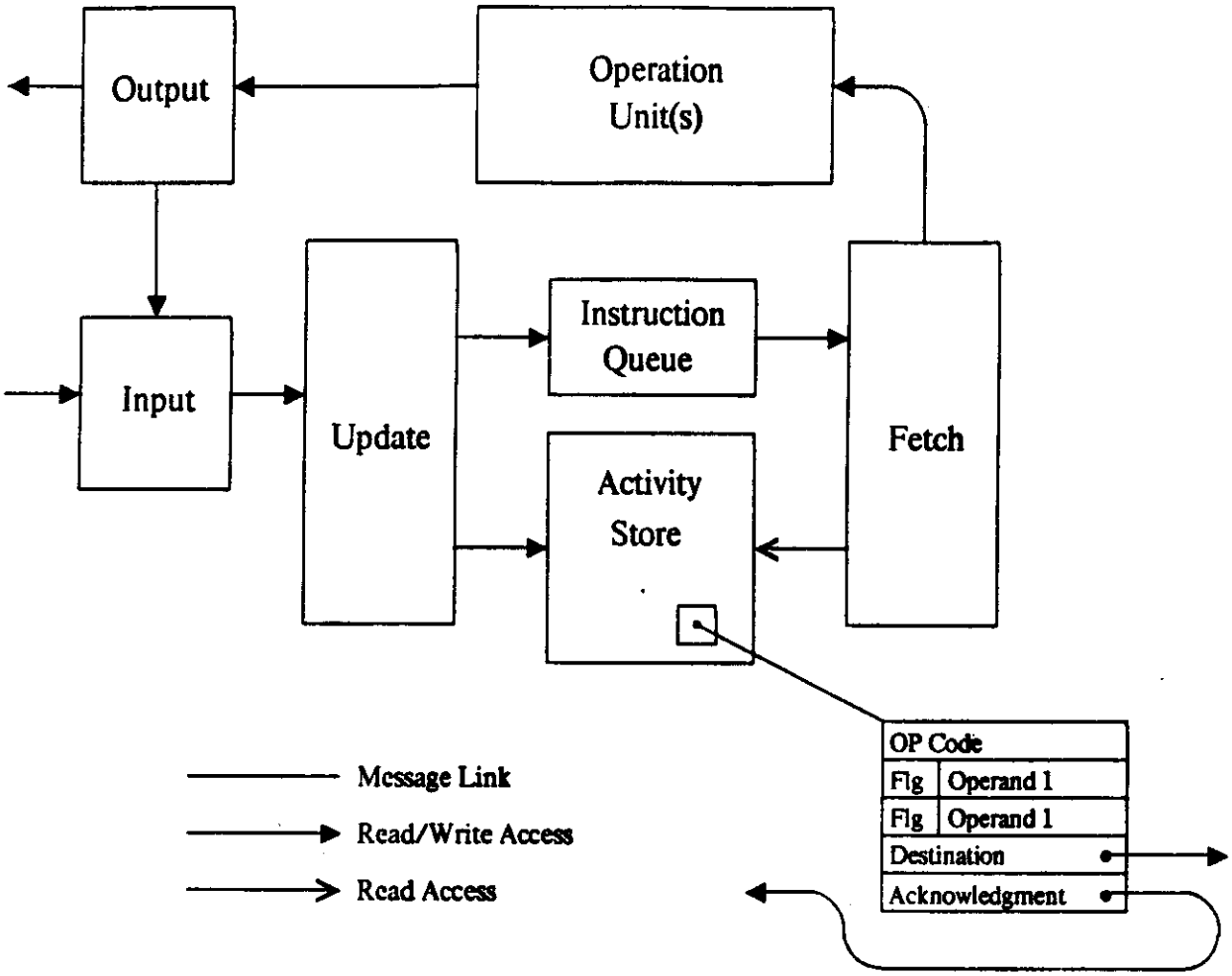

Figure 3-2: One Processing Element of the Static Dataflow Machine (adapted from [11])

counters in the dataflow machine[10]. Instructions which are waiting for operands in the dataflow machine do not block the instruction pipeline in any way. It is this aspect of the dataflow machine which makes it possible to trade program parallelism to reduce latency cost.

Static allocation of operand storage in dataflow machines has some weaknesses too. As explained below, it unnecessarily restricts the kind of parallelism that can be exploited in a program, and makes dynamic invocation of procedures difficult. Since an instruction has space for exactly one set of operands, concurrent enabling of the same instruction with multiple sets of input operands is ruled out. It is easy to show that, at the graph level, many loop programs can produce several inputs for a node (*e.g.*, the program in Figure 3-1). Indeed, such programs can produce erroneous results

[10] One may view the Activity Store as nothing but registers with reservation bits, *a la* Cray. However, unlike Cray machines, the instruction pipeline can not be held up after instruction decoding because only enabled instructions are decoded.

on the static machine if special care is not exercised[11] [23]. Generally, this problem is avoided by introducing an arc, pointing in the opposite direction, corresponding to each arc in the dataflow graph. These so called *acknowledgment* arcs prevent the enabling of the source node until the destination node has an empty slot for the operand (see Figure 3-2). This solution complicates the detection of enabled instructions because all the acknowledgment tokens have to be counted, and it also doubles the token traffic. Further, it increases the time between two firings of an instruction from one delay around the pipeline to two. Implementation of procedures is also rather restricted because static allocation views procedures as "in line functions" or macros; not all procedures can be viewed in this manner.

It is possible to generalize the concept of static dataflow machines by providing support for dynamic loading of a procedure in the Activity Store. This requires mechanisms for directing tokens from the calling procedure to the called procedure, and also a mechanism for returning results. Rather then speculate about these mechanisms we present a more general dataflow model with the hope that it will be straightforward to deduce what mechanisms from it need to be incorporated in the static allocation model to support procedures.

3.3. The Tagged-Token Dataflow Processor

3.3.1. Dynamic Allocation of Operand Store

It is possible to exploit more parallelism in dataflow graphs than Dennis' static machine does. Two groups, one at Manchester, England [18] and the other at University of California, Irvine [5, 17] independently developed the idea of labeling tokens in Dennis' dataflow graphs [10] to achieve this effect. The labels are called *tags*. Tagged-Token dataflow processors allow more than one token to be present on an arc. Therefore, the destination labels in tokens contain some dynamic, or context-sensitive information in addition to the address of the next instruction.

The basic structure of a token is the quadruple: <DATA, TAG, ARITY, OPERAND NUMBER>. The TAG itself is a triple: <CONTEXT, INSTRUCTION POINTER, SEQUENCE>. The CONTEXT and SEQUENCE fields contain dynamic information - the INSTRUCTION POINTER is statically determined. The CONTEXT field identifies the procedure invocation to which this token belongs. Even though the TAGs and, hence, the CONTEXTs, are reused, the CONTEXT is guaranteed to be unique during the lifetime of the procedure invocation.

The D and L operators can now be explained. The purpose of the D operator is to increment the SEQUENCE part of the TAG of each token which passes through it. One can think of this as giving new labels to the tokens associated with different iterations of a loop[12]. D^{-1} resets the SEQUENCE field to zero. The L operator is responsible for logically saving the CONTEXT on incoming tokens *and* generating a new CONTEXT to be substituted for the old one on outgoing tokens. It does so

---

[11] The graph in Figure 3-1 may still produce the correct answer because + is associative and commutative.

[12] Such a mechanism is obviously necessary because we permit more than one token on an arc; unconstrained, cycles in a graph would give rise to ambiguous matchings of tokens and nondeterministic behavior. If one considers only acyclic graphs, then the SEQUENCE part of the TAG is useless. However, it can be viewed as providing a cheap way of allocating and deallocating new contexts for tail recursive programs.

with the help of a resource manager who keeps track of "contexts" in use. L^{-1} reverses the process by substituting the old CONTEXT for the new one on outgoing tokens. The details of context creation are further discussed in section 3.3.3.

Since instructions may have more than one input operand, two more pieces of information are included on each token. These are the *total* number of operands required by its target instruction, that is, its ARITY, and a value to specify which OPERAND NUMBER this token represents.

Unlike the static dataflow machine, tokens (operands) are stored separately from the program in the Tagged-Token architecture. Conceptually, the operand store for the Tagged-Token machine is organized as an association list from which tokens can be extracted by presenting a TAG. The program store, then, contains no data, only pure code. Aside from this difference, instructions in both the static and the tagged-token machines are very similar.

A diagram of the the Tagged-Token dataflow processor is shown in Figure 3-3. Assume that the dataflow graph corresponding to the program to be executed has been stored in the Program Memory. Let us further assume that all instructions require exactly two input operands. The Waiting Matching Section performs the function of the operand store. The TAG of the token entering the processor is compared against the TAGs of the tokens already in the Waiting Matching Section. If a token with the matching TAG is found, the data value on the matched token, the data value on the incoming token, and their (identical) TAG are forwarded to the Instruction Fetch Section. If no match is found, the incoming token is stored in the Waiting Matching Section[13]. The opcode is looked up in the Instruction Fetch Section based on the TAG's CONTEXT and INSTRUCTION POINTER fields. Given the two operands and the opcode, the ALU performs the indicated operation.

The Compute Tag Section derives the new TAG using the old TAG and the DESTINATION information stored with the current instruction. If there is more than one destination, the process is repeated. A token is then formed by appending the new TAG to the ALU's output. At this point, a determination is made based on the new token's logical destination. If the denoted instruction resides on the same processor as the one in which it was created, it is routed back to the Waiting Matching Section. If not, it is passed to a Communication Network which is responsible for delivering the token to the correct processor.

3.3.2. Executing a Procedure on Several Processors

Tagged-Token machines provide a degree of freedom in distributed execution of a procedure not possible in static machines[14]. Like the static machine, it is still possible to divide the dataflow graph of a procedure into several parts and to load these parts into the Program Memories of different processors. Again, if all Program Memories are part of the same global address space, tokens may

---

[13] Thus, storage for operands is allocated dynamically from the common pool of storage available in the Waiting Matching Section.

[14] From now on, our remarks about *Tagged-Token Machines* apply specifically to the MIT Tagged-Token Dataflow project and may not be true for other similar machines (*e.g.*, Manchester [18] and Sigma-1 at the Electrotechnical Laboratory, Japan [33]). In all cases, we can envision integrating these ideas into other Tagged-Token machines without much difficulty, however.

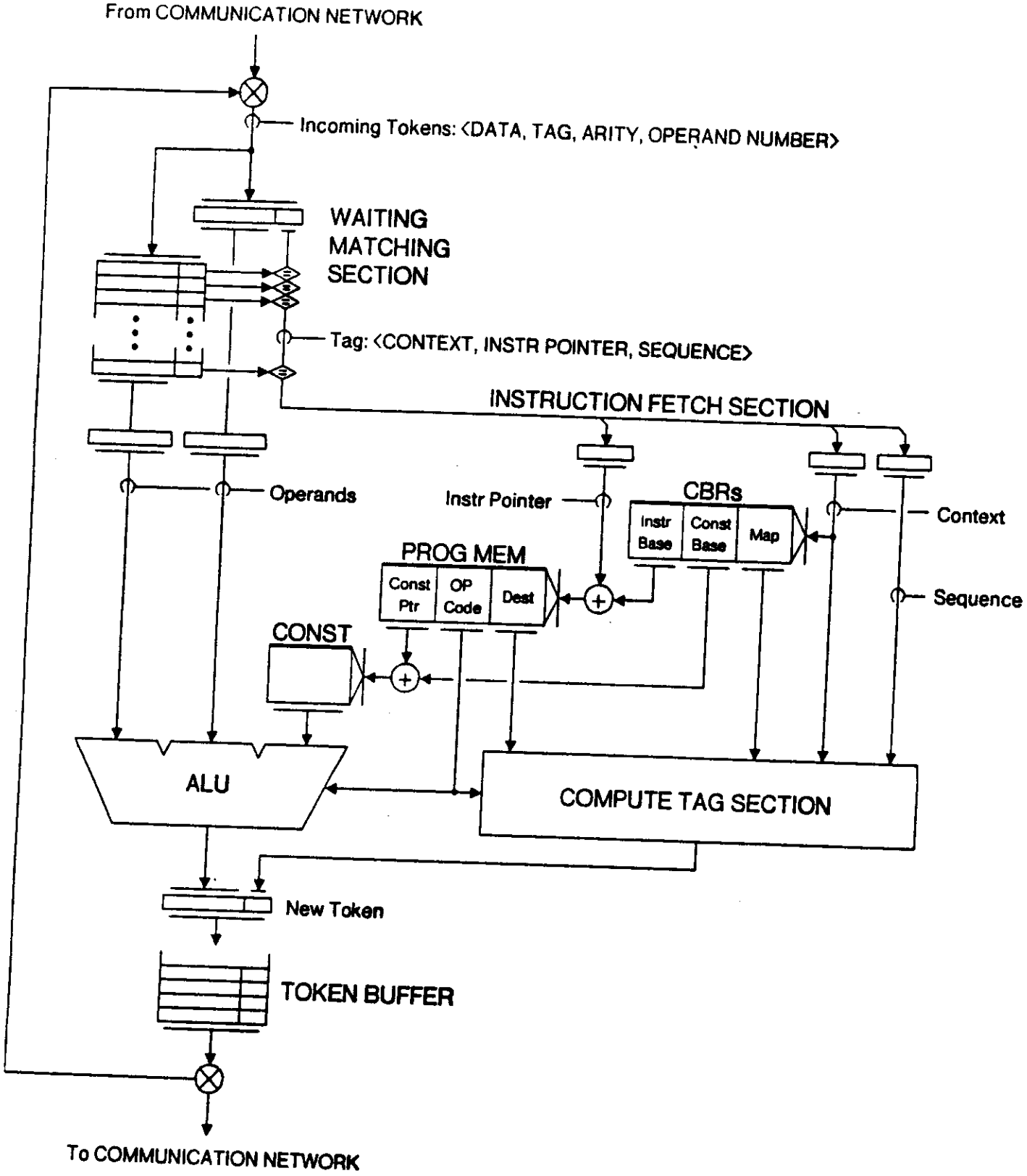

Figure 3-3: Block Diagram of a Tagged-Token Dataflow Processor

be distributed according to their PROCESSOR NUMBER which is trivially derived from the INSTRUCTION POINTER (recall that the INSTRUCTION POINTER is static information). However, it is also possible to load a *complete* copy of the program's graph into each of several processors and to divide up tokens amongst these processors based on the *dynamic* information in the TAG. For example, imagine that a program is loaded onto each of two processors. We can distribute all tokens with *odd* SEQUENCE fields to the first of these processors. Similarly, tokens carrying *even* SEQUENCE fields can be sent to the second processor. To the extent that we can arrange for a uniform distribution of even and odd SEQUENCE fields during the course of the computation, the work will be distributed equitably between these two processors.

The MIT Tagged-Token architecture supports a fairly general program mapping and load distribution scheme along these lines. Each procedure invocation is associated with a particular mapping scheme. It is this CONTEXT specific MAPPING information which is used by the Compute Tag Section (Figure 3-3) in generating output TAGs. A detailed discussion of this mechanism is, however, beyond the scope of this paper and has been described elsewhere [3].

3.3.3. Creating Contexts for Procedure Invocations

We focus our attention now on the mechanisms provided to support creation or allocation of contexts for procedure invocations. The key, of course, is the CONTEXT field carried with each token. Referring again to Figure 3-3, the CONTEXT field of the token's TAG is used to select a CODE BLOCK REGISTER (CBR), one of several key resources allocated at procedure invocation time. Its contents are used to access (1) the program, (2) invocation specific constants, and (3) invocation specific mapping information.

Dataflow graphs are compiled into collections of instructions called *code blocks*. These are dynamically loaded into (and subsequently deleted from) Program Memory, and are accessed *indirectly* via the INSTRUCTION BASE POINTER in the CBR, thus facilitating relocation.

Given that two procedure invocations *do not* share CBR's, the name of the allocated CBR serves the purpose of uniquely identifying a procedure invocation. Both the CONTEXT (the name set) and the CBR (the physical resource) can be reused when the corresponding procedure invocation terminates. Obviously, the number of procedure invocations that can exist simultaneously in our machine is limited by the total number of concurrent contexts the hardware can support.

Regardless of how much other memory is provided, if we run out of CBR's, *i.e.*, contexts, we will have to wait until one is freed up. If no context can be freed then the machine will deadlock. In sequential machines the closest thing to a CONTEXT is the *stack frame base pointer* for the procedure call stack. Frequently, compiler convention dictates that pointers to code and data for the procedure are stored in fixed positions in the stack frame. By analogy, running out of CONTEXTs in the Tagged-Token machine is like running out of procedure call stack memory in conventional machines (with similar results).

Can we imitate the procedure call stack on a parallel machine? In particular, can we take advantage of the size of the main memory to reduce the likelihood of deadlock due to exhaustion of CONTEXTs, a fixed resource? It seems unlikely. If, for example, we assumed that all processors had fast access to a large shared memory we might be able to emulate the traditional solution. We believe that such an assumption is unrealistic, however.

A CONTEXT in our machine is not cheap. Even if the machine can support a large number of CONTEXTs, the time taken to initialize all of the context-specific registers is significant enough to discourage the use of small procedures. Also, in a balanced design, contexts have to be backed up by enough other resources such as program memory and token storage space in the waiting matching section [9]. From a resource management point of view it would be undoubtedly easier if all resources were to come from a single common pool of resources, *e.g.*, the memory.

3.4. Data Structure Operations and I-Structures

3.4.1. Functional Operations on Data Structures

The dataflow model, as proposed by Dennis [10] allowed only *functional* operations on data structures. The two most common functional operations on data structures are SELECT (to select a specific element of a data structure) and APPEND (to generate a new data structure which differs from the input structure in at most one selector position). Conceptually, even a data structure is represented by a single token in a dataflow graph. In any reasonable implementation, however, a token cannot carry arbitrarily large amounts of information. Realistically, data structures reside in a large storage, often called Structure Storage; tokens carry only pointers to the structure. Thus, replicating a structured value token involves merely copying a pointer. With structure storage, a SELECT operation causes the relevant information to be fetched. Implementation of this operation is very different on dataflow machines than on von Neumann machines, and merits a description.

The box containing *x[i]* in the dataflow graph of Figure 3-1, in general, will have two inputs: a descriptor for the structure *x*, and the selector *i*. When such an instruction is enabled in the Static machine (Figure 3-2) a packet containing *x*, *i* and the addresses of the destination instructions (*i.e.*, the address of the box containing *f*) is sent to the Structure Storage. Though not shown in Figure 3-2, the Structure Storage can be imagined as just another functional unit, albeit one shared by all processors. It is very important to realize that, unlike a von Neumann machine, neither the ALUs nor the Instruction Fetch Sections get blocked during a fetch from the Structure Store in the dataflow machine. The output from the Structure Store gets directed to the proper instruction slots in the Activity Store. Very similar behavior can be imagined on the tagged dataflow machines also. Instead of destination instruction addresses, the destination TAGs are sent to the Structure Store, and its output is sent to the Waiting Matching Section.

In a manner similar to the SELECT operation, an enabled APPEND operation causes a packet containing the descriptor *x*, the selector *i*, the value *v*, and the destination addresses to be sent to the Structure Store. However, its effect there is rather complex. The complication stems from the fact that some APPENDs can cause a new copy of a part of the old data structure to be created. Such copying is often expensive. A detailed implementation of structure storage with a lot of internal concurrency is given by Ackerman [1]; a similar solution in the context of the U-interpreter is discussed in [17].

The problems with functional operations on data structures have been discussed extensively in the literature (see, for example, [5]). Besides the copying overhead in even very clever implementations, the structure-on-the-token model implies data structure constructors, *e.g.*, the APPEND operation, to be "strict". That is, the structure output is not produced unless all inputs of the APPEND are available. Consequently, no element of a structure can be used unless all elements

have been stored. The loss of parallelism under such conditions is obvious. I-structures have been proposed to reduce copying and increase parallelism in data structure operations.

3.4.2. I-Structures

From a programmer's perspective, an I-Structure is an array of slots. All slots are initially empty. A programmer is allowed to store into each slot no more than once. A slot can be read as many times as desired, and when there are no outstanding read or write operations for an I-Structure it can be deallocated. At the time the storage is reallocated, all the slots are marked empty again. A interesting aspect of I-Structure semantics is that even if the reading of a location precedes its writing, the value returned does not change. This, as we shall see, is accomplished by delaying the response until the slot is actually filled.

Referring to Figure 3-4, I-Structure storage can be visualized as a word-addressed conventional memory with the addition of a few *presence bits* on each word. Presence bits indicate that the associated word is in one of three possible states:

- PRESENT: The word contains valid data which may be freely read just as in a conventional memory.
- ABSENT: Nothing has been written into this word since the most recent reallocation. No attempt has been made to read the word.
- WAITING: Nothing has been written into this word since the most recent allocation, but at least one attempt to read the word has been made.

The bits change state in obvious ways: during a read operation, the I-Structure storage controller interrogates the presence bits associated with that location, and if the word is marked PRESENT, the contents are retrieved and forwarded to the destination instruction. If the word is ABSENT, the controller puts the read request aside and marks the empty location WAITING to indicate that a read request is outstanding[15]. If the word is already in the WAITING state, the new read request is appended to the existing list.

A write operation similarly interrogates the presence bits. If ABSENT, the datum is written as in a normal memory. if WAITING, the value is both written *and* forwarded to all the instructions on the deferred list. I-Structure semantics are violated if a write is attempted and the word is already marked PRESENT; an error will be signalled.

I-Structures provide the kind of synchronization needed for exploiting the producer-consumer type of parallelism. A programmer is completely freed from the burden of avoiding read-before-write races. The execution time overhead of using I-Structure operations is also minimal as long as most *read* requests precede the corresponding *write* requests. The issues involved with building such a memory, and the design for an I-Structure memory controller are discussed extensively in [19].

Many other issues are involved with the design of a Tagged-Token dataflow machine whose

---

[15] The idea of associating a status bit with each memory cell is not new - the Denelcor HEP multiprocessor [29, 21] uses this idea to synchronize cooperating parallel processes which share registers and/or memory cells. Unsatisfiable requests result in a busy-waiting condition - *i.e.*, there is no such thing as a *deferred read* list.

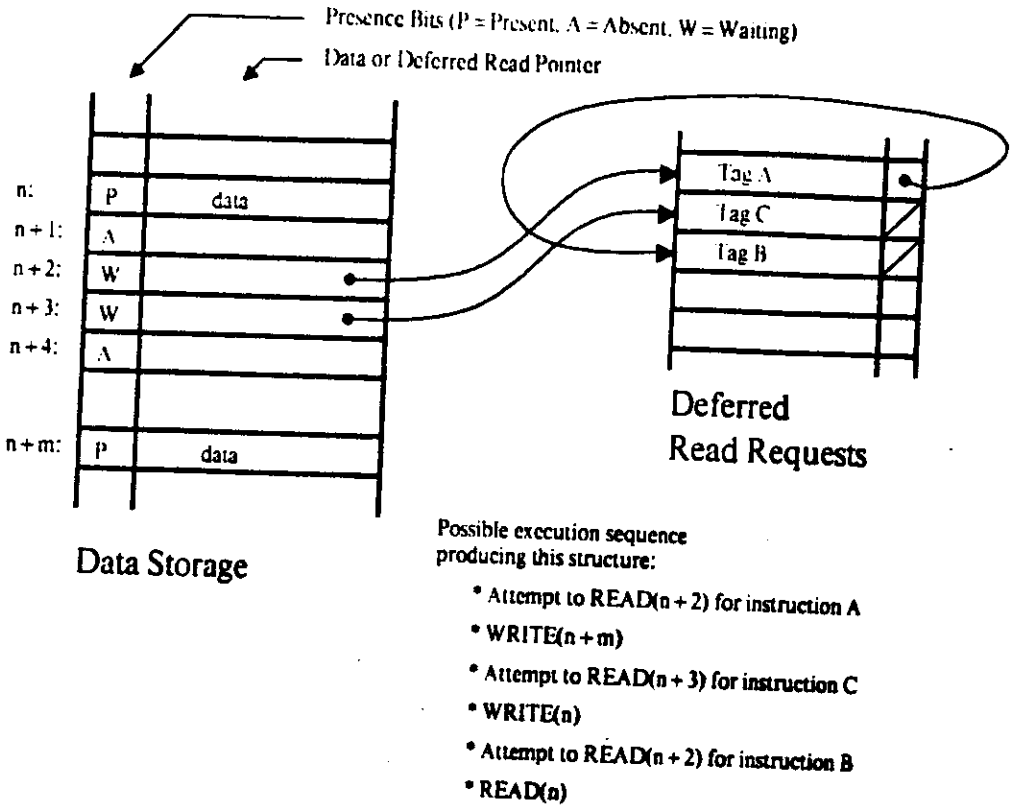

Figure 3-4: I-Structure Storage

details are shown in Figure 3-5). The interested reader will find more details in [3, 2]

3.5. Simultaneously Reducing Latency and Synchronization Costs

The M.I.T. Tagged-Token Dataflow machine exploits parallelism at both the procedure and the instruction level. It is possible that at any given moment several instructions belonging to a procedure may be active in a PE. It is also possible, even likely, that at the same moment instructions belonging to another procedure invocation may be active in the same PE. Instructions from two different procedure invocations automatically share the instruction pipeline and resources such as Waiting Matching Section. Thus, a lack of parallelism at the instruction level within a procedure does not necessarily induce gaps in the instruction pipeline, as it would in a von Neumann machine. The synchronization cost associated with one instruction enabling the next one can be totally absorbed as long as other enabled instructions are available. The hardware feature that makes this possible is the Waiting-Matching Section. It allows instructions to wait for a synchronization event without blocking the instruction pipeline. Further, unlike the register based synchronization in Load/Store architectures, it allocates the name (the TAG) required by a synchronization event at run time.

The latency in fetching an operand from the I-structure store can be significant. However, this can be also totally absorbed because the instruction waiting for the operand does not block the pipeline. Rather, it waits in the Waiting Matching Section. As long as there are instructions

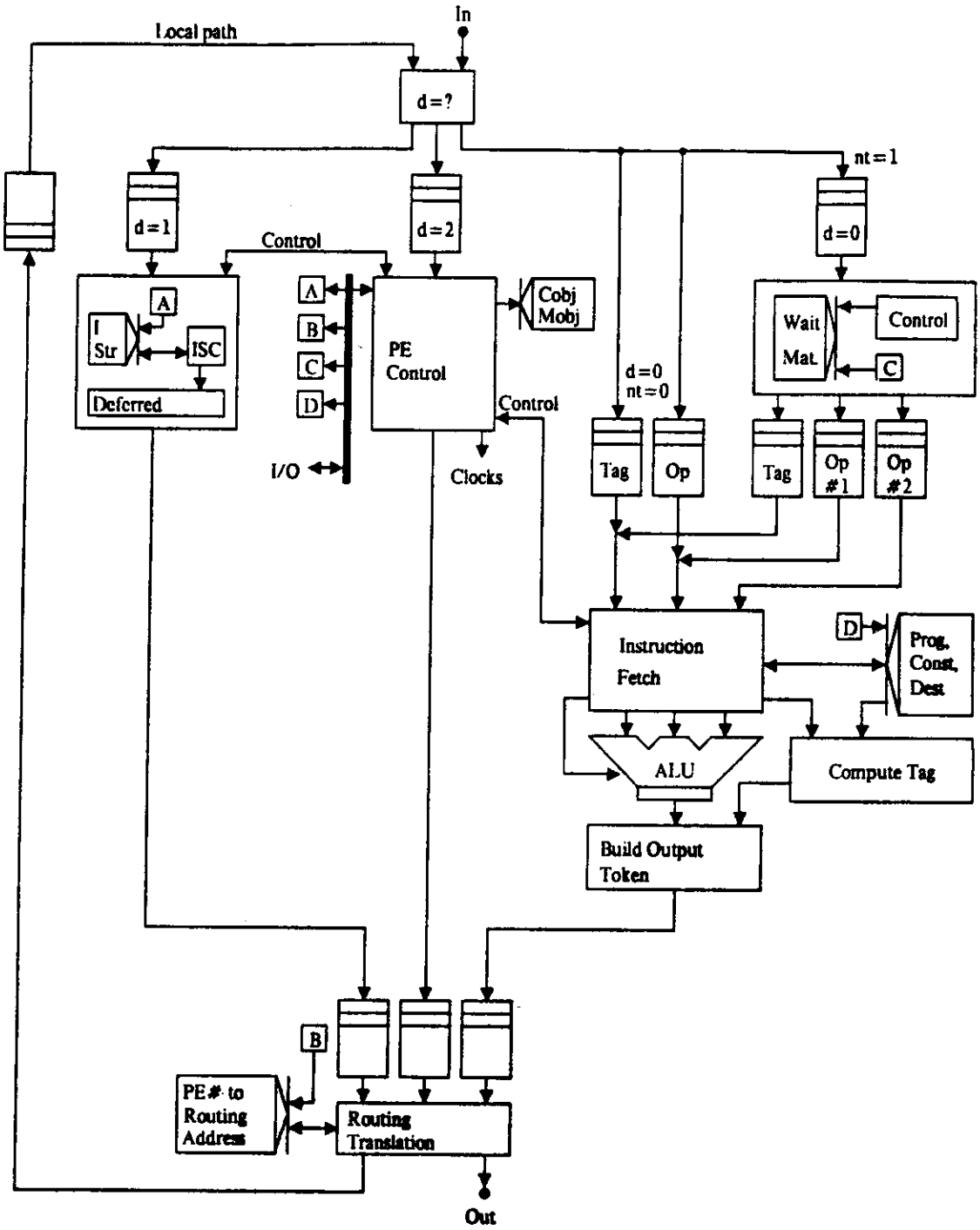

Figure 3-5: Detailed Diagram of the Tagged Token Processing Element

available to keep the pipelines full, no performance degradation should take place. Thus, the Waiting Matching Section plays a crucial role in simultaneously reducing the the latency and synchronization costs.

The role of I-Structure storage in reducing the synchronization cost is harder to explain because it is closely coupled with the manner in which parallelism is expressed. When cast as producer-consumer synchronization, it seems to us that I-structures provide a tremendous opportunity for overlapped execution as well as minimal synchronization overhead. If a read request must wait because the data is not yet available, a fixed penalty in the form of updating the "deferred readers list" is incurred. However, neither the instruction pipeline nor the memory pipeline is blocked by the waiting reader. Processors can keep issuing memory references, even without getting responses to a large number of earlier requests, as long as requests are independent of each other.

4. Future Evolution of Multiprocessors

We have shown that on multiprocessors based on von Neumann processors reducing the losses due to large memory latency results in increased losses due to synchronization waits. Our belief is that this coupling is not fundamental to all multiprocessor architectures, and in particular, dataflow architectures can simultaneously minimize latency and synchronization costs. This idea may be visualized in terms of a three dimensional latency-synchronization-efficiency (LSE) space, in which, specific architectures appear as points (see Figure 4-1). Our assertion about von Neumann machines (page 4) can be thought of as defining a surface in this space which we shall call the *von Neumann Barrier*[16] beyond which no von Neumann design can exist!

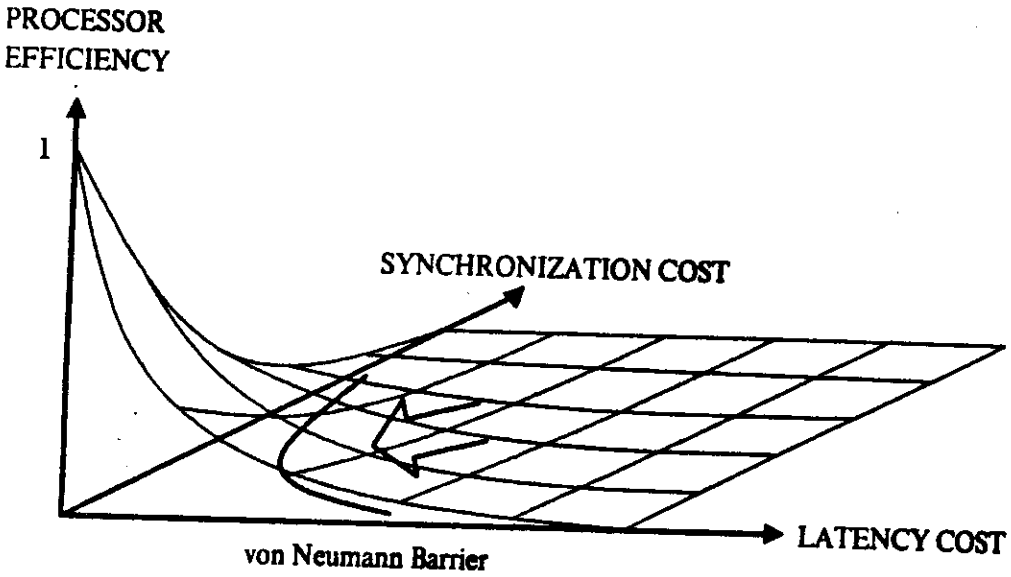

Figure 4-1: The Latency-Synchronization-Efficiency Space

[16] Not to be confused with the much publicized *von Neumann Bottleneck*.

4.1. The Denelcor HEP: A Hybrid Architecture

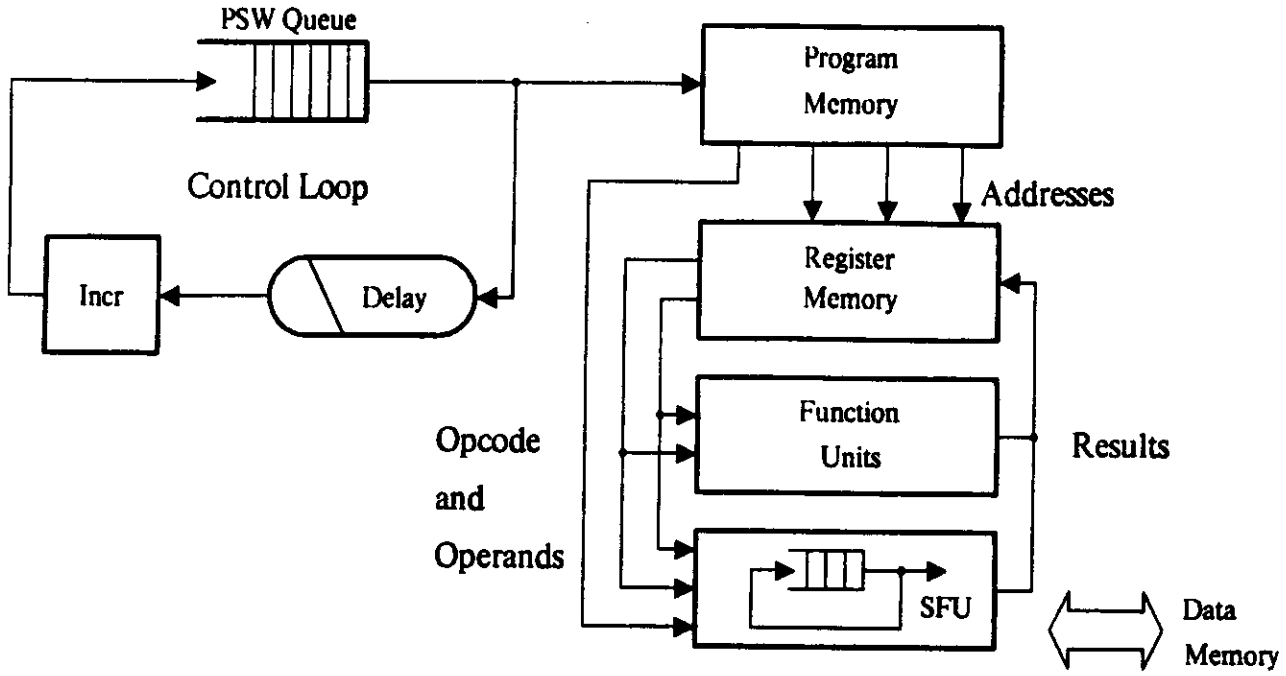

Figure 4-2: Latency Toleration and Synchronization in the HEP

We think there is a large space of possible machine structures other than *the Tagged-Token dataflow* which satisfy the condition set forth in this paper for scalable, general-purpose multiprocessors. One machine which, in some ways, intrigues us is the Denelcor HEP [29, 21]. The basic structure of the HEP processor is shown in Figure 4-2. The processor's data path is built as an eight step pipeline. In parallel with the data path is a control loop which circulates process status words (PSW's). The delay around the control loop is variable because of the queue, but is never shorter than eight pipe steps. This minimum value is intentional to allow the PSW at the head of the queue to initiate an instruction but not return again to the head of the queue until the instruction has completed. If at least eight PSW's, representing eight processes, can be kept in the queue, the processor's pipeline will remain full. This scheme is much like traditional pipelining of instructions, but with an important difference. The inter-instruction dependencies are likely to be weaker here because adjacent instructions in the pipe are always from *different processes.*

Where dependencies *between* instruction streams (*i.e.*, inter-process sharing) must be synchronized, the HEP provides FULL/EMPTY/RESERVED bits on each register and FULL/EMPTY bits on each word in the data memory. An instruction encountering EMPTY or RESERVED registers[17] is effectively NO-OPed: the corresponding PSW which initiated the instruction is not incremented. The result is that process will *busy-wait.*

---

[17] There are 2048 registers in each processor; each process has an index offset into the register array. Inter-process data sharing is possible at the register level via overlapping register allocations.

When a process issues a LOAD or STORE instruction, it is removed from the control loop and is queued separately in the Scheduler Function Unit (SFU) which also issues the memory request. Requests which are not satisfied because of improper FULL/EMPTY status result in recirculation of the PSW within the SFU's loop and also result in reissuance of the request. The SFU matches up memory responses with queued PSW's, updates registers as necessary, and reinserts the PSW's in the control loop.

Thus, the HEP is capable of trading parallelism, to a degree, for memory and communication latency while providing an efficient, low-level synchronization mechanism. Disadvantages of the HEP approach include the limit of *one* suspended memory request per process at a time and the cost of *busy-waiting* when sharing registers across processes.

4.2. Procedure Level Dataflow

A problem with the present Tagged-Token Dataflow architecture is that a dataflow program takes three to five times as many instructions as a comparable sequential program on a conventional von Neumann computer. This does not seem to affect the scalability or the programming generality of the dataflow machine, but it does imply that for comparable performance the dataflow machine may require substantially more hardware than a von Neumann machine. This has motivated us to look for a new and larger independently schedulable entity than a single dataflow instruction.

Several researchers have suggested that dataflow mechanisms should be employed at the procedure rather than at the instruction level, they all seem to have ignored the two fundamental issues discussed here. Suppose that we regard a procedure as the smallest schedulable entity, and assume that each procedure is compiled for some register-based von Neumann machine. A fundamental question then is "when should such an entity be scheduled?". A von Neumann machine has advantage over dataflow machine only when it executes long sequences of instructions without interruptions. This dictates that compiled procedures should be scheduled only when all their inputs are available. This imposes a fairly rigid methodology on compiling programs, and in our estimate would involve an unacceptably large loss of parallelism in programs. Nevertheless, we are looking for hardware structures which can execute procedures as efficiently as a sequential machine without requiring that the code be executed in a non interruptible manner. It seems to us that to keep synchronization costs low, any multiprocessor must allow cheap non-blocking suspensions of schedulable entities.

5. Acknowledgments

The authors wish to thank David Culler for valuable discussions on much of the subject matter of this paper, particularly Load/Store architectures and the structure of the Cray machines. Many people have contributed ideas to the MIT Tagged-Token Dataflow Machine. The architecture presented here is a refinement of the one presented in [3].

References

1. Ackerman, W. B. A Structure Processing Facility for Dataflow Computers. Proc. of the 1978 International Conference on Parallel Processing. July, 1978, pp. 166-172.

2. Arvind, and R. A. Iannucci. Instruction Set definition for a Tagged-token Dataflow Machine. 212-3. Computation Structures Group. Laboratory for Computer Science, MIT, Cambridge, Mass., February, 1983.

3. Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali & R. E. Thomas. "The Tagged Token Dataflow Architecture". Laboratory For Computer Science, MIT, July, 1983. (Prepared for MIT Subject 6.83s).

4. Arvind, K. P. Gostelow, and W. Plouffe. An Asynchronous Programming Language and Computing Machine. 114a. Department of Information and Computer Science, University of California, Irvine, California, December, 1978.

5. Arvind, and Thomas, R. E. I-Structures: An Efficient Data Type for Functional Languages. TM-178, LCS, September, 1980.

6. Block, E. The Engineering Design of the STRETCH Computer. Proceedings of the EJCC, 1959, pp. 48-59.

7. Burks, A., H. H. Goldstine & J. von Neumann. "Preliminary Discussion of the Logical Design of an Electronic Instrument, Part 2". *Datamation 8*, 10 (October 1962), 36-41.

8. Censier, L. M., and P. Feautrier. "A New Solution to the Coherence Problems in Multicache Systems". *IEEE Transactions on Computers C-27*, 12 (December 1978), 1112-1118.

9. Culler, D. E. Resource Management for the Tagged-Token Dataflow Architecture. Master Th., MIT,1985.

10. Dennis, J. B. First Version of a Dataflow Procedure Language. Memo 93, CSG, November, 1973. revised MAC TM61, May, 1975.

11. Dennis, J. B. "Data Flow Supercomputers". *Computer 13*, 11 (November 1980), 48-56.

12. Eckert, J. P., J. C. Chu, A. B. Tonik & W. F. Schmitt. Design of UNIVAC - LARC System: 1. Proceedings of the EJCC, 1959, pp. 59-65.

13. Edler, J., A. Gottlieb, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, M. Snir, P.J. Teller & J. Wilson. Issues Related to MIMD Shared-Memory Computers: The NYU Ultracomputer Approach. Proceedings of the 12th Annual International Symposium On Computer Architecture, Boston, June, 1985, pp. 126-135.

14. Fisher, J. A. Very Long Instruction Word Architectures and the ELI-512. 253, Yale University, Department of Computer Science, December, 1982.

15. Fuller, S. H. "Multi-Microprocessors: An Overview and Working Example". *Proc. of the IEEE 66*, 2 (February 1978), 216-228.

16. Gajski, Daniel D. & Jih-Kwon Peir. "Essential Issues in Multiprocessor Systems". *Computer 18*, 6 (June 1985), 9-27.

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

18. Gurd, John R., C. C. Kirkham & I. Watson. "The Manchester Prototype Dataflow Computer". *Communications of ACM 28*, 1 (January 1985), 34-52.

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

20. Hennessey, John L. "VLSI Processor Architecture". *IEEE Transactions on Computers C-33*, 12 (December 1984), 1221-1246.

21. Jordan, H. F. Performance Measurement on HEP - A Pipelined MIMD Computer. Proceedings of the 10th Annual International Symposium On Computer Architecture, Stockholm, Sweden, June, 1983, pp. 207-212.

22. Lampson, B. W., and K. A. Pier. A Processor for a High-Performance Personal Computer. Xerox Palo Alto Research Center, January, 1981.

23. Misunas, D. P. Deadlock Avoidence in a Data-Flow Architecture. Proceedings of the 3rd ACM-IEEE Milwaukee Symposium on Automatic Computation and Control, April, 1975, pp. 337-343.

24. Moon, D. A. Architecture of the Symbolics 3600. Proceedings of the 12th Annual International Symposium On Computer Architecture, Boston, June, 1985, pp. 76-83.

25. Patterson, David A. "Reduced Instruction Set Computers". *Communications of ACM 28*, 1 (January 1985), 8-21.

26. Radin, G. The 801 Minicomputer. Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, ACM, March, 1982.

27. Rau, B., D. Glaeser, and E. Greenwalt. Architectural Support for the Efficient Generation of Code for Horizontal Architectures. Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, March, 1982. Same as Computer Architecture News 10,2 and SIGPLAN Notices 17,4.

28. Russell, R. M. "The CRAY-1 Computer System". *Communications of ACM 21*, 1 (January 1978), 63-72.

29. Smith, B. J. A Pipelined, Shared Resource MIMD Computer. Proceedings of the 1978 International Conference on Parallel Processing, 1978, pp. 6-8.

30. Takahashi, N. & M. Amamiya. A Dataflow Processor Array System: Design and Analysis. Proceedings of the 10th Annual International Symposium On Computer Architecture, Stockholm, Sweden, June, 1983, pp. 243-251.

31. Thornton, J. E. Parallel Operations in the Control Data 6600. Proceedings of the SJCC, 1964, pp. 33-39.

32. *ALTO: A Personal Computer System - Hardware Manual.* Xerox Palo Alto Research Center, Palo Alto, California, 94304, 1979.

33. Yuba, T., T. Shimada, K. Hiraki, & H. Kashiwagi. "Sigma-1: A Dataflow Computer For Scientific Computation". Electrotechnical Laboratory, 1-1-4 Umesono, Sakuramura, Niiharigun, Ibaraki 305, Japan, 1984.