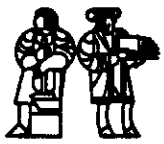
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Two Fundamental Issues in Multiprocessing

Computation Structures Group Memo 226-6

May 5, 1987

Arvind

Robert A. Iannucci

To appear in the Proceedings of
DFVLR - Conference 1987
on
"Parallel Processing in Science and Engineering"
June 25-26, 1987
Bonn-Bad Godesberg

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 contracts N00014-83-K-0125 and N00014-84-K-0099. The second author is employed by the International Business Machines Corporation.

Source File = TFI-6.MSS.5, Last updated 06-MAY-87 at 12:05 PM

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Abstract

A general purpose multiprocessor should be scalable, *i.e.*, show higher performance when more hardware resources are added to the machine. Architects of such multiprocessors must address the loss in processor efficiency due to two fundamental issues: long memory latencies and waits due to synchronization events. It is argued that a well designed processor can overcome these losses provided there is sufficient parallelism in the program being executed. The detrimental effect of long latency can be reduced by instruction pipelining, however, the restriction of a single thread of computation in von Neumann processors severely limits their ability to have more than a few instructions in the pipeline. Furthermore, techniques to reduce the memory latency tend to increase the cost of task switching. The cost of synchronization events in von Neumann machines makes decomposing a program into very small tasks counter-productive. Dataflow machines, on the other hand, treat each instruction as a task, and by paying a small synchronization cost for each instruction executed, offer the ultimate flexibility in scheduling instructions to reduce processor idle time.

Key words and phrases: caches, cache coherence, dataflow architectures, hazard resolution, instruction pipelining, LOAD/STORE architectures, memory latency, multiprocessors, multi-thread architectures, semaphores, synchronization, von Neumann architecture.

This paper is a 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.
CSG Memo 226-3	*Two Fundamental Issues in Multiprocessing: the Dataflow Solution* August, 1985
CSG Memo 226-4	*Two Fundamental Issues in Multiprocessing* February, 1986
CSG Memo 226-5	*Two Fundamental Issues in Multiprocessing* July, 1986

—

Table of Contents

List of Figures

Two Fundamental Issues in Multiprocessing

1. Importance of Processor Architecture

Parallel machines having up to several dozen processors are commercially available now. Most of the designs are based on von Neumann processors operating out of a shared memory. The differences in the architectures of these machines in terms of processor speed, memory organization and communication systems, are significant, but they all use relatively conventional von Neumann processors. These machines represent the general belief that processor architecture is of little importance in designing parallel machines. We will show the fallacy of this assumption on the basis of two issues: *memory latency* and *synchronization*. Our argument is based on the following observations:

1. Most von Neumann processors are likely to "idle" during long memory references, and such references are unavoidable in parallel machines.

2. Waits for synchronization events often require task switching, which is expensive on von Neumann machines. Therefore, only certain types of parallelism can be exploited efficiently.

We believe the effect of these issues on performance to be fundamental, and to a large degree, orthogonal to the effect of circuit technology. We will argue that by designing the processor properly, *the detrimental effect of memory latency on performance can be reduced provided there is parallelism in the program*. However, techniques for reducing the effect of latency tend to increase the synchronization cost.

In the rest of this section, we articulate our assumptions regarding general purpose parallel computers. We then discuss the often neglected issue of quantifying the amount of parallelism in programs. Section 2 develops a framework for defining the issues of latency and synchronization. Section 3 examines the methods to reduce the effect of memory latency in von Neumann computers and discusses their limitations. Section 4 similarly examines synchronization methods and their cost. In Section 5, we discuss multi-threaded computers like HEP and the MIT Tagged-Token Dataflow machine, and show how these machines can tolerate latency and synchronization costs provided there is sufficient parallelism in programs. The last section summarizes our conclusions.

1.1. Scalable Multiprocessors

We are primarily interested in *general purpose parallel computers*, *i.e.*, computers that can exploit parallelism, when present, in any program. Further, we want multiprocessors to be *scalable* in such a manner that adding hardware resources results in higher performance without requiring changes in application programs. The focus of the paper is not on arbitrarily large machines, but machines which range in size from ten to a thousand processors. We expect the processors to be at least as powerful as the current microprocessors and possibly as powerful as the CPU's of the current supercomputers. In particular, the context of the discussion is not machines with millions of one bit ALU's, dozens of which may fit on one chip. The design of such machines will certainly involve fundamental issues in addition to those presented here. Most parallel machines that are available today or likely to be available in the next few years fall within the scope of this paper (*e.g.*, the BBN Butterfly [36], ALICE [13] and now FLAGSHIP, the Cosmic Cube [38] and Intel's iPSC, IBM's RP3 [33], Alliant and CEDAR [26], and GRIP [11]).

If the programming model of a parallel machine reflects the machine configuration, *e.g.*, number of

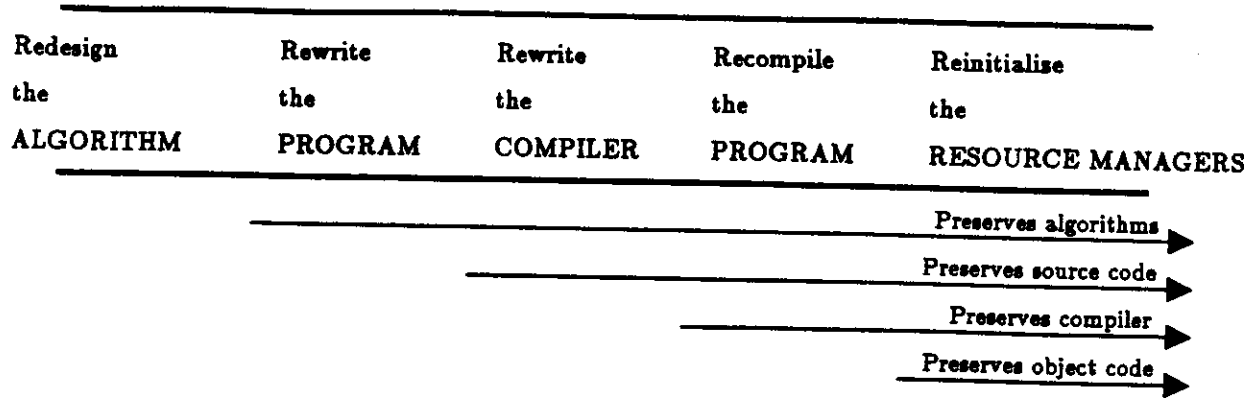


Figure 1: The Effect of Scaling on Software

processors and interconnection topology, the machine is not scalable in a practical sense. Changing the machine configuration should not require changes in application programs or system software; updating tables in the resource management system to reflect the new configuration should be sufficient. However, few multiprocessor designs have taken this stance with regard to scaling. In fact, it is not uncommon to find that source code (and in some cases, algorithms) must be modified in order to run on an altered machine configuration. Figure 1 depicts the range of effects of scaling on the software. Obviously, we consider architectures that support the scenario at the right hand end of the scale to be far more desirable than those at the left. It should be noted that if a parallel machine is not scalable, then it will probably not be fault-tolerant; one failed processor would make the whole machine unusable. It is easy to design hardware in which failed components, *e.g.*, processors, may be masked out. However, if the application code must be rewritten, our guess is that most users would wait for the original machine configuration to be restored.

1.2. Quantifying Parallelism in Programs

Ideally, a parallel machine should speed up the execution of a program in proportion to the number of processors in the machine. Suppose $t(n)$ is the time to execute a program on an n-processor machine. The speed-up as a function of n may be defined as follows:[1].

$$speed\text{-}up(n)=\frac{t(1)}{t(n)}$$

Speed-up is clearly dependent upon the program or programs chosen for the measurement. Naturally, if a program does not have "sufficient" parallelism, no parallel machine can be expected to demonstrate dramatic speedup. Thus, in order to evaluate a parallel machine properly, we need to characterize the inherent or potential parallelism of a program. This presents a difficult problem because the amount of parallelism in the source program that is exposed to the architecture may depend upon the quality of the compiler or programmer annotations. Furthermore, there is no reason to assume that the source program cannot be changed. Undoubtedly, different algorithms for a problem have different amounts of parallelism, and the parallelism of an algorithm can be obscured in coding. The problem is compounded by the fact that most programming languages do not have enough expressive power to show all the

[1]Of course, we are assuming that it is possible to run a program on any number of processors of a machine. In reality often this is not the case.

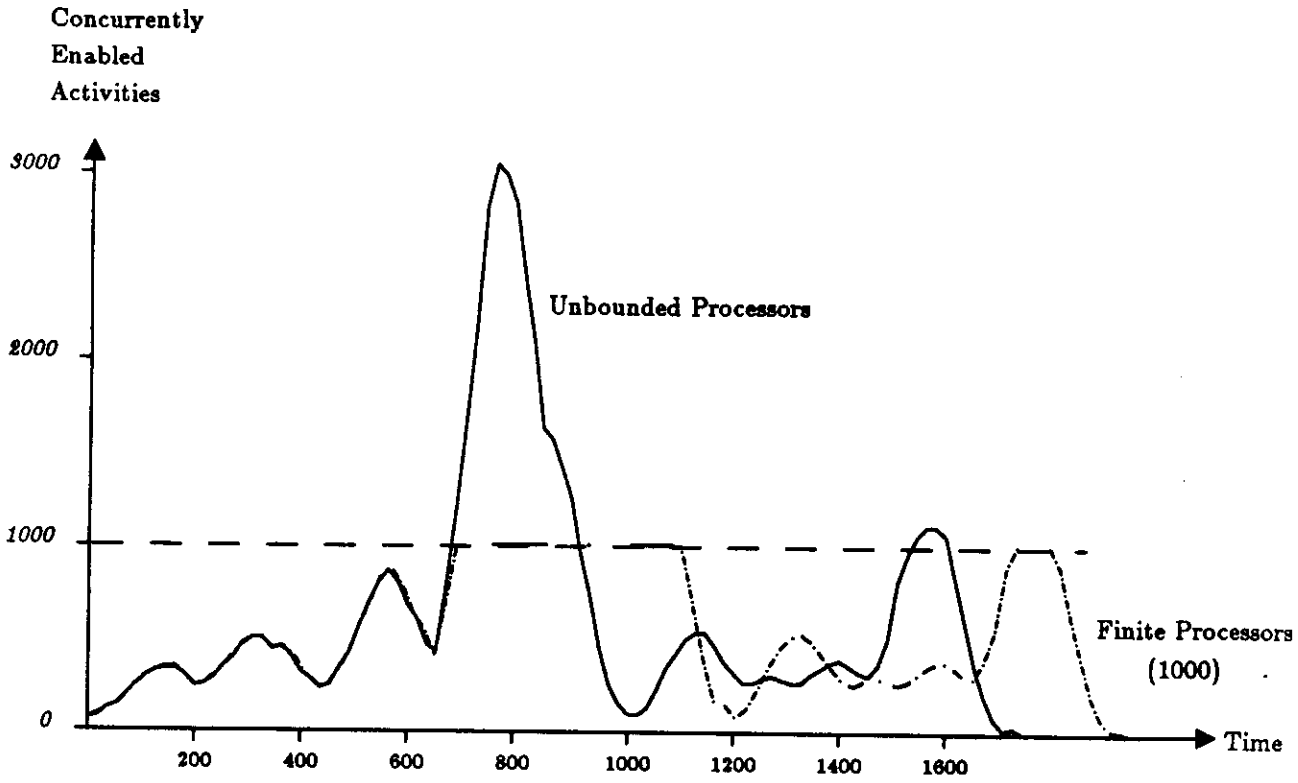


Figure 2: Parallelism Profile of SIMPLE on a 20 × 20 Array

possible parallelism of an algorithm in a program. In spite of all these difficulties, we think it is possible to make some useful estimates of the potential parallelism of an algorithm.

It is possible for us to code algorithms in Id [30], a high-level dataflow language, and compile Id programs into dataflow graphs, where the nodes of the graph represent simple operations such as fixed and floating point arithmetic, logicals, equality tests, and memory loads and stores, and where the edges represent only the *essential* data dependencies between the operations. A graph thus generated can be executed on an interpreter (known as GITA) to produce results and the *parallelism profile*, $pp(t)$, *i.e.*, the number of concurrently executable operators as a function of time on an idealized machine. The idealized machine has unbounded processors and memories, and instantaneous communication. It is further assumed that all operators (instructions) take unit time, and operators are executed as soon as possible. The parallelism profile of a program gives a good estimate of its "inherent parallelism" because it is drawn assuming *the execution of two operators is sequentialized if and only if there is a data dependency between them.* Figure 2 shows the parallelism profile of the SIMPLE code for a representative set of input data. SIMPLE [12], a hydrodynamics and heat flow code kernel, has been extensively studied both analytically [1] and by experimentation.

The solid curve in Figure 2 represents a single outer-loop iteration of SIMPLE on a 20 × 20 mesh, while a typical simulation run performs 100,000 iterations on 100 × 100 mesh. Since there is no significant parallelism between the outer-loop iterations of SIMPLE, the parallelism profile for N iterations can be obtained by repeating the profile in the figure N times. Approximately 75% of the instructions executed involve the usual arithmetic, logical and memory operators; the rest are miscellaneous overhead

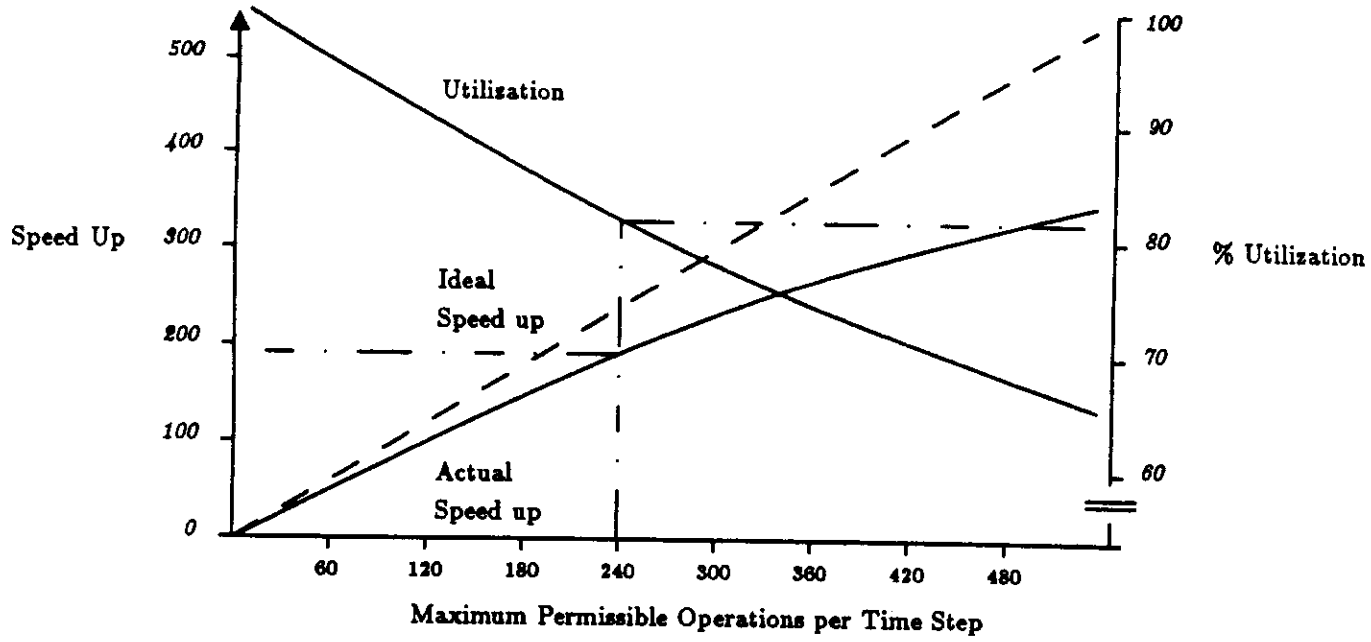


Figure 3: Speed Up and Utilization for 20 × 20 SIMPLE

operators, some of them peculiar to dataflow. One can easily deduce the parallelism profile of any set of operators from the raw data that was used to generate the profile in the figure; however, classifying operators as overhead is not easy in all cases.

The reader may visualize the execution on n processors by drawing a horizontal line at n on the parallelism profile and then "pushing" all the instructions which are above the line to the right and below the line. The dashed curve in Figure 2 shows this for SIMPLE on 1000 processors and was generated by our dataflow graph interpreter by executing the program again with the constraint that no more than n operations were to be performed at any step. However, a good estimate for $t(n)$ can be made, very inexpensively, from the ideal parallelism profile as follows. For any τ, if $pp(\tau) \leq n$, we perform all $pp(\tau)$ operations in time step τ. However, if $pp(\tau) > n$, then we assume it will take the least integer greater than $pp(\tau)/n$ steps to perform $pp(\tau)$ operations. Hence,

$$t(n)=\sum_{\tau=1}^{T_{MAX}} \left\lceil \frac{pp(\tau)}{n} \right\rceil$$

where T_{MAX} is the number of steps in the ideal parallelism profile. Our estimate of $t(n)$ is conservative because the data dependencies in the program may permit the execution of some instructions from $pp(\tau+1)$ in the last time step in which instructions from $pp(\tau)$ are executed.

In our dataflow graphs the number of instructions executed does not change when the program is executed on a different number of processors. Hence, $t(1)$ is simply the area under the parallelism profile. We can now plot *speed–up*$(n)=t(1)/t(n)$ and *utilization*$(n)=t(1)/n \times t(n)$, for SIMPLE as shown in Figure 3. For example, in the case of 240 processors, *speed–up* is 195, and *utilization* is 81%. One way to understand *utilization*(n) is that a program has n parallel operations for only *utilization*(n) fraction of its total $t(n)$ duration.

It can be argued that this problem does not have enough parallelism to keep, say, 1000 processors fully

utilized. On the other hand, if we cannot keep 10 processors fully utilized, we cannot blame the lack of parallelism in the program. Generally, under-utilization of the machine in the presence of massive parallelism stems from aspects of the internal architecture of the processors which preclude exploitation of certain types of parallelism. Machines are seldom designed to exploit inner-loop, outer-loop, as well as instruction-level parallelism simultaneously.

It is noteworthy that the potential parallelism varies tremendously during execution, a behavior which in our experience is typical of even the most highly parallel programs. We believe that any large program that runs for a long time must have sufficient parallelism to keep hundreds of processors utilized; several applications that we have studied support this belief. However, a parallel machine has to be fairly general purpose and programmable for the user to be able to express even the class of partial differential equation-based simulation programs represented by SIMPLE.

2. Latency and Synchronization

We now discuss the issues of latency and synchronization. We believe latency is most strongly a function of the physical decomposition of a multiprocessor, while synchronization is most strongly a function of how programs are logically decomposed.

2.1. Latency: The First Fundamental Issue

Any multiprocessor organization can be thought of as an interconnection of the following three types of modules (see Figure 4):

1. **Processing elements (PE):** Modules which perform arithmetic and logical operations on data. Each processing element has a single *communication port* through which all data values are received. Processing elements interact with other processing elements by sending messages, issuing interrupts or sending and receiving *synchronizing signals* through shared memory. PE's interact with memory elements by issuing LOAD and STORE instructions modified as necessary with atomicity constraints. Processing elements are characterized by the rate at which they can process instructions. As mentioned, we assume the instructions are simple, *e.g.*, fixed and floating point scalar arithmetic. More complex instructions can be counted as multiple instructions for measuring instruction rate.

2. **Memory elements (M):** Modules which store data. Each memory element has a single communication port. Memory elements respond to requests 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 requests[2].

3. **Communication elements (C):** 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 communication ports to one or more of the other communication ports. Communication elements are characterized by the rate of transmission, the time taken per transmission, and the constraints imposed by one transmission on others, *e.g.*, blocking. The maximum amount of data that may be conveyed on a *communication port* per unit time is fixed.

Latency is the time which elapses between making a request and receiving the associated response. The

[2]In many traditional designs, the "memory" subsystem can be simply modeled by one of these M elements. Interleaved memory subsystems are modeled as a collection of M's and C's. Memory subsystems which incorporate processing capability can be modeled with PE's, M's, and C's. Section 4.3 describes one such case.

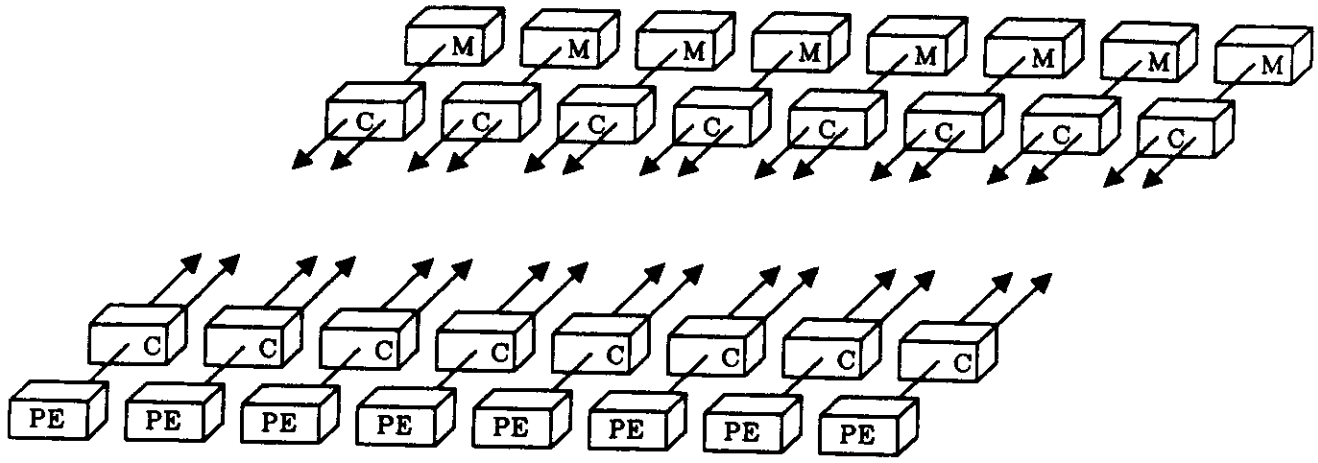


Figure 4: Structural Model of a Multiprocessor

above model implies that *a PE in a multiprocessor system faces larger latency in memory references than in a uniprocessor system* because of the transit time in the communication network between PE's and the memories. The actual interconnection of modules may differ greatly from machine to machine. For example, in the BBN Butterfly machine all memory elements are at an equal distance from all processors, while in IBM's RP3, each processor is closely coupled with a memory element. However, we assume that the average latency in a well designed n-PE machine should be $O(log(n))$. In a von Neumann processor, memory latency determines the time to execute memory reference instructions. Usually, the average memory latency also determines the maximum instruction processing speed. When latency cannot be hidden via overlapped operations, a tangible performance penalty is incurred. We call the cost associated with latency as the total *induced processor idle time* attributable to the latency.

2.2. Synchronization: The Second Fundamental Issue

We will call the basic units of computation into which programs are decomposed for parallel execution *computational tasks* or simply *tasks*. 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 basic operations:

1. *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.

2. *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.

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

The minimal support for synchronization can be provided by including instructions, such as atomic TEST-AND-SET, that operate on variables shared by synchronizing tasks[3]. However, to clarify the true cost

[3]While not strictly necessary, atomic operations such as TEST-AND-SET are certainly a convenient base upon which to build synchronization operations. See Section 4.3.

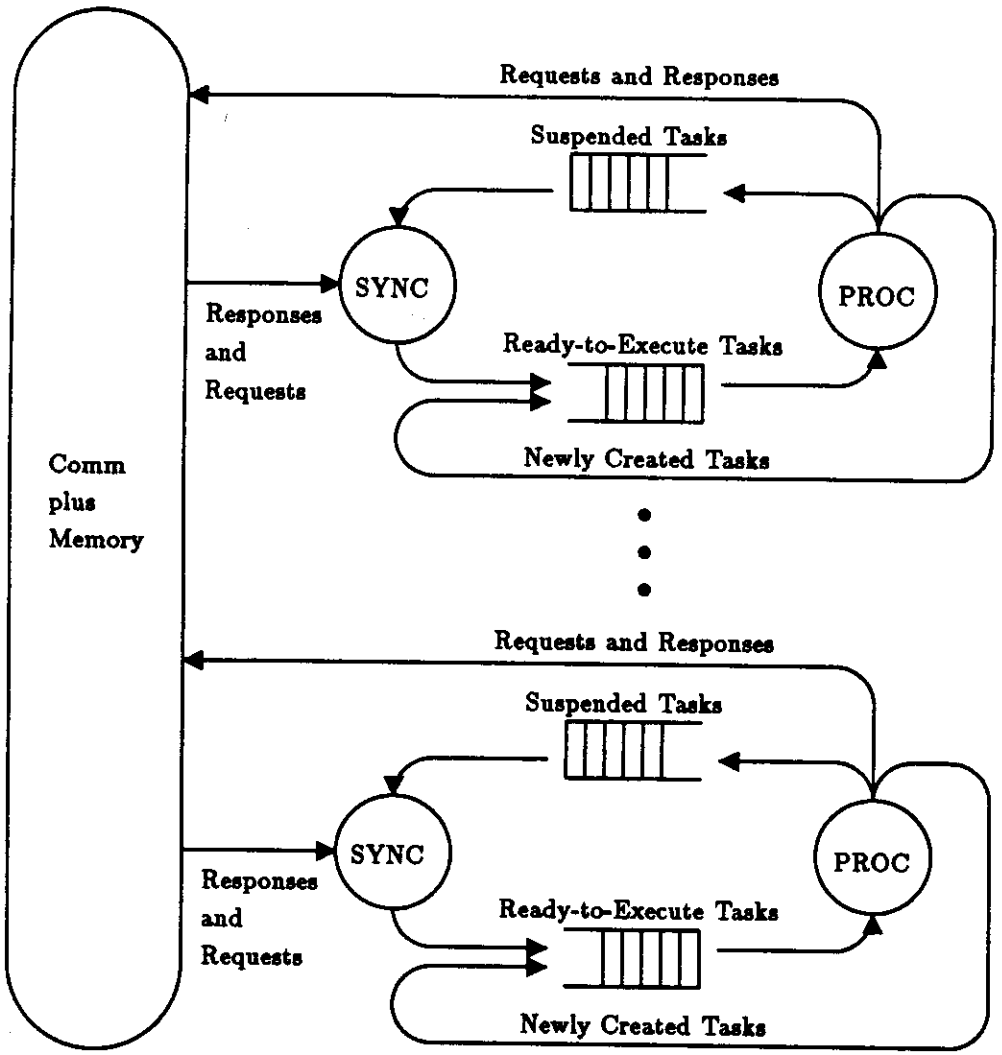


Figure 5: Operational Model of a Multiprocessor

of such instructions, we will use the *Operational Model* presented in Figure 5. Tasks in the operational model have resources, such as registers and memory, associated with them and constitute the smallest unit of independently schedulable work on the machine. A task is in one of the three states: *ready-to-execute*, *executing* or *suspended*. Tasks ready for execution may be queued locally or globally. When selected, a task occupies a processor until either it completes or is suspended waiting for a synchronization signal. A task changes from *suspended* to *ready-to-execute* when another task causes the relevant synchronization event. Generally, a suspended task must be set aside to avoid deadlocks[4]. The cost associated with such a synchronization is *the fixed time to execute the synchronization instruction plus the time taken to switch to another task.* The cost of task switching can be high because it usually involves saving the processor state, that is, the *context* associated with the task.

[4]Consider the case of a single processor system which must execute *n* cooperating tasks.

There are several subtle issues in accounting for synchronization costs. An event to enable or dispatch a task needs a *name*, such as that of a register or a memory location, and thus, synchronization cost should also include the instructions that generate, match and reuse identifiers which name synchronization events. It may not be easy to identify the instructions executed for this purpose. Nevertheless, such instructions represent overhead because they would not be present if the program were written to execute on a single sequential processor. The hardware design usually dictates the number of names available for synchronization as well as the cost of their use.

The other subtle issue has to do with the accounting for *intra-task synchronization*. As we shall see in Section 3, most high performance computers overlap the execution of instructions belonging to one task. The techniques used for synchronization of instructions in such a situation (*e.g.*, instruction dispatch and suspension) are often quite different from techniques for inter-task synchronization. It is usually safer and cheaper not to put aside the instruction waiting for a synchronization event, but rather to idle (or, equivalently, to execute NO-OP instructions while waiting). This is usually done under the assumption that the idle time will be on the order of a few instruction cycles. We define the synchronization cost in such situations to be the *induced processor idle time* attributable to waiting for the synchronization event.

3. Processor Architectures to Tolerate Latency

In this section, we describe those changes in von Neumann architectures that have directly reduced the effect of memory latency on performance. Increasing the processor state and instruction pipelining are the two most effective techniques for reducing the latency cost. Using Cray-1 (perhaps the best pipelined machine design to date), we will illustrate that it is difficult to keep more than 4 or 5 instructions in the pipeline of a von Neumann processor. It will be shown that every change in the processor architecture which has permitted overlapped execution of instructions has necessitated introduction of a cheap synchronization mechanism. Often these synchronization mechanisms are hidden from the user and not used for inter-task synchronization. This discussion will further illustrate that reducing latency frequently increases synchronization costs.

Before describing these evolutionary changes to hide latency, we should point out that the memory system in a multiprocessor setting creates more problems than just increased latency. 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 the following 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.
- No useful bound on the worst case time to fetch an operand may be possible at machine design time because of the scalability assumption. This is at odds with RISC designs which treat memory access time as bounded and fixed.
- If a processor were to issue several (pipelined) memory requests to different remote memory modules, the responses could arrive out of order.

All of these issues are discussed and illustrated in the following sections. A general solution for accepting memory responses out of order requires a synchronization mechanism to match responses with the destination registers (*names* in the task's context) and the instructions waiting on that value. The ill-fated Denelcor HEP [25] is one of the very few architectures which has provided 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 5).

3.1. Increasing the Processor State

Figure 6 depicts the modern-day view of the von Neumann computer [9] (*sans* I/O). In the earliest computers, such as EDSAC, the *processor state* consisted solely of an accumulator, a quotient register, and a program counter. Memories were relatively slow compared to the processors, and thus, the time to fetch an instruction and its operands completely dominated the instruction cycle time. Speeding up the Arithmetic Logic Unit was of little use unless the memory access time could also be reduced.

The appearance of multiple "accumulators" reduced the number of operand fetches and stores, and index registers dramatically reduced the number of instructions executed by essentially eliminating the need for self-modifying code. Since the memory traffic was drastically lower, programs executed much faster than before. However, the enlarged processor state did not reduce the time lost during memory references and, consequently, did not contribute to an overall reduction in cycle time; the basic cycle time improved only with improvements in circuit speeds.

3.2. Instruction Prefetching

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. If instructions and data are kept in separate memories, it is possible to overlap instruction prefetching and operand fetching also. (The IBM STRETCH [7] and Univac LARC [16] represent two of the earliest attempts at implementing this idea.) Prefetching can reduce the cycle time of the machine by twenty to thirty percent 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.

Instruction prefetching works well when the execution of instruction n does not have any effect on either the choice of instructions to fetch (as is the case in a BRANCH) or the content of the fetched instruction (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 shown little performance/cost benefits. Lately, the concept of *delayed* BRANCH instructions from microprogramming has been incorporated, with success, in LOAD/STORE architectures (see Section 3.4). 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 seventy percent of the time a useful instruction can be put in that position.

3.3. Instruction Buffers, Operand Caches and Pipelined Execution

The time to fetch instructions can be further reduced by providing a fast instruction buffer. In machines such as the CDC 6600 [40] and the Cray-1 [37], the instruction buffer is automatically loaded with n instructions in the neighborhood of the referenced instruction (relying on spatial locality in code references), whenever the referenced instruction is found to be missing. To take advantage of instruction buffers, it is also necessary to speed up the operand fetch and execute phases. This is usually done by providing *operand* caches or buffers, and overlapping the operand fetch and execution phases[5]. Of course, balancing the pipeline under these conditions may require further pipelining of the ALU. If successful, 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 four to five instructions in the von Neumann

[5]As we will show in Section 4.4, caches in a multiprocessor setting create special problems.

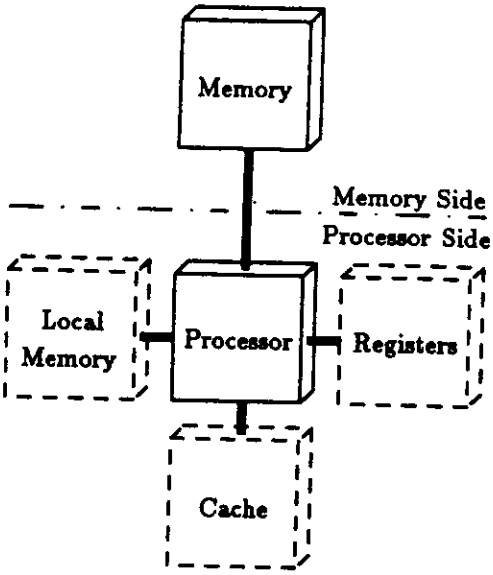


Figure 6: The von Neumann Processor (from Gajski and Peir [20])

framework presents some serious conceptual difficulties, as discussed next.

Designing a well-balanced pipeline requires that the time taken by various pipeline stages be more or less equal, 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 in 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 the complete execution of instruction *n* determines the next instruction to be executed (the aforementioned 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 temporarily able to prevent a new instruction from entering the pipeline when there possibility of interference with the instructions already in the pipe. Detecting and quickly resolving these *hazards* is very difficult with ordinary instruction sets, *e.g.*, IBM 370, VAX 11 or Motorola 68000, due to their complexity.

A major complication in pipelining complex instructions is the variable amount of time taken in each stage of instruction processing (refer to Figure 7). 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.

3.4. Load/Store Architectures

Seymour Cray, in the sixties, pioneered instruction sets (CDC 6600, Cray-1) which separate instructions into two disjoint classes. In one class are instructions which move data *unchanged* between memory and high speed registers. In the other class are instructions which operate on data in the registers. Instructions

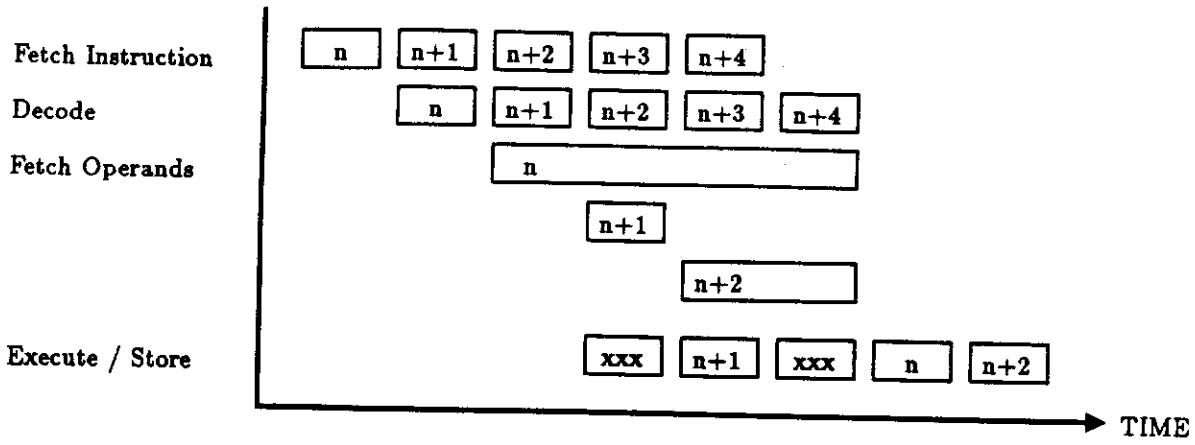


Figure 7: Variable Operand Fetch Time

of the second class *cannot* access the memory. This rigid distinction simplifies instruction scheduling. For each instruction, it is trivial to see if a memory reference will be necessary or not. Moreover, the memory system and the ALU may be viewed as parallel, noninteracting pipelines. An instruction dispatches exactly one unit of work to either one pipe or the other, but never both.

Such architectures have come to be known as LOAD/STORE architectures, and include the machines built by Reduced Instruction Set Computer (RISC) enthusiasts (the IBM 801 [34], Berkeley's RISC [32], and Stanford MIPS [22] are prime examples). LOAD/STORE architectures use the time between instruction decoding and instruction dispatching for hazard detection and resolution (see Figure 8). The design of the instruction pipeline is based on the principle that if an instruction gets past some fixed pipe stage, it should be able to run to completion without incurring any previously unanticipated hazards.

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 them[6]. 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 is 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

[6]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.

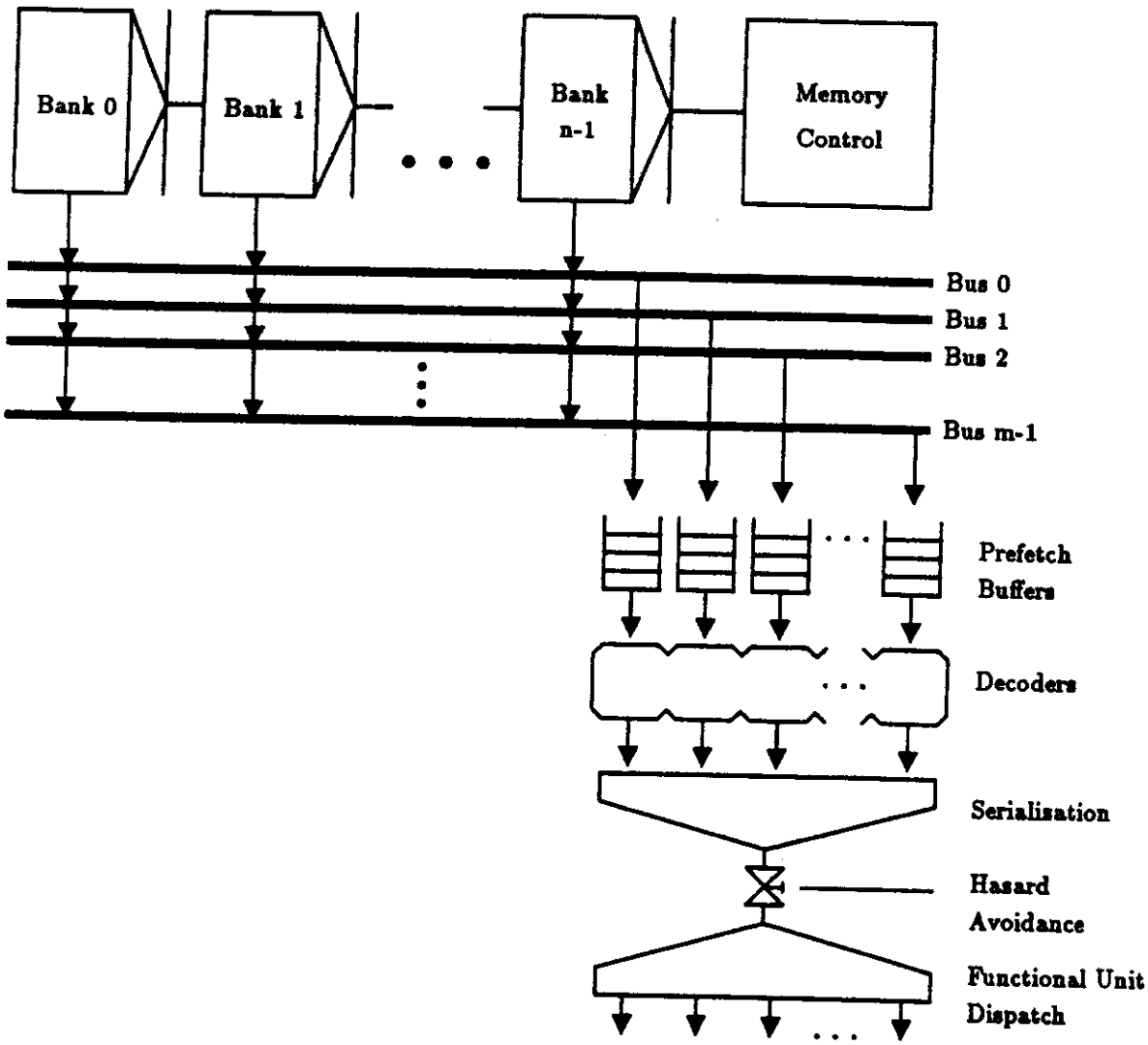


Figure 8: Hazard Avoidance at the Instruction Decode Stage

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 that, like the Floating Point multiplier, 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[7].

From another point of view, latency cost has been reduced by introducing a cheap synchronization mechanism: reservation bits on processor registers. However, the number of *names* available for

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

synchronization, *i.e.*, the size of the task's processor-bound context, is precisely the number of registers, and this restricts the amount of exploitable parallelism and tolerable latency. 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, i_n and i_m. This will require i_n and i_m to be executed sequentially while no such order may have been implied by the source code. *Shadow registers* have been suggested to deal with this class of problems. In fact, shadow registers are an engineering approach to solving a non-engineering problem. The real issue is *naming*. The reason that addition of explicit and implicit registers improves the situation derives from the addition of (explicit and implicit) *names* for synchronization and, hence, a greater opportunity for tolerating latency.

Some LOAD/STORE architectures have eliminated the need for reservation bits on registers by making the compiler responsible for scheduling instructions, such that the result is guaranteed to be available. The compiler can perform hazard resolution only if the time for each operation *e.g.*, ADD, LOAD, is known; it inserts NO-OP instructions wherever necessary. Because the instruction execution times are an intimate part of the object 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 the 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 (one 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 at any given time. For example, in the Cray-1, no more than four independent addresses can be generated during a memory cycle. If the generated address causes a bank conflict, the pipeline is stopped. However, any conflict is resolved in at most three cycles.

LOAD/STORE architectures, because of their simpler instructions, often execute 15% to 50% more instructions than machines with more complex instructions [34]. This increase may be regarded as synchronization cost. However, this is easily compensated by improvements in clock speed made possible by simpler control mechanisms.

4. Synchronization Methods for Multiprocessing

4.1. Global Scheduling on Synchronous machines

For a totally synchronous multiprocessor it is possible to envision a master plan which specifies operations for every cycle on every processor. An analogy can be made between programming such a multiprocessor and coding a horizontally microprogrammed machine. Recent advances in compiling [18] have made such code generation feasible and encouraged researchers to propose and build several different synchronous multiprocessors. Cydrome and Multiflow computers, which are based on proposals in [35] and [19], respectively, are examples of such machines. These machines are generally referred to as *very long instruction word*, or VLIW, machines, because each instruction actually contains multiple smaller instructions (one per functional unit or processing element). The strategy is based on maximizing the use of resources and resolving potential run-time conflicts in the use of resources at compile time. Memory references and control transfers are "anticipated" as in RISC architectures, but here, multiple concurrent threads of computation are being scheduled instead of only one. Given the possibility of decoding and initiating many instructions in parallel, such architectures are highly appealing when one realizes that the fastest machines available now still essentially decode and dispatch instructions one at a time.

We believe that this technique is effective in its currently realized context, *i.e.*, Fortran-based computations on a small number (4 to 8) of processors. Compiling for parallelism beyond this level, however, becomes intractable. It is unclear how problems which rely on dynamic storage allocation or require nondeterministic and real-time constraints will play out on such architectures.

4.2. Interrupts and Low-level Context Switching

Almost all von Neumann machines are capable 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 explicitly, *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 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 state is good because it 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 in the trivial case of infrequent synchronization events or 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 [42], the Xerox Dorado [27], and the Symbolics 3600 family [29] 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, in other words, 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[8]. 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 5) although it should reduce synchronization cost over processors which can hold only a single context. It may be worth thinking about adopting this scheme to reduce the latency cost of a nonlocal memory references as well.

The limitations of this approach are obvious. High performance processors may have a small programmer-visible state (number of registers) but a much larger implicit state (caches). Low-level task switching does not necessarily take care of the overhead of flushing caches[9]. Further, one can only have a small number of independent contexts without completely overshadowing the cost of ALU hardware.

4.3. Semaphores and the Ultracomputer

Next to interrupts, the most commonly supported feature for synchronization is an *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 perform such synchronization with ordinary read and write memory operations, the task is much simpler with an atomic TEST-AND-SET instruction. TEST-AND-SET is powerful enough to implement

---

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

[9]However, solutions such as multicontext caches and multicontext address translation buffers have been used to advantage in reducing this task switching overhead, (*c.f.*, the STO stack mechanism in the IBM 370/168).

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 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 suggested by the NYU Ultracomputer group [17]. Their technique can be illustrated by the atomic FETCH-AND-<OP> instruction (an evolution of the REPLACE-ADD instruction). 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.

An architect must choose between a wide variety of implementations for FETCH-AND-<OP>. One possibility is that the processor may interpret the instruction with a series of more primitive instructions. While possible, such a solution does not find much favor because it will cause considerable memory traffic. A second scheme implements FETCH-AND-<OP> in the memory controller (this is the alternative chosen by the CEDAR project [28]). This typically results in a significant reduction of network traffic because atomicity of memory transactions from the memory's controller happens by default. The scheme suggested by the NYU Ultracomputer group implements the instruction *in the switching nodes of the network.*

This implementation 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,v_i) and FETCH-AND-ADD(A,v_j), the switch extracts the values v_i and v_j, forms a new packet (FETCH-AND-ADD(A,v_i+v_j)), forwards it to the memory, and stores the value of v_i temporarily. When the memory returns the old value of location A, the switch returns two values ((A) and $(A)+v_i$). 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 idle time due to 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 overshadow the advantage of "combining" over other simpler implementations.

The simulation results reported by NYU [17] 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 of a computation hardly share any data, thus, the costly *mutual exclusion* synchronization is rarely needed 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.

The second point may become clearer after reading the next section.

4.4. Cache Coherence Mechanisms

While highly successful for reducing memory latency in uniprocessors, caches in a multiprocessor setting introduce a serious synchronization problem called *cache coherence.* Censier and Feautrier [10] 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."*. It is easy to see that this may be difficult to achieve in multiprocessing.

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 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 copies of location x in caches of other processors, 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 rather than avoiding the possibility of cache incoherence. Generally, *state* information indicating whether the cached data is private or shared, read-only or read-write, etc., is associated with each cache entry. However, this state somehow has to be updated after each memory reference. Implementations of this idea are generally intractable except possibly in the domain of bus-oriented multiprocessors. The so-called *snoopy bus* solution uses the broadcasting capability of buses and purges entry x from all caches when a processor attempts a STORE to x. 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 snoopy bus' ability to handle the coherence-maintaining traffic.

It is possible to improve upon the above solution if some additional state 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. Schemes have also been proposed explicitly to 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.

It is worth noting that, while not obvious, a direct trade-off often exists between decreasing the parallelism and increasing the cachable or non-shared data.

5. Multi-Threaded Architectures

In order to reduce memory latency cost, it is essential that a processor be capable of issuing multiple, overlapped memory requests. The processor 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, with the exception of array references in vector machines. The reasons behind this limitation are fundamental:

1. von Neumann processors must observe instruction sequencing constraints, and
2. since memory references can get out of order in the pipeline, a large number of identifiers to distinguish memory responses must be provided.

One way to overcome the first deficiency is to interleave many threads of sequential computations (as we saw in the very long instruction word architectures of Section 4.1). The second deficiency can be overcome by providing a large register set with suitable reservation bits. It should be noted that these requirements are somewhat in conflict. The situation is further complicated by the need of tasks to communicate with each other. Support for cheap synchronization calls for the processor to switch tasks quickly and to have a non-empty queue of tasks which are ready to run. One way to achieve this is again by interleaving multiple threads of computation and providing some intelligent scheduling mechanism to avoid busy-waits. Machines supporting multiple threads and fancy scheduling of instructions or processes look less and less like von Neumann machines as the number of threads increases.

In this section, we first discuss the erstwhile Denelcor HEP [25, 39]. The HEP was the first commercially available multi-threaded computer. After that we briefly discuss dataflow machines, which may be regarded as an extreme example of machines with multiple threads; machines in which each instruction constitutes an independent thread and only non-suspended threads are scheduled to be executed.

5.1. The Denelcor HEP: A Step Beyond von Neumann Architectures

The basic structure of the HEP processor is shown in Figure 9. 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) of the processes whose threads are to be interleaved for execution. The delay around the control loop varies with the queue size, 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*.

There are 2048 registers in each processor; each process has an index offset into the register array. Inter-process, *i.e.*, inter-thread, communication is possible via these registers by overlapping register allocations. 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 behaves like a NO-OP instruction; the program counter of the process, *i.e.*, PSW, which initiated the instruction is not incremented. The process effectively *busy-waits* but without blocking the processor. 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 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.

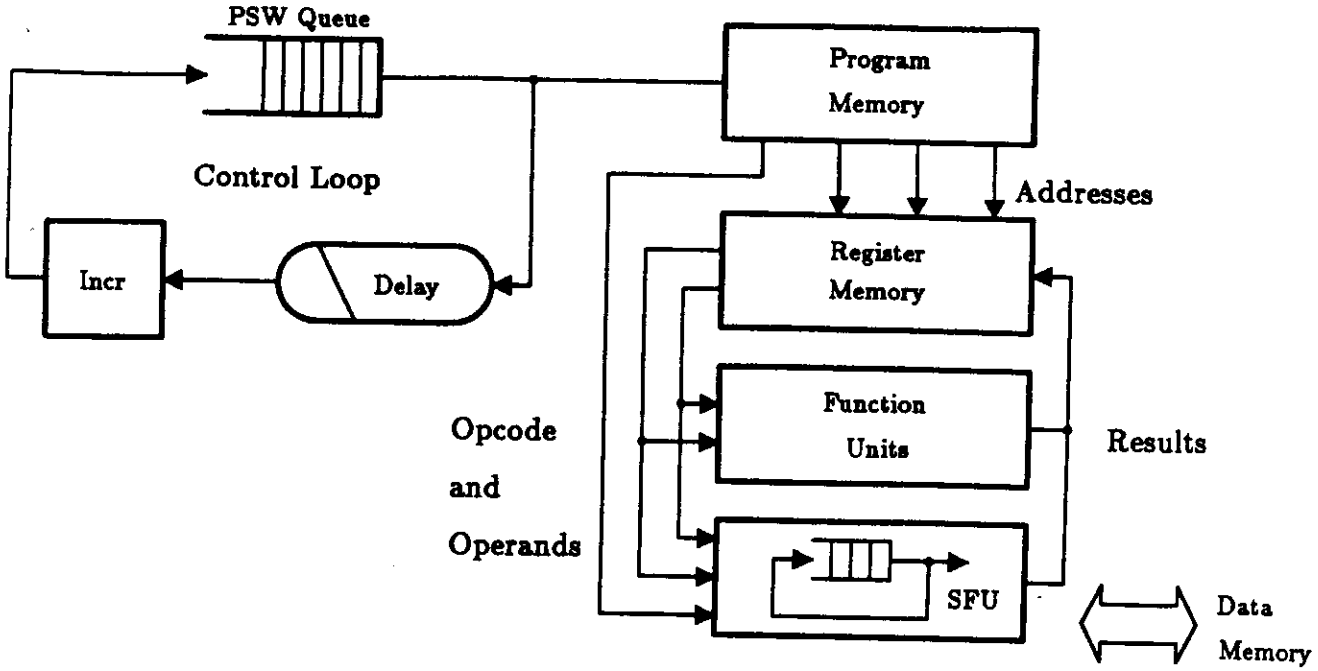


Figure 9: Latency Toleration and Synchronization in the HEP

Thus, the HEP is capable up to a point of using parallelism in programs to hide memory and communication latency. At the same time it provides efficient, low-level synchronization mechanisms in the form of presence-bits in registers and main memory. However, the HEP approach does not go far enough because there is a limit of *one* outstanding memory request per process, and the cost of synchronization through shared registers can be high because of the loss of processor time due to *busy-waiting*. A serious impediment to the software development on HEP was the limit of 64 PSW's in each processor. Though only 8 PSW's may be required to keep the process pipeline full, a much larger number is needed to name all concurrent tasks of a program.

5.2. Dataflow Architectures

Dataflow architectures [2, 15, 21, 23] represent a radical alternative to von Neumann architectures because they use dataflow graphs as their machine language [4, 14]. Dataflow graphs, as opposed to conventional machine languages, specify only a partial order for the execution of instructions and thus provide opportunities for parallel and pipelined execution at the level of individual instructions. For example, the dataflow graph for the expression a*b + c*d only specifies that both multiplications be executed before the addition; however, the multiplications can be executed in any order or even in parallel. The advantage of this flexibility becomes apparent when we consider that the order in which a, b, c and d will become available may not be known at compile time. For example, computations for operands a and b may take longer than computations for c and d or *vice versa*. Another possibility is that the time to fetch different operands may vary due to scheduling and hardware characteristics of the machine. Dataflow graphs do not force unnecessary sequentialization and dataflow processors schedule instructions according to the availability of the operands.

The instruction execution mechanism of a dataflow processor is fundamentally different from that of a

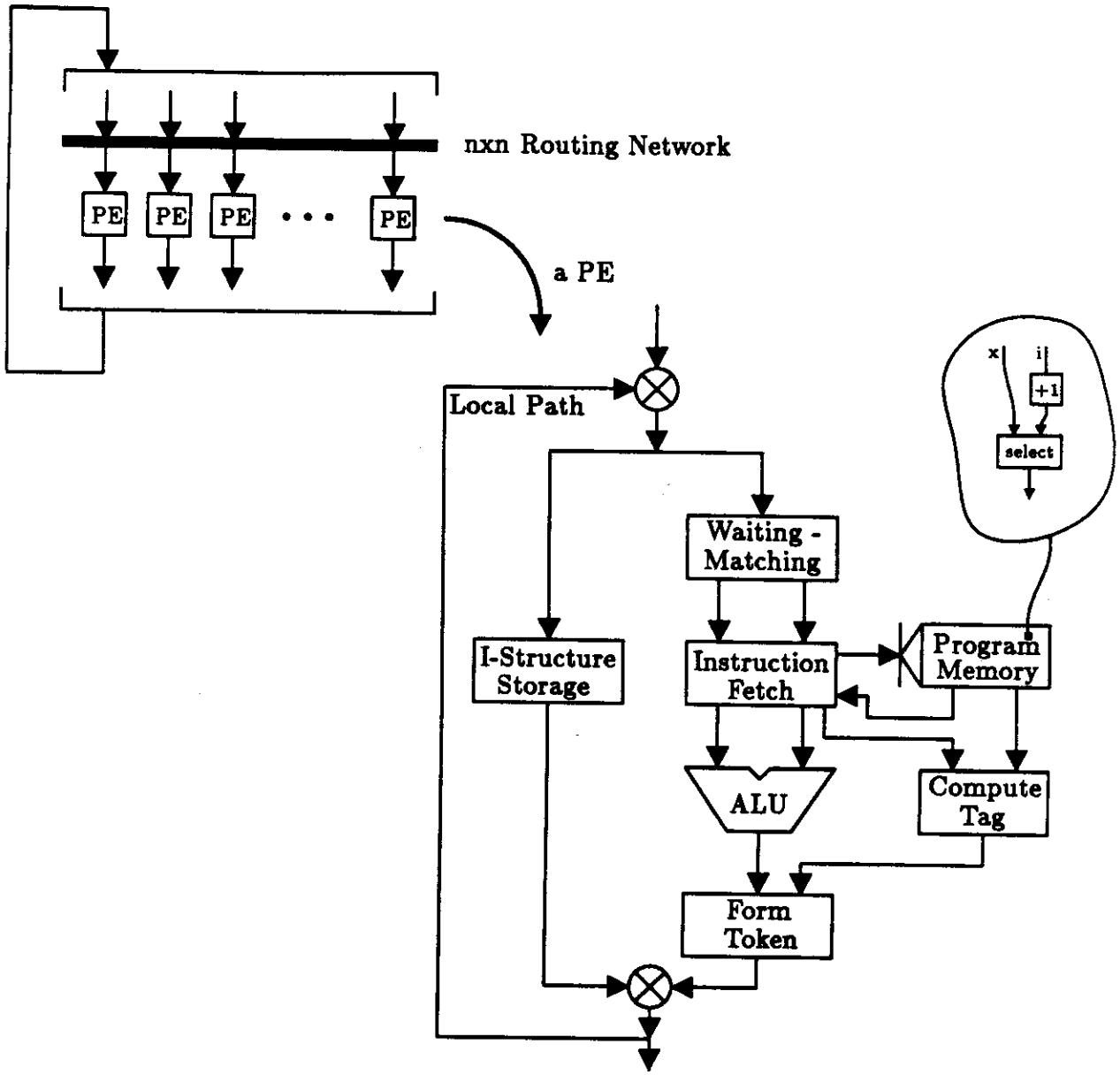


Figure 10: The MIT Tagged-Token Dataflow Machine

von Neumann processor. We will briefly illustrate this using the MIT Tagged-Token architecture (see Figure 10). Rather than following a *Program Counter* for the next instruction to be executed and then fetching operands for that instruction, a dataflow machine provides a low-level synchronization mechanism in the form of *Waiting-Matching* section which dispatches only those instructions for which data are already available. This mechanism relies on *tagging* each datum with the address of the instruction to which it belongs and the context in which the instruction is being executed. One can think of the instruction address as replacing the program counter, and the context identifier replacing the frame base register in traditional von Neumann architecture. It is the machine's job to match up data with the same tag and then to execute the denoted instruction. In so doing, new data will be produced, with a new

tag indicating the successor instruction(s). Thus, each instruction represents a synchronization operation. Note that the number of synchronization names is limited by the size of the tag, which can easily be made much larger than the size of the register array in a von Neumann machine. Note also that the processor pipeline is non-blocking: given that the operands for an instruction are available, the corresponding instruction can be executed without further synchronization.

In addition to the waiting-matching section which is used primarily for dynamic scheduling of instructions, the MIT Tagged-Token machine provides a second synchronization mechanism called *I-Structure Storage*. Each word of I-structure storage has 2 bits associated with it to indicate whether the word is empty, full or has pending read-requests. This greatly facilitates overlapped execution of a producer of a data structure with the consumer of that data structure. There are three instructions at the graph level to manipulate I-structure storage. These are *allocate* - to allocate *n* empty words of storage, *select* - to fetch the contents of the i^{th} word of an array and *store* - to store a value in a specified word. Generally software concerns dictate that a word be written into only once before it is deallocated. The dataflow processor treats all I-structure operations as *split-phase*. For example, when the *select* instruction is executed, a packet containing the tag of the destination instruction of the select instruction is forwarded to the proper address, possibly in a distant I-structure storage module. The actual memory operation may require waiting if the data is not present and thus the result may be returned many instruction times later. The key is that the instruction pipeline need not be suspended during this time. Rather, processing of other instructions may continue immediately after *initiation* of the operation. Matching of memory responses with waiting instructions is done via tags in the waiting-matching section.

One advantage of tagging each datum is that data from different contexts can be mixed freely in the instruction execution pipeline. Thus, instruction-level parallelism of dataflow graphs can effectively absorb the communication latency and minimize the losses due to synchronization waits. We hope it is clear from the prior discussion that even the most highly pipelined von Neumann processor cannot match the flexibility of a dataflow processor in this regard. A more complete discussion of dataflow machines is beyond the scope of this paper. An overview of executing programs on the MIT Tagged-Token Dataflow machine can be found in [6]. A deeper understanding of dataflow machines can be gotten from [2]. Additional, albeit slightly dated, details of the machine and the instruction set are given in [3] and [5], respectively.

6. Conclusions

We have presented the loss of performance due to increased latency and waits for synchronization events as the two fundamental issues in the design of parallel machines. These issues are, to a large degree, independent of the technology differences between various parallel machines. Even though we have not presented it as such, these issues are also independent of the high-level programming model used on a multiprocessor. If a multiprocessor is built out of conventional microprocessors, then degradation in performance due to latency and synchronization will show up regardless of whether a shared-memory, message-passing, reduction or dataflow programming model is employed.

Is it possible to modify a von Neumann processor to make it more suitable as a building block for a parallel machine? In our opinion the answer is a qualified "yes". The two most important characteristics of the dataflow processor are split-phase memory operations and the ability to put aside computations (*i.e.*, processes, instructions, or whatever the scheduling quanta are) without blocking the processor. We think synchronization bits in the storage are essential to support the producer-consumer type of parallelism. However, the more concurrently active threads of computation we have, the greater is the requirement for hardware-supported synchronization names. Iannucci [24] and others [8] are actively exploring designs based on these ideas. Only time will tell if it will be fair to classify such processors as von Neumann processors.

The biggest appeal of von Neumann processors is that they are widely available and familiar. There is a tendency to extrapolate these facts into a belief that von Neumann processors are "simple" and efficient. A technically sound case can be made that well designed von Neumann processors are indeed very efficient in executing sequential codes and require less memory bandwidth than dataflow processors. However, the efficiency of sequential threads disappears fast if there are too many interruptions or if idling of the processor due to latency or data-dependent hazards increases. Papadopoulos [31] is investigating dataflow architectures which will improve the efficiency of the MIT Tagged-Token architecture on sequential codes without sacrificing any of its dataflow advantages. We can assure the reader that none of these changes are tantamount to introducing a program counter in the dataflow architecture.

For lack of space we have not discussed the effect of multi-threaded architectures on the compiling and language issues. It is important to realize that compiling into primitive dataflow operators is a much simpler task than compiling into cooperating sequential threads. Since the cost of inter-process communication in a von Neumann setting is much greater than the cost of communication within a process, there is a preferred process or "grain" size on a given architecture. Furthermore, placement of synchronization instructions in a sequential code requires careful planning because an instruction to wait for a synchronization event may experience very different waiting periods in different locations in the program. Thus, even for a given grain size, it is difficult to decompose a program optimally. Dataflow graphs, on the other hand, provide a uniform view of inter- and intra-procedural synchronization and communication, and as noted earlier, only specify a partial order to enforce data dependencies among the instructions of a program. Though it is very difficult to offer a quantitative measure, we believe that an Id Nouveau compiler to generate code for a multi-threaded von Neumann computer will be significantly more complex than the current compiler [41] which generates fine grain dataflow graphs for the MIT Tagged-Token dataflow machine. Thus dataflow computers, in addition to providing solutions to the fundamental hardware issues raised in this paper, also have compiler technology to exploit their full potential.

Acknowledgment

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. Members of the Computation Structures Group have developed many tools, without which the analysis of the Simple code would have been impossible. In particular, we would like to thank Ken Traub for the ID Compiler and David Culler and Dinarte Morais for GITA. This paper has benefited from numerous discussions with people both inside and outside MIT. We wish to thank Natalie Tarbet, Ken Traub, David Culler, Vinod Kathail and Rishiyur Nikhil for suggestions to improve this manuscript.

References

1. Arvind and R. E. Bryant. Design Considerations for a Partial Equation Machine. Proceedings of Scientific Computer Information Exchange Meeting, Lawrence Livermore Laboratory, Livermore, CA, September, 1979, pp. 94-102.

2. Arvind and D. E. Culler. "Dataflow Architectures". *Annual Reviews of Computer Science 1* (1986), 225-253.

3. Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali, and R. E. Thomas. The Tagged Token Dataflow Architecture. Internal report. (including architectural revisions of October, 1983).

4. Arvind and K. P. Gostelow. "The U-Interpreter". *Computer 15*, 2 (February 1982), 42-49.

5. Arvind and R. A. Iannucci. Instruction Set Definition for a Tagged-Token Data Flow Machine. Computation Structures Group Memo 212-3, Laboratory for Computer Science, MIT, Cambridge, Mass., Cambridge, MA 02139, December, 1981.

6. Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. Proc. PARLE, (Parallel Architectures and Languages Europe), Eindhoven, The Netherlands, June, 1987.

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

8. Buehrer, R. and K. Ekanadham. Dataflow Principles in Multi-processor Systems. ETH, Zurich, and Research Division, Yorktown Heights, IBM Corporation, July, 1986.

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

10. 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.

11. Clack, C. and Peyton-Jones, S. L. The Four-Stroke Reduction Engine. Proceedings of the 1986 ACM Conference on Lisp and Functional Programming, Association for Computing Machinery, August, 1986, pp. 220-232.

12. Crowley, W. P., C. P. Hendrickson, and T. E. Rudy. The SIMPLE Code. Internal Report UCID-17715, Lawrence Livermore Laboratory, Livermore, CA, February, 1978.

13. Darlington, J. and M. Reeve. ALICE: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages. Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, Portsmouth, NH, 1981, pp. 65-76.

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

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

16. 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.

17. 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.

18. Ellis, J. R.. *Bulldog: a Compiler for VLIW Architectures*. The MIT Press, 1986.

19. Fisher, J. A. Very Long Instruction Word Architectures and the ELI-512. Proc. of the 10th, International Symposium on Computer Architecture, IEEE Computer Society, June, 1983.

20. Gajski, D. D. & J-K. Peir. "Essential Issues in Multiprocessor Systems". *Computer 18*, 6 (June 1985), 9-27.

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

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

23. Hiraki, K., S. Sekiguchi, and T. Shimada. System Architecture of a Dataflow Supercomputer. Computer Systems Division, Electrotechnical Laboratory, Japan, 1987.

24. Iannucci, R. A. *A Dataflow / von Neuamnn Hybrid Architecture*. Ph.D. Th., Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., (in preparation) 1987.

25. 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.

26. Kuck, D., E. Davidson, D. Lawrie, and A. Sameh. "Parallel Supercomputing Today and the Cedar Approach". *Science Magazine 231* (February 1986), 967-974.

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

28. Li, Z. and W. Abu-Sufah. A Technique for Reducing Synchronization Overhead in Large Scale Multiprocessors. Proc. of the 12th, International Symposium on Computer Architecture, June, 1985, pp. 284-291.

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

30. Nikhil, R. S., K. Pingali, and Arvind. Id Nouveau. Computation Structures Group Memo 265, Laboratory for Computer Science, MIT, Cambridge, Mass., Cambridge, MA 02139, July, 1986.

31. Papadopoulos, G. M. *Implementation of a General Purpose Dataflow Multiprocessor*. Ph.D. Th., Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., (in preparation) 1987.

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

33. Pfister, G. F., W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. Proceedings of the 1985 International Conference on Parallel Processing, Institute of Electrical and Electronics Engineers, Piscataway, N. J., 08854, August, 1985, pp. 764-771.

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

35. 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.