

Some Computer Organizations and Their Effectiveness

MICHAEL J. FLYNN, MEMBER, IEEE

Abstract—A hierarchical model of computer organizations is developed, based on a tree model using request/service type resources as nodes. Two aspects of the model are distinguished: logical and physical.

General parallel- or multiple-stream organizations are examined as to type and effectiveness—especially regarding intrinsic logical difficulties.

The overlapped simplex processor (SISD) is limited by data dependencies. Branching has a particularly degenerative effect.

The parallel processors [single-instruction stream-multiple-data stream (SIMD)] are analyzed. In particular, a nesting type explanation is offered for Minsky's conjecture—the performance of a parallel processor increases as $\log M$ instead of M (the number of data stream processors).

Multiprocessors (MIMD) are subjected to a saturation syndrome based on general communications lockout. Simplified queuing models indicate that saturation develops when the fraction of task time spent locked out (L/E) approaches $1/n$, where n is the number of processors. Resources sharing in multiprocessors can be used to avoid several other classic organizational problems.

Index Terms—Computer organization, instruction stream, overlapped, parallel processors, resource hierarchy.

INTRODUCTION

ATTEMPTS to codify the structure of a computer have generally been from one of three points of view: 1) automata theoretic or microscopic; 2) individual problem oriented; or 3) global or statistical.

In the microscopic view of computer structure, relationships are described exhaustively. All possible interactions and parameters are considered without respect to their relative importance in a problem environment.

Measurements made by using individual problem yardsticks compare organizations on the basis of their relative performances in a peculiar environment. Such comparisons are usually limited because of their ad hoc nature.

Global comparisons are usually made on the basis of elaborate statistical tabulations of relative performances on various jobs or mixtures of jobs. The difficulty here lies in the fact that the analysis is *ex post facto* and usually of little consequence in the architecture of the system since the premises on which they were based (the particular computer analyzed) have been changed.

The object of this paper is to reexamine the principal interactions within a processor system so as to generate a

more “macroscopic” view, yet without reference to a particular user environment. Clearly, any such effort must be sharply limited in many aspects; some of the more significant are as follows.

1) There is no treatment of I/O problems or I/O as a limiting resource. We assume that all programs of interest will either not be limited by I/O, or the I/O limitations will apply equally to all computer memory configurations. That is, the I/O device sees a “black box” computer with a certain performance. We shall be concerned with *how* the computer attained a performance potential, while it may never be realized due to I/O considerations.

2) We make no assessment of particular instruction sets. It is assumed that there exists a (more or less) ideal set of instructions with a basically uniform execution time—except for data conditional branch instructions whose effects will be discussed.

3) We will emphasize the notion of effectiveness (or efficiency) in the use of internal resources as a criterion for comparing organizations, despite the fact that either condition 1) or 2) may dominate a total performance assessment.

Within these limitations, we will first attempt to classify the forms or gross structures of computer systems by observing the possible interaction patterns between instructions and data. Then we will examine physical and logical attributes that seem fundamental to achieving efficient use of internal resources (execution facilities, memory, etc.) of the system.

CLASSIFICATION: FORMS OF COMPUTING SYSTEMS

Gross Structures

In order to describe a machine structure from a macroscopic point of view, on the one hand, and yet avoid the pitfalls of relating such descriptions to a particular problem, the stream concept will be used [1]. Stream in this context simply means a sequence of items (instructions or data) as executed or operated on by a processor. The notion of “instruction” or “datum” is defined with respect to a reference machine. To avoid trivial cases of parallelism, the reader should consider a reference instruction or datum as similar to those used by familiar machines (e.g., IBM 7090). In this description, organizations are categorized by the magnitude (either in space or time multiplex) of interactions of their instruction and data streams. This immediately gives rise to four broad classifications of machine organizations.

Manuscript received February 26, 1970; revised May 25, 1971, and January 21, 1972. This work was supported by the U. S. Atomic Energy Commission under Contract AT (11-1) 3288.

The author is with the Department of Computer Science, The Johns Hopkins University, Baltimore, Md. 21218.

1) The single-instruction stream–single-data stream organization (SISD), which represents most conventional computing equipment available today.

2) The single-instruction stream–multiple-data stream (SIMD), which includes most array processes, including Solomon [2] (Illiatic IV).

3) Multiple-instruction stream–single-data stream type organizations (MISD), which include specialized streaming organizations using multiple-instruction streams on a single sequence of data and the derivatives thereof. The plug-board machines of a bygone era were a degenerate form of MISD wherein the instruction streams were single instructions, and a derived datum (SD) passed from program step i to program step $i+1$ (MI).

4) Multiple-instruction stream–multiple-data stream (MIMD), which include organizations referred to as “multiprocessor.” Univac [3], among other corporations, has proposed various MIMD structures.

These are qualitative notations. They could be quantified somewhat by specifying the number of streams of each type in the organization or the number of instruction streams per data stream, or vice versa. But in order to attain a better insight into the notions of organization, let us formalize the stream view of computing. Consider a generalized system model consisting of a *requestor* and a *server*. For our purposes we will consider the requestor as synonymous with a program (or user) and the server as synonymous with the resources, both physical and logical, which process the elements of the program. Note that, for now, the distinction between a program and a resource is not precise; indeed, under certain conditions a resource may be a program and vice versa. Then a *problem* can be defined as a stream (or sequence) of requests for service (or resources). Since each request is, in general, a program and can also specify a sequence of requests, we have a request hierarchy.

Let P be a program. A program is defined simply as a request for service by a structured set of resources. P specifies a sequence of other (sub) requests: $R_1, R_2, R_3, R_4, \dots, R_n$, called tasks. While the tasks appear here as a strictly ordered set, in general the tasks will have a more complex control structure associated with them.

Each request, again, consists of a sequence of sub-requests (the process terminates only at the combinatorial circuit level). Regardless of level, any request R_i is a bifurcated function having two roles:

$$R = \{f_i^l, f_i^s\}$$

the logical role of the requestor f_i^l and the combined logical and physical role of a server f_i^s .

1) *Logical Role of Requestor*: The logical role of the requestor is to define a result given an argument and to define a control structure or precedence among the other tasks directly defined by the initiating program P . We

anticipate a hierarchy of requests, where the transition between the initiating level P and the next level tasks $\{R_i\}$ is viewed as the logical role of each of the R_i , while actual service is through a combination of a tree of lower level logical requests and eventual physical service at the leaves of the tree.

Thus consider

$$f_i^l(x, \tau) \rightarrow (y, \tau^*)$$

where f_i^l is a functional mapping (in a mathematical sense) of argument x into result y . Here, $x, y \in B$, where B is the set of values defined by the modulo class of the arithmetic (logical and physical notions of arithmetic should be identical). The τ and τ^* indicate logical time or precedence; τ is a Boolean control variable whose validity is established by one or more predecessor logical functions $\{f_i^l\}$.

The requirement for a τ control variable stems from the need for specification of the validity of f_i^l and its argument x . Notice that two tasks f_i^l, f_j^l are directly dependent if either $\tau_i = 1$ implies $\tau_j^* = 1$ or $\tau_j = 1$ implies $\tau_i^* = 1$. This precedence specification may be performed implicitly by use of a restrictive convention (e.g., by strict sequence)—that the physical time t at which the i th task control variable τ_i becomes valid, $t(\tau_i = 1)$, has $t(\tau_i = 1) \geq t(\tau_{i-1}^* = 1)$, for all i —or by explicit control of τ and τ^* .

That there can be no general way of rearranging the request sequence f_i^l (or finding an alternate sequence g_i^l) such that the precedence requirement vanishes, is a consequence of the composition requirement $f(g(x))$, intrinsic to the definition of a computable function. That is, f cannot be applied to an argument until $g(x)$ has been completed.

The notion of an explicit precedence control has been formalized by Leiner [16] and others by use of a precedence matrix.

Given an unordered set of requests (tasks) $\{f_j^l | 1 \leq j \leq n\}$, an $n \times n$ matrix is defined so that: $a_{ij} = 1$ if we require $t(\tau_j = 1) \geq t(\tau_i^* = 1)$, i.e., task f_i must be completed before f_j can be begun. Otherwise, $a_{ij} = 0$.

The matrix M so defined identifies the initial priority. By determining M^2 (in the conventional matrix product sense), secondary implicit precedence is determined. This process is continued until

$$M^{P+1} = 0.$$

The fully determined precedence matrix H is defined as

$$H = M + M^2 + M^3 + \dots + M^P, \quad P \leq n$$

where $+$ is the Boolean union operation: $(a+b)_{ij} = a_{ij} + b_{ij}$.

Thus H defines a scheduling of precedence among the n tasks. At any moment of logical time (τ_i), perhaps a set of tasks $\{f_k^l; k | \text{either } a_{jk} = 0, \text{ for all } j, \text{ or if } a_{jk} = 1, \text{ then } \tau_j^* = 1\}$ are independently executable.

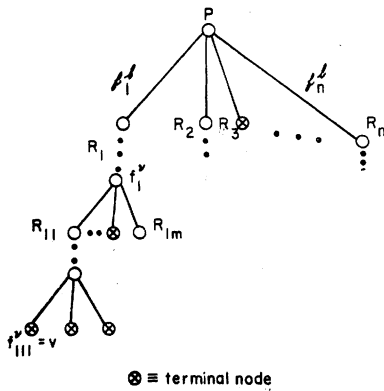


Fig. 1. Service hierarchy.

Since “task” and “instruction” are logically equivalent requests for service, the preceding also applies to the problem of detecting independent instructions in a sequence. In practice, tasks are conditionally issued. This limits the use of the precedence computation to the “while running” or dynamic environment.

2) *Service Role of Request*: Thus far we have been concerned with precedence or task control at a single level. The service for node R_i is defined by f_i^v , which is a subtree structure (see Fig. 1) terminating in the physical resources of the system, i.e., the physically executable primitives of the system. Thus f_i^v defines the transition from P to R_i and among R_i , ($i=1, \dots, n$), at a given level, while f_i^v defines the substructure under node R_i . Thus f_i^v is actually a hierarchy of subrequests terminating in a primitive physical service resource.

These terminal nodes are defined as a request for service lying within the physical resource vocabulary of the system, i.e.,

$$f_{ijk}^v \dots \equiv v \mid v \in V$$

where V is the set of available physical resources. Note that the request is generally for any element in a particular resource class rather than a specific resource v . The elements $\{v\}$ are usually of two types: operational or storage. A storage resource is a device capable of retaining a representation of a datum after the initial excitation is removed. The specification of a device is usually performed by coordinate location, contents, or implication. An operational resource is a combinational circuit primitive (e.g., add, shift, transfer, ...) that performs a (usually) binary mapping $S \times S \rightarrow S$; S the set storage (space) resource.

Strictly speaking, since a request for a physical resource v is a request for accessing that resource, there is no “request for storage resource” but rather a request to an allocation algorithm to access or modify the memory map. Thus a storage resource has operational characteristics if we include the accessing mechanism in the storage resource partition.

Partitions are defined on physical resources which

define a memory hierarchy (space) or a set of primitive operations. Note that since partitions may not be unique, resulting tree structures may also differ. Also note that while leaves of the tree must be requests for physical resources, higher level nodes may also be if all inferior nodes are physical.

Since physical time is associated with a physical activity, at the terminal nodes we have

$$f_{ijk}^v \dots (S \times S, t_b) \rightarrow S, t_f$$

where t_b is the initiation time and t_f is the completion time. Initiation is conditioned on the availability of operational resource v and the validity of both source operands ($\tau=1$). When the operation is complete, the result is placed in a specified sink location, and the control variable τ^* is set to “1.”

The advantage of this model is that it allows a perspective of a system at a consistent hierarchical level of control structure. One may replace the subtree with its equivalent physical execution time (actual or mean value) and mean operational and spatial resource requirements if task contention is a factor. The net effect is to establish a vector space with a basis defined by the reference of the observer. The vector of operations O_k may appear as physical resources at a particular level k in the tree, but actually may represent a subtree of requests—similarly with the storage vector S_k . The control structure defined by the set of request functions $\{f_{ki}^v \mid 1 \leq i \leq n\}$ will determine the program structure as an interaction of resources on $O_k \times S_k$. The reader may note the similarity, in hierarchy treatment at least, between the preceding model and the general model of Bell and Newell [28].

Implicit in the foregoing statement is the notion that a physical resource exists in space-time, not space or time alone. For example, N requests may be made for an “add” operational resource—these may be served by N servers each completing its operation in time T or equivalently by one resource that operates in time T/N . Two parameters are useful in characterizing physical resources [1]—latency and bandwidth. *Latency* is the total time associated with the processing (from excitation to response) of a particular data unit at a phase in the computing process. *Bandwidth* is an expression of time-rate of occurrence. In particular, operational bandwidth would be the number of operand pairs processed per unit time.

If, as a hierarchical reference point, we choose operations and operands as used by, for example, an IBM 7090, we can explore arrangements of, and interactions between, familiar physical resources. The IBM 7090 itself has a trivial control tree. In particular, we have the SISD—single operational resource operating on a single pair of storage resources. The multiple-stream organizations are more interesting, however, as well as two considerations: 1) the latency for interstream com-

munication; and 2) the possibilities for high computational bandwidth within a stream.

Interstream Communications

There are two aspects of communications: operational resource accessing of a storage item ($\bar{O} \times \bar{S}$) and storage to storage transfer ($\bar{S} \times \bar{S}$). Both aspects can be represented by communications matrices each of whose entry t_{ij} is the time to transfer a datum in the j th storage resource to the i th resource (operational or storage). The operational communication matrix is quite useful for MIMD organizations, while the storage communications matrix is usually more interesting for SIMD organizations. An ($\bar{O} \times \bar{O}$) matrix can also be defined for describing MISD organizations.

An alternate form of the communications matrix, called the connection matrix, can also be developed for the square matrix cases. This avoids possibly large or infinite entries possible in the communications matrix (when interstream communications fail to exist). The reciprocal of the normalized access time t_{ii}/t_{ij} (assuming t_{ii} is the minimum entry for a row) is entered for the access time of an element of the i th data storage resource by the j th operational or storage resource d_{ij} . The minimum access time (resolution) is 1. If a particular item were inaccessible, there would be a zero entry. Notice that in comparing parallel organizations to the serial organization, the latter has immediate access to corresponding data. While it appears that under certain conditions an element expression can be zero due to lack of communication between resources, in practice this does not occur since data can be transferred from one stream to another in finite time, however slow. Usually such transfers occur in a common storage hierarchy.

Stream Inertia

It is well known that the action of a single-instruction stream may be telescoped for maximum performance by overlapping the various constituents of the execution of an individual instruction [4]. Such overlapping usually does not exceed the issuing of one instruction per instruction decode resolution time Δt . This avoids the possibly exponentially increasing number of decision elements required in such a decoder [1], [5]. A recent study [13] provides an analysis of the multiple-instruction issuing problem in a single-overlapped instruction stream. In any event, a certain number of instructions in a single-instruction stream are being processed during the latency time for one instruction execution. This number may be referred to as the confluence factor or inertia factor J of the processor per individual instruction stream. Thus the maximum performance per instruction stream can be enhanced by a factor J . If the average instruction execution time is $L \cdot \Delta t$ time units, the maximum performance per stream would be

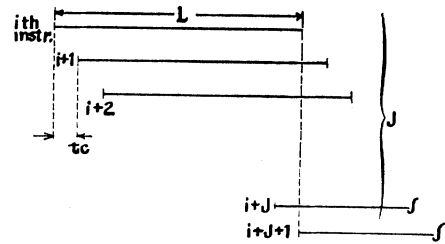


Fig. 2. Stream inertia.

$$\text{perf.}_{\max} = \frac{J}{L \cdot \Delta t}$$

This is illustrated in Fig. 2. Successive instructions are offset in this example by Δt time units.

System Classification

Then to summarize, a technology independent macroscopic specification of a large computing system would include: 1) the number of instruction streams and the number of data streams—the “instruction” and “data” unit should be taken with respect to a convenient reference; 2) the appropriate communications (or connection) matrices; and 3) the stream inertia factor J and the number of time units of instruction execution latency L .

EFFECTIVENESS IN PERFORMING THE COMPUTING PROCESS

Resolution of Entropy

Measures of the effectiveness are necessarily problem based. Therefore, comparisons between parallel and simplex organizations frequently are misleading since such comparisons can be based on different problem environments. The historic view of parallelism in problems is probably represented best by Amdahl [6] and is shown in Fig. 3. This viewpoint is developed by the observation that certain operations in a problem environment must be done in an absolutely sequential basis. These operations include, for example, the ordinary housekeeping operations in a program. In order to achieve any effectiveness at all, from this point of view, parallel organizations processing N streams must have substantially less than $1/N \times 100$ percent of absolutely sequential instruction segments. One can then proceed to show that typically for large N this does not exist in conventional programs. A major difficulty with this analysis lies in the concept of “conventional programs” since this implies that what exists today in the way of programming procedures and algorithms must also exist in the future. Another difficulty is that it ignores the possibility of overlapping some of this sequential processing with the execution of “parallel” tasks.

To review this problem from a general perspective, consider a problem in which N_1 words each of p bits

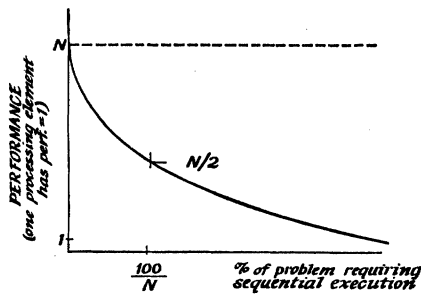


Fig. 3. "Parallelism" in problems.

serve as input data. The program or algorithm operates on this input data and produces output results corresponding to N_2 words each of p bits. If we presume that as a maximum each of the input bits could affect each of the bits in the output, then there are $N_1 \times p$ bits of uncertainty or entropy which specify an output bit. Thus a table-oriented solution algorithm (i.e., generate a table of solutions—one solution for each input combination) requires 2^{pN_1} entries and uses pN_2 bits per entry. The total number of bits required is $pN_2 2^{pN_1}$.

We could generalize this for variable length input and output by treating N_2 and N_1 as the maximum number of words required to specify output or input information. Note that this imposes a limit on the size of output strings, and hence the table algorithm is too restrictive for generalized computable functions. However, within this restricted class the entropy Q to be resolved by an algorithm is

$$pN_2 \leq Q \leq pN_2 \cdot 2^{pN_1}.$$

The lower bound also needs special interpretation for trivial algorithms (e.g., strings of identical ones); the pN_2 bits were assumed to be independently derived. Hence the pN_2 bits must be reduced by their dependency. (See Kolmogorov [10] for an alternate notion on the specification of Q .)

In any event, Q bits of uncertainty must be resolved. This resolution may be performed in space or time. Typically it is resolved in space by combinatorial logic. Each decision element may resolve between zero and one bit of information depending upon the associated probabilities of the binary outcomes. Usually a useful repertoire or vocabulary of logical decisions is available to an instruction stream. Let an element of the vocabulary consist of M decisions. Depending upon the type of operation to be performed, these execution unit decision elements resolve less than M bits of information; thus

$$Q \leq m' M N_t$$

where m' is the number of data stream execution units and N_t is the number of time operations that were used (i.e., number of sequential instructions). In order to retire the complete algorithm, of course, a sequence of operations is performed to execute the instruction stream; each operation in the sequence resolves or exceeds the required amount of entropy for a solution.

Thus from this most general point of view there is little difference in the resolution of entropy in space or time. The reader will note, however, that space resolution is not necessarily as efficient as time sequences. In fact, the number of time sequence operations N_t is a linear function of input size n :

$$N_t = k \cdot n$$

while Muller [32] has shown that for combinational circuits of arbitrary functional basis a measure for the number of circuits required N_s is

$$k_1 \frac{2^n}{n} \leq N_s \leq k_2 \frac{2^n}{n}$$

where n is the number of input lines. See Cook [33] and Cook and Flynn [34] for a more general and complete treatment of space-time functional measures. Later in this paper we will discuss similar inefficiencies in parallel SIMD processors.

Recursive Functions

Substantial difficulties arise when the preceding "general point of view" is reduced to the specific. In particular, the class of functions for which arbitrary "space-time" transformations can be developed is not equivalent to the class of all recursive (computable) functions. Recursion (in Kleene's sense [7]) is based on the application of the composition operation on a finite set of initial functions. This composition operation [8] is the association of functions $h(X_1, X_2, \dots, X_n)$ with $F(X_1, \dots, X_n)$, $g_1(X_1, \dots, X_n)$, $g_2(X_1, \dots, X_n)$, \dots , $g_n(X_1, \dots, X_n)$, so that $h(X_1, \dots, X_n) = F(g_1(X_1, \dots, X_n), \dots, g_n(X_1, \dots, X_n))$. Clearly, the application of F on the results of g_i is a sequential operation. Any general attempts to resolve the functional entropy without such sequential (time) dependence leads to "recursion" based on infinite sets of initial functions.

Bernstein [9] has developed three (rather strong) sufficient conditions for two programs to operate in parallel based on the referencing of partitions of storage. As in the preceding discussion, these conditions specify limitations on the interactions between programs.

Notice that none of the foregoing serves to limit the capability of a processor organized in a parallel fashion to perform computation, but rather serves as a limit on the efficiency of such an operation. Also note that the composing function F may induce similar inefficiencies in a confluent simplex processor (depending on the nature of F and the last g_i to be evaluated). Such performance degradation will be discussed later. The composition mechanism that causes a problem here is an interprogram action, while the stream inertia difficulty occurs more prominently in purely intraprogram conditional actions. Indeed, there are techniques for the elimination of branches in simple programs by use of Boolean test variables (0 or 1) operating multiplica-

tively on each of two alternate task paths [14]. This is a direct "branch" to "composition" transformation.

SYSTEM ORGANIZATIONS AND THEIR EFFECTIVENESS
IN RESOURCE USE

SISD and Stream Inertia

Serious inefficiencies may arise in confluent SISD organizations due to turbulence when data interacts with the instruction stream. Thus an instruction may require an argument that is not yet available from a preceding instruction (direct composition), or (more seriously) an address calculation is incomplete (indirect composition). Alternately, when a data conditional branch is issued, testable data must be available before the branch can be fully executed (although both paths can be prepared). In conventional programs delays due to branching usually dominate the other considerations; then in the following we make the simplified assumption that inertia delay is due exclusively to branching. In addition to providing a simple model of performance degradation, we can relate it to certain test data derived from a recent study. The reader should beware, however, that elimination of branching by the introduction of composition delay will not alter the turbulence situation. For a machine issuing an instruction per Δt , if the data interaction with the stream is determined by the i th instruction and the usage of such data is required by the $(i+1)$ th instruction, a condition of maximum turbulence is encountered, and this generates the maximum serial latency time $(L-1)\Delta t$ which must be inserted into the stream until the overlapped issuing conditions can be reestablished.

If we treat the expected serial latency time of an instruction $L \cdot \Delta t$ as being equivalent to the total execution time for the average instruction (from initiation to completion), we must also consider an anticipation factor N as the average number of instructions between (inclusively) the instruction stating a condition and the instruction which tests or uses this result. Clearly, for $N \geq J/L$ instructions no turbulence (or delay) will result.

Thus under ideal conditions one instruction each $L \cdot \Delta t / J$ (time units) would be executed. Turbulence adds a delay:

$$\text{delay} = \left(L - \frac{NL}{J} \right) \Delta t, \quad \text{for } N < \frac{J}{L}$$

$$= 0, \quad \text{for } N \geq \frac{J}{L}.$$

Given a block of M instructions with a probability of encountering a turbulence causing instruction p the total time to execute these instructions would be

$$T = \left[\frac{L}{J} [M(1-p)] + 1 + pM \left(L - \frac{NL}{J} \right) \right] \Delta t.$$

The additional "1" in the expression is due to the

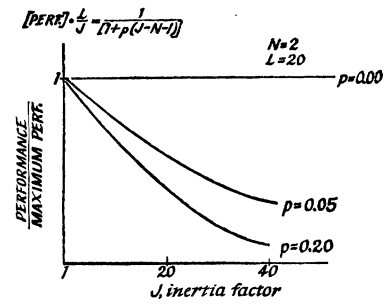


Fig. 4. Stream inertia and effects of branch.

startup of the instruction overlapping. If we define performance as the number of instructions executed per unit time, then

$$\text{perf.} = \frac{M}{T} = \frac{M}{\left[\frac{L}{J} [M(1-p)] + pM \left(L - \frac{NL}{J} \right) + 1 \right] \Delta t}$$

$$= \frac{1}{\frac{L \cdot \Delta t}{J} \left[(1-p) + p(J-N) + \frac{J}{L \cdot M} \right]}$$

The last term in the denominator drops out as M becomes large. Then

$$\text{perf.} = \frac{J}{L \Delta t} \cdot \frac{1}{[1 + p(J-N-1)]}$$

Fig. 4 shows the effect of turbulence probability p for various J and L . In particular, if $J=20$ and L were 20 time units, $N=2$ instructions and a turbulence probability of 10 percent, the performance of a system would degrade from a maximum possible of 1 (instructions/time unit) to $1/2.7$ or about 30 percent of its potential.

A major cause of turbulence in conventional programs is the conditional branch; typically the output of a computer would include 10–20-percent conditional branches. A study by O'Regan [12] on the branch problem was made using the foregoing type of analysis. For a typically scientific problem mix (five problems: root finding; ordinary differential equation; partial differential equations; matrix inversion; and Polish string manipulation), O'Regan attempted to eliminate as many data conditional branches as possible using a variety of processor architectures (single and multiple accumulators, etc.). O'Regan did not resort to extreme tactics, however, such as multiplying loop sizes or transformations to a Boolean test (mentioned earlier). For four of the problems selected the best (i.e., minimum) conditional branch probability attainable varied from $p=0.02$ to $p=0.10$. The partial differential equation results depend on grid size, but $p=0.001$ seems attainable. The best (largest) attainable N (the set-test offset) average was less than 3. No attempt was made in the study to evaluate degradation due to other than branch dependent turbulence.

SIMD and Its Effectiveness

There are three basic types of SIMD processors, that is, processors characterized by a master instruction applied over a vector of related operands. These include (Fig. 5) the following types.

1) *The Array Processor*: One control unit and m directly connected processing elements. Each processing element is independent, i.e., has its own registers and storage, but only operates on command from the control unit.

2) *The Pipelined Processor*: A time-multiplexed version of the array processor, that is, a number of functional execution units, each tailored to a particular function. The units are arranged in a production line fashion, staged to accept a pair of operands every Δt time units. The control unit issues a vector operation to memory. Memory is arranged so that it is suitable to a high-speed data transfer and produces the source operands which are entered a pair every Δt time units into the designated function. The result stream returns to memory.

3) *The Associative Processor*: This is a variation of the array processor. Processing elements are not directly addressed. The processing elements of the associative processor are activated when a generalized match relation is satisfied between an input register and characteristic data contained in each of the processing elements. For those designated elements the control unit instruction is carried out. The other units remain idle.

A number of difficulties can be anticipated for the SIMD organization. These would include the following problems.

- 1) Communications between processing elements.
- 2) *Vector Fitting*: This is the matching of size between the logical vector to be performed and the size of the physical array which will process the vector.
- 3) *Sequential Code (Nonvector)*: This includes house-keeping and bookkeeping operations associated with the preparation of a vector instruction. This corresponds to the Amdahl effect. Degradation due to this effect can be masked out by overlapping the sequential instructions with the execution of vector type instructions.
- 4) *Degradation Due to Branching*: When a branch point occurs, several of the executing elements will be in one state, and the remainder will be in another. The master controller can essentially control only one of the two states; thus the other goes idle.
- 5) Empirically, Minsky and Papert [29] have observed that the SIMD organization has performance proportional to the $\log_2 m$ (m , the number of data streams per instruction stream) rather than linear. If this is generally true, it is undoubtedly due to all of the preceding effects (and perhaps others). We will demonstrate an interpretation of it based upon branching degradation.

Communication in SIMD organizations has been

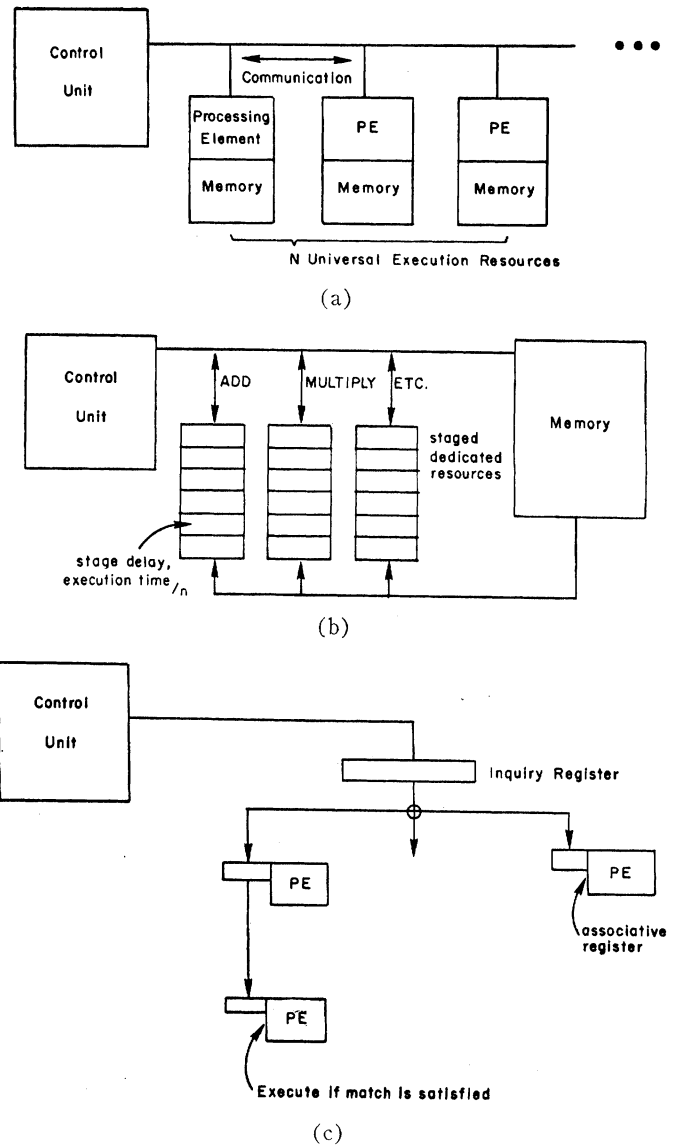


Fig. 5. SIMD processors. (a) Array processor. (b) Pipelined processor. (c) Associative processor.

widely studied [17]–[20]. Results to date, however, indicate that it is not as significant a problem as was earlier anticipated. Neuhauser [17], in an analysis of several classical SIMD programs, noted that communications time for an array-type organization rarely exceeded 40 percent of total job time and for the matrix inversion case was about 15 percent.

The fitting problem is illustrated in Fig. 6. Given a source vector of size m , performance is effected in an array processor when the M physical processing elements do not divide m [21]. However, so long as m is substantially larger than M , this effect will not contribute significant performance degradation. The pipeline processor exhibits similar behavior, as will be discussed later.

The Amdahl effect is caused by a lack of “parallelism” in the source program; this can be troublesome in any multistream organization. Several SIMD organizations

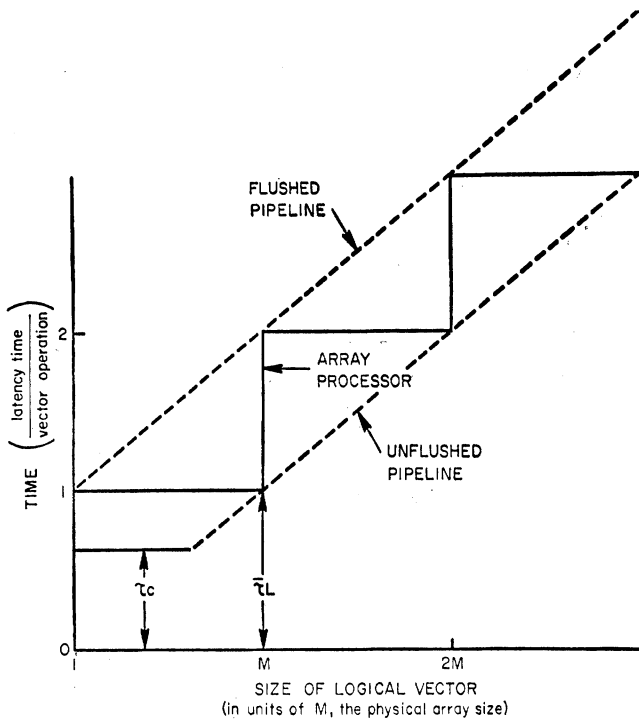


Fig. 6. Fitting problem.

use overlapping of “sequential type” control unit instructions with “vector operations” to avoid this effect, with some apparent success.

Multiple-execution organizations such as SIMD have potential difficulty in the use of the execution resources. The reason for this is that all units must process the same instruction at a particular unit of time. When nested decisions are considered (Fig. 7), difficulty arises because the execution units are not available to work on any other task.

Consider an SIMD system with M data streams and an average instruction execution time (per data stream) of $L \cdot \Delta t$ time units. Now a single instruction will act uniformly on M pairs of operands. With respect to our reference instruction I (which operates on only a pair of operands) the SIMD instruction, designated I^* , has M times the effect. Ignoring possible overlap, a single I^* instruction will be executed in a least time $L \cdot \Delta t$, while the conventional unoverlapped SISD system would execute in $M \cdot L \cdot \Delta t$.

To achieve close to the $1/M$ bound, the problem must be partitionable in M identical code segments. When a conditional branch is encountered if at least one of the M data differs in its condition, the alternate path instructions must be fully executed. We now make a simplifying assumption: the number of source instructions are the same for the primary branch path and the alternate. Since the number of data items required to be processed by a branch stream is M , only the fraction available will be executed initially and the task will be reexecuted for the remainder. Thus a branch identifying

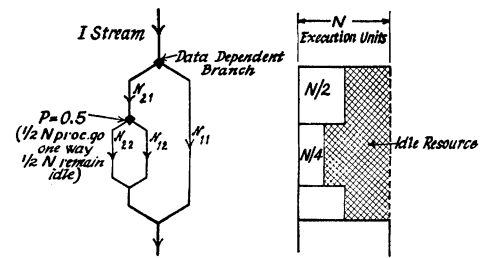


Fig. 7. SIMD branching.

two separate tasks, each of length N , will take twice the amount of time as their unconditional expectation. Thus the time to execute a block of N^* source instructions (each operating on M data streams) with equal probability on primary and alternate branch paths is

$$T = L \cdot \sum_i N_{i,0} + L \cdot \sum_i N_{i,1} \cdot 2 + L \cdot \sum_i N_{i,2} \cdot 4 + \dots + L \cdot \sum_i N_{i,j} \cdot 2^j$$

$$T = L \cdot \sum_j \sum_i N_{i,j} 2^j$$

where j is the level of nesting of a branch and $N_{i,j}$ is the number of source instructions in the primary (or alternate) branch path for the i th branch at level j . As the level of nesting increases, fewer resources are available to execute the M data streams, and the time required increases—the situation would be similar if we had used other branch probabilities ($p \neq 0.5$). Thus the performance is

$$\text{perf.} = \frac{I}{T} = \frac{N^*}{L \cdot \sum_j \sum_i N_{i,j} 2^j}$$

The factor $\sum_i N_{i,j} / N^*$ is the probability of encountering a source instruction:

$$P_j = \sum_i N_{i,j} / N^* \leq 1.$$

This analysis assumed that the overhead for reassigning execution elements to alternate paths tasks is prohibitive. This is usually true when the task size $N_{i,j}$ is small or when the swapping overhead is large (an array processor, each of whose data streams has a private data storage). Based on empirical evaluation of program performance in a general scientific environment (i.e., not the well-known “parallel type” programs such as matrix inversion, etc.) it has been suggested [29] that the actual performance of the SIMD processor is proportional to the \log_2 of the number of slave processing elements rather than the hoped for linear relation. This has been called Minsky’s conjecture:

$$\text{perf.}_{\text{SIMD}} \approx \log_2 M.$$

While this degradation is undoubtedly due to many causes, it is interesting to interpret it as a branching

degradation. Assume that the probability of being at one of the lower levels of nesting is uniform. That is, it is equally likely to be at level 1, 2, \dots , $\lceil \log_2 M \rceil$. Since beyond this level of nesting no further degradation occurs, assume $P_1 = P_2 = \dots = P_j = P_{\lceil \log_2 M \rceil - 1}$ and $P_1 = \sum_{k=\lceil \log_2 M \rceil}^{\infty} P_k$. Now

$$P_j = \frac{1}{\lceil \log_2 M \rceil} \begin{cases} j = \lceil \log_2 M \rceil - 1 \\ j = 0 \end{cases}$$

and the earlier performance relation can be restated:

$$\text{perf.} = \frac{1}{L} \cdot \frac{1}{\sum_j P_j 2^j}$$

This was derived for an absolute model, i.e., number of SIMD instructions per unit time. If we wish to discuss performance *relative* to an SISD unoverlapped processor with equivalent latency characteristics, then

$$\text{perf. relative} = \frac{M}{\sum_j P_j 2^j}$$

Thus the SIMD organization is M times faster if we have no degradation. Now

$$\text{perf. relative} = \frac{M}{\sum_j \frac{2^j}{\lceil \log_2 M \rceil}}$$

or, ignoring the effect of nonintegral values of $\log_2 M$

$$\text{perf. relative} = \frac{M}{\frac{2M - 1}{\log_2 M}}$$

for M large:

$$\text{perf. relative} \approx \frac{\log_2 M}{2}$$

Note that if we had made the less restrictive assumption that

$$P_j = 2^{-j}$$

then

$$\text{perf. relative} \approx \frac{M}{\log_2 M}$$

Thus we have two plausible performance relations based on alternate nesting assumptions. Of course this degradation is not due to idle resources alone; in fact, programs can be restructured to keep processing elements busy. The important open question is whether these restructured programs truly enhance the performance of the program as distinct from just keeping the resource busy. Empirical evidence suggests that the most effi-

cient single-stream program organization for this larger class of problems is presently substantially more efficient than an equivalent program organization suited to the SIMD processors. Undoubtedly this degradation is a combination of effects; however, branching seems to be an important contributor—or rather the ability to efficiently branch in a simple SISD organization substantially enhances its performance.

Certain SIMD configurations, e.g., pipelined processors, which use a common data storage may appear to suffer less from the nested branch degradation, but actually the pipelined processor should exhibit an equivalent behavior. In a system with source operand vector $\mathbf{A} = \{a_s, a_1, \dots, a_i, \dots, a_n\}$ and $\mathbf{B} = \{b_0, b_1, \dots, b_i, \dots, b_n\}$, a sink vector $\mathbf{C} = \{c_0, c_1, \dots, c_i, \dots, c_n\}$ is the resultant. Several members of \mathbf{C} will satisfy a certain criterion for a type of future processing, and others will not. Elements failing this criterion are tagged and not processed further, but the vector \mathbf{C} is usually left unaltered. If one rearranges \mathbf{C} , filters the dissenting elements, and compresses the vector, then an overhead akin to task swapping the array processor is introduced. Notice that the automatic hardware generation of the compressed vector is not practical at the high data rates required by the pipeline.

If the pipelined processor is logically equivalent to other forms of SIMD, how does one interpret the number of data streams? This question is related to the vector fitting problem. Fig. 6 illustrates the equivalence of an array processor to the two main categories of pipeline processors.

1) *Flushed*: The control unit does not issue the next vector instruction until the last elements of the present vector operation have completed their functional processing (gone through the last stage of the functional pipeline).

2) *Unflushed*: The next vector instruction is issued as soon as the last elements of the present vector operation have been initiated (entered the first state of the pipeline).

Assuming that the minimum time for the control unit to prepare a vector instruction τ_c is less than the average functional unit latency $\bar{\tau}_L$, for the flushed case the equivalent number of data streams per instruction stream m is

$$m = \frac{\bar{\tau}_L}{\Delta t} \text{ flushed pipeline}$$

where Δt is the average stage time in the pipeline.

With the unflushed case, again assuming the $\bar{\tau}_L > \tau_c$, the equivalent m is

$$m = \frac{\tau_c}{\Delta t} \text{ unflushed pipeline.}$$

Notice that when $\tau_c = \Delta t$, $m = 1$, and we no longer have

SIMD. In fact, we have returned to the overlapped SISD.

MIMD and Its Effectiveness

The multiple-instruction stream organizations (the “multiprocessors”) include at least two types.

1) *True Multiprocessors*: Configurations in which several physically complete and independent SI processors share storage at some level for the cooperative execution of a multitask program.

2) *Shared Resource Multiprocessor*: As the name implies, skeleton processors are arranged to share the system resources. These arrangements will be discussed later.

Traditional MIMD organizational problems include: 1) communications/composition overhead; 2) cost increases linearly with additional processors, while performance increases at a lesser rate (due to interference); and 3) providing a method for dynamic reconfiguration of resources to match changing program environment, (critical tasks)—this is related to 1).

Shared resource organization may provide limited answers to these problems, as we will discuss later.

Communications and composition is a primary source of degradation in MI systems. When several instruction streams are processing their respective data streams on a common problem set, passing of data points is inevitable. Even if there is naturally a favorable precedence relationship among parallel instruction streams insofar as use of the data is concerned, composition delays may ensue, especially if the task execution time is variable. The time one instruction stream spends waiting for data to be passed to it from another is a macroscopic form of the strictly sequential problem of one instruction waiting for a condition to be established by its immediate predecessor.

Even if the precedence problem (which is quite program dependent) is ignored, the “lockout” problem associated with multiple-instruction streams sharing common data may cause serious degradation. Note that multiple-instruction stream programs without data sharing are certainly as sterile as a single-instruction stream program without branches.

Madnick [11] provides an interesting model of software lockout in the MIMD environment. Assume that an individual processor (instruction stream control unit) has expected task execution time (without conflicts) of E time units. Suppose a processor is “locked out” from accessing needed data for L time units. This locking out may be due to interstream communications (or accessing) problems (especially if the shared storage is an I/O device). Then the lockout time for the j th processor (or instruction stream) is

$$L_j = \sum_i p_{ij} T_{ij}$$

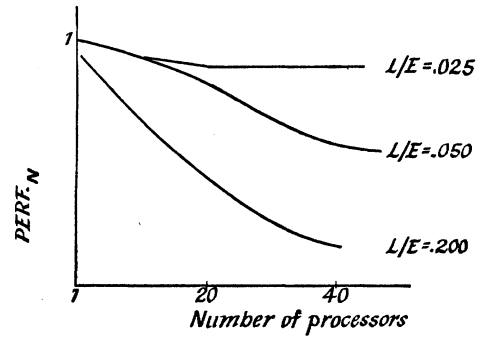


Fig. 8. MIMD lockout.

where T_{ij} is the communications time discussed earlier and p_{ij} is the probability of task j accessing data from data stream i . Note that the lockout here may be due to the broader communications problem of the j th processor requesting a logical data stream i . This includes the physical data stream accessing problem as well as additional sources of lockout due to control, allocation, etc.

In any event, Madnick [11] used a Markov model to derive the following relationship:

$$\epsilon (\text{idle}) = \sum_{i=2}^n \frac{(i-1)}{\left(\frac{E}{L}\right)^i (n-i)!} \bigg/ \sum_{i=0}^n \frac{1}{\left(\frac{E}{L}\right)^i (n-i)!}$$

where ϵ (idle) is the expected number of locked-out processors and n is the total number of processors. If a single processor has unit performance, then for n processors

$$\text{perf.} = n - \epsilon (\text{idle})$$

and normalized performance (max = 1) is given by

$$\text{perf.}_N = \frac{n - \epsilon (\text{idle})}{n}$$

Fig. 8 is an evaluation of the normalized performance as the number of processors (instruction stream–data stream pairs) are increased for various interaction activity ratios L/E .

Regis [15] has recently completed a study substantially extending the simple Markovian model previously described (homogeneous resources, identical processors, etc.) by developing a queuing model that allows for vectors of requests to a vector of service resources. Lehman [30] presents some interesting simulation results related to the communications interference problem.

Since shared resource MIMD structures provide some promising (though perhaps limited) answers to the MI problems, we will outline these arrangements. The execution resources of an SISD overlapped computer (adders, multiplier, etc.—most of the system exclusive of registers and minimal control) are rarely efficiently used, as discussed in the next section.

In order to effectively use this execution potential,

consider the use of multiple-skeleton processors sharing the execution resources. A "skeleton" processor consists of only the basic registers of a system and a minimum of control (enough to execute a load or store type instruction). From a program point of view, however, the skeleton processor is a complete and independent logical computer.

These processors may share the resources in space [22], or time [23]–[25]. If we completely rely on space sharing, we have a "cross-bar" type switch of processors—each trying to access one of n resources. This is usually unsatisfactory since the cross-bar switching time overhead can be formidable. On the other hand, time-phased sharing (or time-multiplexing) of the resources can be attractive in that no switching overhead is involved and control is quite simple if the number of multiplexed processors are suitably related to each of the pipelining factors. The limitation here is that again only one of the m available resources is used at any one moment.

A possible optimal arrangement is a combination of space-time switching (Fig. 9). The time factor is the number of skeleton processors multiplexed on a time-phase ring, while the space factor is the number of multiplexed processor "rings" K which simultaneously request resources. Note that K processors will contend for the resources and, up to $K - 1$, may be denied service at that moment. Thus a rotating priority among the rings is suggested to guarantee a minimum performance. The partitioning of the resources should be determined by the expected request statistics.

When the amount of "parallelism" (or number of identifiable tasks) is less than the available processors, we are faced with the problem of accelerating these tasks. This can be accomplished by designing certain of the processors in each ring with additional staging and interlock [13] (the ability to issue multiple instructions simultaneously) facilities. The processor could issue multiple-instruction execution requests in a single-ring revolution. For example, in a ring $N = 16$, 8 processors could issue 2 request/revolutions, or 4 processors could issue 4 request/revolutions; or 2 processors could issue 8 request/revolutions; or 1 processor could issue 16 request/revolutions. This partition is illustrated in Fig. 10. Of course mixed strategies are possible. For a more detailed discussion the reader is referred to [25], [26], and [31].

SOME CONSIDERATIONS ON SYSTEMS RESOURCES

The gross resources of a system consist of execution, instruction control, primary storage, and secondary storage.

Execution Resources

The execution resources of a large system include the decision elements which actually perform the operations

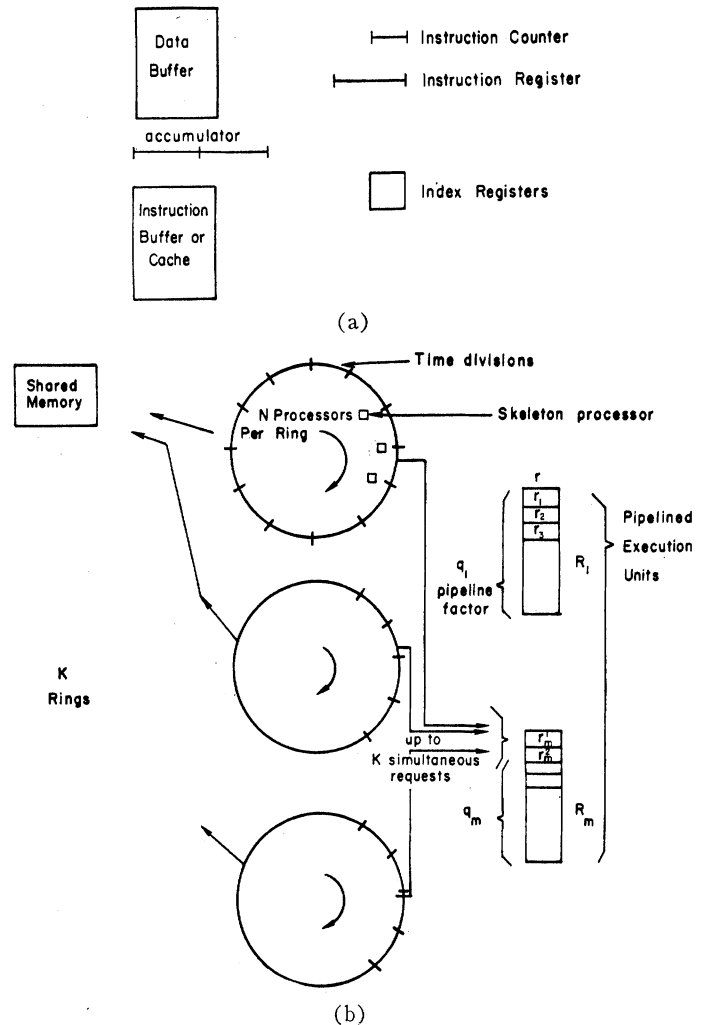


Fig. 9. (a) Skeleton processor. (b) Time-multiplexed multiprocessing.

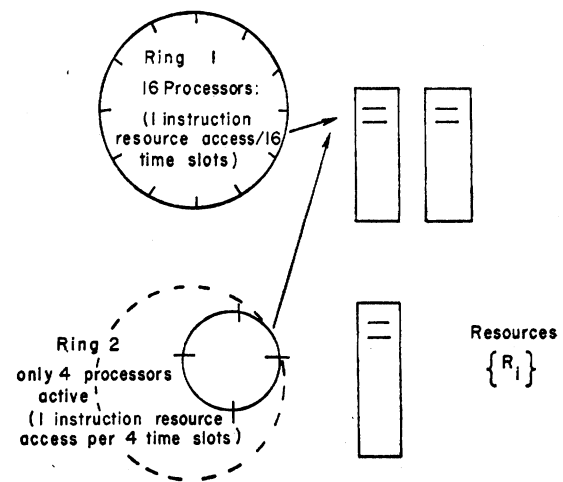


Fig. 10. Subcommutation of processors on ring.

implied in the instruction upon the specified operands. The execution bandwidth of a system is usually referred to as being the maximum number of operations that can be performed per unit time by the execution area. Notice that due to bottlenecks in issuing instructions, for example, the execution bandwidth is usually substantially in excess of the maximum performance of a sys-

tem. This overabundance of execution bandwidth has become a major characteristic of large modern systems.

In highly confluent SISD organizations specific execution units are made independently and designed for maximum performance. The autonomy is required for ease of implementation since from topological considerations an individual independent unit executing a particular class of operands gives much better performance on the class than an integrated universal type of unit [1]. The other notable characteristic of the SISD organization is that each of these independent subunits must in itself be capable of large or high bandwidths since the class of operations which successive instructions perform are not statistically independent, and in fact they are usually closely correlated; thus, for example, systems like the CDC 6600 and the IBM 360/91 both have execution facilities almost an order of magnitude in excess of the average overall instruction retirement rate. Fig. 11 illustrates the bandwidth concept. Given N specialized execution units, with execution pipelining factor P_i for the i th unit (the pipelining factor is the number of different operations being performed by one unit at one time—it is the execution analog of confluence), then if t_i is the time required to fully execute the i th type of operation, the execution bandwidth is

$$\text{execution bandwidth} = \sum_{i=1}^N \frac{P_i}{t_i}.$$

Note the bandwidth is in reality a vector partitioned by the resource class i . We sum the components as a scalar to assess gross capability. Notice that the shared resource MI organizations are, by definition, optimized for the efficient use of the execution resources.

Instruction Control

The control area is responsible for communications in addition to operational control. The communication function is essentially a process of identification of operand sink and source positions. The control area of a system is proportionally much larger in number of decision elements when confluence is introduced in the instruction stream since additional entropy must be resolved due to possible interactions between successive instructions. These precedence decisions must be made to assure normal sequential operation of the system. The analog situation is present in many parallel systems. For example, in SIMD those data streams which are activated by a particular instruction stream must be identified as well as those which do not participate. Notice that elaborations for controls, whether be it due to confluence or parallelism, basically resolve no entropy with respect to the original absolutely sequential instruction stream (and hence none with respect to the problem).

The necessary hardware to establish these sophistications is strictly in the nature of an overhead for the premium performance. From an instruction unit or con-

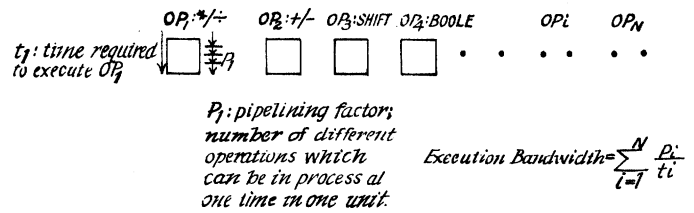


Fig. 11. Execution bandwidth.

trol unit point of view, the maximum decision efficiency is generated when the control unit has relatively simple proportions (i.e., handles a minimum number of interlocks or exceptional conditions—as when J is small per instruction stream) and when it is being continually utilized (a minimum idle time due to interstream or intrastream turbulence or lockout). While shared resource MI organizations have an advantage over the confluent SISD and the SIMD arrangements insofar as complexity of control, the control must be repeated M times. An overlapped SIMD processor would probably have the simplest control structure of the three.

Primary and Secondary Storage

The optimization of storage as a resource is a relatively simple matter to express. The task, both program and data, must move through the storage as expeditiously as possible; thus the less time a particular task spends in primary storage, the more efficiently the storage has been used with respect to this particular resource.,

In essence we have storage efficiency measured by the space-time product of problem usage at each level of the storage hierarchy. The “cost” of storage for a particular program can be defined as

$$\text{storage cost} = \sum_i c_i \cdot s_i \cdot t_i$$

where i is the level of storage hierarchy, c_i is the cost per word at that level, s_i is the average number of words of the program used, and t_i is time spent at that level.

While the preceding overly simplifies the situation by ignoring the dynamic nature of storage requirements, some observations can be made. The MI organizational structure, by nature, will require both task program and data sets for each of the instruction units to be simultaneously available at low levels of the hierarchy. The SIMD arrangement requires only simultaneous access to the M data sets, while the SISD has the least intrinsic storage demands. Thus in general

$$s_1 |_{\text{MIMD}} \geq s_1 |_{\text{SIMD}} \geq s_1 |_{\text{SISD}}.$$

Thus the MIMD and SIMD must be, respectively, more efficient in program execution t_i to have optimized the use of the storage resource.

ACKNOWLEDGMENT

The author is particularly indebted to C. Neuhauser, R. Regis, and G. Tjaden, students at The Johns Hopkins

University, for many interesting discussions on this subject. In particular, the hierarchical model presented here contains many thoughtful suggestions of R. Regis. The analysis of SIMD organizations was substantially assisted by C. Neuhauser. The author would also like to thank Prof. M. Halstead and Dr. R. Noonan of Purdue University for introducing him to the work of Aiken [14].

REFERENCES

- [1] M. J. Flynn, "Very high-speed computing systems," *Proc. IEEE*, vol. 54, pp. 1901-1909, Dec. 1966.
- [2] D. L. Slotnick, W. C. Borch, and R. C. McReynolds, "The Soloman computer—a preliminary report," in *Proc. 1962 Workshop on Computer Organization*. Washington, D. C.: Spartan, 1963, p. 66.
- [3] D. R. Lewis and G. E. Mellen, "Stretching LARC's, capability by 100—a new multiprocessor system," presented at the 1964 Symp. Microelectronics and Large Systems, Washington, D. C.
- [4] W. Buchholz, Ed., *Planning a Computer System*. New York: McGraw-Hill, 1962.
- [5] D. N. Senzig, "Observations on high performance machines," in *1967 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 31. Washington, D. C.: Thompson, 1967.
- [6] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *1967 Spring Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 30. Washington, D. C.: Thompson, 1967, p. 483.
- [7] Kleene, "A note on recursive functions," *Bull. Amer. Math. Soc.*, vol. 42, p. 544, 1936.
- [8] M. Davis, *Computability and Unsolvability*. New York: McGraw-Hill, 1958, p. 36.
- [9] A. J. Bernstein, "Analysis of programs for parallel processing," *IEEE Trans. Electron. Comput.*, vol. EC-15, pp. 757-763, Oct. 1966.
- [10] A. N. Kolmogorov, "Logical basis for information theory and probability theory," *IEEE Trans. Inform. Theory*, vol. IT-14, pp. 662-664, Sept. 1968.
- [11] S. E. Madnick, "Multi-processor software lockout," in *Proc. 1968 Ass. Comput. Mach. Nat. Conf.*, pp. 19-24.
- [12] M. E. O'Regan, "A study on the effect of the data dependent branch on high speed computing systems," M.S. thesis, Dep. Ind. Eng., Northwestern Univ., Evanston, Ill., 1969.
- [13] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of independent instructions," *IEEE Trans. Comput.*, vol. C-19, pp. 889-895, Oct. 1970.
- [14] Aiken, *Dynamic Algebra*, see R. Noonan, "Computer programming with a dynamic algebra," Ph.D. dissertation, Dep. Comput. Sci., Purdue Univ., Lafayette, Ind., 1971.
- [15] R. Regis, "Models of computer organizations," The Johns Hopkins Univ., Baltimore, Md., Comput. Res. Rep. 8, May 1971.
- [16] A. L. Leiner, W. A. Notz, J. L. Smith, and R. B. Marimont, "Concurrently operating computer systems," *IFIPS Proc.*, UNESCO, 1959, pp. 353-361.
- [17] C. Neuhauser, "Communications in parallel processors," The Johns Hopkins Univ., Baltimore, Md., Comput. Res. Rep. 18, Dec. 1971.
- [18] H. S. Stone, "The organization of high-speed memory for parallel block transfer of data," *IEEE Trans. Comput.*, vol. C-19, pp. 47-53, Jan. 1970.
- [19] M. C. Pease, "An adaptation of the fast Fourier transform for parallel processing," *J. Ass. Comput. Mach.*, vol. 15, pp. 252-264, Apr. 1968.
- [20] —, "Matrix inversion using parallel processing," *J. Ass. Comput. Mach.*, vol. 14, pp. 757-764, Oct. 1967.
- [21] T. C. Chen, "Parallelism, pipelining and computer efficiency," *Comput. Des.*, vol. 10, pp. 69-74, 1971.
- [22] P. Dreyfus, "Programming design features of the Gamma 60 computer," in *Proc. 1958 Eastern Joint Comput. Conf.*, pp. 174-181.
- [23] J. E. Thornton, "Parallel operation in control data 6600," in *1964 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 26. Washington, D. C.: Spartan, 1964, pp. 33-41.
- [24] R. A. Ashenbrenner, M. J. Flynn, and G. A. Robinson, "Intrinsic multiprocessing," in *1967 Spring Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 30. Washington, D. C.: Thompson, 1967, pp. 81-86.
- [25] M. J. Flynn, A. Podvin, and K. Shimizu, "A multiple instruction stream processor with shared resources," in *Parallel Processor Systems*, C. Hobbs, Ed. Washington, D. C.: Spartan, 1970.
- [26] M. J. Flynn, "Shared internal resources in a multiprocessor," in *1971 IFIPS Congr. Proc.*
- [27] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of independent instructions," *IEEE Trans. Comput.*, vol. C-19, pp. 889-895, Oct. 1970.
- [28] C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*. New York: McGraw-Hill, 1971.
- [29] M. Minsky and S. Papert, "On some associative, parallel, and analog computations," in *Associative Information Techniques*, E. J. Jacks, Ed. New York: Elsevier, 1971.
- [30] M. Lehman, "A survey of problems and preliminary results concerning parallel processing and parallel processors," *Proc. IEEE*, vol. 54, pp. 1889-1901, Dec. 1966.
- [31] C. C. Foster, "Uncoupling central processor and storage device speeds," *Comput. J.*, vol. 14, pp. 45-48, Feb. 1971.
- [32] D. E. Muller, "Complexity in electronic switching circuits," *IEEE Trans. Electron. Comput.*, vol. EC-5, pp. 15-19, Mar. 1956.
- [33] R. Cook, "Algorithmic measures," Ph.D. dissertation, Dep. Elec. Eng., Northwestern Univ., Evanston, Ill., 1970.
- [34] R. Cook and M. J. Flynn, "Time and space measures of finite functions," Dep. Comput. Sci., The Johns Hopkins Univ., Baltimore, Md., Comput. Res. Rep. 6, 1971.