A DATA-DRIVEN MACHINE ARCHITECTURE SUITABLE FOR VLSI IMPLEMENTATION

A. L. Davis
Computer Science Department
University of Utah
Salt Lake City, Utah 84112

ABSTRACT:

A machine architecture is presented which is capable of supporting very high levels of concurrency. The machine language of the class of machines described here is a graphical program schema known as data-driven nets. The machine architecture is arbitrarily extensible and consists of a recursively organized hierarchy of homogeneous processor-store modules. System control is decentralized, and each module is a completely asynchronous processing site, capable of executing any machine language program. Resource allocation is performed dynamically on the basis of the amount of available concurrency in the program and on the availability of physical resources.

Key Words: VLSI, concurrency, pipelining, recursive hierarchy, data-flow.

I. INTRODUCTION

In an attempt to increase the performance of computing machines, there appears to be two main approaches: 1) to use faster components in existing architectures, and 2) to design new architectures and programming methods, which are capable of exploiting high degrees of concurrency. The first approach is inherently limited in that the effects of reduced integrated circuit geometry, and new logic families can be expected to increase overall system performance by only a couple of orders of magnitude. While this is initially impressive, it does not meet the desired machine performance estimates necessary to solve large physics problems, or that needed for accurate weather prediction [16]. The second approach is not inherently limited by the physical properties of switching devices. There are numerous levels at which concurrency can be exploited in digital computers, i.e. multiple data paths, more concurrent realization of low-level circuit functions, overlap and pipeline processing within a single processing element, multiple processors, etc. In developing any new "fast as possible" machine, it is important to attempt to implement all of the above suggestions. The work reported here will mainly be concerned with solving the problem of how to utilize and organize systems containing large numbers of independent processors.

In attempting to escape the performance bounds imposed by Von Neumann architectures, it is insufficient to modify only a few aspects of the Von Neumann style system ideas. Alternative proposals to the "clock-driven" Von Neumann architectures are numerous. There are two areas which have some promise. One is the "demand-driven" approach espoused by Friedman and Wise [10]; Backus [3]; and Berkling [5]. Another is the "data-driven" approach taken by Dennis [8]; Bahrs [4]; Davis [6]; and Arvind, Gostelow, and Plouffe [2]. The work described here is of the data-driven variety due to the difficulty with which demand-driven systems support intra-process pipelining. In addition the propagation of demands takes time, and while demand-driven programs do allow for increased expressive power, the emphasis here is on performance. The data-driven approach naturally describes both pipelined (vertical) concurrency, and independent operation (horizontal) concurrency.

The work reported in this paper was supported by Burroughs Corporation. DDM1, a prototype module of the architecture described in this paper, is a hard wired data-driven machine, and was completed in July 1976 at the Burroughs Interactive Research Center in La Jolla, California. Many of the early systems ideas were developed in conjunction with Robert S. Barton. Gary Hodgman, Lawrence Rogers, and Karl Boekelheide were instrumental in the conceptualization and implementation of the actual machine. DDM1 now resides at the University of Utah, where the project continues under the support of the Burroughs Corporation.

It is clear that any machine architecture intended to have a general commercial appeal must be viable with respect to the changing constraints of integrated circuit technology. For architectures which fit nicely into the VLSI realm, the advantages are numerous. Among these are lower cost, increased reliability, increased speed, and decreased power consumption.

The actual machine language of the class of machines presented here is a directed graph schema called data-driven nets [6] or DDN's. DDN's are very similar to the data-flow nets of Dennis [8] and Rodriguez [15]. The terms data-driven and data-flow are used synonymously. The asynchronous nature of DDN's makes it easy to decompose a given net into a set of concurrent subnets, which can then be allocated to independent physical resources. The main distinction between the Dennis nets and DDN's is that in DDN's no distinction is made between the net tokens which are used for control purposes, and other net tokens. All DDN tokens are considered to be data items, and no explicit distinction is made to distinguish between classes of tokens. Another difference is that the primitive DDN cell types are slightly more high level than the Dennis nets. Finally some primitive activities which are explicitly specified in the Dennis schema are implicit in DDN's. An example is the Dennis "link" which serves as both a transmission and copy site. The functions of such links are implicitly incorporated into the output mechanism of DDN operators. The result is that both the DDN and Dennis schemas share the same properties with respect to ease of program verification, ease of program conceptualization, and ease of machine evaluation. Due to slightly higher level primitives and a less explicit schema a DDN program graph will typically have less vertices (cells) and arcs (data paths) than a functionally equivalent net in the Dennis schema. This difference is mostly one of style and is not particularly significant, although the differences are reflected in the respective architectures.

The only sequencing constraint in DDN's is that of data dependence, and since no weaker sequencing constraint exists without doing non-productive computation [14], DDN's are naturally a maximally concurrent representation of a given algorithm. While such concurrency may add to the "naturalness" of the programming experience, it is useless as a speed-up mechanism unless it can be mapped onto a set of physical resources capable of exploiting this concurrency. If this mapping is done at run-time, then the time to map must not overshadow the speed-up attained as a result of the concurrent execution.

Lastly, a number of additional goals for the machine structures presented here are felt to be desirable. Namely it is intended that these machines be general purpose, extensible, reliable, easily programmable, support very high levels of concurrency, and also be economical with respect to their performance and existing technology. In particular this effort is not concerned with one of a kind or special purpose machines. Special

purpose machines are perhaps ideal for a given environment, but suffer from inherent limits in their applicability to other problems.

II. THE IMPACT OF VLSI

The advantages of high density integrated circuit technology are so overwhelming that the constraints of VLSI must be considered as a primary force on future architectures. A detailed analysis of these effects is beyond the scope of this paper, but the global influences are summarized here. Due to the tremendous commercial emphasis on MOS VLSI, the following discussion will mainly be concerned with the properties of MOS device integration.

The most highly publicized VLSI benefits are those involving cost. A single custom VLSI chip (64 pin package) currently costs about $80,000 to $300,000 to produce. Even then, production typically must be guaranteed for about a quarter of a million parts at an additional cost of $7 to $10 per part. This clearly indicates that VLSI cost advantages can be obtained only if any given chip can be used in very large volumes. If a part does not have universal appeal, then the use of such a part in a new architecture brings about some high pressure constraints. Either the part must be used a large number of times in a single system, or a single system must have a very high sales volume, or some combination of the two. The number of part types in a given system is also a major concern in that it becomes a multiplicative factor in the system development cost.

Another factor heavily influenced by a VLSI implementation is speed. The dominant speed factor is due to the capacitive effects on a given transmission path. Typical off chip loads are on the order of 100 picofarads, while on chip loads are approximately one picofarad. Since delay times are proportional to the capacitive load (for constant drive current), this implies that signals which can remain on the chip will be driven about 2 orders of magnitude faster than those which must be driven to destinations off the chip. Additional speed-up can be obtained from the decreased geometries of switching elements and conductor path lengths. This is a very strong argument for architectures which attempt to maximize locality of processing. For architectures in which processing and local storage can not be done at the same locality, massive off chip delays must be incurred as a result. The only way around the slow off chip drive problem is to drive more current off the chip. This requires a series of relatively large output drivers, which are extremely costly in terms of chip real estate and power consumption. In addition, locality of processing will reduce the amount of contention for a given system transmission path. This contention is important in a highly parallel system in that the resultant sequencing will yield reduced system efficiency.

The number of pins is an important VLSI metric. The pin count is a primary factor in determining whether a given system module is nicely implementable as a VLSI circuit. Techniques to decrease physical pin count, such as time division multiplexing are applicable in certain situations but can not be considered a general solution. In addition, if chip types are used in sufficient quantities to amortize the initial layout cost, then the physical cost to manufacture the system becomes approximately linear with pin count. In addition increasing the number of pins on a particular chip causes decreased yield due to bonding problems. Increased pin count also implies that even more silicon area must be allocated to connection pads and pin drivers.

VLSI implementation also yields the more commonly discussed advantages such as: 1) increased system reliability due to reduced part count, 2) decreased system power consumption since voltages on a given chip scale with physical feature size, and 3) decreased system maintenance cost as chip replacement policies become more effective in highly integrated systems.

The extent to which these VLSI advantages can be realized is proportional to the logic/pin ratio of the proposed system modules. If the logic/pin ratio is relatively small then the situation is very much that of an SSI machine. If the logic/pin ratio is very high then true VLSI advantages can be obtained. This is a challenge to architects to devise systems which can be modularized into high complexity modules which communicate with their environment using relatively few signals. Furthermore as integration technology advances causing feature sizes to shrink even more, these new architectures must remain viable.

III. ARCHITECTURAL PRINCIPLES

The VLSI constraints indicate that future architectures to support very high levels of concurrency should consist of a set of processing sites capable of performing localized storage and computation of a reasonable complexity. These sites should be essentially the same physical module, which can be constructed from one or a set of part types. An additional goal of the architecture presented here is that of extensibility. More specifically, the architecture should be extensible without bound in the following way:

1. Machine power should be enhanced by the addition of more modules (i.e. allow greater concurrency due to the increased number of processing sites);

2. The addition of new modules should not require any change to the existing operating system in order to manage the resulting larger system;

3. Additional resources should be added simply by "plugging in new modules" without any special tuning of the existing hardware to create consistent system timing and communication for the expanding system; and

4. Extensions should be available in small quantums.

The first and last points indicate that a user should be able to purchase only the power needed and not much more or much less. The other points indicate that the manufacturer only needs to support a single module, rather than a large number of system configurations.

Systems such as these cannot be implemented in a synchronous, centrally controlled manner. Central control of arbitrarily extensible systems implies that the control must be able to function on an arbitrarily large amount of state information, which either slows the control drastically or requires controller modification to access the new state information. In an arbitrarily extensible synchronous system the problem of unbounded clock skew (maximum difference in the perceived clock time between any two processing sites in the system) will cause failure. The systems described here will therefore be asynchronous, fully distributed systems. Fully distributed systems have the following characteristics: 1) no module of a fully distributed system can determine the total system state, and 2) no module of a fully distributed system can enforce simultaneity in other modules. Holt [13] has shown that the notion of total system state in complex asynchronous systems is counter productive. Furthermore the enforcement of simultaneity in physically separate, asynchronous devices is impossible.

There are many ways to organize an extensible set of modules in a distributed control system. The advantages of hierarchical organizations are: 1) simplification in the amount of complexity to be dealt with at a given level, 2) verification by inductive methods can be done for uniform hierarchic systems, and 3) the superior-inferior relationship can be utilized to resolve problems such as contention and deadlock in multi-resource systems. It will be seen that hierarchy also facilitates a nice resource allocation policy. Recursive hierarchies are of particular interest in that they imply that the same module (and ultimately the same chip) can be used at each level.

Recursive systems are nicely extensible. A recursively structured machine is one which has exactly the same structure at every level. Clearly physical recursion must terminate at some point. This point will be seen to be the deepest set of resources in the physical hierarchy. Additional advantages of recursively structured systems have been demonstrated by Glushkov[11]. It will be shown that the width of a level in these recursive hierarchic structures can be used to execute independent operations, while the depth of the hierarchy will be used to facilitate pipelined operations.

IV. THE ARCHITECTURE

The architecture consists of a set of asynchronous modules which communicate by passing messages. The basic computational unit of the architecture is a processor-store element (PSE). A PSE consists of a processor module (P) and its associated local storage module (S). Any PSE can execute any machine language program, providing that it has a sufficient amount of local storage. No module that is not a PSE can perform this function. The architecture is a recursively organized set of these PSE's. The recursive definition of the structure is:

$$\langle PSE_n \rangle ::= \langle P_n \rangle \langle S_n \rangle$$

$$\langle S_n \rangle ::= \langle ASU_n \rangle$$

$$\langle P_n \rangle ::= \langle AP_n \rangle \mid \langle AP_n \rangle \langle PSE.GROUP_{n+1} \rangle$$

$$\langle PSE.GROUP_{n+1} \rangle ::= \langle PSE_{n+1} \rangle \mid \langle PSE_{n+1} \rangle \langle PSE.GROUP_{n+1} \rangle$$

Subscripts denote the recursive level at which the module physically resides. <AP> is an atomic processor module, which has no further sub-structure (contains no PSE's). Similarly an atomic storage unit <ASU> has no PSE substructure. The width of a <PSE.GROUP> has a physical bound. For the DDM1 prototype, this bound is eight. The structure is depicted in Figure 1.

This structure allows for a hierarchical distributed storage organization. Any S or ASU may consist of an arbitrary amount of storage of any desired medium. Higher levels of PSE's are considered logically superior to lower level PSE's. It is advantageous if higher level stores (ASU's) are slower and larger than the stores of lower levels. The interface and functional ability of any ASU (regardless of size, speed, and level) is the same. The structure also allows for an arbitrary number of processors to be used concurrently. It is important to note that all AP's are identical regardless of level. However, the processors at higher levels

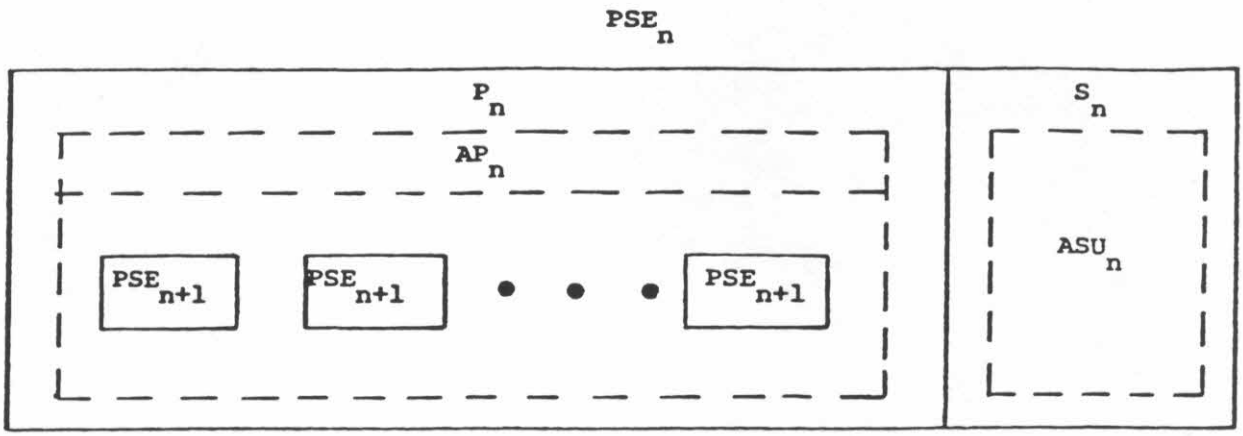

Figure 1: Recursive definition of PSE at level n.

will be more powerful, in that they contain more substructure than the processors at lower levels. More substructure implies more internal concurrent processing capability.

When viewed non-recursively this structure is simply a tree structure with a single root and a possibility for up to eight sons at any node. Each node of the tree is a PSE and is capable of executing any machine language program. The leaf nodes have no substructure and therefore consist of an AP and an ASU. At each node the fan-out is fixed but the depth of the tree is arbitrary. In this manner the architecture allows any desired number of PSE's to be configured for a given machine. The desired goal is for machine performance to improve with the addition of more PSE's.

There are a number of ways to enforce this logical tree structure onto a collection of PSE's. All involve some form of a connection network to implement the desired communication paths. A number of general interconnect networks have been considered: busses, crossbars, Banyan nets [12], and

permutation networks [17]. For tree-like machines, full connectivity is not required. The expense of crossbar switches vary as the square of the connected elements. Bus conflict would drastically reduce actual parallelism in the machine. Permutation networks present a tremendous problem in that they may need to be totally reconfigured when a single new connection is necessary. This is difficult to do reliably in a multi-path distributed control environment. Banyan networks have some merit, but do not easily allow for the desired hierarchic pipelined communication. Therefore in the DDM1 prototype, a simple 1 to 8 switch was chosen as the interface unit between successive levels of PSE's. The result is that the physical and logical recursive structures are the same. The structure is fixed and cannot be dynamically changed.

Information is passed between PSE's as messages which are variable length character strings. Upward traveling messages are passed on by the switch in an arbiter like manner. Downward going messages contain header fields which indicate their destination. This header is deleted by the switch as the message is passed. Downward and upward messages are dealt with by independent hardware, and therefore are controlled concurrently. This character serial nature of the machine has the following advantages:

1. Hardware modules are made simpler and more applicable for VLSI implementation due to the reduced pin count.

2. Hardware communication paths are more general in that variable length information units can be transmitted as varying numbers of fixed-width base characters. This facilitates a hardware substitution strategy for modules. Each module can interpret the variable length message and perform the indicated function.

These advantages aid in greatly reducing the cost of the hardware modules. Some low-level performance is lost by doing everything serially. The philosophy for this architecture is to regain that lost performance many times over by providing a systems organization that allows for many highly concurrent levels of activity.

Physical queues are placed between levels of PSE's in order to facilitate pipelining and increase physical module independence. Without queues, the sender of a message would need to wait on receiver availability. If a queue becomes full, only then must the sender wait until the receiver has freed up some queue space. If queue sizes are adjusted so that a sender is rarely required to wait for space, then the system would be well tuned for efficient processing. Optimal queue size depends on the average message length. It is therefore impossible to guarantee that no waiting will occur.

Strict hierarchical control and a restricted process structure insures that the system does not deadlock. A block diagram of the PSE structure is shown in Figure 2. In the DDM1 prototype, all communication paths except for the path between the ASU and the AP, consist of 6 wires (a 2 wire request-acknowledge control link and a 4 wire, character-width data bus).

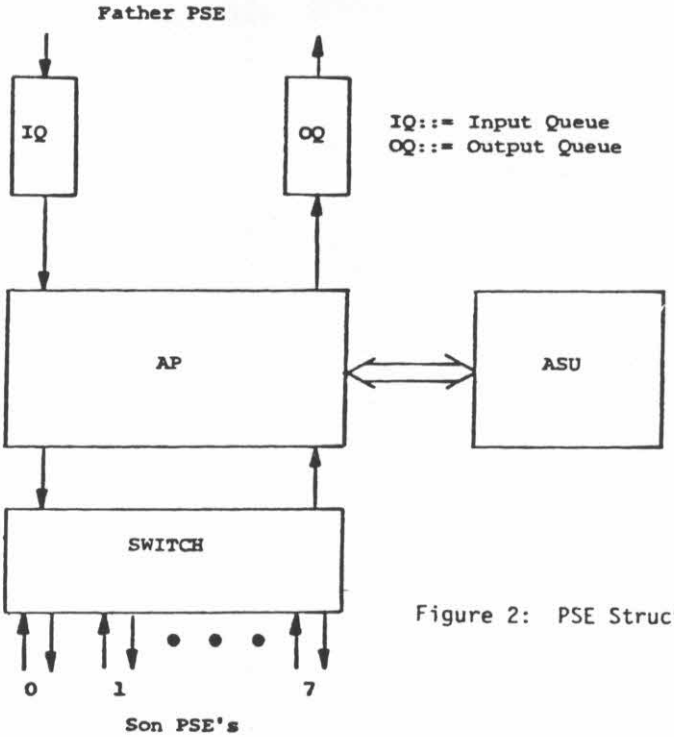

Figure 2: PSE Structure

The variable length, character serial message structure and DDN representation indicate that the ASU should be a highly flexible storage structure. Further requirements are that the ASU deal with the pipelining of data items and the continual destruction of data items due to cell firings. In order to increase efficiency of the PSE, all storage management functions are performed internally by the ASU. The ASU appears as a variable field length file system, which directly executes commands, such as: initialize, skip, insert, read, write, delete, and index. The free space is managed automatically by the ASU.

This PSE structure allows for a high degree of processing locality in that any PSE can execute any DDN program (assuming that there is sufficient storage in its local ASU). In addition the PSE admits nicely to VLSI implementation. The 1 to 8 switch can be implemented using a set of 1 to 2 switches of similar function. Using 1:2 switches, module complexities for the DDM1 prototype (pin and gate count) are shown in Figure 3. The pin counts include pins for power, ground, initialization, and extension. The indicated module pin counts are rounded up to coincide with standard package sizes.

Module	Gate Count	Pin Count
IQ, OQ (1K Characters)	3,000	16
AP	20,000	64
ASU (4K Characters)	47,000	64
1:2 Switch	2,000	40
Ap + ASU	67,000	64
Ap + ASU + IQ + OQ	73,000	64
Ap + ASU + Switch	69,000	64
PSE	75,000	64

Figure 3: PSE Module Complexities

These complexities are all within reason for VLSI designs, and are attractive with respect to the logic/pin ratio.

V. AUTOMATIC RESOURCE ALLOCATION AND EVALUATION

When a message corresponding to a DDN program enters a PSE at any level, the PSE may take one of two actions:

1. DECOMPOSITION AND ALLOCATION: if the PSE has substructure and if there exists some set of concurrent subnets in the DDN process, then the PSE may split the DDN and send concurrent subnets to PSE's at the next lower level.

2. EXECUTION: if the PSE has no subresources, or if there is no exploitable concurrency in the DDN, then the PSE executes the DDN at that level.

To aid the decomposition process, a structural descriptor may precede the incoming DDN in the message structure. This additional storage can greatly reduce time required for decomposition decisions in the PSE. In addition, each PSE must contain information about the number of available PSE's and the sizes of their respective stores. Problems would result if a DDN were sent to a PSE that was too large to fit in its local store. Only the local store sizes of immediate subresources are known. This insures the recursive nature of the decomposition process.

The decomposition process takes some time. It is important that the speed-up gained by extra concurrency resulting from the decomposition is not overshadowed by the time to decompose. Experiments have indicated that a "first fit" decomposition is almost always better than a "best fit" decomposition strategy. It is also not generally worthwhile to decompose the DDN structure completely on this architecture. At fine granularities, the slowdown resulting from loss of locality is not regained by the concurrent execution of very small subtasks. The exception to this rule is in the case of pipelining, where subtasks remain allocated for relatively long periods of time and sustain high activity at each site.

If decomposition and resource allocation occur at run-time, it is important that they be simplified as much as possible. It is possible to perform these tasks completely at compile-time. This however is inadvisable since it depends on knowing the run-time availability of PSE's in the system. In a system containing large numbers of PSE's, the probability is high that some PSE's will fail or be busy doing other things. In addition large portions of a process may only be evaluated conditionally. A compile-time allocation would have to allocate tasks which may never be executed. The strategy is taken here to split the decomposition task into two phases: 1) at compile time do all of the resource and condition

independent work, and 2) at run-time, dynamically make the actual allocation of executable tasks to available physical resources.

DDN's are quite randomly structured graphs and the data-driven architecture is a very regularly structured set of resources. Direct run-time allocation would be too slow, due to the structural disparity between program and machine. At compile-time, the two-terminal DDN process structure is transformed into a well structured and functionally equivalent series parallel graph (SP-graph). Two-terminal means that the graph contains a single "first" cell and a single "last" cell. This facilitates the determination of net termination and initiation. SP-graphs are acyclic, two terminal, directed graph structures which can be formed by successively combining cells and/or SP-graphs in series or in parallel. The SP-graph structures are then allocated as necessary at run-time. Data-flow graphs in general admit nicely to arbitrary restructuring due to their asynchronous and local control characteristics.

VI. CONCLUSIONS

An architecture and evaluation scheme for data-flow programs has been presented. The architecture exploits recursive hierarchy to reduce complexity and allows for the arbitrary expansion of system resources. Physical resources are organized such that they can be used to exploit both pipelined and independent tasks. The system exploits the notion of locality that is important for both increased speed and decreased cost aspects of a VLSI implementation. This notion of locality also indicates that this system is not intended to exploit concurrency at the lowest possible level. It is felt that the additional overhead involved to do this would actually reduce overall performance levels.

The main points of departure of this approach and that of Dennis [9] is the use of a recursive hierarchy of physical resources, the exploitation of physical locality to decrease message frequency and increase the speed of VLSI implementations, dynamic hierarchical resource allocation, the lack of specialized functional modules to reduce the chip type count, and a slight difference in the structure of the low-level schema. The architecture of DDM1 differs from that of Arvind and Gostelow [1] in that it does not try to achieve concurrency at all possible levels (because of the locality issue), the interconnection scheme is much simpler which results in reduced communication path contention, no special address space management needs to be done, allocated tasks may consist of many cells rather than just a single operation, and tasks are allocated only when all of their necessary input operands are present.

Analysis based on a working prototype module indicates that the machine architecture described is nicely implemented in VLSI. It has been shown that it is possible to implement an entire processor-store element as a 64 pin chip containing about a quarter of a million MOS transistors. This logic/pin ratio indicates that true VLSI benefits can be obtained.

The disadvantages of the system described here are:

1. The current ASU design is not nicely extensible to allow more storage capacity to just be "plugged in".

2. The fixed, hard-wire tree structure is not reconfigurable and may result in a situation where certain PSE's in one subtree remain idle when another heavily loaded subtree badly needs more resources.

3. There is currently not enough empirical data from test runs on very large programs to accurately quantify the overhead involved in decomposition and resource allocation.

BIBLIOGRAPHY

Arvind, Gostelow, and Plouffe. The ID Report: An Asynchronous Programming Language and Computing Machine. University of California at Irvine, Computer Science Department, Technical Report #114, (1978).

Arvind, and K. P. Gostelow. A computer capable of exchanging processors for time. Information Processing '77, North Holland, New York (1977), pp. 849 - 854.

Backus, J.. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. CACM, Vol. 21, No. 8, pp. 613 - 614, (August 1978).

Bahrs, A.. Programming language semantics and closed applicative languages. Proceedings of the ACM Symposium on Principles of Programming Languages, pp. 71 - 86, (1972).

Berkling, K. J.. Reduction Languages for Reduction Machines. Interner Bericht ISF-76-8, GMD, (1977).

Davis, A.. The Architecture of DDM1: A Recursively Structured Data-Driven Machine. University of Utah, Computer Science Department, Technical Report UUCS-77-113, (1977).

Davis, A.. Data-Driven Nets: A Maximally Concurrent, Procedural, Parallel Process Representation for Distributed Control Systems. University of Utah, Computer Science Department, Technical Report UUCS-78-108, (1978).

Dennis, J. B.. Data Flow Schemas. Lecture Notes in Computer Science 5, Springer-Verlag, pp. 187 - 216, (1972).

Dennis, J. B., and D. P. Misunas. A computer architecture for highly parallel signal processing. Proceedings of the ACM National Conference, pp. 402 - 409, (1974).

Friedman, D. P., and Wise, D. S.. Aspects of Applicative Programming for Parallel Processing. IEEE TC, Vol. C-27, No. 4, pp. 289 - 296, (April 1978).

Glushkov, V. M., et al. Recursive Machines and Computing Technology. IFIPS Proceedings 1974, North Holland, New York, pp. 65 - 70, (1974).

Goke, L. R.. Banyan Networks for Partitioning Multiprocessor Systems. Ph.D. Dissertation, University of Florida (1976).

Holt, A., and F. Commoner.. Events and Conditions. Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, pp. 3 - 52, (1970).

Linderman, J. P.. Productivity in Parallel Computation Schemata. MIT Project MAC, TR-111, (1973).

Rodriguez, J. D.. A Graph Model for Parallel Computations. MIT Project MAC, TR-64, (1969).

Rudy, T. E.. Megaflops from Multiprocessors? Proceedings of the 2nd Rocky Mountain Symposium on Microcomputers, pp. 99 - 107, (1978).

Waksman, A.. A Permutation Network. JACM, Vol. 15, No. 1, pp. 159 - 163, (1968).