DYPP - A VLSI DYNAMIC-GRAPH ENSEMBLE MACHINE

Marius V. A. Hâncu
Bell-Northern Research Ltd.
P.O. Box 3511, Station C, Ottawa, Ontario K1Y 4H7, Canada

Kenneth C. Smith
Department of Electrical Engineering
University of Toronto
Toronto, Ontario M5S 1A4
Canada

Abstract

In a previous paper, we proposed a special environment and techniques for the implementation of highly parallel processing. A new VLSI supercomputer architecture, DYPP (DYnamically Programmable multi-Processor), was introduced, with the unique property of being able *to embed, and execute directly, program graphs both statically and dynamically*. Either asynchronous (dataflow) or synchronous implementations of program graphs can be realized in DYPP. In the present paper, we provide further insight in its operation.

In the proposed ensemble architecture, *we separate processing and communication* into two distinct, though overlapping and interacting layers. Separate, simpler (and thus more reliable), processors are assigned to the connectivity layer, which becomes active and self-adaptive, thus being able to detect and compensate for malfunctions in the underlying layer of main processing elements.

There is no global control at either level. Rather in a first, static, version, the program graph (incorporating both connectivity information and operators, that is instructions), is "injected" in a preliminary, separate, phase via the connectivity-layer processors. In this phase, the connectivity graph is embedded between live (operational) main processing elements. In the second phase, processing takes place.

A more advanced option makes the connectivity layer fully dynamic. In this case, the program graph is continuously injected (embedded) in a flow fashion to interact with the flow of data and intermediate results, which flow is said to become *levitating*. This can greatly reduce the need for local program memory and large numbers of PEs, and correspondingly the required VLSI area. As well, it can dispel the need for (large) resident, localized, static programs characteristically present in von Neumann architectures. Based on data levitation, the *generalization of systolic arrays* becomes feasible.

1. Introduction

There is presently an ongoing search for full implementations in VLSI of highly parallel supercomputers with regular, array-like structures ([43], [46]). However, this search implies the solving of numerous problems. Some of the most critical of these problems are that:

- Supercomputer structures, implemented with conventional logic families, or present-day microprocessors [21], [44] must be completely reworked for total integration.
- Present densities (and areas) fall far short of the real need. Hedlund and Snyder [24] mention the possibility of implementing only about 300 processing elements (PEs) on a wafer, even using wafer-scale integration [33], but do not reveal the complexity of the PE type considered.
- Communication difficulties necessitate neighbour-to-neighbour solutions [29], [33].
- New reliability approaches must be discovered in order to provide a truly fault-tolerant design.

In Hâncu and Smith [22], we have introduced a new VLSI supercomputer architecture, the DYPP (DYnamically Programmable multi-Processor), intended to contribute to the solution of the difficulties outlined. In the present paper, we detail its description. DYPP embodies:

- a new two-level fault-tolerant structure, corresponding to a reconfigurable, highly parallel supercomputer in which, in contrast to the supercomputer proposed in Snyder [22], *the connectivity layer contains simple processing elements* and requires no *global* control of configuration;
- a new concept, called *"connectivity injection"*, whereby the *program graph* is located in the computational ensemble via a "program-wave" radiated into, and propagating (in a parallel or serial fashion) through, the connectivity layer serving as an adaptive support structure;
- a new concept, called *"data levitation"*, concerned with the way data and program flow information should interact in a highly parallel computational structure in order to ensure minimization of VLSI area, minimization of local fixed memory (i.e. ROM), as well as generality of application, which can be seen as a *generalization of the systolic array concept*.

The idea of massive parallelism in supercomputers has been advocated for a long time (see for example [5]). In [28], is given an interesting overview of "pre-VLSI" implementations.

Snyder [45], [46] was one of the first to outline the merits and difficulties of a *full VLSI* implementation of supercomputers and to propose an architecture, the CHiP (Configurable Highly Parallel Processor), which has become a term of reference. CHiP has three main architectural layers:

a) a regular bidimensional array of PEs;

b) a switch lattice (with *passive* switches);

c) a controller.

In CHiP, the controller is intended to be the element which *broadcasts* the switch-programming pattern. The switch lattice provides the only medium for interprocessor communication. Various architectures can be realized by modifying the switch lattice.

This work was done while the first author was a graduate student at the University of Toronto.

One problem with the CHiP architecture is the *global* controller and the means by which it obtains access to all of the switching elements in the connectivity lattice. In VLSI implementations, any global connection in addition to power, ground and clock (the latter, obviously, in synchronous systems) is not normally recommended. This is one of the reasons why *systolic arrays* [29], based on systematic neighbour-to-neighbour connections, have created such interest.

2. DYPP - A Fault-Tolerant Architecture with Self-Adaptive Connectivity

In contrast to the CHiP architecture, the proposed architecture for DYPP (Fig. 1) consists of only two conceptual levels or layers:

a) a regular array of main processing elements (MPEs);

b) a connectivity network (lattice) consisting of wires and switches controlled by an array of *connectivity processing elements* (CPEs).

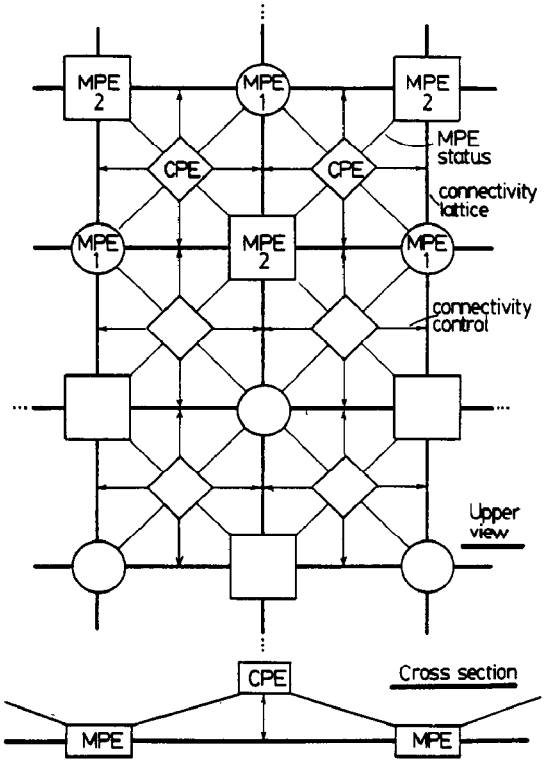


Fig. 1 DYPP architecture.

The main processing elements (MPEs) are each able to perform a predefined set of instructions. They could be all identical to each other, or could represent a *mosaic-like* spatial distribution of distinct processors, able to cover in combination all the data-processing operations required by the applications environment (see for example the distinction between MPE1 and MPE2 in Fig. 1). The sets of instructions pertaining to each processor could be customized via individual writable control stores (WCSs).

In contrast to schemes previously presented, the proposed connectivity lattice is not simply static, but is adaptable, under distributed control, to the needs of the current program.

Traditionally, VLSI interconnection networks have been thought of as being purely passive, being collections of switches and wires. Many configurations have been imagined: trees, shuffles, single or multiple buses, etc. [8].

However, in the VLSI environment, testing *by direct access* for non-functional units at the component, block or even chip level (the latter for example in the case of wafer-scale integration schemes) is quite difficult, if not impossible (in a majority of cases). This results from:

a) circuit complexity with extreme miniaturization;

b) access difficulty (the mechanical or optical resolution of available tools is limited);

c) cost of provision of additional testing pads often associated with drivers powerful enough to support external loads;

d) software complexity, which requires quite intricate *sequences* of tests.

As a result, designers are forced to include *built-in* testing procedures and provisions for fault tolerance.

In our present approach to VLSI supercomputer design, the interconnection lattice of DYPP includes, and is controlled by, a set of processors, the connectivity processing elements (CPEs). They serve an essential role both in the (re)configurability and fault tolerance of the structure.

The CPEs are the architectural blocks controlling, and thus configuring, the switch lattice. Their decisions are based on:

a) *connectivity* information provided by the user as part of the array-programming and customizing process (see section 3);

b) *functionality* information provided by the MPEs as status bits (indicating fault status) and by tests performed by the CPEs themselves on the passive switch lattice and adjacent connections.

The functionality information is obtained either during a preliminary self-test step, or in specially allocated time intervals, interspersed with the real processing of data (the latter approach is intended for continuous array maintenance).

Our approach to the fault tolerance of the processing array is based on additional perceived factors:

1. The necessity of delegating as much of the reliability testing and repair process to the chip itself because of the difficulty of access and surgery in VLSI by external means.
2. The higher reliability of the interconnects over the processing elements (PEs) themselves which are much more complex structurally [23], [24], [33].
3. The necessity of a more hierarchical distribution over different layers of PEs and interconnects; this leads naturally to the idea *of CPEs as being processors which are relatively simple* in comparison with the MPEs.
4. The necessity in wafer-scale integration for self-test and self-reconfiguration. This is seen to be in antithesis with the exclusive use of external techniques, like discretionary wiring [38] or laser restructuring [33].

The two-layer approach to chip (or wafer) reliability is even more appealing if we take into consideration some current trends towards a more 3-dimensional layout [41]. In such a layout, the separation between the processing layers could be even clearer.

Note that the granularity of the active array can be varied, thus for example making provision for the availability of spare functional components (MPEs, CPEs and interconnections) in the immediate vicinity of the defective ones.

Various fault-tolerant strategies can be implemented in this architectural environment to cover different fault instances. Some of them will be covered in the following sections.

3. Dataflow and Connectivity Injection

The *program graph* will, for the present purposes, be interpreted in a dataflow sense [21], which leads, in the end, towards an asynchronous implementation. However, other interpretations are possible, some of them leading to synchronous schemes (as those presented in section 8).

The dataflow model of computation has been advocated over the last three decades [12], [15], [20], [21], [26], [40], [50], as an advantageous means of performing highly parallel computations. The main advantages lie in the areas of implementation (as no synchronization is required for operand communication) and programming (reducing side effects, as only values, and not variables, exist).

A feature distinguishing dataflow architectures from each other is the degree of directness with which they can bridge the gap between the *program graph* (used as program representation) and its implementation.

In the majority of dataflow approaches [11], [15], [21], [48], the nodes of the program graph represent functions (or processes in a more general setting) which are enabled (fired) only when all the (input) operands become available. The *arcs* of the graph provide communication, that is they carry data (or sometimes control) tokens labelled with their destination, constituting the specific input of a specific instruction (operator).

As a defining attribute of the proposed architecture (DYPP), the program graph nodes are seen to be located at the MPEs (main processing elements). The program graph arcs will be implemented via the physical links created between the source MPEs and destination MPEs when applying appropriate connectivity instructions to the CPEs defining the communication paths.

Certainly, DYPP is optimized for running highly parallel programs, where a majority of its processors are busy at a certain time. This means, too, a requirement for a certain locality of data dependencies, which should translate logically into neighbour-to-neighbour communications.

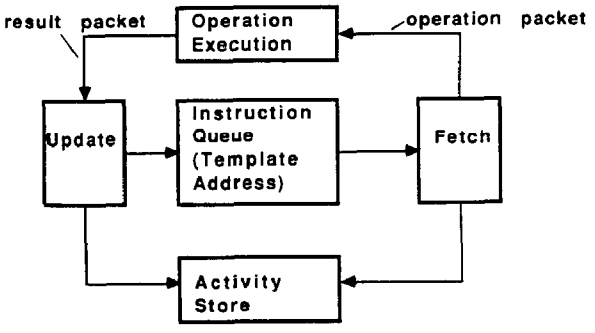


Fig. 2 The basic dataflow-instruction mechanism (Dennis [13]).

The most well-known dataflow architectures are those based on the *circular pipeline* mechanism of instruction execution proposed by Dennis [13], (Fig. 2). The program graph is stored in *the Activity Store*, where the operators are represented as *templates* (Fig. 3) having as component fields the opcode, the operand inputs and destination address. The addresses of the templates which have received all their operands (corresponding to instructions which can be fired) are read by the *Fetch Unit* from the *Instruction Queue*. Based on these addresses, the Fetch Unit reads the activity template from the activity store, assembles it in an operation packet and sends it to an appropriate *Operation Execution Unit*. The result packet is directed via the *Update* unit to its destination template. The receipt of all necessary operands by a template (in order to fire) is also checked by the Update unit. The complete templates (those ready to fire) are queued in the Instruction Queue. Several operation and result packets can coexist on the ring, reflecting the parallel (but unsynchronized) execution of several instructions forming the circular pipeline.

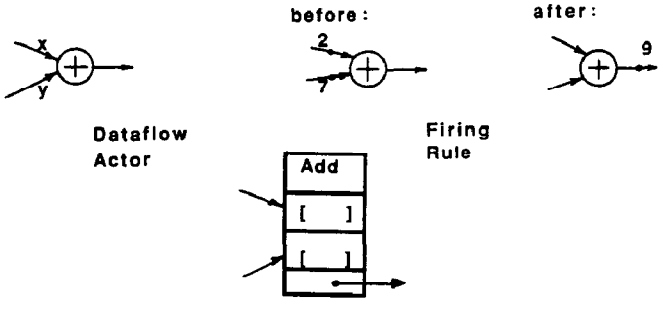


Fig. 3 Dataflow actors and templates.

The degree of parallelism in execution can be improved by increasing the number of PEs implementing the operation units. Dennis [13] proposed a dataflow multiprocessor for which a VLSI implementation under current development is described in [20].

In Dennis' multiprocessor architecture (Fig. 4) the main element besides the PEs is the communication network directing the result packets to the appropriate PEs, which fire upon receiving all the necessary operands.

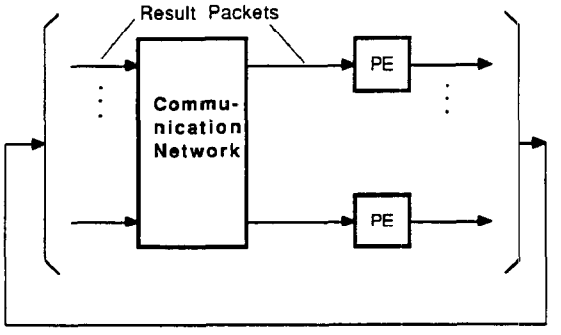


Fig. 4 Dataflow multiprocessor (Dennis [13]).

While these conventional circular dataflow architectures are advantageous in terms of minimizing the number of PEs, our major point is that *they abandon contact with the program graph level*, which does not appear explicitly in the communication scheme. As a result, too many intermediate conceptual levels appear in the process of programming, and as a result direct graphical programming is quite impossible.

With the advent of VLSI and the prospect of WSI (wafer-scale integration) near at hand, the number of PEs present on a chip or wafer is increasing considerably. Thus another design philosophy can be pursued, based on an abundance of PEs, in which *we design a truly distributed dataflow architecture, able to replicate the program graph naturally within its communication scheme*. Thus, programming at the program graph level would be facilitated tremendously.

For the more conventional dataflow architectures, high-level languages like VAL [2] or ID [3] have been created and corresponding compilers or interpreters have been designed. These conventional dataflow languages are functional or applicative languages whose dominant properties are freedom of side effects, locality of effect and the single-assignment convention [2]. They include special constructs able to describe highly parallel activities in a compact manner.

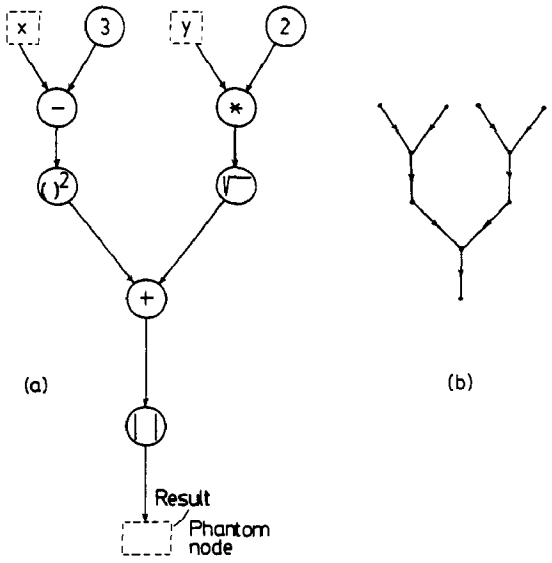
However, when using novel architectures, like CHiP and DYPP, both fully distributed and providing an easy view of the communication paths, one feels pressed to abandon language-based high-level programming. Instead, a direct *graphical-programming* approach can potentially be used where in fact the programmer locates the functions (or processes, in a more complex case) at the PEs and defines the communication links along which the data (or control) tokens are exchanged. In [45], is presented such a parallel programming environment for CHiP. The corresponding development for DYPP (in the static graph-embedding case) is similar, though not the subject of this paper. In fact, the programmer can operate directly at the program-graph level. Of course, time is required until such new programming approaches, unconventional but straightforward and arguably very productive, will be adopted.

Our confidence in the potential for direct programming at the program graph level has influenced the writing of this paper. Thus here the graph is the main program support to be embedded statically or dynamically in DYPP, with no intermediates.

An extra assumption (although not essential for the DYPP definition), will be made about the program graph, in order to limit the intersection of communication arcs in the embedding. In particular we will assume that the program graph is composed of *sections*, each of binary-

tree form [8], [9]. In Fig. 5a a simple program graph is given in the dataflow form, while in Fig. 5b the associated connectivity (communication) graph is presented. In Fig.5c the operators (functions) are represented, located at the nodes, with the same graph "perpendicularized" in order to fit a presumed rectangular mesh of main processing elements (MPEs). Phantom nodes (Figs. 5a and 5c) contain no operators [11]; they are used exclusively for I/O transfers.

The spatial location of the operators in the MPE mesh (Fig. 5c) will be obtained by a mapping program which has, as its input, the program graph. It is provided also with information about the available arrays of MPEs and CPEs and outputs the (presumably final) positions of the operators (provided no faults are considered for MPEs, CPEs or interconnects.)



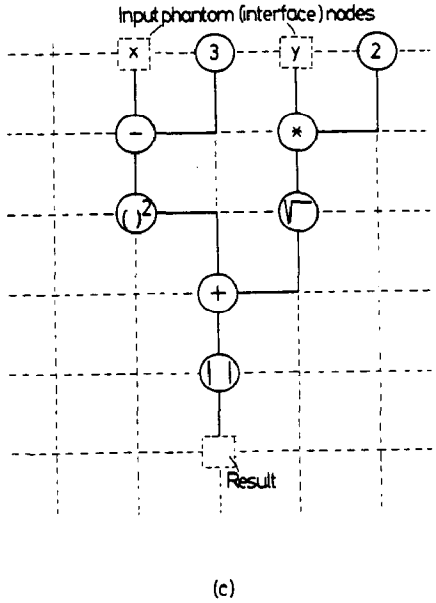


Fig. 5. Program-graph "injection". (a) Dataflow program for $|(x-3)^2 + \sqrt{2y}|$. (b) Associated connectivity (communication) graph. (c) Locating the operators in the MPEs (same graph).

In the next step, this operand-location and connectivity information must be transposed onto the real array, by programming each MPE to perform its corresponding instruction (elementary operator or function) and asking the CPEs to configure the connectivity lattice correspondingly.

The newly concocted term of *connectivity injection*, presented in the title of this section, refers to the mapping (embedding) of the program graph in the available array of processors provided by the DYPP environment.

Program-graph injection can take place before or during program execution. In this respect, it is defined as being respectively *static* or *dynamic*. The next two sections treat these approaches successively.

4. Static Program-Graph Injection

The *mapping programs* which must be used to obtain the physical location of the MPEs and CPEs in DYPP, for static program-graph injection, could be similar to those derived in [6], [7] for CHiP. Thus their implementation as such is not our main topic of study.

The techniques used in this section are said to be static because the (whole) graph is "frozen in place" in the array (since the MPEs are programmed with the same operators, and CPEs define the connections for the whole duration of program *execution*).

Instead, we tried to devise simple techniques for injecting the program graph in DYPP, taking into account the fact that no special configuration controller exists in our design, in contrast to the situation in the CHiP design.

Two techniques have been devised to cope with the final stage of the graph-injection process: a parallel one and a serial one.

The programming of the array takes place in two steps, scheduled entirely before program execution is initiated in the static program-graph-injection process:

1. The MPEs are provided, in a global parallel or serial fashion (Figs. 6 and 7), with their operators and with the addresses of their two children (assumming the graph is a *binary* tree)

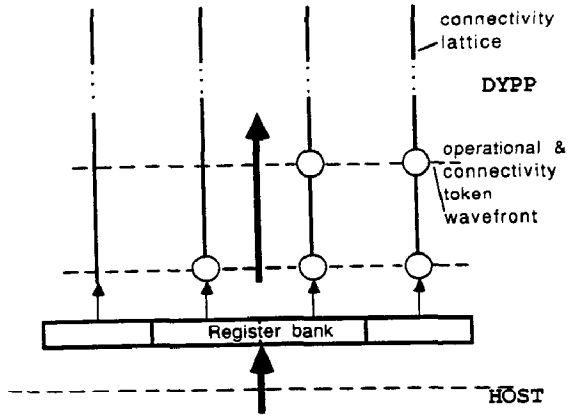


Fig. 6 Parallel, static, programming of DYPP (parallel injection).

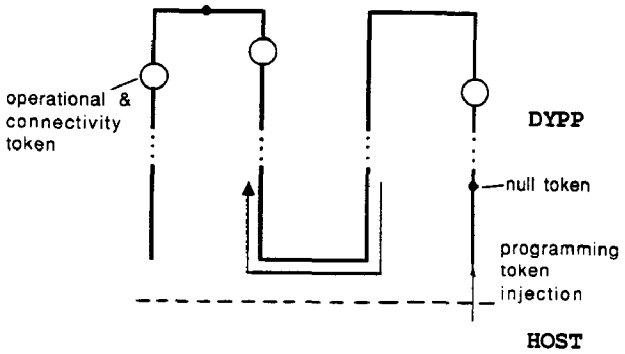


Fig. 7 Serial, static, programming of DYPP (serial injection).

2. The MPEs ask the adjacent connectivity processors (CPEs) to connect them with their children.

It is probably clear by now that the intermediate mapping program should be provided with some information about the dimensions of the array, and about the number of levels in the program graph. The latter should be smaller than the maximum *integer* coordinates in any communicating direction in the processor mesh (a value which is equal to the number of processors in that direction).

The programming step in the program-graph-injection process takes place as presented in Fig. 6 and 7.. In both approaches, the mechanism for the advancement of the *operation and connectivity tokens* (containing the operators and the addresses of specific connected children of the node) can be synchronous (under the influence of a clock) or asynchronous (as in the data-wavefront approach in [31].

The operation and connectivity tokens are provided *by the host*, based on the output of the intermediate-mapping program previously mentioned.

The MPEs not participating in the computational process are assigned a null operator, and obviously do not ask the CPEs for a connection.

The *parallel* program-graph injection process, shown in Fig. 6, is initiated after power-up when the connectivity lattice is automatically organized by the CPEs *into columns*, on which the operational-connectivity *tokens* are pushed from the host. The register bank at the lower side of the array provides the conversion from the field width of the host interface to the array communication width. The necessary numbers of pins for host I/O might otherwise be prohibitive. In any event, this connectivity issue is a serious problem for any VLSI supercomputer, given the present packaging techniques. The operational-connectivity tokens are pushed in a pipelined fashion by the clock (in the synchronous approach) or pulled by handshaking (in the asynchronous version) until they reach the upper edge of the array.

The tokens should advance for a number of steps precisely equal to the vertical integer coordinate of the processor array (corresponding to the number of processors in the vertical direction). This is ensured by defining an appropriate number of clock pulses in the synchronous approach or providing the upper edge of the array with a permanent NONREAD response, in the asynchronous (handshaking) technique.

The tokens are provided by the register bank in layers, starting with those in the upper row. The tokens propagate in waves until they fill in the array, thereby fully programming and configuring it.

In the *serial* scheme, presented in Fig. 7, the tokens are injected, one by one, at one corner (shown to be the lower-right in the figure) of the array. They propagate along the columns configured in the above-mentioned manner. At the end of each column, they travel horizontally through an extra interconnection, provided to connect all columns end to end.

The injection (filling) process of the array ceases when all the MPEs have received their programming tokens.

Note that the content of the tokens can be modified to increase the number of neighbours to which each MPE is connected.

5. Data Levitation and the Dynamic-Graph Machine

Very long parallel programs (represented by graphs with many levels), might require a considerable area if we use the *static* embedding of program graphs discussed in the previous section.

Assuming that we are using a dataflow type of computation, the *data* tokens travel, from the root(s) and other injection points, along the graph. Only a limited number of operators are fired (active) at a given time, depending on the data already available for input to each operator. Correspondingly, only a limited number of MPEs are used at a given time. leading potentially to low resource utilization.

Conceptually, we could use a movable "activity window" to frame the graph area (Fig. 8) containing the active operators (instructions) at any time. The window would move along the graph, from the root to the pendant vertices (which are connected to the exit points from which emerge

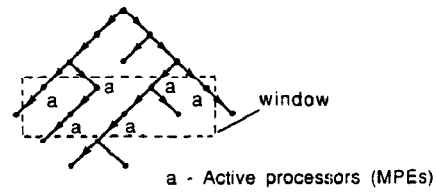


Fig. 8 A program-graph activity window.

the final results).

Even if, for the graph in question, the height of this window (that is the number of graph levels included in it) is much smaller than the total number of levels in the graph, a static graph injection (embedding) in the DYPP would normally lead to a correct implementation only if *the whole graph* is embedded in the processor array.

This is because the static-injection approach causes the nodes of the graph to be stationarily located at certain MPEs in the processor array, while the data tokens propagate, at program runtime, along the arcs of the graph. This means that *all* the graph nodes must be injected in the programming phase (Figs. 6 and 7) and no modification can be made to the operators (functions) performed by each MPE during program running. This approach might in general require a prohibitively large VLSI area, even with wafer-scale integration.

To cure this problem, we propose a new concept of "levitating" data (including partial results) via dynamic modification of MPE and CPE programming.

In this approach, the size of the array would be prescribed by the maximum activity window at any time, and not by the total number of graph levels.

The corresponding array is no longer statically programmed. Instead, the *program graph is continuously injected through the array, dynamically changing the functions (and the connections) of all MPEs* (Fig. 9). The programming and execution phases must now take place in parallel.

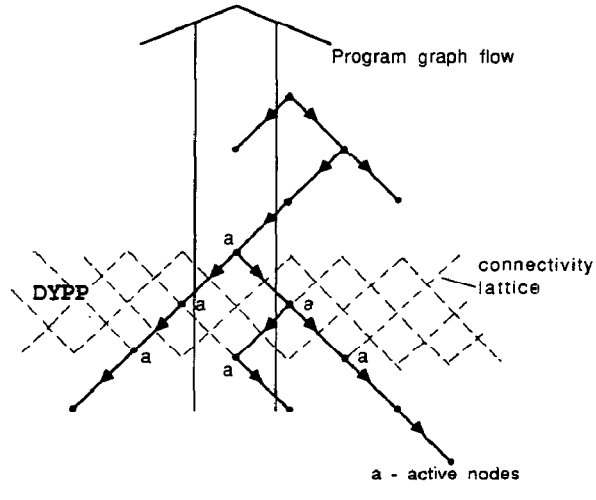


Fig. 9 The data-levitation process.

The processor array must accommodate an area at least equal to the *maximum* activity window required at any time.

All the *data* tokens resulting from previous operations, or *current* input, should *levitate* (be present dynamically) in the array. Conceptually, this means keeping all the arcs of the graph having unresolved data tokens in the array (hardware), dynamically mapped onto communication links.

If we compare this with the static model of embedding studied in section 4, we realize that the program graph moves itself through the array in a wave-like manner, carrying with it *the operation and connectivity tokens* of the current activity window. Synchronous or asynchronous schemes can be devised to support this movement, devised to maintain dynamically the activity window inside DYPP.

This can eventually be seen as a *generalization of the systolic array concept*. In systolic arrays several flows of data and results coexist in interaction in the array. In our proposed dynamic architecture, the *functional definitions of the operators* implemented by the PEs find themselves also in a flow movement, causing dynamic reconfigurability of the array (Fig. 10).

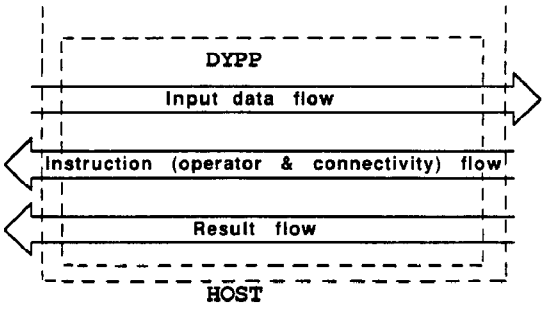


Fig. 10 A DYPP generalization of systolic arrays.

This architecture can be identified as being both *dynamic and non-von Neumann* (the program and data memories are equally distributed to all MPEs).

The concept leads potentially to a massive reduction in the array area and local ROM size. But, at the same time, it forces the host computer to perform another function, namely the injection of *instruction flow*, as well as providing input data flow and accepting result data flow.

The concept of *data levitation* can possibly be applied successfully to general-purpose computation. However one reservation remains. It concerns the implementation of program loops in a VLSI context where global connections are to be avoided (and are, as a matter of principle, not recommended in DYPP), and where neighbour-to-neighbour transactions should be the rule.

As a matter of fact, considerable current research in VLSI algorithms is directed toward creating algorithms matching the VLSI constraints [29], [37], [45].

6. Two-Level Fault-Tolerance

The hierarchical fault-tolerant structure of DYPP is organized on two conceptual levels: processing and connectivity. Because an effective implementation of parallel computation in such an ensemble of PEs requires a significant number of them, the techniques destined to provide fault tolerance described in [34] for large arrays of processing elements, and those recommended in [24], based on wafer-scale integration, should be appropriate.

While both synchronous and asynchronous (dataflow) architectures can be implemented in DYPP, we feel that a *dataflow* architecture is more appropriate as it makes full use of the direct graph-embedding feature of DYPP. At the same time, a dataflow architecture is more recommended in terms of VLSI implementation for large DYPP structures, where the effects of clock distribution can generate considerable difficulty.

The research on fault-tolerant dataflow and corresponding architectures is at an early and evolving state. The most significant contributions now known are [20], [47], [48]. In view of the current interest and novelty of this area, we will concentrate mostly on the problem of implementing such *fault-tolerant dataflow architectures in DYPP*.

The two-level architecture of DYPP is entirely appropriate for the application of the "loading arm method" of test and reconfiguration proposed by Manning [34]. The "loading arm" is conceived to be a set of contiguous cells, grown gradually in the array. It is able both to advance and to retract. The cell located at the top of the arm first tests and then programs its neighbours. The intermediate cells, located between the controller and the top cell are used collectively as a communication link through which the test or reconfiguration information is passed. This method can be used for the static embedding of the program graph in a DYPP structure containing faulty processing elements. These faults are expected to be located mainly at the MPE level, as the main processing elements are by design more complex and occupy more area than the CPEs, thus being on the average more prone to defects. But faults in the CPEs can be detected and acted upon, as well.

In Manning's concept, an external or internal central controller projects into the array of PEs a *loading arm* which can advance or retract in all directions (a square array was the original case), avoiding faulty PEs (Fig.11). Manning was the first to propose the use of such an arm in *flawed* arrays [34]. The PEs at the front of advancement are tested by the controller via the loading arm or else it merely reads the results of individual PE self-tests and sends them back to the controller.

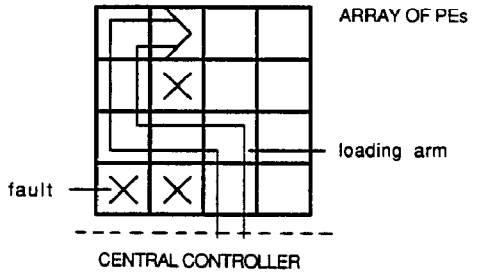


Fig. 11 Manning's loading-arm method for test and configuration of an array with faults.

The loading arm grows progressively from functional PEs tested at the previous step. In DYPP, the processing and loading layers as conceived by Manning are implemented respectively in the processing layer (containing MPEs) and in the communication layer (containing CPEs).

In the case of DYPP, the *controller* could be the *external host* or a *group of internal PEs* working as a diagnostic and reconfiguration processor (Fig. 12a & b), located preferably at the DYPP periphery, for convenient I/O with the host.

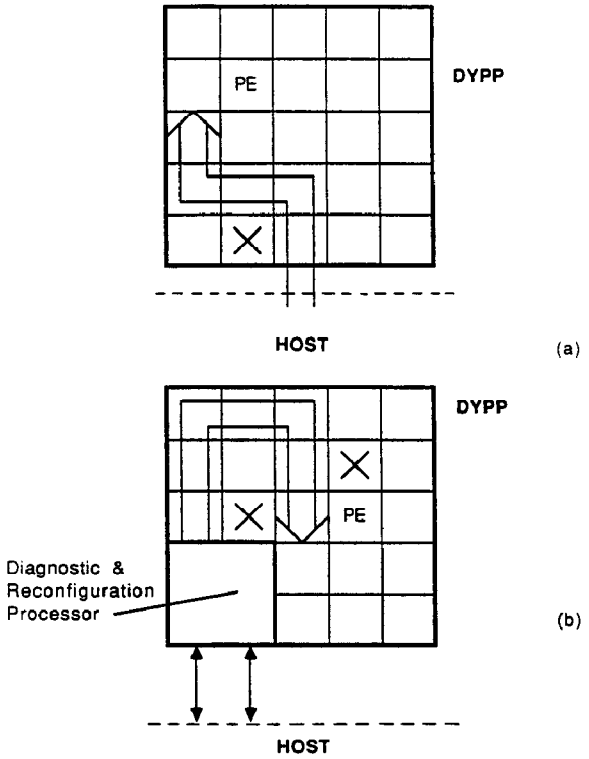


Fig. 12 External (a) and internal (b) implementation of the controller for the Manning's method of test and reconfiguration of DYPP.

The loading arm also acts to program the array of PEs, performing in fact in the case of DYPP the above-mentioned *static* embedding of the program graph. The use of the loading arm for dynamic embedding of the program graph is not appropriate, as the presence of the arm prohibits movement of the graph through the array.

The loading arm is composed of two parts:

- the *arm itself*, implemented in the CPE layer, has to communicate the test and programming instructions from the controller to the current cell under test and configuration;
- the *head of the arm*, implemented in the MPE layer, is able to test all adjacent neighbours and relocate in any of them to advance or retract the arm.

The MPEs and CPEs can be designed with full or partial self-test facilities so that after power-up the positions of the faults are implicitly known. These fault positions are discovered in the first pass (used for test) of the process of arm advancement. Thus the controller is quickly provided with a fault distribution. Taking into consideration the graph to be embedded and the available functional processors, a mapping procedure is carried out. At a second pass (used for configuration or graph embedding), the arm programs the MPEs and CPEs to their required functions, based upon the results of the mapping procedure. The second pass thus substitutes the programming steps described in Figs. 6 and 7.

Two passes of the arm are necessary in our opinion, because the mapping algorithms normally require *global* information on the location of faults in the array and such information can be available only at the end of the first pass. This, in fact, is one of the reasons why fully distributed fault-tolerant systems (using distributed test and reconfiguration) are not very popular in the literature .

The approach used by Manning is one in which no explicit reference is made to the specific nature of the computation to be performed by the array. However, the ideas on fault-tolerance by Srini [47], [48] are directly related to fault-tolerance in *dataflow computers* and are thus more immediately relevant to DYPP, as already mentioned in this section.

Srini's approach to fault-tolerance is also centralized, as he uses a *diagnostic processor* to which all the nodes in the dataflow graph report their states (determined via self-test) previously to and during node finding. When applying these concepts to DYPP, upon detection of a fault during node firing, the node and its operands (located in its input buffers) will be reassigned to a neighbour MPE, based on tables of replacement. This reconfiguration is managed by the diagnostic processor.

In the process of reassigning the processor to a functional node, Srini suggests that one should neglect the output buffer and as a result start the computation from the first instance assigned to that processor because the memory effect of the output buffer dissappears.

A difficulty arises when the input buffer is itself defective. In this case, Srini suggests that the system memory provide a backup copy of the token stored in the buffer. As DYPP is a distributed computer system, the closest we could come to his original solution (because the absence of a unique memory) is to ask the source processor for a backup copy. Such copies might also be stored by the communication processor (CPE) supervising the corresponding link.

Several ideas concerning the fault tolerance of a *distributed* dataflow computer using multiple chips have been proposed by Gaudiot et al. [20]. Some of them could be used in the DYPP environment as well. Gaudiot et al. distinguish two aspects of *static* fault tolerance:

- immediate recovery from faults;
- long-term recovery from faults.

For the *immediate recovery* from faults (after only a few seconds), an error memory will be provided in all MPEs and CPEs. Upon failure, a processing element informs all neighbours (or the whole array, by broadcast, superceding normal operation, in the case where long program-graph arcs have to be embedded) of his address, which is stored in the corresponding error memories. A certain amount of redundancy being designed in DYPP, we can use some of the unused processors for replacement based on tables stored in each of them. Usually, a close neighbour will be the substitute. The replacement can be executed in both the static and dynamic program-graph embedding (injection). Of course, a solution is not always available, as is the case when several faults are crowded in the same area and replacements do not exist for all of them.

For *long-term recovery*, the host could execute test and maintenance programs on the DYPP array before program embeddings (static or dynamic) and use the resulting distribution of faults in the mapping process.

Another useful method outlined in [20] concerns *dynamic* fault tolerance, implemented in our case by running identical code sections in parallel through different DYPP PEs and *voting* on the results. The same authors underline the advantages of a dataflow architecture in avoiding associated synchronization problems.

7. A Dataflow Implementation of Data Levitation

As mentioned in previous sections, program graphs can propagate through the DYPP structure either synchronously or asynchronously. In this section, we will insist on the latter approach, as it is more appropriate for large DYPP structures, in order to avoid the problems of clock distribution. As well, the program-graph representation is natural in the dataflow model of computation.

The approach used for the asynchronous movement of the graph in DYPP is based on the following steps:

- each MPE, after sending its result tokens to their destinations, relocates itself at the next available position in the PE lattice;
- after relocating itself at the new position, it reestabishes the data links (graph arcs) to its destination MPEs by using CPEs (relocated, too, in the process) as path-support points (in Fig. 13, the CPEs are not represented).

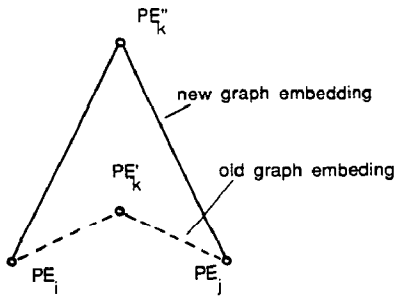


Fig. 13 Segments of a program graph relocating in the course of data levitation.

The first step of this procedure necessitates the introduction of several definitions. In the active program window dynamically implemented in DYPP, each operator (actor) of the program graph must fire a finite number of times (and should not occupy DYPP area subsequently). We will call this number the *firing count*. The firing count of each graph node can be obtained at compile time, if the organization of the data structures to be treated is known at that time, including their sizes (but certainly, not their composition). The organization of the data structures under discussion can be variable (like sizes of matrices or vectors), but their dynamics must be entirely known at compile time in order to generate an appropriate program graph and to use the firing count as an enabling-operator characteristic.

In other words, the instruction flow as represented in Fig. 10, must be known entirely as a result of compilation and mapping, at the time the dynamic injection of the graph (in the data-levitation technique) takes place.

In the process of relocating the program graph, *control* tokens must be used. They are sent from the MPE initiating the move to the connecting CPEs, in order to disable them and the corresponding old paths. Other control tokens are sent by the new "impersonation" of the operator (program graph node) to its adjacent CPEs at the new location, to activate them and install new data links.

In order to simplify the mapping, a bidimensional rectangular mesh will be used for DYPP. The source and destination processors on each data link will be separated by at least two mesh increments in order to guarantee non-overlapping movement of processor embeddings.

All token communications are asynchronous, on a handshake basis (confirmation tokens are thus used as well to confirm receipt of an originating token).

In Fig. 14, we present the process of graph advancement in the DYPP. A simple binary tree has been chosen for the graph. The execution proceeds in several stages:

a) After the input data token has arrived at the root node, it is processed and the output data tokens are released towards their destination nodes (Fig. 14a).

b) After *both* (in order to compensate for data link delays) acknowledge (ACK) signals from the destinations have been received, the node (MPE) disables the side (adjacent) CPEs, via control tokens (Fig. 14b & c).

c) The graph node relocates itself by sending a control token up its own column (we assume that nodes travel constantly in the same column of DYPP). The control tokens contain the operation code and the mesh coordinates of the locations (a list of such locations is obtained for each node in the mapping process). The control token is checked by each MPE situated along that column and upon the first coordinate coincidence, the new embedding of the node is enabled (Fig. 14d & e).

d) The process of node relocation and graph advancement continues asynchronously and independently as described at a) - c) at all graph levels. (Fig. 14e).

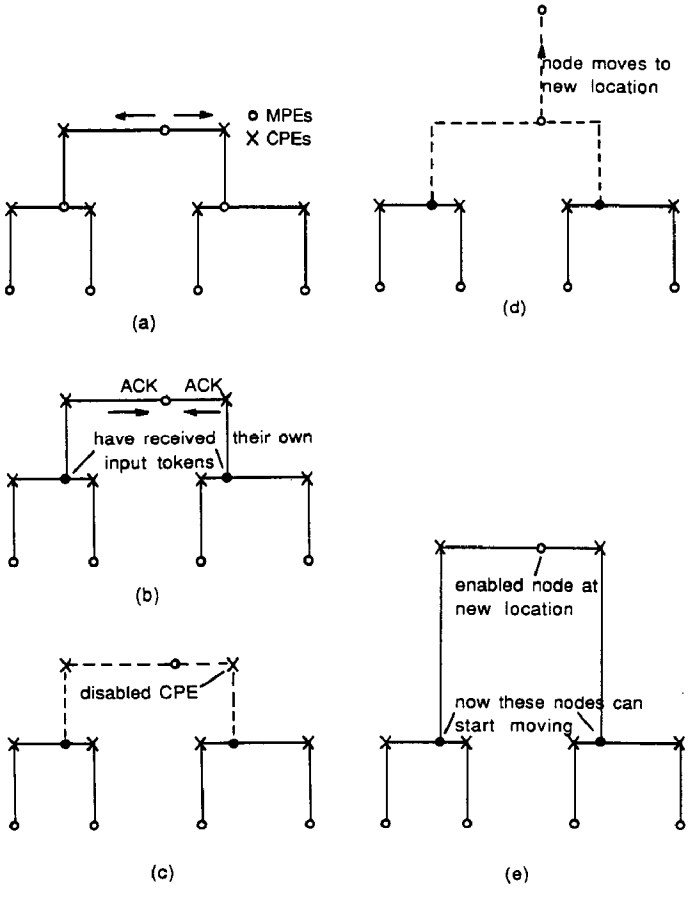


Fig. 14 The process of asynchronous graph advancement in DYPP.

In order to provide a suggestive illustration of how the above process works, we would associate it visually with the way a worm advances, by successive constriction and expansion of its body segments.

After each node (operator) firing, the corresponding node decreases its firing count by one. When the firing count reaches zero, no data will be available for processing by that graph node while it moves in the levitation window of DYPP. As such, it is permanently disabled.

In a fault tolerant-approach, the control tokens used to relocate a node would reach the center of a cluster of MPEs or CPEs and then would settle themselves in one of the functional ones, activating it.

8. A Synchronous Implementation of Data Levitation

As an alternative to the dataflow approach presented in the previous section, a synchronous implementation of the data levitation (or "windowing") technique is proposed in this section.

In order at the same time to introduce a solution to the problem of executing loops via data levitation in DYPP, the program graph used as an example contains a conditional loop. As previously mentioned, loops are more difficult to implement via data levitation, as they contain arcs extending themselves over many levels in the program graph. At the same time, the direction of data flow on such loops is generally opposed to the direction of propagation of data in the main-graph body in data levitation. Thus, they cannot be implemented via single neighbour-to-neighbour connections, but only by concatenations of such connections. Moreover, their length could exceed the height of the window available in DYPP for data levitation, if this window height is determined by the operator activity in the main graph body alone.

As is to be expected in a synchronous implementation, but in contrast to the operators present in the dataflow model of computation used in the previous section, the operators included in this program graph (Fig. 15) are timed (synchronous). Their positional shift in DYPP, contributing to the program graph movement, is also synchronous.

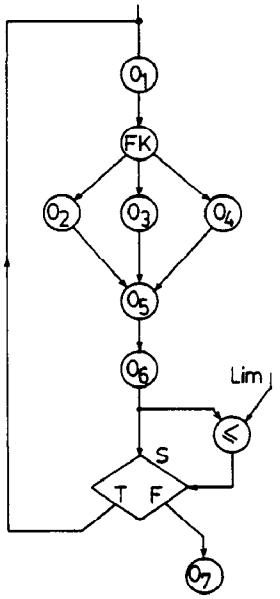


Fig. 15 Conditional loop.

A DYPP array of 2 × 3 MPEs and CPEs is used as the hardware window (Fig. 16). For simplicity of representation, the MPEs and CPEs are superimposed at the same position, localized in distinct, but communicating layers.

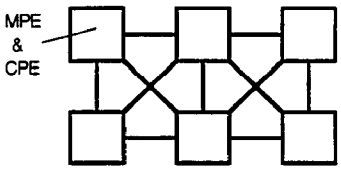


Fig. 16 Implementing a 2×3 data-levitation window in DYPP. Processors and their possible interconnection links.

The width of the array used is related to the maximum number of operators found in a graph crossection (layer) at any time. This is the case with the O2, O3, O4 set of operators (Fig. 15). The operators O1 to O6 can be general arithmetic processors, for example. The selector operator (S) closes the conditional loop based on the result provided by the output of an inequality (≤) operator. For control input=*T* (true), the loop is closed for another iteration. For an *F* (false) control input, the data token provided by the output of operator O6 is passed to O7 as a loop output.

The FK ("fork") operator simply replicates its input token on all of its outputs. The input data arrives as a synchronized token on the input of O1.

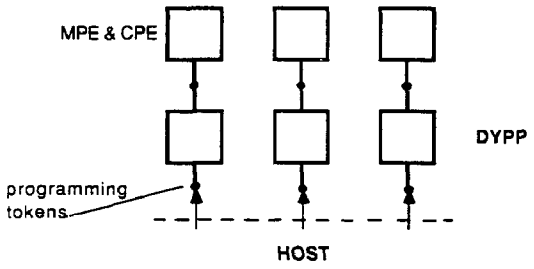


Fig. 17 Interconnections during phase Φ2 (parallel injection for data levitation).

In the proposed synchronous implementation, the interconnections between MPEs are activated on clock phase Φ1 for processing purposes. On clock phase Φ2 (only) the column interconnections are turned on, providing the paths along which the programming tokens are injected by the host, in order to implement the new graph segment in the data-levitation window. (Fig. 18).

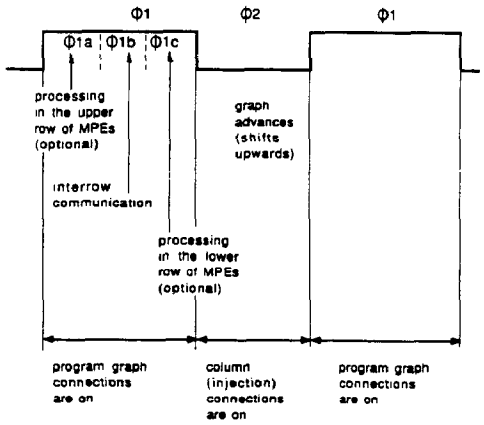


Fig. 18 Timing of processing and programming phases in the synchronous implementation of data levitation.

As mentioned in section 5, in the data-levitation approach, the program graph is gradually executed by a DYPP array implementing a window swept over the full graph. In our example, two levels of the graph will be executed at any time. The clock period will be conservatively adjusted to cover the longest operation ever arising in the graph execution and the associated communication delays.

The programming tokens injected by the host in phase Φ2 have the role of defining the operations performed by the MPEs and their interconnections (via the corresponding CPEs).

In Fig. 19 data tokens are represented by black dots, while control tokens (used for the control of the selector operator used in the example graph) by empty circles. Input data tokens are represented in the upper part of the processor cells, while result data tokens are represented in the lower part of the processor cells.

Because of the synchronous nature of the computation, we know at all times where the different operators will be located in the levitation window. This is in marked contrast to the asynchronous implementation where only areas in which operators exist can be eventually specified, based on the estimated maximum duration of processing and communication.

While shifting upwards in phase Φ2, the programming tokens *are also used as carriers*, able to carry input or output data tokens to their new locations.

In Fig. 19, we present a representation of the step-by-step execution of the example program loop, following the timing in Fig.18.

In order to condition the programming tokens to enable the MPEs, and connect them (via CPEs and links) with their neighbours only at the required positions in the levitation window, special *enabling lists* are used. They are stored by the host in the programming tokens and contain the row coordinates of the enabling positions. While advancing through the columns, during the upward movement of the graph in the levitation window, the contents of the enabling list are compared with the coordinates of the current DYPP node, and upon coincidence, the operators are enabled and the correct connections established. In Fig. 19, we *underline enabled operators.*

The enabling lists are established in the program graph-mapping process, prior of course to graph injection.

In clock period T7 of the sequence represented in Fig. 19, the value of the control tokens directs the interconnections. In this respect all the operation and connecting tokens provided by the host have two components (fields):

- an operation field (explicitly indicated as O1, O2, etc);
- a connectivity field (implicitly indicated in the pattern of interconnections enabled for the MPE where the token is currently located).

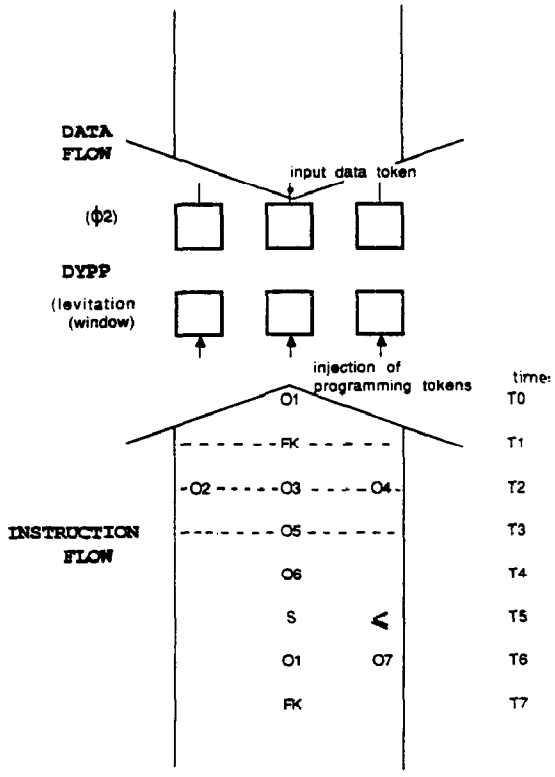


Fig. 19a Synchronous execution of a program loop by data levitation. Detailed instuction-token flow.

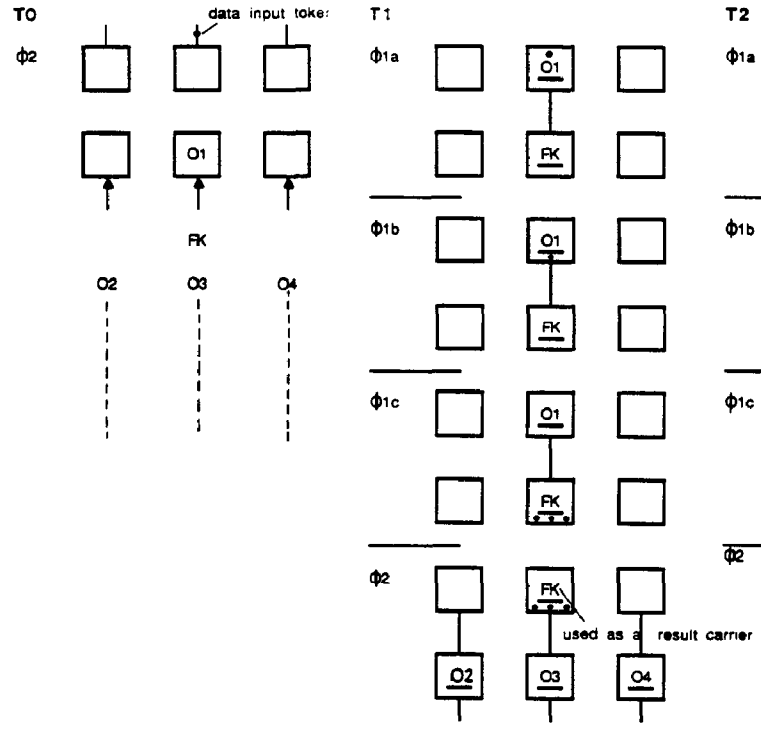


Fig. 19b Synchronous execution of a program loop by data levitation.

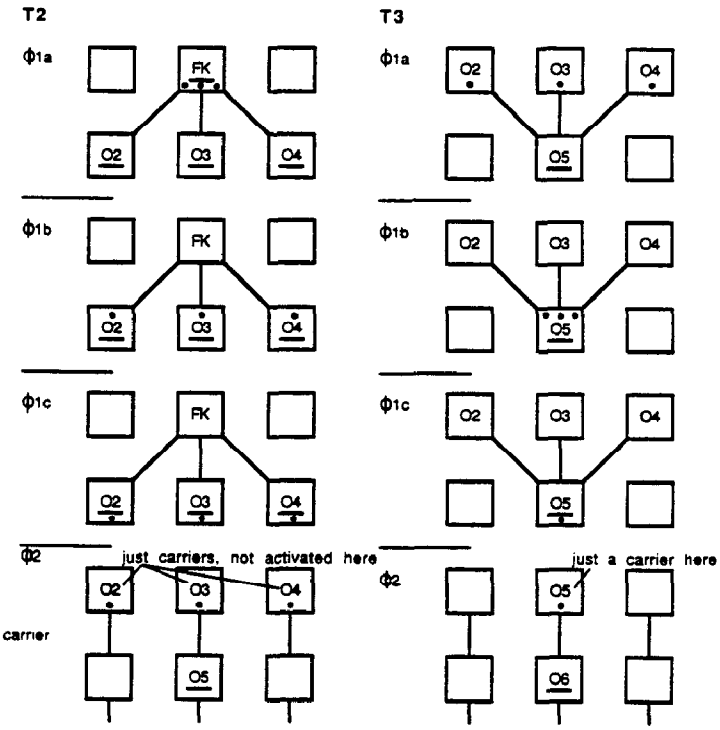


Fig. 19-c

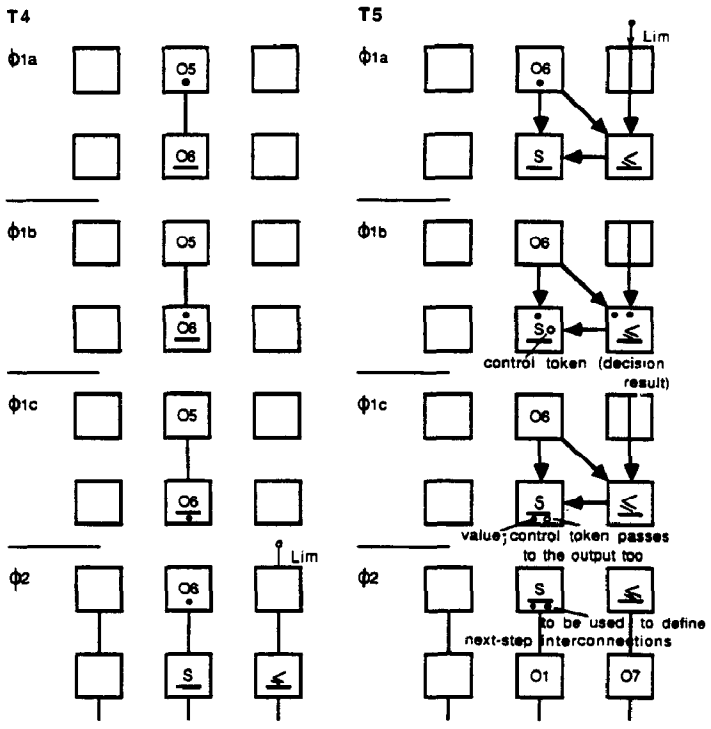


Fig. 19d

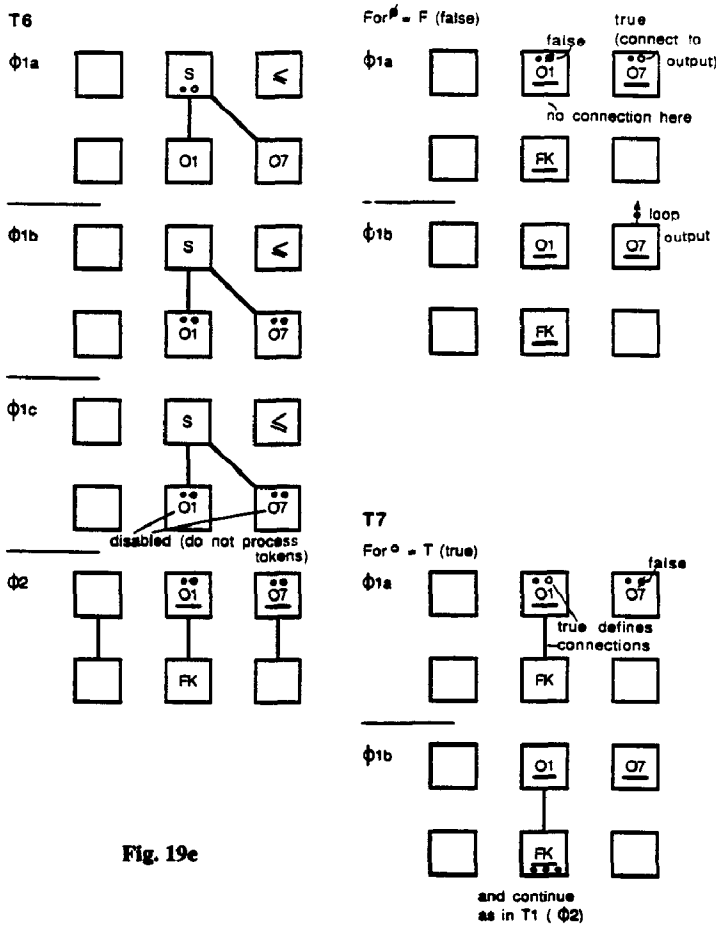


Fig. 19e

9. Conclusions

A new ensemble-array architecture has been proposed for a dynamically programmable (DYPP) VLSI supercomputer, having as salient features:

- *A hierarchical (two-level) fault-tolerant structure* separated on two conceptual levels (processing and connectivity), suitable for wafer-scale integration;
- Provision of static and dynamic *programmability* in the form of *graph injection*;
- Support for *full dynamic reconfigurability* (via the application of a new concept, called *data levitation* applied to program graphs, preferably in dataflow form).

The new fully distributed supercomputer architecture is considered to be very appropriate for running highly parallel programs. It provides the outstanding feature of *directly embedding and executing program graphs in both static and dynamic versions.*

An important generalization of the systolic concept is implemented within the data-levitation structure, where instructions are also provided in a continuous flow to interact with the data flow in the environment provided by DYPP. In principle, this was made possible by using CPEs (Connectivity Processing Elements) instead of passive switches in the distributed mesh-architecture.

References

[1] W.B. Ackerman, "A Structure Processing Facility for Data Flow Computers", *Proc. Int.'l Conf. on Parallel Processing*, pp.166-172, 1978.

[2] W.B. Ackerman, "Data Flow Languages," *Proc. 1979 Nat. Computer Conf.* AFIPS Press, Arlington, Va., pp.1087-1095, 1979.

[3] Arvind, K.P. Gostelow, and W. Plouffe, "The (Preliminary) Id Report," Tech. Rep. 114a, Dept. of Information and Computer Science, Univ. Calif. Irvine, Calif., May 1978.

[4] Arvind and R.E. Thomas, "I-Structures, An Efficient Data Type for Functiona Languages," Laboratory for Computer Science, MIT, TM-178, 1980.

[5] K.E. Batcher, "Design of a Massively Parallel Processor," *IEEE Trans. Comput.*, vol. C-29, no. 9, pp.836-840, 1980.

[6] F. Berman, "Edge Grammars and Parallel Computation," *1983 Allerton Conference on Circuits, Communications & Computers*, Urbana, Ill, pp.214-223, 1983.

[7] F. Berman, M. Goodrich, Ch. Koelbel, W.J. Robison III and K. Showell, "Prep-P: A Mapping Processor for CHiP Computers," *Proc. Int'l Conf. on Parallel Processing*, pp.731-733, 1985.

[8] S.N. Bhatt and C.E. Leiserson, "How to Assemble Tree Machines," MIT, Laboratory for Computer Science, Technical Memo TM-255, 1984.

[9] S. Browning "The Tree Machine: A Highly Concurrent Computing Environment," Ph.D. Thesis, Dept. of Computer Science, California Institute for Technology, 1980.

[10] J.L. Caluwaerts, J. Debacker and J.A. Peperstraete, "Implementing Streams on a Data Flow Computer System with Paged Memory," *Proc. Int'l Symp. on Computer Architecture*, pp.76-83, 1983.

[11] A.L. Davis and R.N. Keller, "Data Flow Program Graphs," *IEEE Computer*, vol.15, no.2, pp.26-41, Feb. 1982.

[12] J.B. Dennis, "First Version of a Data Flow Procedure Language," *in Programming Symp.: Proc. Colloque sur la Programmation (Paris, France, April 1974). Lecture Notes in Computer Science, Vol. 19.* New York: Springer-Verlag, pp.362-376, 1974.

[13] J.B. Dennis, "The Varieties of Data Flow Computers," *Proc. 1st Int'l Conf. Distributed Computing Systems*, pp.430-439, Oct. 1979.

[14] J.B. Dennis and K.K.-S. Weng, "An Abstract Implementation for Concurrent Computation with Streams," *Proc. Int'l Conf. on Parallel Processing*, pp.35-45, 1979.

[15] J.B. Dennis "Data Flow Supercomputers," *IEEE Computer*, vol. 13, no. 11, pp.48-56, November 1980.

[16] A.N. Despain and D.A. Paterson, "X-Tree, A Tree structured Multiprocessor Computer Architecture," *Proc. Symp. on Computer Architecture*, pp.144-151, 1978.

[17] D.D. Gajski, D.A. Padua, D.J. Kuck and R.H. Kuhn, "A Second Opinion on Data Flow Machines and Languages," *IEEE Computer*, vol. 15, no. 2, pp.58-69, Feb. 1982.

[18] J.-L. Gaudiot and M.D. Ercegovac, "A Scheme for Handling Arrays in Data-Flow Systems," *Proc. Int'l Conf. on Distributed Systems*, pp.724-729, 1982.

[19] J.-L. Gaudiot, "Methods for Handling Structures in Data-Flow Systems," *Proc. Int'l Symp. on Computer Architecture*, pp.352-358, 1985.

[20] J.-L. Gaudiot, R.W. Vedder, G.K. Tucker, D. Finn and M.L. Campbell, "A Distributed VLSI Architecture for Efficient Signal and Data Processing," *IEEE Trans. Comput.*, vol. C-34, pp.1072-1077, December 1985.

[21] J.R. Gurd, C.C. Kirkham and I. Watson, "The Manchester Prototype Dataflow Computer," *Comm. of the ACM*, vol.28, no.1, pp.34-52, Jan. 1985.

[22] M.V.A. Hâncu, and K.C. Smith, "DYPP - A VLSI Supercomputer Architecture Supporting Two-Level Fault-Tolerance, Program Graph Injection and Data Levitation Concepts," *Proc. 1986 ACM Computer Science Conference*, pp.115-120, February 1986.

[23] K.S. Hedlund, "Wafer-Scale Integration of Parallel Processors," Ph.D. Thesis, Comp.Sci. Dept., Purdue Univ., Aug. 1982.

[24] K.S. Hedlund and L. Snyder, "Wafer-Scale Integration of Configurable, Highly Parallel Processors," *Proc. Int. Conf. Parallel Process.*, IEEE, pp.262-264, 1982.

[25] K. Hwang and F. Briggs, *Computer Architectures and Parallel Processing*, McGraw-Hill, New York, 1984.

[26] R.M. Karp and R.E. Miller, "Properties of a Model for Parallel Computations," *SIAM J. of Applied Math.*, vol.14, no.6, pp.1390-1411, Nov. 1966.

[27] S.P. Kartashev and S.I. Kartashev, "Architectures for Supersystems of the 80's," *Proc. 1980 AFIPS National Computing Conference*, pp.165-180, 1980.

[28] R.H. Kuhn and D.A. Padua, (Eds.), *Tutorial on Parallel Processing*, IEEE Computer Society Press, New York, 1981.

[29] H.T. Kung and C.E. Leiserson, "Systolic Arrays (for VLSI)," *Sparse Matrix Proceedings 1978*, Duff, I.S. and Stewart, G.H., (Eds.), SIAM, pp.256-282, 1979. (An earlier version appears in Chapter 8 of Mead and Conway [35]).

[30] S.-Y. Kung and R.V. Gal-Ezer, "Synchronous vs. Asynchronous Computation in VLSI Array Processors," *Proc. SPIE Conf.*, Arlington, VA, May 1982.

[31] S.-Y. Kung, K.S. Arun, R.J. Gal-Ezer and D.V. Bhaskar Rao, "Wavefront Array Processor: Language, Architecture and Applications," *IEEE Trans. Comput.*vol. C-31, no.11 , pp.1054-1066, Nov. 1982.

[32] S.-Y. Kung, "On Supercomputing with Systolic/Wavefront Array Processors," *Proc. IEEE*, vol.72, no.7, pp.867-884, July 1984.

[33] F.T. Leighton and C.E. Leiserson, "Wafer-Scale Integration of Systolic Arrays," *Proc. 23rd Annual Symp. on Foundations of Computer Science*, pp.297-311, 1982.

[34] F.B. Manning, "An Approach to Highly Integrated, Computer-Maintained Cellular Arrays," *IEEE Trans. Computers*, vol. C-26, no.6, pp.536-552, June 1977.

[35] C.A. Mead and L.A. Conway, *Introduction to VLSI Systems*, Reading, MA, Addison-Wesley, 1980.

[36] D. Misunas, "A Computer Architecture for Data-Flow Computation," Laboratory for Computer Science, MIT, TM-100, March 1978.

[37] D.I. Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays," *Proc. IEEE*, vol.71, no. 1, pp.113-120, Jan. 1983.

[38] R.L. Petritz, "Current Status of Large Scale Technology," *IEEE J. Solid-State Circuits*, SC-2, pp.130-147, Dec. 1967.

[39] F.P. Preparata and J. Vuillemin, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," *Comm. of the ACM*, vol.24, no.5, pp.300-309, May 1981.

[40] J.E. Rodriguez, "A Graph Model for Parallel Computation," Ph.D. Thesis (Also Report MAC-TR-64, Project MAC), MIT, Cambridge, MA, Sept. 1969.

[41] A.L. Rosenberg, "Three-Dimensional VLSI: A Case Study," *JACM*, vol. 30, no.3, pp.397-416, 1983.

[42] C.L. Seitz, "Ensemble Architectures for VLSI - A Survey and Taxonomy," in *Proc. Conf. Advanced Res. VLSI*, Massachussets Inst. Technol., Cambridge, Jan. 1982. Dedham, MA: Artech., pp.130-135, 1982.

[43] C.L. Seitz, "Concurrent VLSI Architectures," *IEEE Trans. Comput.*, vol.C-33, no. 12, pp.1247-1265, 1984.

[44] C.L. Seitz, "The Cosmic Cube," *Comm. of the ACM*, vol.28, no.1, pp.22-33, Jan. 1985.

[45] L. Snyder, "Introduction to the Poker Parallel Programming Environment," *Proc. Int'l Conf. Parallel Processing*, pp.289-292, 1983.

[46] L. Snyder, "Supercomputers and VLSI: The Effect of Large-Scale Integration on Computer Architecture," *in Yovits, M.Y. (Ed.), Advances in Computers*, vol.23, Academic Press, pp.1-33, 1984.

[47] V.P. Srini, "An Architecture for Extended Abstract Data Flow," *Proc. Symp. Computer Architecture*, pp.303-325, 1981.

[48] V.P. Srini, "A Fault-Tolerant Dataflow System," *IEEE Computer*, vol. 18, no.3, pp.54-68, March 1985.

[49] P.J. Varman, "Wafer-Scale Integration of Linear Processor Arrays," Ph.D. Thesis, University of Texas at Austin, 1983.

[50] J.W. Young and H.K. Kent, "Abstract Formulation of Data Processing Problems," Internal Report, Product Specification Dept., The National Cash Register Company, Hawthorne, CA, 1958.