DYPP - A VLSI Supercomputer Architecture Supporting Two-Level Fault Tolerance, Program Graph Injection and Data Levitation Concepts*

Marius V.A. Hâncu and Kenneth C. Smith

Department of Electrical Engineering, University of Toronto

Toronto, Ontario M5S 1A4, Canada

Abstract

It is generally accepted that in the environment of a VLSI array of processors the interconnections are much more reliable than the processing elements (PEs) themselves. In the proposed 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 the program graph (connectivity and operations, that is instructions), is "injected" in a preliminary phase via the connectivity 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 in a flow fashion to interact with the flow of data which is said to become *levitating*. This can reduce greatly the need for local program memory, 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, *generalizations of systolic arrays* become feasible.

The project is presently at the stage of architectural refinement leading to potential simulation.

1. Introduction

There is presently an ongoing search for full implementations in VLSI of highly parallel supercomputers with array-like structures [1], [3]. However, this search implies the solving of numerous problems. Some of the most critical of these problems are that:

1) Supercomputer structures, implemented with conventional logic families or present-day microprocessors [6], [7], [8], must be completely reworked for total integration.

2) Present chip and area fall far short of the real need ([2] mentions the possibility of implementing only about 300 processor elements (PEs) on a wafer, even using wafer-scale integration [5], but does not reveal the complexity of the PE type considered).
3) Communication difficulties necessitate neighbour-to-neighbour solutions [4], [5].
4) New reliability approaches must be discovered in order to provide a truly fault-tolerant design.

This paper introduces constructs which are intended to contribute to the solution of the difficulties outlined. These new constructs define a novel organization, the DYPP (DYnamically Programmable Processor). DYPP embodies:

a) a new structure, corresponding to a reconfigurable, highly parallel supercomputer in which, as opposed to [1], the connectivity layer becomes self-adaptive to failures in the PEs and requires no *global* control of configuration;
b) a new concept, called "connectivity injection", whereby the program graph is self-locating via a "program-wave" radiated into and propagating (in a parallel or serial fashion) through the connectivity layer serving as sensitive support;
c) a new concept, called "data levitation" concerning the way data and program information should interact in a highly parallel computational array in order to ensure minimization of VLSI area, minimization of local fixed memory (i.e. ROM) as well as generality of application (not a special-purpose engine). This can be seen as a *generalization of the systolic array concept*.

2. Two-level fault tolerance and self-adaptive connectivity

The idea of massive parallelism in supercomputers has been advocated for a long time [9]. Reference [11] gives an interesting overview of "pre-VLSI" implementations.

Snyder [1], [2] 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). Hedlund and Snyder [2], [12] suggested wafer-scale integration techniques in order to solve the problems of fault tolerance.

CHiP has three main architectural layers:

a) a regular bidimensional array of PEs;
b) a switch lattice (with *passive* switches);
c) a controller.

*Ref.no. 85112010.

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

One possible problem with the CHiP architecture is the *global* controller and the means by which it might have 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.

The proposed architecture for DYPP (Fig. 1) consists only of 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).

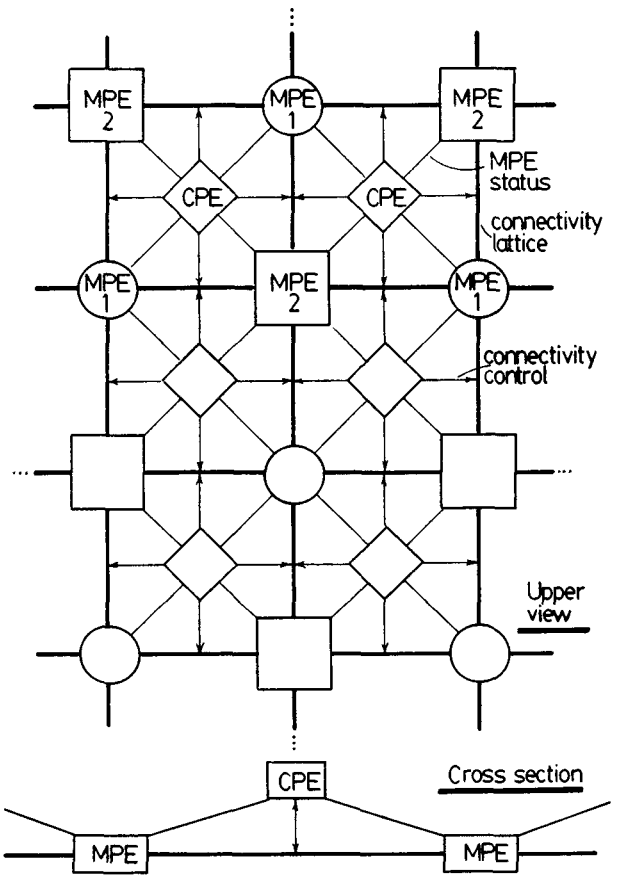

Fig. 1. DYPP architecture.

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 together 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 with 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. [20].

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

c) 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 approach, the interconnection lattice 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 [2], [5], [12].

3. The necessity of a more hierarchical distribution of reliability 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. Wafer-scale integration is considered by us to be possible only with self-test and self-reconfiguration, rather than with exclusively external techniques, like discretionary wiring [17] or laser restructuring [5]; we feel that the price to be paid in this approach is the necessity of more built-in redundancy and complexity.

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 [10]. In such a layout, the separation between the processing and connectivity layers would be even clearer.

The granularity of the array can be varied, thus for example making provision for the availability of functional components (MPEs, CPEs and interconnects) in the immediate vicinity of the defective ones.

Various fault-tolerant strategies can be implemented in this architectural environment to cover different fault instances. However, they will not be covered here.

3. Connectivity injection

This newly concocted term refers to the mapping (embedding) of the *program graph* in the available array of processors provided by the DYPP environment.

The program graph will, in the present paper, be interpreted in a data flow sense [8], which means, in the end, an asynchronous implementation. However, other interpretations are straightforward, some of them leading to synchronous schemes.

In one of the data flow approaches [8], [25], the nodes of the program graph represent functions which are enabled (fired) only when all the (input) operands become available. The arcs of the graph carry data tokens (packets with data accompanied by labels designating their appropriate function destinations).

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

An extra assumption will be made about the program graph, in order to limit intersection of communication arcs in the embedding. We will assume that the program graph is composed of *sections* of binary tree form [19], [20]. In Fig. 2(a) the program graph is given in the data flow form [25], while in Fig. 2(b) the associated connectivity (communication) graph is presented. In Fig. 2(c) 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. 2(a) and 2(c)) contain no operators [25]; they are used exclusively for I/O transfers.

The spatial location of the operators in the MPE mesh (Fig. 2(c)) will be obtained by a mapping program which has as input the program graph. It is provided also with information about the available grid of MPEs and outputs the (presumably final) positions of the operators (if no faults are considered for MPEs, CPEs or interconnects).

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

Two techniques have been devised to cope with this final stage of the graph injection process: a parallel and a serial one. They are presently under analysis. Simulation remains to be done.

The programming of the array takes place in two steps:

1. The MPEs are provided, in a global parallel or serial fashion (Figs. 3 and 4), with their operators and with the addresses of their two children (in a *binary* tree); thus they are programmed to perform their corresponding functions.
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 of the number of levels in the program graph. The latter should be smaller than the maximum *integer* coordinates in any possible communicating direction in the processor mesh (which equals the number of processors in that direction).

The programming step in the program graph injection process takes place as presented in Figs. 3 and 4. In both approaches, the motor for the advancement of the *operational 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 [21]).

These tokens, the operational and connectivity tokens, as well as the data tokens, are provided by the host, based on the output of the intermediate mapping program previously mentioned.

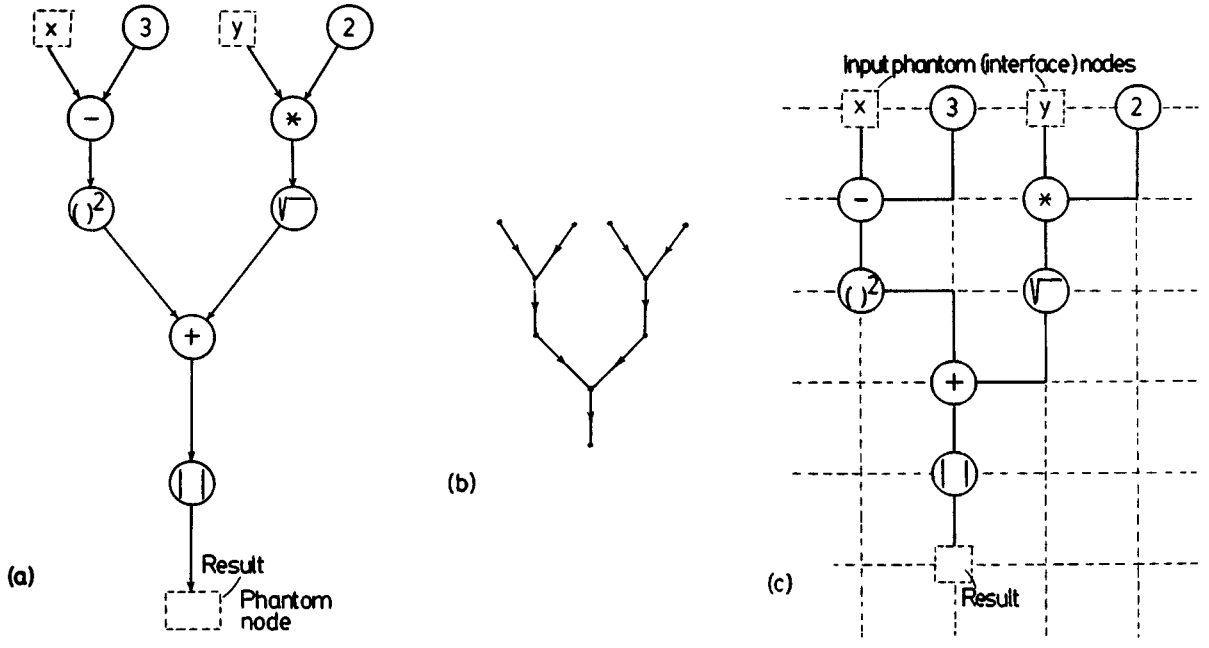

Fig. 2. Program graph "injection".
(a) Data flow program graph for $|(x-3)^2 + \sqrt{2}y|$.
(b) Associated connectivity (communication) graph.
(c) Locating the operators in the MPEs (same graph).

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

In Fig. 3 (showing parallel program graph injection), after power-up the connectivity lattice is automatically organized by the CPEs *in columns*, on which the *operational-connectivity tokens* are pushed in 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 is a serious problem for any VLSI supercomputer, given the present packaging techniques. The operational-connectivity tokens are pushed 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 precisely a number of steps 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 NONREADY response, in the asynchronous (handshaking) technique.

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

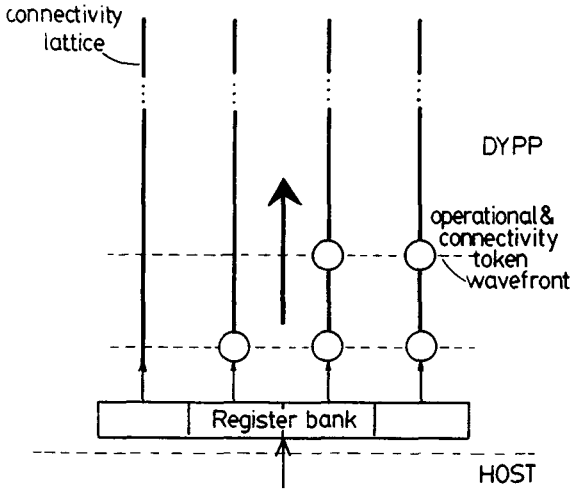

Fig. 3. Parallel, static, programming of DYPP (parallel injection).

In the serial scheme, Fig. 4, 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

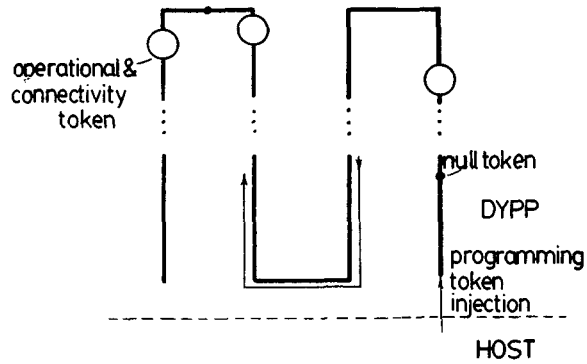

Fig. 4. Serial, static, programming of DYPP (serial injection).

above-mentioned manner. At the end of each column, they travel horizontally through some extra interconnections, provided to connect all columns end to end.

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

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

4. Levitating data by dynamic programming of DYPP

DYPP is intended to run programs with high parallelism. It certainly can be used to implement *systolic arrays*, for example.

Very long parallel programs (having many levels), might require a considerable area if we use the *static* techniques discussed in Section 3. These techniques 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 connections for the whole duration of program running).

Assuming that we are using a data flow type of computation, the *data* tokens are travelling, from the root(s) and other injection points, along the graph. Only a limited number of levels 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. 5) containing the active operators (instructions) at any time. This window would move along the graph, from the root to the pendant vertices (which are connected to the exit points from which emerge the final results).

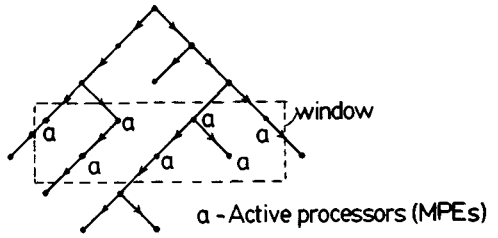

Fig. 5. A program graph activity window.

Even if for the graph in question, the height of this window (the number of graph levels included in it) is much smaller than the total number of levels in the graph, a static graph injection in the DYPP would normally lead to a correct implementation only if *the whole graph* is injected (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 run time, along the arcs of the graph. This means that *all* the graph nodes must be injected in the programming phase (Figs. 3,4) 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 here a new concept of *levitating data 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. 6). The programming and execution phases must now take place in parallel.

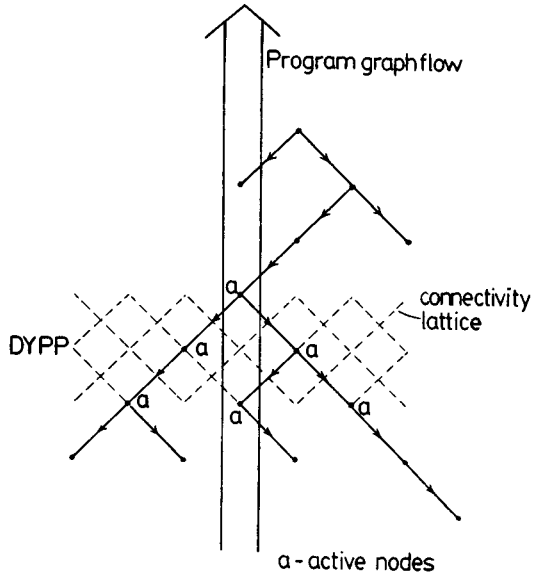

Fig. 6. The data levitation process.

The processor array must accomodate 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 (hardware) array, dynamically mapped into communication links.

If we compare this with the model of injection studied in section 2, we realize that the program graph moves itself through the array in a wave-like manner, carrying with it the *operation and connectivity tokens* mainly from the current "activity window". Synchronous or asynchronous schemes can be devised to support this movement, devised to keep dynamically the "activity window" inside DYPP.

This can be eventually seen as a *generalization of the systolic array concept*. In systolic arrays several flows of data and results in interaction coexist in the array. In our proposed dynamic architecture the *functional definitions* of the PEs find themselves in a flow movement, causing dynamic reconfigurability of the array (Fig. 7).

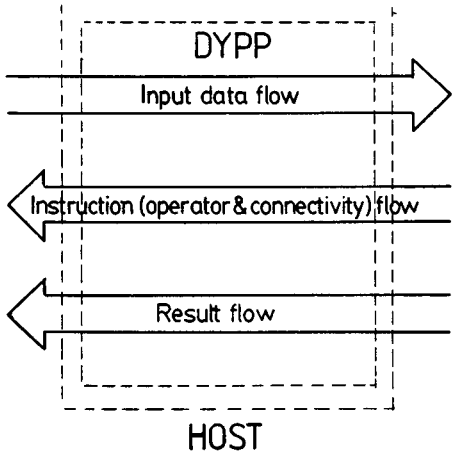

Fig. 7. A DYPP generalization of systolic arrays.

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 massive reductions 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 the input data flow and accepting the results data flow.

The concept of *data levitation* can possibly be applied successfully to general-purpose computations. 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 transaction should be the rule.

As a matter of fact, considerable current research in VLSI algorithms is directed toward creating algorithms matching the VLSI constraints [22], [23].

5. Conclusions and future directions for research

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

a) *A hierarchical (two-level) fault tolerant structure* separated on two conceptual levels (processing and connectivity), suitable for wafer-scale integration;

b) Provision of the *graph injection feature of programmability*;

c) Support for full *dynamic reconfigurability* (via the application of a new concept, called *data levitation* applied to program graphs, preferably in data flow form).

This architecture can implement *generalizations of systolic arrays*.

As directions of future research one can consider: fault-tolerant strategies for reconfiguration, implementation of long program loops in the presence of data levitation, mapping programs for graph injection and of course simulation.

6. Acknowledgments

The authors wish to thank the anonymous reviewers for their constructive suggestions which have helped in improving the revised version of the manuscript.

This research was supported by the Natural Sciences and Engineering Research Council of Canada through Operating Grant A1753.

7. References

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

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

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

[4] Kung, H.T. and Leiserson, C.E., "Systolic Arrays (for VLSI)", *Sparse Matrix Proceedings 1978*, Duff, I.S. and Stewart, G.H., (Eds.), SIAM, 1979, pp.256-282. (An earlier version appears in Chapter 8 of [3]).

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

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

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

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

[9] Batcher, K.E., "Design of a Massively Parallel Processor", *IEEE Trans. Comput.*, 1980, C-29(9), pp.48-56.

[10] Rosenberg, A.L., "Three-Dimensional VLSI: A Case Study", *J.A.C.M.*, 1983, 30(3), pp.397-416.

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

[12] Hedlund, K.S., *Wafer-Scale Integration of Parallel Processors*, Ph.D. thesis, Comp. Sci. Dept., Purdue Univ., Aug. 1982.

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

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

[15] Dennis, J.B., "Data Flow Supercomputers", *IEEE Computer*, vol.13, Nov. 1980, pp.48-56.

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

[17] Petritz, R.L., "Current Status of Large Scale Technology", *IEEE J. Solid-State Circuits*, SC-2, 4, Dec. 1967, pp.130-147.

[18] Varman, P.J., *Wafer-Scale Integration of Linear Processor Arrays*, Ph.D. thesis, University of Texas at Austin, 1983.

[19] Browning, S., *The Tree Machine: A Highly Concurrent Computing Environment*, Ph.D. thesis, Dept. of Computer Science, California Institute of Technology, 1980. See also [3], pp.295-312.

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

[21] Kung, S.-Y., Arun, K.S., Gal-Ezer, R.J. and Bhaskar Rao, D.V., "Wavefront Array Processor: Language, Architecture and Applications", *IEEE Trans. Comput.*, C-31(11), Nov. 1982, pp.1054-1066.

[22] Kung, H.T., "Let's Design Algorithms for VLSI Systems", *Proc. Caltech Conf. on VLSI*, Jan. 1979, pp. 65-90.

[23] Moldovan, D.I., "On the Design of Algorithms for VLSI Systolic Arrays", *Proc. IEEE*, Jan. 1983, pp.113-120.

[24] Gacs, P., *Reliable Computation with Cellular Automata*, Dept. of Computer Science, University of Rochester, Technical Report TR132, 1983.

[25] Davis, A.L. and Keller, R.M., "Data Flow Program Graphs", *IEEE Computer*, Feb. 1982, pp.26-41.

[26] Wills, D.W., *Ultra-fine Grain Processing Architectures*, M.Sc. Thesis, MIT, 1985.

[27] Seitz, Ch., *Experiments with VLSI Ensemble Machines*, Technical Report 5102-83, Computer Science, California Institute for Technology, 1983.

[28] Seitz, Ch., "Concurrent VLSI Architectures", *IEEE Trans. Comput.*, Dec. 1984, pp. 1247-1265.

[29] Siegel, H.J., Siegel, L.J., Kemmerer, F.C., Mueller, P.T., Smalley, H.E., Jr. and Smith, D.S., "PASM: A Partionable SIMD/MIMD System for Image Processing and Pattern Recognition", *IEEE Trans. Comp.*, Dec. 1981, vol.C-20, pp.934-947.