Wavefront Array Processor: Language, Architecture, and Applications

SUN-YUAN KUNG, MEMBER, IEEE, K. S. ARUN, STUDENT MEMBER, IEEE, RON J. GAL-EZER, STUDENT MEMBER, IEEE, AND D. V. BHASKAR RAO, STUDENT MEMBER, IEEE

***Abstract*—This paper describes the development of a wavefront-based language and architecture for a programmable special-purpose multiprocessor array. Based on the notion of computational wavefront, the hardware of the processor array is designed to provide a computing medium that preserves the key properties of the wavefront. In conjunction, a wavefront language (MDFL) is introduced that drastically reduces the complexity of the description of parallel algorithms and simulates the wavefront propagation across the computing network. Together, the hardware and the language lead to a programmable wavefront array processor (WAP). The WAP blends the advantages of the dedicated systolic array and the general-purpose data-flow machine, and provides a powerful tool for the high-speed execution of a large class of matrix operations and related algorithms which have widespread applications.**

***Index Terms*—Asynchrony, computational wavefront, concurrency, data-flow computing, matrix data-flow language, signal processing, systolic array, VLSI array processor, wavefront architecture.**

I. INTRODUCTION

A. VLSI Signal Processing

WITH the rapidly growing microelectronics technology leading the way, modern signal processing is undergoing a major revolution. The past two decades have witnessed a steep increase in the complexity of computations, processing speed requirements, and the volume of data handled in various signal processing applications. The availability of low cost, high density, fast VLSI devices has opened a new avenue for implementing these increasingly sophisticated algorithms and systems [1], [2]. While a major improvement in device speed is foreseen, it is in no way comparable to the rate of increase of throughput required by modern real-time signal processing. In order to achieve such increases in throughput rate, the only effective solution appears to be highly concurrent processing. Consequently, it has become a major challenge to update the current signal processing techniques so as to maximally exploit their potential for concurrent execution.

In a broad sense, the answer to this challenge lies in a cross-disciplinary research encompassing the areas of algorithm analysis, parallel computer design, and system applications. In this paper, we therefore introduce an integrated research approach aimed at incorporating the vast VLSI computational capability into modern signal processing applications.

The traditional design of parallel computers and languages is deemed unsuitable for our purposes. It usually suffers from heavy supervisory overhead incurred by synchronization, communication, and scheduling tasks, which severely hamper the throughput rate which is critical to real-time signal processing. In fact, these are the key barriers inherent in very large scale computing structure design. Moreover, although VLSI provides the capability of implementing a large array of processors on one chip, it imposes its own constraints on the system. Large design and layout costs [2] suggest the utilization of a repetitive modular structure. In addition, communication, which costs the most in VLSI chips in terms of area, time, and energy, has to be restricted (to *localized communication*) [1]. In general, highly concurrent systems require this locality property in order to reduce interdependence and ensuing waiting delays that result from excessive communication [1]. This locality constraint prevents the utilization of centralized control and global synchronization. The resulting use of *asynchronous distributed control* and *localized data flow* is an effective approach to the design of very large scale, highly concurrent computing structures.

B. A Special-Purpose VLSI Array Processor

The above restrictions imposed by VLSI will render the general-purpose array processor rather inefficient. We therefore restrict ourselves to a special class of applications, i.e., recursive[1] and local data-dependent algorithms, to conform with the constraints imposed by VLSI. This restriction, however, incurs little loss of generality, as a great majority of signal processing algorithms possess these properties. One typical example is a class of matrix algorithms. It has recently been indicated that a major portion of the computational needs for signal processing and applied mathematical problems can, in fact, be reduced to a basic set of matrix operations and other related algorithms [3], [4]. Therefore, a special-purpose parallel machine for processing these typical computational algorithms will be cost effective and attractive in VLSI system design.

Manuscript received January 11, 1982; revised June 18, 1982. This work was supported in part by the Office of Naval Research under Contract N00014-80-C-0457, N00014-81-K-0191, by the National Science Foundation under Grant ECS-80-16581, and by the Defense Advanced Research Projects Agency under Contract MDA903-79-C-0680.

The authors are with the Department of Electrical Engineering–Systems, University of Southern California, Los Angeles, CA 90089.

[1] In a recursive algorithm, all processors do nearly identical tasks, and each processor repeats a fixed set of tasks on sequentially available data.

0018-9340/82/1100-1054$00.75

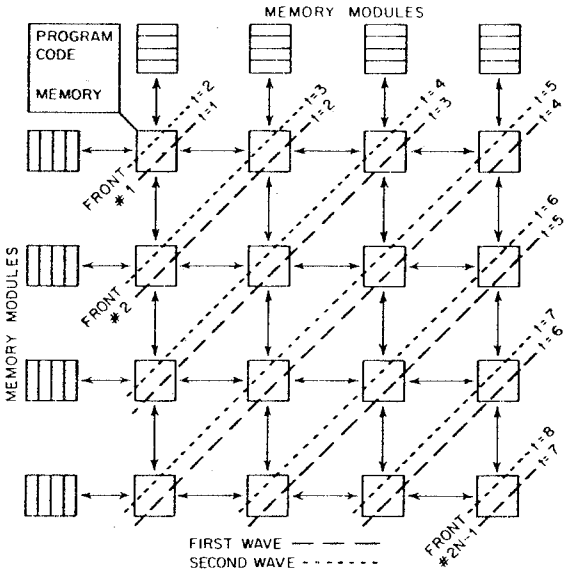


Fig. 1. The WAP configuration.

Very significantly, these algorithms involve repeated application of relatively simple operations with regular localized data flow in a homogeneous computing network. This leads to an important notion of *computational wavefront*, which portrays the computation activities in a manner resembling a wave propagation phenomenon. More precisely, the recursive nature of the algorithm, in conjunction with the localized data dependency, points to a continuously advancing wave of data and computational activity. The computational sequence starts with one element and propagates through the processor array, closely resembling a physical wave phenomenon (cf. Fig. 1). In fact, all algorithms that have *locality* and *recursivity* (and are thus implementable on our VLSI array processor) will exhibit this wave phenomenon. Therefore, the notion of a computational wavefront has attracted the attention of many researchers [2], [5]–[13].

The wavefront concept provides a firm theoretical foundation for the design of highly parallel array processors and concurrent languages. In addition, this concept appears to have some distinct advantages.

First, the wavefront notion drastically reduces the complexity in the description of parallel algorithms. The mechanism provided for this description is a special-purpose, wavefront-oriented language [7], [9], [10]. Rather than requiring a program for each processor in the array, this language allows the programmer to *address an entire front of processors.*

Second, the wavefront notion leads to a wavefront-based architecture which preserves Huygen's principle [14], and ensures that wavefronts never intersect. Therefore, a wavefront architecture can provide *asynchronous waiting* capability, and consequently, can cope with timing uncertainties, such as local clocking, random delay in communications, and fluctuations of computing times. In short, the notion lends itself to a (asynchronous) data-flow computing structure that conforms well with the constraints of VLSI.

The integration of the wavefront concept, the wavefront language, and the wavefront architecture leads to a programmable computing network, which we will call the wavefront array processor (WAP). The WAP is, in a sense, an *optimal tradeoff* between the *globally synchronized* and *dedicated systolic array* [1], [15], [16] (that works on a similar set of algorithms), and the *general-purpose data-flow multiprocessors* [17]–[23]. It provides a powerful tool for the high-speed execution of a large class of algorithms which have widespread applications.

C. Organization

The organization of the rest of the paper is as follows. Section II elaborates on the computational wavefront. Section III proposes a wavefront-oriented language (MDFL) for programming the WAP, and provides a programming methodology. Section IV explains a possible hardware organization and architecture for the WAP. Section V illustrates some applications of the WAP by means of MDFL programs. Section VI compares the WAP to other array processors, such as the systolic array, data-flow multiprocessors, and conventional SIMD arrays like the Illiac IV.

II. Concept of Computational Wavefront

The wavefront array processor is configured in a square array of $N \times N$ processing elements with regular and local interconnections (cf. Fig. 1).

The computing network serves as a (data) wave-propagating medium. The notion of computational wavefront can be best explained by a simple example. To this end, we shall consider matrix multiplication as being representative. Let $A = \{a_{ij}\}$, $B = \{b_{ij}\}$, and $C = A \times B = \{c_{ij}\}$ all be $N \times N$ matrices. The matrix A can be decomposed into columns A_i and matrix B into rows B_j, and therefore,

$$C = A_1 * B_1 + A_2 * B_2 + \cdots + A_N * B_N. \tag{1}$$

The matrix multiplication can then be carried out in N recursions, executing

$$C^{(k)} = C^{(k-1)} + A_k * B_k \tag{2}$$

recursively for $k = 1, 2, \cdots, N$.

The exploitation of parallelism is now evident with the availability of $N \times N$ processors. A parallel algorithm for matrix multiplication is fairly simple. However, most existing parallel programs would need global interconnections in the computing network, while localized interconnections and data flow are much more desirable in VLSI systems.

The topology of the matrix multiplication algorithm can be mapped naturally onto the square, orthogonal $N \times N$ matrix array of the WAP (cf. Fig. 1). To create a smooth data movement in a localized communication network, we make use of the computational wavefront concept. For the purpose of this example, a wavefront in the processing array will correspond to a mathematical recursion in the algorithm. Successive pipelining of the wavefronts will accomplish the computation of all recursions.

As an example, the computational wavefront for the first recursion in matrix multiplication will now be examined.

Suppose that the registers of all the processing elements (PE's) are initially set to zero:

$$C_{ij}^{(0)} = 0 \qquad \text{for all } (i, j);$$

the entries of A are stored in the memory modules to the left (in columns), and those of B in the memory modules on the top (in rows). The process starts with PE (1, 1), where

$$C_{11}^{(1)} = C_{11}^{(0)} + a_{11} * b_{11} \tag{3}$$

is computed. The computational activity then propagates to the neighboring PE's (1, 2) and (2, 1), which will execute

$$C_{12}^{(1)} = C_{12}^{(0)} + a_{11} * b_{12} \tag{4}$$

and

$$C_{21}^{(1)} = C_{21}^{(0)} + a_{21} * b_{11}. \tag{5}$$

The next front of activity will be at PE's (3, 1), (2, 2), and (1, 3), thus creating a computation wavefront traveling down the processor array. This computational wavefront is similar to optical wavefronts (they both obey Huygen's principle) since each processor acts as a secondary source and is responsible for the propagation of the wavefront. It may be noted that wave propagation implies localized data flow. Once the wavefront sweeps through all the cells, the first recursion is over. As the first wave propagates, we can execute an *identical* second recursion in parallel by *pipelining* a second wavefront immediately after the first one. For example, the (1, 1) processor will execute

$$C_{11}^{(2)} = C_{11}^{(1)} + a_{12} * b_{21} \tag{6}$$

and so on. In general, the (i, j)th processor will execute the kth recursion,

$$C_{ij}^{(k)} = a_{i1} * b_{1j} + a_{i2} * b_{2j} + \cdots + a_{ik} * b_{kj}. \tag{7}$$

The pipelining is feasible because the wavefronts of two successive recursions will never intersect (Huygen's wavefront principle), as the processors executing the recursions at any given instant will be different, thus avoiding any contention problems.

Locality, *regularity*, *recursivity*, and *concurrency* lead to the wavefront phenomenon. Thus, all algorithms which possess these properties will exhibit computational wavefronts. The notion of computational wavefronts leads to a wavefront-based language (a modified data-flow language) to program the processor array. This language (called the matrix data-flow language) is especially powerful for matrix and other related algorithms. It will be developed in the next section.

III. A Wavefront Language: Matrix Data-Flow Language

In contrast to the heavy burden of scheduling, resource sharing, as well as the control of processor interactions encountered in programming a general-purpose multiprocessor, the computational wavefront notion can facilitate the description of parallel algorithms and drastically reduce the complexity of parallel programming. In this section, we introduce a wavefront language that is tailored towards the description of computational wavefronts and the corresponding data flow in a large class of algorithms (which exhibit the recursivity and locality mentioned earlier).

Since matrix algorithms are typical of this class, we call the language the matrix data-flow language (MDFL). This wavefront language permits the array processor to be programmable, broadens the range of applications, and also makes possible the simulation and verification of parallel algorithms.

A. Matrix Data-Flow Language

There exist two approaches to programming the WAP: a local approach describing the actions of each processing element, and a global approach describing the actions of each wavefront. To allow the user to program the WAP in both of these fashions, two versions of MDFL are proposed: global and local MDFL.

A global MDFL program describes the algorithm from the viewpoint of a wavefront, while a local MDFL program describes the operations of an individual processor. More precisely, the perspective of a global MDFL programmer is of *one wavefront passing across all the processors*, while the perspective of a local MDFL programmer is that of *one processor encountering a series of wavefronts*.

From the macroscopic point of view, a higher level language, closer to the algorithm, is desired for reducing the heavy burden on the programmer. Global MDFL provides such a tool, as it is easier to view the algorithm as a series of wavefronts. At the microscopic level, each PE executes its own set of instructions and performs localized interprocessor transactions. Implementing a program on such a system requires transforming the high level description (in global MDFL) into a set of lower level programs (in local MDFL) for the individual processors.[2] We have developed a compiler to do such a mapping of global wavefront-oriented MDFL programs into local programs for individual processors.

B. MDFL Instruction Set

At each front of activity, the computational wavefront performs similar tasks in all the processors involved (cf. Section II, matrix multiplication example and Section III-D on programming methodology). Hence, global (wavefront) instructions and local (processor) instructions are nearly identical. Most of the MDFL instruction set is, therefore, common to both global and local MDFL.

Table I is a complete list of the MDFL instruction repertoire. For the complete semantics and more detailed syntax, the reader is referred to a recent publication [10]. Unless otherwise specified, the following instructions belong to both global and local MDFL. The proposed instruction set is functionally complete, but improvements and modifications are still underway to incorporate additional features such as double precision, etc. There also appears to be room for the

[2] It suffices, in general, to describe a wavefront algorithm at four locations of the array: corner, FirstRow, FirstColumn, and Interior PE's. Consequently, every global MDFL program gets compiled into four slightly different local programs.

TABLE 1
MDFL INSTRUCTION SET

```
Data Transfer Instructions

FLOW <SOURCE REGISTER>, <DIRECTION>;
FETCH <DESTINATION REGISTER>, <DIRECTION>;
READ;


Recursion Oriented Instructions

REPEAT ... UNTIL TERMINATED;
WHILE WAVEFRONT IN ARRAY DO
        BEGIN ... END;
SET COUNT <NUMBER OF WAVEFRONTS>;
DECREMENT COUNT;
ENDPROGRAM.


Conditional Instructions

IF EQUAL THEN <STATEMENT>;
IF NOT-EQUAL THEN <STATEMENT>;
IF GREATER THEN <STATEMENT>;
IF LESS-THAN THEN <STATEMENT>;
IF <DIRECTION> DISABLED THEN <STATEMENT>;
CASE KIND =
        (1,1) : <STATEMENT>;
        (1,*) : <STATEMENT>;
        (*,1) : <STATEMENT>;
        INT   : <STATEMENT>;
   ENDCASE;


Internal Processor Instructions

TSR     <SOURCE>, <DESTINATION>;
ADD     <SOURCE #1>, <SOURCE #2>, <DESTINATION>;
SUB     <SOURCE #1>, <SOURCE #2>, <DESTINATION>;
MULT    <SOURCE #1>, <SOURCE #2>, <DESTINATION>;
DIV     <SOURCE #1>, <SOURCE #2>, <DESTINATION>;
SQRT    <SOURCE>, <DESTINATION>;
CMP     <SOURCE #1>, <SOURCE #2>;
TST     <SOURCE>;
STORE;
NOP;
RESET;
BEGIN ... END;
DISABLE-SELF;
```

development of a higher level language suitable for the nonspecialist programmer.

C. Programming Methodology

In this subsection, we provide guidelines for programming in global MDFL. The most straightforward method of programming the WAP would be to explicitly spell out the actions of each wavefront at each of its $(2n-1)$ positions (fronts) (cf. Fig. 1). Nevertheless, the regularity and recursivity in almost all matrix algorithms allows us to assume the following.

1) Space Invariance: The tasks performed by a wavefront in a particular kind of processor must be identical at all $(2n-1)$ fronts.

2) Time Invariance: Recursions are identical.

Accordingly, global MDFL provides two repetitive constructs, the space repetitive construct

WHILE WAVEFRONT IN ARRAY DO
BEGIN ⟨TASK T⟩ END

(so that T is repeated at all fronts), and the time repetitive construct

REPEAT ⟨ONE RECURSION⟩ UNTIL TERMINATED

(so that the same recursion is repeated).

The REPEAT construct is inherently concurrent in that successive wavefronts are pipelined through the array. As soon as the kth wavefront is propagated, the (1, 1) processor initiates the $(k+1)$st wavefront.

To allow for more than one wavefront per recursion, the complete global MDFL program will have the syntax

```
BEGIN
  SET COUNT ( );
  REPEAT
      ⟨TASKS A⟩;
      WHILE WAVEFRONT IN ARRAY DO
        BEGIN
          ⟨TASKS B⟩;
        END;
      WHILE WAVEFRONT IN ARRAY DO
        BEGIN
          ⟨TASKS C⟩;
        END;
      ⟨TASKS D⟩;
      DECREMENT COUNT;
  UNTIL TERMINATED;
ENDPROGRAM.
```

Each recursion will execute the instructions within the REPEAT ··· UNTIL construct. The number of recursions is set by SET COUNT. In this example, a recursion consists of two wavefronts. At the start, tasks A are performed only at the (1, 1) processor. The first wavefront of each recursion will perform tasks B at each of its $(2n-1)$ fronts. The second wave will execute tasks C in each of these fronts immediately after tasks B have been concluded. It should be noted that the number of different wavefronts within a recursion may vary from one application to another. At the end of each recursion, COUNT is decremented. When it becomes zero, TERMINATED is set and a "phase" of identical recursions is over.

The corresponding local MDFL program for interior processors (cf. Section III-A) will be

```
REPEAT
    ⟨TASKS B'⟩
    ⟨TASKS C'⟩
UNTIL TERMINATED;
```

where B' and C' are the compiled versions of B and C, with only relevant portions of the CASE statement extracted. The conversion of a global program into its local versions is thus fairly straightforward.

Certain syntax rules [10] are needed to ensure that there is no circular waiting for data between adjacent PE's. They also ensure that neighbors will not contend for the interprocessor bus at the same time. Concisely, the rules dictate that FETCHes in one direction precede (and equal in number) the FLOWs in the opposite direction, and that the data FETCHing precede any computation (cf. data-driven computation in Section IV-A). The complete proof of the claim that these rules will prevent deadlock and bus contention is omitted here [24].

Based on the above guidelines, a complete global MDFL program for matrix multiplication follows.

More programming examples will be provided in Section V.

Program 1. Matrix multiplication.

```
Array Size:     N x N

Computation:    C = A x B
                 th                (k)     (k-1)
                k   wavefront:  c    =  c      + a   b
                                 ij      ij       ik kj

                                k = 1, 2, ... , N

Initial:    Matrix A is stored in the Memory Module  (MM)

        on the left (stored row by row).  Matrix B is in

        MM on  the top and is  stored  column by column.

Final:      The result will be in the C registers.


1:      BEGIN
        SET COUNT 3;
        REPEAT;
           WHILE WAVEFRONT IN ARRAY DO
5:           BEGIN
                FETCH B, UP;
                FETCH A, LEFT;
                FLOW A, RIGHT;
                FLOW B, DOWN;
                ! C <- C + AB; *
10:             MULT A, B, D;
                ADD C, D, C;
             END;
           DECREMENT COUNT;
        UNTIL TERMINATED;
15:     ENDPROGRAM.

* Comments are enclosed between "!" and ";"
```

D. Summary of MDFL Features

Future VLSI multiprocessors must support massive concurrency to achieve a significant increase in performance; consequently, a base language for parallel computers must allow expression of concurrency of program execution on a large scale [20]. However, few languages support the idea of large scale concurrency, and only weak notions of locality exist in most of them [25].

It must be noted that a parallel program is not just an ensemble of separate programs for individual processors. More importantly, it should also define coordination between PE's including their interdependence for data and the sequencing of their tasks. As such, it is a tall order.

At present, data-flow languages appear to be the best candidate as the base language of parallel computers. Data-flow languages are asynchronous, data driven, and algorithmic. They avoid centralized control and shared memory to achieve *asynchrony* and *maximal concurrency* [20], [26]–[29].

MDFL basically possesses all these properties of data-flow languages. It shares the principle of *data-activated computation* and its consequent advantages. In addition, MDFL has a rather distinctive feature of regularity, and is built around the notion of locality. It is very close to the algorithms; hence, MDFL programs are modular and easy to understand. MDFL permits the programmer to address a front of processors at the same time, instead of programming individual processors separately. Being wavefront oriented, it permits viewing the algorithm as the repetition of a computational sequence (i.e., the wavefront) progressing through the array. In short, the wavefront notion makes it possible to program an array of asynchronous processors in a simplistic fashion, and leads to simple readable MDFL programs.

IV. Wavefront Architecture

The hardware of the processing array is designed to support MDFL. The main architectural considerations include four general aspects: 1) interprocessor communications, based on wavefront requirements; 2) the basic PE configuration; 3) interfacing with the host computer; and finally, 4) the extendability and reconfigurability issues.

A. Interprocessor Communication

The configuration of the processor array is as shown in the schematic diagram of Fig. 1. It provides for data and control communications between orthogonally adjacent processors and links to memory modules through the first row and first column of processors.

To simulate the phenomenon of wavefront propagation, the processors in the array must *wait* for a primary wavefront (of data), then perform its computation and, finally act as a *secondary source* of new wavefronts. To implement this wait, processors are provided with data transfer buffers. Hence, a FETCHing of data involves an inherent WAITing for the buffer to be filled (DATA READY) by the adjacent data sourcing processor. Thus, if the software ensures that the processor always performs data FETCH before the computation (cf. syntax rules of Section III-C), *the processing will not be initiated until the arrival of the data wavefront* (this is similar to the concept of data flow machines [17]–[23], [26]–[29]). Each processor can FLOW data to the input buffers of the neighboring PE's, thus acting as a secondary source of data wavefronts (Huygen's principle). To avoid the overrunning of data wavefronts (in conformation with Huygen's principle), the processor hardware ensures that a processor cannot send new data to the buffer unless the old data have been used by the neighbor. Thus, the wavefront concept suggests that interprocessor communication employ buffers and "DATA READY/DATA USED" flags between adjacent processors.

The WAITs for wavefronts of data allow for *globally asynchronous* operation of processors, i.e., there is no need for global synchronization. Synchronization is a very critical issue in parallel processing, especially when one considers large scale systems. Two opposite timing schemes come to mind, namely, the synchronous and the asynchronous timing approaches. In the synchronous scheme, there is a global clock network which distributes the clocking signals over the entire chip. The global clock beats out the rhythm to which all the PE's in the array execute their tasks. In the basic synchronous configuration, all the PE's operate in unison, all performing the same identical operation. In contrast, the asynchronous scheme involves no global clock, and information transfer is by mutual convenience between each PE and its immediate neighbors. Whenever the data are available, the transmitting PE informs the receiver of the fact, and the receiver accepts the data when it needs them. It then conveys to the sender the information that the data have been used. This scheme can be implemented by means of a simple handshaking protocol [10], [30] (cf. Fig. 2).

B. PE Configuration

The proposed hardware for the processing elements of the

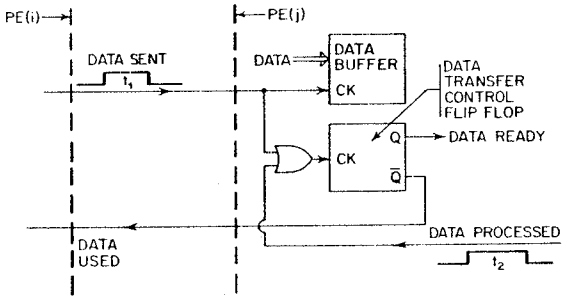


Fig. 2. The handshaking scheme.

WAP involves a simple modular structure. A basic architecture is schematically shown in Fig. 3. Each processor is a hardware interpreter of local MDFL, and reflects a relatively conventional architecture consisting of an internal program memory, a control unit, an ALU, and a set of registers. The only exception is the PE's interfacing with its neighbors. Every processing element has a bidirectional buffer and independent status and control flags to each of its four adjacent elements (cf. Fig. 3). These buffers are supported by an appropriate multiplexing subsystem, under the control unit's supervision. Except for the extended capabilities of the (1, 1) processor and the memory links of the first row and first column of processors, all of the processing elements are essentially identical. Hence, from both the software (cf. footnote 2, Section III-A) and hardware viewpoints, it suffices to group the PE's into four types: Corner, FirstRow, FirstCol, and Interior processors.

For signal processing applications, the most important requirements are the accuracy and speed of the arithmetic units. These are determined by extensive numerical error analysis of the algorithms under consideration. Fortunately, most matrix algorithms have a rather complete error analysis documentation [31], [32]. Another important factor in the selection of the ALU is the speed/hardware tradeoff. The decision between serial versus parallel and fixed- versus floating-point arithmetic will be affected by this consideration. In addition, the potential for concurrent execution and internal pipelining within the processing elements may be exploited for further speedup of the processing rate.

C. *Interfacing with the Host Computer and Extendability*

The interaction between the host computer and the WAP includes loading of programs and I/O of data before and after processing. All these transactions may be carried out by means of a smart interface, the nature of which is currently under investigation. This interfacing will minimize the host computer's involvement in the transaction, permitting efficient transfer of large blocks of data.

The word "extendability" refers to the capability of handling matrix operations of larger order with a smaller sized array processor. Software solutions to the extendability problem are often adopted via efficient partitioning (decomposition) of the algorithms' tasks (e.g., the divide-and-conquer method for matrix inversion [16], [33]) and intensive utilization of the host computer's memory.

In order to handle a larger matrix without incurring the great delay involved in accessing the host computer frequently, we propose a globally parallel, locally sequential (GPLS) scheme, which is explained below.

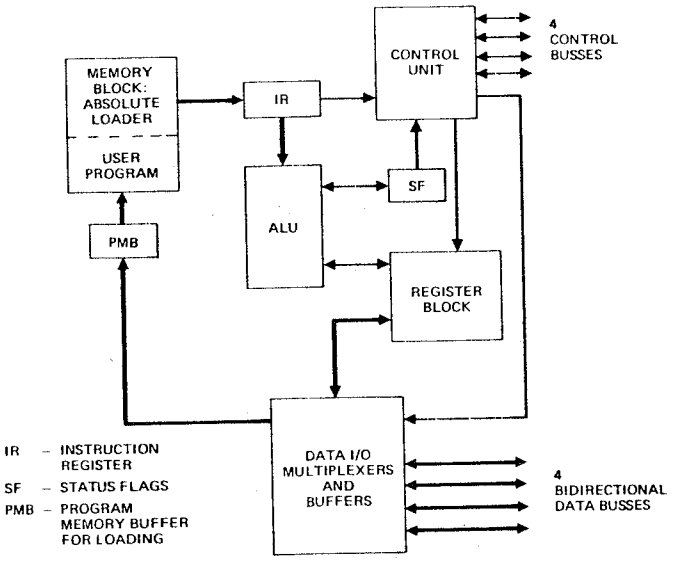


Fig. 3. Architecture of an INTerior PE.

Suppose that the array size is $N \times N$, and the size of the operand matrix A is $N' \times N'$ where $N' = mN$ and m is an integer. Then a partitioning of A into submatrices of size m will be

$$A = [A_{i,j}], \qquad i = 1, \cdots, N; j = 1, \cdots, N$$

and

$$A_{i,j} = [a_{(i-1)m+k,(j-1)m+p}], \qquad k, p = 1, \cdots, m.$$

Each processor then works on an $m \times m$ submatrix of data stored in its local memory. While the processors can work in parallel, an individual processor works on its $m \times m$ data sequentially, one after the other; hence, the term GPLS. We assert that the inclusion of local memory [two-dimensional array $M(i, j)$ and linear array $G(i)$] will not affect the global interconnection. More curiously, global MDFL also needs only a minimal modification for the extension. An extended MDFL syntax and semantics have been developed to effectively (i.e., transparently) handle larger size matrix operations. A sample program for multiplication of matrices of larger size ($mN \times mN$) is listed here.

Program 2. Extended matrix multiplication.

```
BEGIN
    EQUIVALENCE (B, G(1));
    EQUIVALENCE (C,M);
    SET COUNT mN;
    REPEAT
        WHILE WAVEFRONT IN ARRAY DO
        SCAN BY ROW 1 TO m DO
        BEGIN
            FETCH B, UP;
            FETCH A, LEFT;
            MULT  A, B, R;
            ADD   C, R, C;
            FLOW  A, RIGHT;
            FLOW  B, DOWN;
            MOVE  RIGHT, DOWN;
        END;
        DECREMENT COUNT;
    UNTIL TERMINATED;
ENDPROGRAM.
```

Here, each PE scans its submatrix in local memory $M(i, j)$ row by row, and C refers to $M(i, j)$ where i and j are the current values of the scan counters I and J. Similarly, B refers to $G(i)$.

There exist other solutions to the extendability problem. One class of solutions involves reconfiguring the WAP to simulate, for instance, a linear array of processors or a bilinear array [10]. This will involve additional and adaptable interconnections and a much expanded software; several possibilities are being explored.

In summary, the wavefront concept dictates, to some extent, the type of interprocessor communication and the architecture of the individual processing elements, First, the regularity of tasks performed at each activity front allows the processors to be identical. Hence, it calls for a *modular* design, with each processor as a basic functional module. Second, it requires interprocessor communication to be *localized* and "buffered," and that the PE's be *data driven*. This permits the WAP to be *globally asynchronous* and locally synchronized. As a result, the WAP is well suited for VLSI implementation, and has the key advantages of both the systolic array and data-flow multiprocessors (cf. Section VI).

V. Applications and Programming Examples

The power and flexibility of the WAP is best demonstrated by the broad range of algorithms that can be programmed in MDFL. The WAP is applicable to all algorithms that possess recursivity and locality. Such algorithms can be roughly classified into three groups.

1) Basic Matrix Operations:

a) Matrix Multiplication: array size: $N \times N$; processing time: $N(t_a + t_m)$.

b) LU Decomposition: array size: $N \times N$; processing time: $N(t_a + 2t_m + t_d)$.

c) LU Decomposition with Localized Pivoting: array size: $N \times N$; processing time: $N(t_a + 2t_m + t_d)$.

d) Given's Algorithm: array size: $N \times N$; processing time: $N(4t_a + 8t_m + 4t_d + 2t_q)$.

e) Back Substitution: array size: $N \times N$; processing time: $N(t_a + t_m + t_d)$.

f) Null Space Solution: array size: $N \times N$; processing time: $N(t_a + 2t_m + t_d)$.

g) Matrix Inversion: array size: $N \times N$; processing time: $N(2t_a + 3t_m + t_d)$.

Parallel algorithms for eigenvalue and singular value decomposition are also being investigated.

2) Special Signal Processing Algorithms:

a) Toeplitz System Solver: array size: $2 \times N$; processing time: $N(t_a + t_m + t_d)$.

b) Linear Convolution: array size: $1 \times N$; processing time: $N(t_a + t_m)$.

c) Recursive Filtering: array size: $1 \times N$; processing time: $2N(t_a + t_m)$.

We are also looking into adaptive filtering and discrete Fourier transformation, etc.

3) Other Algorithms:

a) Partial Difference Equation Solution: array size: $N \times N$; processing time: $3m(3t_a + t_d)$ for m iterations.

b) Sorting: array size: $1 \times N$; processing time: $3nt_c$.

Here, t_a, t_m, t_d, t_c, and t_q stand for addition, multiplication, division, comparison, and square root execution times, respectively. Each of the above three categories will be separately discussed in Sections V-A, V-B, and V-C, respectively. Ultimately, real system applications will be the only objective criterion for justifying and reconfirming the usefulness of the WAP. Investigations are, therefore, being conducted into its use in such diverse fields as image analysis, speech recognition, Kalman filtering, nonlinear filtering, adaptive filtering, smoothing, prediction, optimization, detection, spectrum analysis, and other advanced signal processing applications [12].

In fact, some recent reports have indicated a promising and encouraging trend. For instance, a computing structure similar to the WAP has been applied for parallel processing (for dynamic time warping [34]) in speech recognition. It has also been recently reported [13] that the wavefront concept can be effectively utilized to solve the singular value decomposition (SVD) problem, which is crucial for image processing and beam-forming applications. We are currently looking into an MDFL program for such an SVD algorithm. Finally, for a system application in real-time signal processing, a case of adaptive array processing systems for high resolution directivity pattern analysis is studied in an earlier publication [10]. There, a combined application of the WAP, MDFL, and modern spectrum analysis is presented.

A. Matrix-Based Algorithms

Example 1—LU Decomposition: For solving the linear system of equations $Ax = b$, a possible approach is decomposing the square matrix A into a product of a lower triangular matrix L (with unit diagonal elements) and an upper triangular matrix U. We present here the first recursion of the LU decomposition to illustrate the algorithm.

Let $A^{(0)} = A$; then

$$A^{(0)} - L_1 U_1 = \begin{bmatrix} 0 & 0 \cdots 0 \\ 0 & \\ \vdots & A^{(1)} \\ 0 & \end{bmatrix}$$

where

$$L_1 = -\frac{1}{a_{11}}\{a_{11}, a_{21}, \cdots, a_{N1}\}^T \tag{i}$$

and

$$U_1 = \{a_{11}, a_{12}, \cdots, a_{1N}\}. \tag{ii}$$

The other recursions will follow along the same lines [7], [8]. A global MDFL program for LU decomposition is provided below.

Program 3. LU decomposition

```
Array Size:     N x N with dynamic reduction.

Computation:    Decompose  A = LU.

                                                    -1
      th              (k)    (k-1)   (k-1) (k-1)  (k-1)
   k  wavefront:  a     = a     - a     a     a     a
                   ij     ij      ik    kk     kj

Initial:        A is stored in the A registers.

Final:          L and U will be in the left and upper Memory
                Modules, respectively.
```

```
1:      BEGIN
        ! Initialization phase: one wavefront;
        SET COUNT 1;
        REPEAT
           WHILE WAVEFRONT IN ARRAY DO
5:           BEGIN
                FLOW A, RIGHT;
                CASE KIND =
                (1,*), INT:    BEGIN
                                  FETCH B, LEFT;
10:                               FLOW B, LEFT;
                               END;
                END CASE;
             END;
           DECREMENT COUNT;
15:     UNTIL TERMINATED;

        SET COUNT <N>;
        REPEAT
           WHILE WAVEFRONT IN ARRAY DO
             BEGIN
20:             FETCH A, RIGHT;
                CASE KIND =
                (1,1):     BEGIN
                                  DIV 1, A, B;
                                  FLOW A, UP;
25:                        END;
                (1,*):     BEGIN
                                  TSR A, B;
                                  FLOW A, UP;
                           END;
30:             (*,1):     BEGIN
                                  FETCH B, UP;
                                  MULT A, B, C;
                                  FLOW C, LEFT;
                                  FLOW C, RIGHT;
35:                        END;
                INT  :     BEGIN
                                  FETCH C, LEFT;
                                  FETCH B, UP;
                                  MULT B, C, D;
40:                               SUB A, D, A;
                                  FLOW C, RIGHT;
                                  FLOW A, UP;
                                  END;
                END CASE;

45:             FLOW B, DOWN;
                IF DOWN DISABLED THEN DISABLE-SELF;
             END;

           WHILE WAVEFRONT IN ARRAY DO
             BEGIN
50:             CASE KIND =
                (1,*), INT:    BEGIN
                                  FETCH A, DOWN;
                                  FLOW A, LEFT;
                               END;
55:             END CASE;

                IF RIGHT DISABLED THEN DISABLE-SELF;
             END;
           DECREMENT COUNT;
        UNTIL TERMINATED;
60:     ENDPROGRAM.
```

Each recursion consists of a main wavefront (lines 18–47) and an alternate wavefront (48–57). The main wavefront computes L_i at the $(*, i)$ processors and sends it left to form the columns of L in the left memory modules. It also computes U_i at the $(1, *)$ processors and sends it up to form the rows of U. In the interior processors, the outer product $L_i * U_i$ is computed and $A^{(i)}$ is formed. The main wavefront must also send the new a_{ij}'s up, as the first step in the north–west movement of the matrix $A(1)$. To complete the second step of the north–west movement of data, we need an alternate wavefront (lines 48–57).

The MDFL program for LU decomposition also illustrates how the array size is dynamically reduced (lines 46, 56). The REPEAT loop (lines 17–59) causes the above sequence to be performed N times. Because there is data flow in a direction opposite that of the wave propagation (line 20, for example), there must be an intialization phase (syntax rule 3), and lines 3–15 perform this initialization.

Example 2—QR Decomposition by Givens Transformation: A sequence of Givens rotations can be employed to achieve QR decomposition of a matrix A. This is done by annihilating elements of the matrix A to finally yield a triangular matrix R. Annihilation of an element $a_{p,1}$ is illustrated below.

$$\begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & c & -s & \\ & & s & c & \\ & & & & \ddots \\ & & & & & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & x & \cdots \\ \vdots & \vdots & \\ 0 & x & \\ a'_{p+1,1} & x & \\ \vdots & \vdots & \\ x & x & \cdots \end{bmatrix} = \begin{bmatrix} 0 & x & \cdots \\ & & \\ 0 & x & \\ 0 & x & \\ a'_{p+1,1} & x & \\ \vdots & \vdots & \\ x & x & \cdots \end{bmatrix}$$

where

$$r = \{a_{p,1}^2 + a_{p+1,1}^2\}^{1/2}$$

$$c_p = \frac{a_{P+1,1}}{r}; \quad s_p = \frac{a_{p,1}}{r}.$$

Elements in row p are modified to

$$a'_{p,j} = c \cdot a_{p,j} - s \cdot a_{p+1,j} \tag{i}$$

and in row $p + 1$ to

$$a'_{p+1,j} = s \cdot a_{p,j} + c \cdot a_{p+1,j}. \tag{ii}$$

Referring to the figure, similar computations can be used to annihilate $(n - 1)$ elements of the first column. This will constitute one recursion. Other recursions will be on the same lines, except that, similar to LU, the matrix has to be moved west.

Program 4. Given's algorithm.

```
Array Size:     N x N

Computation:    Givens rotation; each recursion annihilates one column.

Initial:        A is stored in the A registers.

Final:          R is stored in the A registers.

1:      BEGIN
        SET COUNT 1;
        REPEAT
           WHILE WAVEFRONT IN ARRAY DO
5:            BEGIN
                 CASE KIND =
                 (*,1), INT:  FLOW A, UP;
                 ENDCASE;
                 IF DOWN DISABLED THEN DISABLE-SELF;
10:           END;
           DECREMENT COUNT;
        UNTIL TERMINATED;

        SET COUNT <N>;
        REPEAT
15:        WHILE WAVEFRONT IN ARRAY DO
              BEGIN
                 FETCH B, DOWN;
                 CASE KIND =
                 (*,1), INT:  FETCH A, UP;
20:              (*,1),(1,1): BEGIN      ! R1 = √(A^2 + B^2)   ;
                                 MULT A, A, R1;
                                 MULT B, B, R2;
                                 ADD R1, R2, R1;
                                 SQRT R1, R1;
25:                              DIV B, R1, C;
                                 DIV A, R1, S;
                                 TSR R1, B;
                              END;

                 (1,*), INT:  BEGIN
30:                              FETCH C, LEFT;
                                 FETCH S, LEFT;
                                 ! A <- CP - SP and B <- SP + CP;
                                 MULT C, A, R1;
                                 MULT S, B, R2;
                                 SUB R1, R2, R1;
35:                              MULT S, A, R2;
                                 TSR R1, A;
                                 MULT C, B, R1;
                                 ADD R1, R2, B;
                                 FLOW A, LEFT;
40:                           END;
                 ENDCASE;
```

(*continued*)

```
                FLOW B, DOWN;
                FLOW C, RIGHT;
                FLOW S, RIGHT;
45:             IF RIGHT DISABLED THEN DISABLE-SELF;
              END;

          WHILE WAVEFRONT IN ARRAY DO
              BEGIN
                FETCH A, RIGHT;
50:             CASE KIND =
                (*,1), INT:  FLOW A, UP;
                ENDCASE;
                IF DOWN DISABLED THEN DISABLE-SELF;
              END;    ! OF second wavefront;

55:       DECREMENT COUNT;
          UNTIL TERMINATED;
        ENDPROGRAM.
```

The Given's algorithm, like LU, requires an initialization phase (lines 3–12) due to a FETCH from DOWN instruction in line 17 (syntax rule 3). Here, as in LU, two wavefronts per recursion are called for. The alternate wavefront completes the second step of the backwards data flow (47–54). The main wavefront computes C_p and S_p from $A_{p,1}$ and $A_{p+1,1}$ in the first column processors $(p, 1)$ (lines 20–28), and uses it to modify the interior elements in the INT processors. With each new recursion, the effective size of the new matrix reduces by one. Hence, the main wavefront has to flow the new A^1 to the left and reduce the effective sizes of the processor array (lines 39, 45, 53). We are looking into the possibility of using cordic and noncordic ALU's to compute the rotation coefficients C_p and S_p.

B. Signal Processing Algorithms

Example 3—Recursive Filter: A digital infinite impulse response (IIR) filter can be implemented recursively, as illustrated in the following difference equation:

$$y(n) = \sum_{k=0}^{N} a(k) * x(n-k) - \sum_{k=1}^{N} b(k) * y(n-k).$$

Here, recursive implementation means that past values of the output are used along with the input sequence to determine the current output.

Filter parameters $\{a(k)\}$ and $\{b(k)\}$ are sequences of length $N+1$ and N, respectively. The input sequence is $\{x(n)\}$ and the output sequence is $\{y(n)\}$. Algorithm analysis indicates that the computation can be done in parallel in a linear array of N processors, with the basic kth recursive operation defined by

$$C(k, i-1) = [a(i) * x(k)] - [b(i) * C(k-1, i-1)] + C(k-1, i)$$

where $C(n, -1) = y(n)$ gives the output sequence.

This parallel algorithm also possesses recursivity and locality and demonstrates the wavefront phenomenon. The MDFL program for this algorithm is listed below.

Program 5. Recursive filtering.

```
Array Size:     1 x N

Computation:    y(n) =    a(k)x(n-k) +    b(k)y(n-k)

Initial:          Coefficients a(k) are stored in registers A and b(k) in
                registers B.  {x(k)} is input from the left Memory Module
                to registers D.

Final:            The resulting  {y(n)} will be output from register C to
                the left.


1:      BEGIN
        SET COUNT 1;
        REPEAT
          WHILE WAVEFRONT IN ARRAY DO
5:            BEGIN
                FLOW C, RIGHT;
                CASE KIND =
                (1,*):BEGIN
                        FETCH A, LEFT;
10:                     FLOW  A, LEFT;
                      END;
                ENDCASE;
              END;
          DECREMENT COUNT;
15:     UNTIL TERMINATED;

        SET COUNT < >;
        REPEAT
          WHILE WAVEFRONT IN ARRAY DO
              BEGIN
20:             FETCH D, LEFT;
                MULT D, A, R1;
                FETCH C, RIGHT;
                MULT C, B, R2;
                SUB R1, R2, R1;
25:             ADD C, R1, C;
                FLOW D, RIGHT;
                FLOW C, LEFT;
              END;
          DECREMENT COUNT;
30:     UNTIL TERMINATED;
        ENDPROGRAM.
```

Here, line 22 dictates the need for an initialization phase. Also note the equal number of FLOW and FETCH instructions as required by the syntax rules.

Example 4—Toeplitz System Solver: We would like to mention another key matrix operation in modern signal processing, the solution of Toeplitz systems [10], [35]:

$$Tx = y$$

where T is Toeplitz, i.e., $T(i, j) = T(i+k, j+k)$ for all k.

In applications, T is often a covariance matrix of a stochastic signal. It is possible to derive a pipelined wavefront-oriented algorithm to solve the system in $O(n)$ time using a bilinear array of $2N$ processors [35], [36].

C. Other Algorithms

Example 5—Partial Difference Equations: Second-order partial difference equations in two variables, expressed in terms of central differences, can be solved by the relaxation algorithm [37], which involves repeated iterations on an $N \times N$ matrix of elements. The Laplace PDE

$$\nabla^2 u = 0$$

with boundary conditions can be solved by the iterative algorithm defined by the basic computation

$$u(x, y) = \tfrac{1}{4}\{u(x-1, y) + u(x+1, y) + u(x, y-1) + u(x, y+1)\}.$$

In every PE in the WAP, this turns into the iterative operation

$$u(\mathrm{PE}) = \tfrac{1}{4}\{u(\mathrm{left}) + u(\mathrm{right}) + u(\mathrm{up}) + u(\mathrm{down})\}.$$

Boundary conditions are preloaded in the WAP elements prior to the recursive iterations described in the program below.

Program 6. PDE relaxation algorithm.

```
Array Size:     N x N

Computation:    Solution of      2
                                 ∇   A = 0.
                                  x,y
                 th
                k    wavefront:
                  (k)              (k)          (k)
                 a    (i,j) = (1/4){a   (i-1,j) + a   (i,j-1) +
                                    (k-1)          (k-1)
                               + a     (i+1,j) + a     (i,j+1)}

Initial:          Boundary conditions stored in the B, C, D and
                E registers of the boundary processors.
```

```
Final:          Solution will be stored in the A register.


1:      BEGIN
        ! Initialization phase:  one wavefront;
        SET COUNT 1;
        REPEAT
           WHILE WAVEFRONT IN ARRAY DO
5:            BEGIN
              CASE KIND =
                 (1,*):     FLOW A, LEFT;
                 (*,1):     FLOW A, UP;
                 INT  :     BEGIN
10:                           FLOW A, LEFT;
                              FLOW A, UP;
                           END;
              ENDCASE;
           END;
15:        DECREMENT COUNT;
        UNTIL TERMINATED;

        SET COUNT <P>;  !Represents the number of iterations.;
        REPEAT
           WHILE WAVEFRONT IN ARRAY DO
20:           BEGIN
                 CASE KIND =
                 (*,1):     FETCH E, UP;
                 (1,*):     FETCH B, LEFT;
                 INT  :     BEGIN
25:                           FETCH E, UP;
                              FETCH B, LEFT;
                           END;
                 END CASE;

                 ADD B, E, A
30:              FETCH D, RIGHT;
                 FETCH C, DOWN;
                 ADD D, A, A;
                 ADD C, A, A;
                 DIV A, 4, A;

35:              FLOW A, RIGHT;
                 FLOW A, DOWN;

                 CASE KIND =
                 (*,1):     FLOW A, UP;
                 (1,*):     FLOW A, LEFT;
40:              INT  :     BEGIN
                              FLOW A, UP;
                              FLOW A, LEFT;
                           END;
                 ENDCASE;

45:           END;  !Of WHILE block;
              DECREMENT COUNT;
        UNTIL TERMINATED;
        ENDPROGRAM.
```

Each recursion consists of only one wavefront. Every wavefront computes, at each PE, the average of the contents of the neighboring four PE's. Recursions are continued until the iterations converge. Because of the backward flow of data (lines 30, 31), an initialization phase is needed (lines 2–16).

In summary, the WAP is applicable to a large class of algorithms, and a few MDFL programs were illustrated in this section as prototypical examples. Further examples may be obtained from earlier publications [7]–[10]. In all these applications, processor utilization, and subsequently the gross efficiency factor,[3] is fairly high, except in cases where the array size is dynamically reduced. But in such applications as well, it may be possible to utilize the free PE's for some other tasks. For instance, in LU decomposition and QR factorization algorithms, the free processors may be used for back substitution.

VI. Comparison to Other Array Processors

Different kinds of solutions have been proposed to overcome the inherent problems in the design of large scale multiprocessor arrays. Popular solutions have included restricting the application to a dedicated system, complete global synchronization of all processors, restriction to localized communication, and restriction to a special class of applications. The systolic array [1], [15], [16], data-flow computers [17]–[23], and the Illiac IV [38]–[40] are representative examples of different attempts at solving the parallel processing problem. It is naturally interesting and useful to compare the WAP to these multiprocessor arrays. However, a totally objective comparison is almost impossible because it depends on the complicated tradeoff between numerous criterion such as programmability, software simplicity, modularity, synchronization, interprocessor communication, etc.

A. Systolic Array

One solution amenable to VLSI implementation is the systolic array processor [1], [15], [16]. It restricts itself to a special class of algorithms and takes advantage of their regular localized data flow. The systolic array features the important properties of modularity, regularity, local interconnection, and highly pipelined and highly synchronized multiprocessing. However, the systolic array requires global synchronization, which is a major barrier toward the VLSI implementation. Furthermore, the systolic array is strictly dedicated, and its lack of programmability undoubtedly reduces much of its cost-effectiveness for mass production.

The concept of the WAP focuses on getting around the global synchronization requirements. It utilizes the powerful notion of the computational wavefront to deal with issues of programmability, extendability, etc. (cf. Sections III and IV). For a detailed comparison between the synchronous and asynchronous array processors, the reader is referred to many recent studies [30], [41], [42]. Whereas the asynchronous model incurs a fixed time delay overhead due to the handshaking processes, the synchronous time delay is primarily due to the clock skew which changes dramatically with the size of the array N. A detailed analysis [30] indicates that in an H-tree clock distribution, the clock skew grows with the array size at the significant rate of $O(N^3)$. An immediate conclusion from this analysis is that, while for small N a globally synchronized array may be easier to implement, for large values of N, an asynchronous system may become more favorable. However, there are (and we are looking into) some other important factors such as programmability, hardware complexity, extendability, testability, etc., and the final choice between the two array processors hinges upon the specific requirements of the intended application.

B. Data-Flow Multiprocessor

A data-flow multiprocessor [20], [26]–[29] is a concurrent multiprocessor which runs programs expressed in data-flow notation. The execution of its instructions is "data driven," i.e., it depends only on the availability of operands required by the instructions. Thus, unrelated instructions can be executed concurrently without interference [20]. Moreover, all interactions among the modules in a data-flow network are asynchronous. The principal advantages of data-flow multiprocessors over conventional multiprocessors are a simple representation of concurrent activity, relative independence of in-

[3] The gross efficiency factor is defined as the ratio of computation time on a uniprocessor to the product of array size and computation time on an array processor.

dividual PE's, greater use of pipelining, and reduced use of centralized control and global memory.

The WAP shares all these advantages. The basic principle of wavefront architecture is that each PE waits for the arrival of a primary wavefront, then executes the required computation, and acts as a secondary source of new wavefronts. This is equivalent to the key concept in data-flow machines: the arrival of data fires each PE, which subsequently sends relevant data to the next PE. Hence, the WAP can be regarded as an array of homogeneous data-flow processing elements. The "Wait" for data feature, provided by handshaking, allows for the globally asynchronous operation of processors, i.e., there is no need for global synchronization.

For both data-flow and WAP concepts, scheduling and synchronization are built in at the hardware level, and are distributed, not centralized. This allows for efficient fine-grain parallelism. Being language-based architectures, they support the mapping of application algorithms directly onto the multiprocessor in a way that achieves high performance. Due to its general-purpose nature, a data-flow machine often involves a great amount of data and resource management, and needs a powerful supervising system. By the same token, the construction of a working data-flow machine has not demonstrated convincing effectiveness. It is therefore extremely advantageous to focus on a special (yet major) class of applicational algorithms, and achieve a much greater efficiency. This, in fact, leads to the wavefront notion and the concept of the WAP.

The inherent advantages of restricting applications to a special class are the WAP's programming ease, functional modularity, and consequent reduction in design costs. However, again, a tradeoff exists between the general-purpose nature of data-flow multiprocessors versus the simplicity of the WAP. A research effort is being conducted into how to optimally compromise between specialized applications (of the WAP) and the general-purpose approach (of conventional data-flow multiprocessors).

Both the WAP and data-flow multiprocessors share the key common feature of being globally asynchronous. The adoption of global asynchrony has been a controversial issue, and has already attracted a great deal of attention. Nevertheless, it is bound to have a major impact on the design of future VLSI systems.

C. Illiac IV

The Illiac IV system (a SIMD computer implemented as an array of arithmetic processors) has been used as a typical example for the study of array computers [38]-[40]. Examples of other large scale SIMD array computers implemented in LSI technology are NASA's Massively Parallel Processor and ICL's Distributed Array Processor. The instructions of these systems are stored in a global main memory, together with data. The central control unit directs the operation of all PE's, communicating with them via a global broadcasting network. As in the systolic array, the processors are all synchronized. Unfortunately, many of the Illiac IV features appear to be incompatible with VLSI constraints, and major modifications will be required to make it amenable to VLSI implementation.

While centralized control and global synchronization are easily implementable via today's LSI technology, asynchronous distributed control and localized communication (for both data and control) are more attractive in VLSI. It is a relatively simple task to add a few delay buffers to the control signals to compensate for clock skew in large scale LSI systems. However, due to large *random* clock skew and expensive communication in VLSI, the adoption of the SIMD scheme in future multiprocessors has to be carefully reviewed.

In contrast to the Illiac IV, the WAP employs local instruction storage, data-flow-based control, local communication, and needs no global synchronization, all of which make the WAP very appealing for VLSI. Owing to the drastically simpler structure of the WAP, its range of applications is naturally much more limited than that of the Illiac IV. Nevertheless, the WAP is applicable to a large number of algorithms with inherent locality and recursivity. In this special, but rather broad class of applications, the WAP concept is a more promising candidate for VLSI systems.

In summary, the wavefront array processor possesses the advantages of the systolic array processor, such as extensive pipelining and multiprocessing, regularity, and modularity. More significantly, it also shares the asynchronous waiting capability of data-flow machines and can, therefore, accommodate the critical problem of timing uncertainties, which is bound to be a cause for concern in future VLSI systems. In other words, the WAP concept is very suitable for efficient and reliable design of large scale computing systems, and it is especially appealing for future VLSI implementation.

VII. Conclusions

To aid in the design of the WAP and its future development and experimentation, three software packages have been developed. The first package is an *Interpreter* which simulates the execution of a global wavefront-oriented MDFL program. The Interpreter executes the MDFL programs by "freezing" the time count to sequentially perform all the supposedly parallel operations. The output of the Interpreter includes sampled snapshots of the data processing activities. The user can, therefore, very easily validate the MDFL program at the algorithm level. The second software package is a *Compiler* which converts global MDFL programs to its local versions. The last software package is a *Simulator* which enables simulation of the WAP hardware down to the register level. This enables the user to test various architectural approaches in the design stage without invoking the enormous expenses involved in the actual hardware construction.

A current short-term project to build a microprocessor-based wavefront array processor [10] is in progress. It will consist of an 8×8 array of processors. The processing element under consideration is a hybrid between a control-oriented microprocessor and a sophisticated arithmetic processing unit. With the experience gained from this experimental task, a longer term project on VLSI implentation of the WAP is being undertaken.

In summary, the development of the WAP demonstrates an effective top-down design procedure with the algorithm analysis dictating the language structure which, in turn, determines the architecture of the computing network. The resulting wavefront language (MDFL) strongly supports the notions of locality and the idea of large scale concurrency which is lacking in most current languages [25]. At the same time, the wavefront architecture supports the mapping of concurrent algorithms onto the computing network in a way that achieves high performance. The array processor itself satisfies the constraints of VLSI, and is applicable to a large class of VLSI signal processing algorithms. Interestingly, the WAP is, in a sense, an optimal tradeoff between the globally synchronous and dedicated systolic array and the asynchronous and general purpose data-flow multiprocessor. Finally, the development of the language, architecture, and system applications of the WAP has reconfirmed the power of the wavefront notion and has demonstrated an efficient approach to the design of special purpose VLSI array processors.

ACKNOWLEDGMENT

The authors wish to acknowledge the contributions of Y. H. Hu of the University of Southern California, Los Angeles, to the WAP software development and subsequent discussions. They also wish to thank H. J. Whitehouse and J. M. Speiser of NOSC, San Diego, CA and N. Powell of General Electric, Syracuse, NY, for their very valuable advice throughout this research period.

REFERENCES

[1] C. Mead and L. Conway, *Introduction of VLSI Systems*. Reading, MA: Addison-Wesley, 1980.

[2] I. E. Sutherland and C. Mead, "Microelectronics and computer science," *Sci. Amer.*, vol. 237, pp. 210–228, Sept. 1977.

[3] S. Y. Kung, "VLSI array processor for signal processing," presented at the Conf. Advanced Res. in Integrated Circuits, M.I.T., Cambridge, Jan. 28–30, 1980.

[4] J. M. Speiser and H. J. Whitehouse, "Architectures for real time matrix operations," in *Proc. GOMAC*, Nov. 1980.

[5] Y. Muraoka, "Parallelism exposure and exploitation in programs," Ph.D. dissertation, Dep. Comput. Sci., Univ. Illinois, Urbana-Champaign, Feb. 1971.

[6] D. J. Kuck, Y. Muraoka, and S. C. Chen, "On the number of operations executable in Fortran-like programs and their resulting speed-up," *IEEE Trans. Comput.*, vol. C-21, pp. 1293–1310, Dec. 1972.

[7] S. Y. Kung, "Matrix data flow language for matrix operation dedicated array processors," in *Proc. ECCTD 1981*, The Hague, The Netherlands, Aug. 1981, pp. 393–398.

[8] S. Y. Kung and D. V. Bhaskar Rao, "Highly parallel architectures for solving linear equations," in *Proc. ICASSP 1981*, Atlanta, GA, Mar. 1981, pp. 39–42.

[9] S. Y. Kung, K. S. Arun, D. V. Bhaskar Rao, and Y. H. Hu, "A matrix data flow language/architecture for parallel matrix operations based on computational wavefront concept," in *Proc. CMU Conf. VLSI Syst. Computations*. Comput. Sci. Press, Oct. 1981, pp. 235–244.

[10] S. Y. Kung, R. J. Gal-Ezer, and K. S. Arun, "Wavefront array processor: Architecture, language and applications," presented at the M.I.T. Conf. Advanced Res. in VLSI, M.I.T., Cambridge, Jan. 1982.

[11] U. Weiser and A. Davis, "A wavefront notation tool for VLSI array design," in *Proc. CMU Conf. VLSI Syst. Computations*. Comput. Sci. Press, Oct. 1981, pp. 226–234.

[12] "Highly parallel modern signal processing," S. Y. Kung, Ed., Univ. Southern California, Los Angeles, USC-SRO Project Rep. 1, part XI, Oct. 1981.

[13] A. M. Finn, F. T. Lux, and C. Pottle, "Systolic array computation of the singular value decomposition," presented at the SPIE Conf., Arlington, VA, May 1982.

[14] D. Halliday and R. Resnick, *Physics for Students of Science and Engineering*. New York: Wiley, 1960.

[15] H. T. Kung, "Let's design algorithms for VLSI systems," in *Proc. CalTech Conf. VLSI*, Jan. 1979, pp. 65–90.

[16] ——, "The structure of parallel algorithms," in *Advances in Computers*, vol. 19. New York: Academic, 1980, pp. 65–111.

[17] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data flow processor," in *Proc. 2nd Annu. IEEE Symp. Comput. Architecture*, Jan. 1974, p. 126.

[18] J. E. Rumbaugh, "A data flow multiprocessor," *IEEE Trans. Comput.*, vol. C-26, pp. 138–146, Feb. 1977.

[19] J. B. Dennis, "The varieties of data flow computers," in *Proc. 1st Int. Conf. Distributed Computing Syst.*, Oct. 1979, pp. 430–439.

[20] ——, "Data flow supercomputers," *IEEE Computer*, pp. 48–56, Nov. 1980.

[21] K. P. Gostelow and R. E. Thomas, "Performance of a simulated data flow computer," *IEEE Trans. Comput.*, vol. C-29, pp. 905–919, Oct. 1980.

[22] I. Watson and J. Gurd, "A prototype data flow computer with token labelling," in *Proc. AFIPS Conf.*, vol. 48. New York: NCC, June 1979, pp. 623–628.

[23] A. Davis, "A data flow evaluation system based on the concept of recursive locality," in *Proc. AFIPS Conf.*, vol. 48. New York: NCC, June 1979, pp. 1079–1086.

[24] S. Y. Kung *et al.*, "MDFL user manual," Univ. Southern California, Los Angeles, USC-SRO Rep. 2.

[25] C. Mead, "The impact of VLSI on computer science education," in *Proc. 12th Asilomar Conf. Circuits, Syst., Comput.*, Pacific Grove, CA, Nov. 1978, pp. 350–351.

[26] W. B. Ackerman and J. B. Dennis, "VAL: A value-oriented algorithmic language, Preliminary reference manual," Lab. for Comput. Sci., M.I.T. Tech. Rep. TR-218, June 1979.

[27] Arvind, K. P. Gostelow, and W. Plouffe, "An asynchronous programming language and computing machine," Dep. Inform. Comput. Sci., Univ. California, Irvine, Tech. Rep. 114a, Dec. 1978.

[28] W. B. Ackerman, "Data flow languages," in *Proc. AFIPS Conf.*, vol. 48. New York: NCC, June 1979, pp. 1087–1095.

[29] J. Backus, "Can programming be liberated from the Von Neumann style—A functional style and its algebra of programs," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 613–641, Aug. 1978.

[30] S. Y. Kung and R. J. Galezer, "Synchronous vs asynchronous computation in VLSI array processors," presented at the SPIE Conf., Arlington, VA, May 1982.

[31] G. W. Stewart, *Introduction to Matrix Computations*. New York: Academic, 1973.

[32] W. Miller and C. Wrathall, *Software for Roundoff Analysis of Matrix Algorithms*. New York: Academic, 1980.

[33] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.

[34] D. J. Burr, B. Ackland, and N. Weste, "A high speed computer for dynamic time warping," in *Proc. ICASSP*, Atlanta, GA, Apr. 1981, pp. 471–474.

[35] S. Y. Kung and Y. H. Hu, "Fast and parallel algorithms for solving Toeplitz systems," presented at the Int. Symp. Mini and Microcomputers in Contr. and Measurement, San Francisco, CA, May 1981.

[36] ——, "A highly concurrent algorithm and pipelined architecture for solving Toeplitz systems," submitted for publication.

[37] S. A. Hovanessian and L. A. Pipes, *Digital Computer Methods in Engineering*. New York: McGraw-Hill, 1969.

[38] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Stonick, and R. A. Stokes, "The Illiac IV computer," *IEEE Trans. Comput.*, vol. C-17, pp. 746–757, Aug. 1968.

[39] D. J. Kuck, "Illiac IV software and application programming," *IEEE Trans. Comput.*, vol. C-17, pp. 758–770, Aug. 1968.

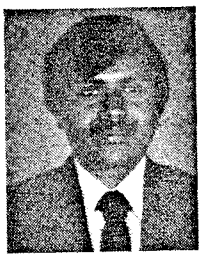
[40] W. J. BouKnight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. M. Sameh, and D. L. Slotnick, "The Illiac IV system," *Proc. IEEE*, vol. 60, pp. 369–388, Apr. 1972.

[41] M. Franklin and D. Wann, "Asynchronous and clocked control structures for VLSI based interconnection networks," presented at the 9th Annu. Symp. Comput. Architecture, Austin, TX, Apr. 1982.

[42] A. L. Fisher and H. T. Kung, "Synchronizing large systolic arrays," presented at the SPIE Conf., Arlington, VA, May 1982.

Sun-Yuan Kung (M'77) received the B.S. degree in electrical engineering from the National Taiwan University, Taipei, Taiwan, in 1971, the M.S. degree in electrical engineering from the University of Rochester, Rochester, NY, in 1974, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1977.

In 1973 he held a fellowship at the University of Rochester. In 1974 he joined the Amdahl Corporation, Sunnyvale, CA, as an Associate Engineer. From 1974 to 1977 he was a Research Assistant at the Information Systems Laboratories, Stanford University. Since July 1977 he has been on the faculty of Electrical Engineering–Systems, University of Southern California, Los Angeles, where he is presently an Associate Professor. Since 1979 he has also been a Research Consultant with Stanford University and General Electric, Syracuse, NY. His research interests are in the areas of multivariable and two-dimensional system theory, digital signal processing, modern spectrum analysis, and VLSI parallel processing.



K. S. Arun (S'81) was born in Bangalore, India, on July 23, 1959. He received the B.Tech. degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, in 1980, and the M.S.E.E. degree in computer engineering from the University of Southern California, Los Angeles, in 1981. He is currently working towards the Ph.D. degree in electrical engineering at USC.

Since 1980 he has been a Teaching Assistant and a Research Assistant in the Department of Electrical Engineering–Systems, USC. His current research interests are spectral estimation and adaptive signal processing, with emphasis on parallel processing.

Mr. Arun is a member of Eta Kappa Nu.

Ron J. Gal-Ezer (S'79) was born in Israel in 1947. He received the B.Sc. and M.Sc. degrees in 1971 and 1977, respectively, from the Technion—Israel Institute of Technology, Haifa.

He worked for the Israeli Ministry of Defence in digital system design (1971–1978) and as an Electronic Systems Engineer involved in the Boeing 767 program (1979–1982). During 1978–1979 he was an Assistant Professor in the Department of Electrical and Computer Engineering, California State Polytechnic University, Pomona. He is a Research Assistant and is currently completing the requirements for the Ph.D. degree in the Department of Electrical Engineering, University of Southern California, Los Angeles.

Mr. Gal-Ezer is a member of Eta Kappa Nu.

D. V. Bhaskar Rao (S'80) was born in Cuddapah, India, on November 8, 1957. He received the B.Tech. degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, in 1979, and the M.S. degree in computer engineering from the University of Southern California, Los Angeles. At present he is working towards the completion of the Ph.D. degree in signal processing.

Since 1979 he has been a Teaching and Research Assistant in the Department of Electrical Engineering–systems, University of Southern California. His research interests include VLSI, spectral estimation, and adaptive signal processing.

Mr. Bhaskar Rao is a member of Eta Kappa Nu.