A Model of SIMD Machines and a Comparison of Various Interconnection Networks

HOWARD JAY SIEGEL, MEMBER, IEEE

Abstract—**A formal mathematical model of single instruction stream–multiple data stream (SIMD) machines is defined. It is used as a basis for analyzing various types of interconnection networks that have been discussed in the literature. The networks are evaluated in terms of the lower and upper bounds on the time required for each of the networks discussed to simulate the other networks. SIMD machine algorithms are presented as proofs of the upper time bounds on these simulation tasks. These simulations are used to demonstrate techniques for proving the correctness of SIMD machine algorithms, i.e., analyzing the simultaneous flow of N data items among N processors. Processor address masks, a concise notation for activating and deactivating processors, are used in the algorithms. The methods used to prove the lower bounds and to construct (and prove correct) simulation algorithms to show the upper bounds can be generalized and applied to the analysis of other networks.**

Index Terms—**Algorithm correctness, array processors, computer architecture, Illiac IV, interconnection networks, n-cube array, parallel processing, perfect shuffle, permutation networks, SIMD machines, STARAN.**

I. Introduction

A single instruction stream–multiple data stream (SIMD) machine is a computer system consisting of a control unit, N processing elements, and an interconnection network. The control unit broadcasts instructions to the processing elements, and all active processing elements execute the same instruction at the same time. Each active processing element executes the instruction on data in its own memory. The interconnection network provides communication among the processing elements. This type of machine structure is designed to exploit the parallelism of tasks such as vector and matrix operations.

A formal mathematical model of SIMD machines is defined. The model was designed to provide a basis for the evaluation of various SIMD computer system components. It is used to define and analyze five interconnection networks. These networks include the types used in the Illiac IV [1], [6], Omen [14], DAP [12], RAP [7], SIMDA [32], and STARAN[1] [2], [3] machines. They also include the types discussed by Feng [10], Lang [15], Lang and Stone [16], Lawrie [17], Orcutt [18], Pease [19], Siegel [21], [23]–[25], [27], Smith and Siegel [28], [29], and Stone [30].

Designing an interconnection network has been recognized as an important problem by computer architects (e.g., [11]). The five networks evaluated here were chosen because they form the basis of most of the SIMD machine interconnection networks that have been proposed and shown to be useful in the literature.

The ability of an interconnection network to simulate the actions of an interconnection not in that network is important to SIMD machine designers, who must choose a set of interconnections to implement in the hardware of the system. The number of interconnections which may be included is constrained by such factors as cost and hardware complexity. Therefore, architects must consider the ability of the interconnection network that is chosen to simulate other interconnections which may be necessary for the machine to perform various computational tasks.

To study this simulation capability, the times required for each of the five networks to simulate the other networks are examined. Algorithms are presented as proofs of upper time bounds on these simulation tasks. Techniques for proving the correctness of SIMD machine algorithms are demonstrated. The lower time bounds are an extension of the work presented in [21]. In the analyses very few assumptions are made about the exact architecture of SIMD machines, so the results are applicable to a variety of actual machines. Therefore, an SIMD architect can use the information on the time bounds for the various simulations, and even the simulation algorithms themselves, if one of these networks is implemented.

The various methods used to construct SIMD machine algorithms and prove their correctness can be generalized and applied to other interconnection networks. Thus, the significance of this paper lies not only in the specific results, but also in the methods used to obtain these results.

II. SIMD Machines

One way to view the physical structure of an *SIMD* machine [13] is as a set of N *processing elements* (*PE*'s) (where each PE consists of a processor with its own memory), interconnected by a network, and fed instructions by a control unit. The network connects each PE to some subset of the other PE's. An interprocessor transfer instruction causes data to be moved from each PE to one of the PE's to which the element is connected by the network. To move data between two PE's that are not directly connected, the

Manuscript received August 20, 1977; revised August 18, 1978 and August 6, 1979. This work was supported in part by the National Science Foundation under Grant DCR74-21939, and by the Air Force Office of Scientific Research, Air Force Systems Commands, U.S. Air Force, under Grant AFOSR-78-3581. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

The author is with the School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.

[1] Note the name STARAN, which is used throughout this paper, is a registered trademark of Goodyear Aerospace.

0018-9340/79/1200-0907$00.75

data must be passed through intermediate PE's by executing a programmed sequence of data transfers. An alternative structure is to position the network between the processors and the memories. The PE-to-PE configuration is used here, although the results presented are also valid for the processor-to-memory type of structure.

The model of an SIMD machine presented here consists of four parts: processing elements, interconnection functions, machine instructions, and masking schemes. It is a mathematical model that provides a common formal basis for evaluating and comparing the various components of different SIMD machines. The model was designed to reflect the features of actual SIMD computer systems.

Each *PE* is a processor together with its own memory, a set of fast access registers, and a data transfer register (*DTR*). It is assumed that there are at least three fast access registers, which will be referred to as *A*, *B*, and *C*. The DTR of each PE is connected to the DTR's of the other PE's via the interconnection network. When data transfers among PE's occur it is the DTR contents of each PE that are transferred. There are N PE's, each assigned an address from 0 to $N-1$, where $N = 2^m$; i.e., $\log_2 N = m$. Each PE has a register *ADDR* which contains the address of that PE. A PE is shown in Fig. 1.

An alternative processor organization would have two DTR's, one for input and one for output. In this paper the single DTR organization is used in the various SIMD machine algorithms. This is done for two reasons: 1) the algorithms can easily be made to operate for the two DTR organization by adding an instruction to move the data from the input DTR to the output DTR immediately after each interprocessor data transfer, and 2) the single DTR implementation manifests problems the two DTR organization does not, as will be discussed later in this section.

Each PE is either in the active or the inactive mode. If a PE is *active* it executes the instructions broadcast to it by the control unit. If a PE is *inactive* it will not execute the instructions broadcast to it. The masking schemes are used to specify which PE's will be active.

An *interconnection network* is a set of interconnection functions. Each *interconnection function* is a bijection on the set of PE addresses. When an interconnection function f is applied, PE_i copies the contents of its DTR into the DTR of $PE_{f(i)}$. This occurs for all i simultaneously, for $0 \le i < N$ and PE_i active. An inactive PE may receive data from another PE if an interconnection function is executed, but it cannot send data. To pass data from one PE to another PE a programmed sequence of interconnection functions must be executed. This sequence of functions moves the data from one PE's DTR to the other's by a single transfer or by passing the data through intermediate PE's.

Five interconnection networks are defined and compared, using $p_{m-1} \cdots p_1 p_0$ to denote the binary representation of an arbitrary PE address and $\bar{p}_i$ to denote the complement of p_i.

The *perfect shuffle* (*PS*) network consists of a shuffle function and an exchange function. The *shuffle* is defined by

$$\text{shuf}\,(p_{m-1}p_{m-2}\cdots p_1p_0) = p_{m-2}p_{m-3}\cdots p_1p_0p_{m-1}$$

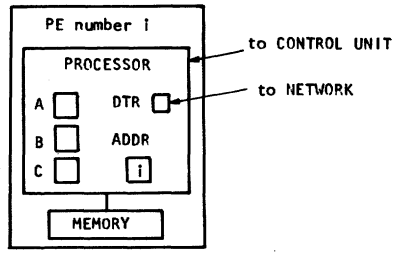

Fig. 1. Processing element.

and the *exchange* is defined by

$$\text{exch}\,(p_{m-1}p_{m-2}\cdots p_1p_0) = p_{m-1}p_{m-2}\cdots p_1\bar{p}_0.$$

In Fig. 2 these interconnections are shown. This network has been shown to be useful by Lang [15], Lang and Stone [16], and Stone [30]. It is the basis of Lawrie's Omega network [17] and is included in the networks of the Omen [14] and RAP [7] systems.

The *Illiac* network consists of four functions defined as follows:

$$\text{Il}_{+1}(x) = x + 1 \bmod N$$
$$\text{Il}_{-1}(x) = x - 1 \bmod N$$
$$\text{Il}_{+n}(x) = x + n \bmod N$$
$$\text{Il}_{-n}(x) = x - n \bmod N$$

where n is the square root of N, which is assumed to be a perfect square. If the PE's are considered as a $n \times n$ array, then each PE is connected to its north, south, east, and west neighbors, as shown in Fig. 3. This is the network implemented in the Illiac IV [6] and DAP [12] systems. Its ability to perform various tasks is discussed in [1], [6], [12], [18].

The *cube* network consists of m functions defined by

$$\text{cube}_i\,(p_{m-1}\cdots p_{i+1}p_ip_{i-1}\cdots p_0)$$
$$= p_{m-1}\cdots p_{i+1}\bar{p}_ip_{i-1}\cdots p_0$$

for $0 \le i < m$. When the PE addresses are considered as the corners of an m-dimensional cube this network connects each PE to its m neighbors, as shown in Fig. 4. Pease's binary n-cube [19], the network used in STARAN [2], [3] and the network proposed for Phoenix [9] are each wired series of cube functions. In [2], [4], [19] the applicability of this network to practical problems is discussed.

The *plus–minus* 2^i (*PM2I*) network consists of $2m$ functions defined by

$$\text{PM}_{+i}(j) = j + 2^i \bmod N$$
$$\text{PM}_{-i}(j) = j - 2^i \bmod N$$

for $0 \le i < m$. Note that $\text{PM}_{+(m-1)} = \text{PM}_{-(m-1)}$. Fig. 5 shows the PM_{+i} interconnections for $N = 8$. Diagramatically, PM_{-i} is the same as PM_{+i} except the direction is reversed. A network similar to the PM2I is included in the network of the Omen computer [14]. The concept underlying the SIMDA array machine's interconnection network is similar to that of the PM2I [32]. Feng describes a network that is basically a wired series of PM2I functions in [10], where various practical permutations this network can perform are discussed.

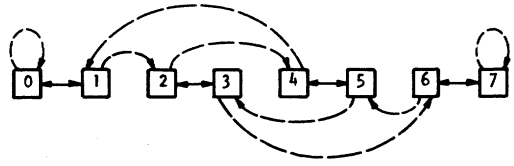

Fig. 2. PS network for $N = 8$; solid line is exchange and dashed line is shuffle.

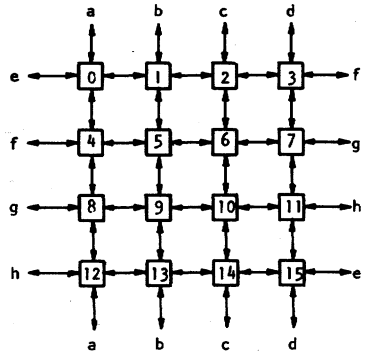

Fig. 3. Illiac network for $N = 16$; vertical lines are Π_{+n} and Π_{-n} connections, horizontals are Π_{+1} and Π_{-1}.

The *wrap-around plus-minus* 2^i (*WPM2I*) network consists of $2m$ functions defined by

$$\mathrm{WPM}_{+i}(p_{m-1} \cdots p_i \cdots p_0) = q_{m-1} \cdots q_i \cdots q_0,$$

where

$$q_{i-1} \cdots q_0 q_{m-1} \cdots q_{i+1} q_i = (p_{i-1} \cdots p_0 p_{m-1} \cdots p_{i+1} p_i) + 1 \bmod N$$

and

$$\mathrm{WPM}_{-i}(p_{m-1} \cdots p_i \cdots p_0) = q_{m-1} \cdots q_i \cdots q_0,$$

where

$$q_{i-1} \cdots q_0 q_{m-1} \cdots q_{i+1} q_i = (p_{i-1} \cdots p_0 p_{m-1} \cdots p_{i+1} p_i) - 1 \bmod N,$$

for $0 \le i < m$. The function WPM_{+i} adds 2^i to the PE addresses, with any "carry" wrapping around up to and including the $i-1$st bit position. A carry cannot affect the ith bit position. The function WPM_{-i} behaves analogously. Fig. 6 shows WPM_{+i} interconnections for $N = 8$. Diagramatically, WPM_{-i} is the same as WPM_{+i} except the direction is reversed. WPM2I is similar to PM2I, except any "carry" or "borrow" will "wrap-around" through the $i-1$st bit position. For example, if $N = 8$, then $\mathrm{WPM}_{+2}(101) = 010$ whereas $\mathrm{PM}_{+2}(101) = 001$. When the networks are treated as sets of permutations on the integers from 0 to $N-1$, the WPM2I network has the ability to simulate any other interconnection function, i.e., in terms of group theory, WPM2I can generate the entire group of permutations on N elements. Of the five networks presented here, only the WPM2I network has this ability [21].

Cross bar networks, Benes networks [5], and Swanson's networks [31] are not considered here for reasons discussed in [21]. Note that the PEPE machine has no interconnection network [8].

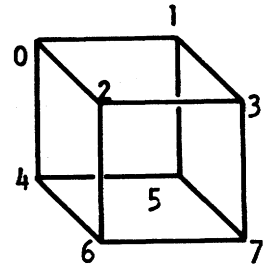

Fig. 4. Cube network for $N = 8$; horizontal lines are cube_0 connections, diagonals are cube_1, and verticals are cube_2.

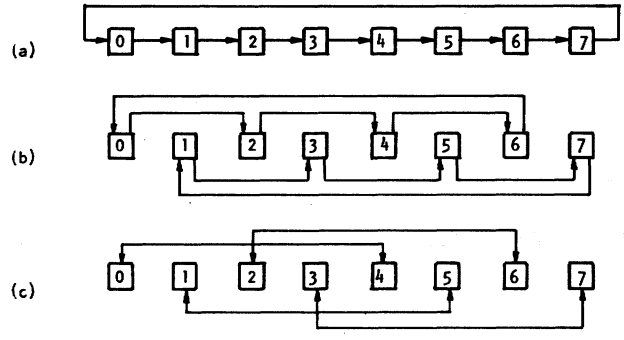

Fig. 5. PM2I network for $N = 8$; (a) PM_{+0}, (b) PM_{+1}, and (c) PM_{+2}.

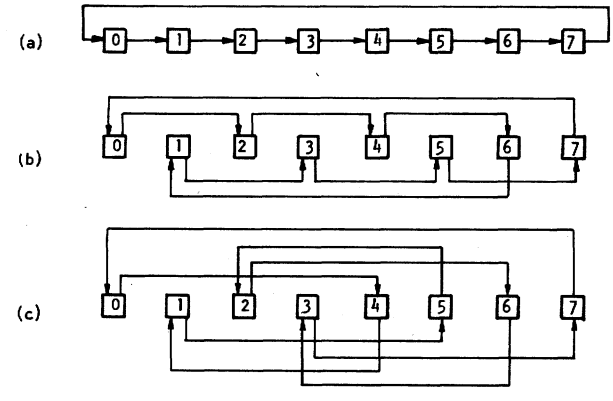

Fig. 6. WPM2I network for $N = 8$; (a) WPM_{+0}, (b) WPM_{+1}, and (c) WPM_{+2}.

The *machine instructions* part of the model consists of those operations that each processor can perform on data in its individual memory or registers. There is a separate *control unit* (*CU*) computer which stores the programs and broadcasts the instructions to the PE's. The CU also executes control flow instructions, such as "*for* $i = 1$ *until* Q *do* $\cdots$" All active PE's execute the instruction broadcast by the CU at the same time, but on possibly different data. It is assumed the set of machine instructions includes the capability to move data among the registers. The notation $X \leftarrow Y$ means the contents of register Y are copied into register X. The notation $X \leftrightarrow Y$ is an abbreviation for two registers exchanging their contents by using a third register for temporary storage.

Interconnection functions are broadcast by the CU in the same way as the machine instructions. Thus, a stream of

instructions may contain both machine instructions and interprocessor data transfer instructions.

The *masking schemes* component of the model consists of methods for controlling the active/inactive status of the PE's. Masking schemes provide the system user with devices to enable some PE's and disable others. For some masking schemes, the set of PE's that can be activated is any member of the power set of integers 0, 1, $\cdots$, $N-1$. For other schemes, the set of PE's that can be activated is more restricted. In this paper, two masking schemes are used: PE address masks and data conditional masks.

The *PE address masking* scheme uses an m-position mask to specify which PE's are to be activated, each position of the mask corresponding to a bit position in the addresses of the PE's. Each position of the mask will contain either a 0, 1, or X ("don't care"). The only PE's that will be active are those whose address matches the mask: 0 matches 0, 1 matches 1, and either 0 or 1 matches X.

Superscripts will be used as repetition factors when describing masks or PE addresses (e.g., $1^2 0^2 = 1100$). Square brackets will be used to denote a mask. A PE address mask could accompany each machine instruction and interconnection function, or a separate mask instruction could be executed whenever a change in the active status of the PE's is required. A compiler or assembler could be used to convert from one method to the other. For the sake of clarity, each instruction will be accompanied with a mask. For example, executing "$A \leftarrow B + C$ $[X^{m-1}1]$" causes each PE with an odd numbered address to add the contents of B and C, and store the sum in A.

The way this masking scheme interacts with various interconnection networks is examined in [21]. In [22] variations of this masking scheme and methods to implement it are discussed.

Data conditional masks are the implicit result of performing a conditional branch dependent on local data in an SIMD machine environment, where the result of different PE's evaluations may differ. The notation

$$\textit{where } \langle \text{data condition} \rangle \textit{ do } \cdots \textit{ elsewhere } \cdots$$

will be used. Thus, as a result of a conditional *where statement* each PE will set its own internal flag to activate itself for either the "do" or the "elsewhere," but not both. The execution of the "elsewhere" statements must follow the "do" statements; i.e., the "do" and "elsewhere" statements cannot be executed simultaneously. For example, as a result of executing the statement

$$\textit{where } A > B \textit{ do } C \leftarrow A \textit{ elsewhere } C \leftarrow B$$

each PE will load its C register with the maximum of its A and B registers, i.e., some PE's will execute "$C \leftarrow A$," and then the rest will execute "$C \leftarrow B$." This type of masking is used in such machines as the Illiac IV [1] and PEPE [8]. "Where statements" may be nested using a run-time control stack. This is described in [26].

Data conditional statements are an essential part of all programming languages, so it is reasonable to assume they would be present in all SIMD machines. The results of this paper would still be valid even if only data conditional masks were used. This is because if each PE knows its own address, then data conditional masks could be used to simulate PE address masks using no additional interprocessor data transfers.

When PE address masks and data conditional masks are used together, PE address masks must accompany each instruction in the "do" block and in the "elsewhere" block. Thus, in order for a PE to be active it must be in active mode as a result of the where statement and match the PE address mask accompanying the instruction.

When an interconnection function is executed with all PE's in the active state no DTR data is destroyed, it is just transferred to another DTR. However, if some PE's are inactive, then their DTR contents can be destroyed, i.e., overwritten and not transferred. This is the problem, referred to earlier, that the single DTR organization has and the two DTR organization does not. For example, when $N = 8$, the data transfer instruction "shuf [001]" would overwrite and destroy the DTR contents of PE_2. To prevent the loss of the data in the DTR of PE_2 a copy of it must be saved prior to executing the data transfer instruction. An analysis is presented in [21] of the combinations of PE address masks and interconnection functions which will destroy data, i.e., are not bijections. In the algorithms in the next section, only when data will be destroyed by an interprocessor data transfer which is not a bijection will this issue be discussed.

In summary, an SIMD machine can be formally represented as the 4-tuple (N, F, I, M), where

1) N is a positive integer, representing the number of processing elements in the machine,

2) F is the set of interconnection functions (i.e., the interconnection network), where each function is a bijection on the set $\{0, 1, \cdots, N-1\}$, which determines the communication links among the PE's,

3) I is the set of machine instructions, instructions that are executed by each active PE and act on data within that PE, and

4) M is the set of masking schemes, where each mask partitions the set $\{0, 1, \cdots, N-1\}$ into two disjoint sets, the enabled PE's and the disabled PE's.

A particular SIMD machine architecture can be described by specifying N, F, I, and M. In this paper $N = 2^m$, F is varied, I includes instructions for moving data between the DTR and other registers of the same PE, and M includes PE address and data conditional masks.

III. Interconnection Network Simulations

Methods for proving lower and upper time bounds of SIMD machine interconnection network simulation algorithms are presented. Various examples are given by examining the time required for each of the networks defined to simulate the others. These specific simulations are used since each of the networks discussed here has been proposed in some form in the literature and shown to be useful. However,

TABLE I
THE ENTRIES IN ROW x AND COLUMN y ARE LOWER AND UPPER BOUNDS ON THE TIME REQUIRED FOR NETWORK x TO SIMULATE NETWORK y

		PM2I	Cube	Illiac	PS	WPM2I
PM2I	lower	-	2	1	m	2
	upper	-	2	1	2m-2	2
Cube	lower	m	-	m	$2\lfloor m/2 \rfloor$	m
	upper	m	-	m	m	m
Illiac	lower	n/2	(n/2)+1	-	(n/2)+1	(n/2)+1
	upper	n/2	(n/2)+1	-	3n-4	(n/2)+1
PS	lower	2m-1	m+1	2m-1	-	2m-1
	upper	2m	m+1	2m	-	2m
WPM2I	lower	3	2	3	m/2	-
	upper	3	2	3	2m-2	-

there is little in the literature directly comparing the simulation abilities of these types of networks. In [23], the ability of networks to simulate an arbitrary interconnection by sorting destination tags (in $0(m^2)$ time for most of these networks) is discussed. This section is a comparison of the abilities of networks to simulate a particular interconnection (in less time than it would take to simulate an arbitrary one using sorting techniques).

The following assumptions are made.

1) When simulating the interconnection function f, the data that is originally in the DTR of PE_x must be transferred to the DTR of $PE_{f(x)}$, $0 \le x < N$.

2) The interconnection function is to be simulated as if it were executed with all PE's being active. It will be shown after the theorem how this restriction can be removed.

3) The bounds are in terms of the number of executions of interconnection functions required to perform the simulation.

4) The interconnection function of the network to be simulated which requires the most time to simulate will determine the time bounds for the simulation of that network.

The instructions in the simulation algorithms can be divided into three categories: control unit operations, interprocessor data transfers, and register-to-register operations. Control unit operations, such as incrementing an index register in a "for loop," can usually be done in parallel with the previously broadcast PE instruction, thus, taking no additional time. Register-to-register operations, within a PE, will probably involve a single chip, or at worst adjacent chips. The interprocessor data transfers will involve setting the controls of the interconnection network and passing data among the PE's, involving board-to-board, and probably rack-to-rack, distances. Furthermore, if there is a sequence of data transfers and the two DTR model is implemented (to reduce clocking problems), then an input DTR to output DTR move will be required after each transfer. Thus, unless the number of register-to-register operations is much greater than the number of interprocessor data transfers, the time for the interprocessor transfers will be the dominating factor in determining the execution time.

In the theorem that follows, the lower bound and upper bound on the time required to perform each simulation are shown. In most cases these bounds are tight. Unless indicated otherwise, the lower bound is proven in [21]. Each upper bound is based on the time complexity of the algorithm presented to do that simulation. Each algorithm is proven correct. Most proofs are only sketched and are provided as an explanation of the algorithm; details are in [20]. Several proofs are given in detail to demonstrate the techniques used. In the proofs, the transferring of data from PE_x to PE_y will also be referred to as *mapping the address x to y*, since the interconnections have been defined as functions on the PE addresses.

Theorem: In Table I the entries in row x and column y are lower and upper bounds on the time required for network x to simulate network y.

Proof: The notation "network 1 → network 2" means "the case where network 1 is used to simulate network 2." In the algorithms ":" indicates a comment. Proofs for PM_{-i} and WPM_{-i} are omitted since they are similar to PM_{+i} and WPM_{+i}, respectively, $0 \le i < m$.

PM2I → Cube (Lower Bound):
For $0 \le i < m-1$, no single PM2I function is equivalent to $cube_i$.

PM2I → Cube (Upper Bound):
For $cube_{m-1}$, use $PM_{+(m-1)}$, since these functions are equivalent.
For $cube_i$, $0 \le i < m-1$:
(S1) $PM_{+i}[X^m]$: all PE's execute PM_{+i}
(S2) $PM_{-(i+1)}[X^{m-(i+1)}0X^i]$
: active PE's execute $PM_{-(i+1)}$

For example, when $i = 1$ and $N > 4$, the data from the DTR of PE_3 is moved to PE_5, and then to PE_1. For address P whose ith bit is 0, Step 1 maps P to $P + 2^i = cube_i(P)$, which does not match the mask in Step 2. For address P whose ith bit is 1, Step 1 maps P to $P + 2^i$ and Step 2 maps $P + 2^i$ to $P - 2^i = cube_i(P)$.

PM2I → Illiac (Lower and Upper Bound):
$Il_{+1} = PM_{+0}$, $Il_{-1} = PM_{-0}$, $Il_{+n} = PM_{+(m/2)}$,
$$Il_{-n} = PM_{-(m/2)}.$$

PM2I → PS (Upper Bound):
For the exchange: see PM2I → cube analysis, since $exch = cube_0$.

For the shuffle: use *hoba* as an abbreviation for "high order bit position of its address," and *loba* as an abbreviation for "low order bit position of its address." In the algorithm below, Step 2 simulates the shuffle on the DTR contents of each PE where hoba = 0. It operates by moving the data from PE $0p_{m-2}p_{m-3}\cdots p_1p_0$ to $p_{m-2}0p_{m-3}p_{m-4}\cdots p_1p_0$, to $p_{m-2}p_{m-3}0p_{m-2}\cdots p_1p_0$, and so on, to $p_{m-2}p_{m-3}p_{m-4}\cdots p_1p_00 = \text{shuf}(0p_{m-2}\cdots p_1p_0)$. Step 3 saves this data and

Step 5 reloads it. Step 4 is similar to Step 2, and simulates the shuffle where hoba = 1.

(S1) $A \leftarrow$ DTR $[X^m]$
: all PE's save original DTR data in A registers
(S2) *for* $i = m-2$ *until* 0 *step* -1 *do*
$\text{PM}_{+i}\ [X^{m-(i+2)}01X^i]$: shuf for hoba = 0
(S3) DTR $\leftrightarrow A\ [X^m]$
: all PE's save DTR data, reload original DTR data
(S4) *for* $i = m-2$ *until* 0 *step* -1 *do*
$\text{PM}_{-i}\ [X^{m-(i+2)}10X^i]$: shuf for hoba = 1
(S5) DTR $\leftarrow A\ [X^{m-1}0]$
: active PE's reload DTR data saved in S3

For example, when $N = 8$, the contents of the DTR of PE_6 are saved in Step 1, restored in Step 3, and moved to PE_5 in Step 4 (when $i = 0$).

Case 1: PE where hoba = 0. Induction on i can be used to prove the induction hypothesis that after "$\text{PM}_{+i}\ [X^{m-(i+2)}01X^i]$" in Step 2 is executed the data that was originally in the DTR of PE $0p_{m-2}\cdots p_1p_0$ is in the DTR of PE $p_{m-2}p_{m-3}\cdots p_i0p_{i-1}\cdots p_1p_0$ (see Appendix I). After Step 2 is executed with $i = 0$, the data from PE $0p_{m-2}\cdots p_1p_0$ will have been transferred to PE $p_{m-2}\cdots p_1p_00$. Step 3 saves this data and Step 5 reloads it.

The data transfers resulting from Step 2 will not be bijections, but the data that was originally from the DTR's of PE's that had a 0 in the hoba will never be destroyed by these transfers. Consider Step 2 when $i = j$. The transfer moves data into PE's whose addresses have 1 in their p_{j+1} position. The induction hypothesis above shows that after Step 2 executes with $i = j+1$ the data being shuffled are in PE's whose p_{j+1} address bit is 0.

Case 2: PE where hoba = 1. The original DTR data are saved in Step 1 and restored in Step 3. The correctness proof for Step 4 is analogous to that for Step 2.

This algorithm uses three register-to-register operations.

PM2I → WPM2I (Lower Bound):

For $0 < i < m$, no single PM2I function is equivalent to WPM_{+i}.

PM2I → WPM2I (Upper Bound):

For WPM_{+0} use PM_{+0}.

For WPM_{+i}, $0 < i < m$:

(S1) $A \leftarrow$ DTR $[0^m]$
: save PE_0's DTR contents in its A register
(S2) $\text{PM}_{+0}\ [1^{m-i}X^i]$
: active PE's use PM_{+0} for wrap-around carry
(S3) $A \leftrightarrow$ DTR $[0^m]$
: in PE_0 save current DTR, reload original contents
(S4) $\text{PM}_{+i}\ [X^m]$: all PE's execute PM_{+i}
(S5) DTR $\leftarrow A\ [0^m]$
: data saved in S3 restored to DTR of PE_0

For example, when $i = 1$ and $N = 8$, the data from the DTR of PE_6 is moved to PE_7 by Step 2, and then to PE_1 by Step 4. For all PE's other than 1^m, Step 2 simulates any possible effect of the "wrap-around" carry and Step 4 completes the simulation. Step 2 transfers the data from PE 1^m to $0^m = \text{WPM}_{+i}\ (1^m)$. Step 2 is not a bijection, but PE_0's DTR contents, the only data destroyed, is saved in Step 1 and reloaded in Step 3. This algorithm uses three register-to-register operations.

Cube → PM2I (Upper Bound):

For $\text{PM}_{+(m-1)}$, use cube_{m-1}, since these functions are equivalent.

For PM_{+i}, $0 \le i \le m-2$:

(S1) $\text{cube}_i\ [X^m]$
: complements ith bit of all PE addresses
(S2) *for* $j = i+1$ *until* $m-1$ *do* $\text{cube}_j\ [X^{m-j}0^{j-i}X^i]$
: simulates carry

For example, when $i = 1$ and $N = 8$, the data from the DTR of PE_6 is moved to PE_4 by Step 1, and then moved to PE_0 by Step 2 when $j = 2$. The algorithm simulates the action of PM_{+i} by first complementing bit i, then complementing bit $i+1$ if bit i is now 0 (indicating it was a 1), then complementing bit $i+2$ if bits i and $i+1$ are 0's (indicating they were 1's), etc. By induction on k it can be proven that Step 2 will map $p_{m-1}\cdots p_{i+k}1^kp_{i-1}\cdots p_0$ to $p_{m-1}\cdots \bar{p}_{i+k}0^kp_{i-1}\cdots p_0$, when algorithm variable $j = k+i$.

Cube → Illiac (Upper Bound):

Follows from the Cube → PM2I and PM2I → Illiac analyses.

Cube → PS (Upper Bound):

For the exchange use cube_0.

For the shuffle: let the ith bit of a PE's address be ADDR(i).

(S1) *where* ADDR($m-1$) = ADDR(0)
do $A \leftarrow$ DTR $[X^m]$
elsewhere $\text{cube}_0\ [X^m]$
(S2) *for* $j = 1$ *to* $m-1$ *do*
(S3) *where* ADDR(j) $\neq$ ADDR($j-1$)
do $A \leftrightarrow$ DTR $[X^m]$
elsewhere null
(S4) $\text{cube}_j\ [X^m]$
(S5) *where* ADDR($m-1$) = ADDR(0)
do DTR $\leftarrow A\ [X^m]$
elsewhere null

For example, when $N = 8$, the data in the DTR of PE_6 is moved to the DTR of PE_7 by Step 1, to the DTR of PE_5 by Step 4 when $j = 1$, to the A register of PE_5 by Step 3 when $j = 2$, and to the DTR of PE_5 by Step 5.

Recall that shuf $(p_{m-1}\cdots p_1p_0) = p_0p_{m-1}\cdots p_1$. By induction on i, $0 \le i \le m-2$, one can prove the induction hypothesis that in PE $p_{m-1}\cdots p_1p_0$, where $p_{m-1} = p_0$, after cube_i is executed, the DTR will contain the data that was originally in PE $p_0p_{m-2}\cdots p_{i+1}\bar{p}_ip_ip_{i-1}\cdots p_1$ and the A register will contain the data that was originally in PE $p_0p_{m-2}\cdots p_{i+1}p_ip_ip_{i-1}\cdots p_1$ (see Appendix II). When $i = m-2$, the DTR of $p_{m-1}\cdots p_1p_0$, where $p_{m-1} = p_0$, contains the data originally in $p_0\bar{p}_{m-2}p_{m-2}\cdots p_1$ and the A register contains the data originally in $p_0p_{m-2}p_{m-2}\cdots p_1$. Consider the data in the DTR of $p_{m-1}\cdots p_1p_0$.

Case 1: $p_{m-1} = p_{m-2}$. In Step 4 this data is sent to $\bar{p}_{m-1}p_{m-2}\cdots p_0$ when $j = m-1$. This is correct since the DTR contained the data originally in $p_0\bar{p}_{m-2}p_{m-2}\cdots p_1 = p_0\bar{p}_{m-1}p_{m-2}\cdots p_1$.

Case 2: $p_{m-1} \neq p_{m-2}$. In Step 3 this data is stored in A when $j = m - 1$, and in Step 5 it is reloaded into the DTR. This is correct since it was originally in $p_0 \bar{p}_{m-2} p_{m-2} \cdots p_1 = p_0 p_{m-1} p_{m-2} \cdots p_1$.

Consider what happens to the data in the A register of $p_{m-1} \cdots p_1 p_0$.

Case 1: $p_{m-1} = p_{m-2}$. In Step 2 this data is loaded into the DTR. This is correct since it was originally in $p_0 p_{m-2} p_{m-2} \cdots p_1 = p_0 p_{m-1} p_{m-2} \cdots p_1$.

Case 2: $p_{m-1} \neq p_{m-2}$. In Step 3 this data is loaded into the DTR and Step 4 sends it to $\bar{p}_{m-1} p_{m-2} \cdots p_0$ when $j = m - 1$. This is correct since it was originally in $p_0 p_{m-2} p_{m-2} \cdots p_1 = p_0 \bar{p}_{m-1} p_{m-2} \cdots p_1$.

This algorithm uses m comparisons and m register-to-register moves.

Cube → WPM2I (Upper Bound):

For WPM_{+0}: see the Cube → PM2I analysis, since $\text{PM}_{+0} = \text{WPM}_{+0}$.

For WPM_{+i}, $0 < i < m$: the algorithm and the correctness proof are similar to that of the Cube → PM2I case.

(S1) $\text{cube}_i\ [X^m]$
: complement ith bit of all PE addresses

(S2) *for* $j = i + 1$ *until* $m - 1$ *do* $\text{cube}_j\ [X^{m-j}0^{j-i}X^i]$
: simulate carry

(S3) $\text{cube}_0\ [0^{m-i}X^i]$
: simulate wrap-around carry on 0th bit

(S4) *for* $j = 1$ *until* $i - 1$ *do* $\text{cube}_j\ [0^{m-i}X^{i-j}0^j]$
: rest of wrap-around

For example, when $i = 1$ and $N = 8$, the data in the DTR of PE_6 is moved to PE_4 by Step 1, to PE_0 by Step 2 when $j = 2$, and to PE_1 by Step 3.

Illiac → PM2I (Upper Bound):

For PM_{+i}, $m/2 \leq i < m$:

for $j = 1$ *until* $2^i/n$ *do* $\text{Il}_{+n}\ [X^m]$
: all PE's execute Il_{+n} $2^i/n$ times

For example, when $i = 3$ and $N = 16$, the data in the DTR of PE_6 is moved to PE_{10} when $j = 1$, and then to PE_{14} when $j = 2$. Il_{+n} executed $2^i/n$ times equals PM_{+i}.

For PM_{+i}, $0 \leq i < m/2$: the algorithm is the same as the case above, except both "n's" are changed to "1's."

Illiac → Cube (Lower Bound):

Let $d(x, y) = |x - y|$, the absolute difference. This measurement function is a *metric* [20]. If f is an interconnection function and d is a metric, then $d(x, f(x))$ is the "distance" that f can "move" PE address x according to the metric d. Let $j = (m/2) - 1$. Then $d(0, \text{cube}_j(0)) = n/2$.

Case 1: Il_{+n} or Il_{-n} is used. For $0 \leq x < N$, $d(x, \text{Il}_{+n}(x)) = d(x, \text{Il}_{-n}(x)) = n$, so to move a distance of $n/2$ the Il_{+1} and/or Il_{-1} functions must be used also, e.g., if Il_{+n} was used to map 0 to $n/2$, it would have to be followed by $n/2$ executions of Il_{-1}.

Case 2: Neither Il_{+n} nor Il_{-n} is used. For $0 \leq x < N$, $d(x, \text{Il}_{+1}(x)) = d(x, \text{Il}_{-1}(x)) = 1$. The only way to map 0 to $n/2$ in $n/2$ steps is to execute Il_{+1} $n/2$ times. But $\text{cube}_j(n/2) = 0$, and no subsequence of $(\text{Il}_{+1})^{n/2}$ can perform this mapping.

Illiac → Cube (Upper Bound):

For cube_{m-1}: see the Illiac → PM2I analysis, since $\text{cube}_{m-1} = \text{PM}_{+m-1}$.

For cube_i, $m/2 \leq i \leq m - 2$:

(S1) $A \leftarrow \text{DTR}\ [X^{m-(i+1)}1X^i]$
: active PE's save DTR contents in A

(S2) *for* $j = 1$ *until* $2^i/n$ *do* $\text{Il}_{+n}\ [X^m]$
: use Il_{+n} $2^i/n$ times $(= +2^i)$

(S3) $A \leftrightarrow \text{DTR}\ [X^{m-(i+1)}1X^i]$
: active PE's switch DTR and A contents

(S4) *for* $j = 1$ *until* $2^i/n$ *do* $\text{Il}_{-n}\ [X^m]$
: use Il_{-n} $2^i/n$ times $(= -2^i)$

(S5) $\text{DTR} \leftarrow A\ [X^{m-(i+1)}1X^i]$
: active PE's reload data saved in S3

For example, when $i = 2$ and $N = 16$, the data from the DTR of PE_6 is moved to the A register by Step 1, back to the DTR by Step 3, and then to PE_2 by Step 4. For $p_i = 0$, Step 2 maps $p_{m-1} \cdots p_{i+1}0p_{i-1} \cdots p_0$ to $p_{m-1} \cdots p_{i+1}1p_{i-1} \cdots p_0$ and Step 3 saves the data. For $p_i = 1$, Step 1 saves the data, Step 3 reloads them, and Step 4 maps $p_{m-1} \cdots p_{i+1}1p_{i-1} \cdots p_0$ to $p_{m-1} \cdots p_{i+1}0p_{i-1} \cdots p_0$. This algorithm uses three register-to-register operations.

For $\text{cube}_{(m/2)-1}$:

(S1) *for* $j = 1$ *until* $n/2$ *do* $\text{Il}_{+1}\ [X^m]$
: all PE's execute Il_{+1} $n/2$ times

(S2) $\text{Il}_{-n}\ [X^{m/2}0X^{(m/2)-1}]$: active PE's execute Il_{-n}

For example, when $N = 16$, the data from the DTR of PE_6 is moved to PE_7 by Step 1 when $j = 1$, to PE_8 by Step 1 when $j = 2$, and then to PE_4 by Step 3. Step 1 maps address P to $P + (n/2) = \text{cube}_{(m/2)-1}(P)$, where the $(m/2) - 1$st bit of P is 0. Step 1 maps address P to $P + (n/2)$ and Step 2 maps it to $P - (n/2) = \text{cube}_{(m/2)-1}(P)$, where the $(m/2) - 1$st bit of P is 1.

For cube_i, $0 \leq i \leq (m/2) - 2$: the algorithm is the same as the $m/2 \leq i \leq m - 2$ case above, if "n" is changed to "1" in lines Step 2 and Step 4.

Illiac → PS (Upper Bound):

For the exchange: see the Illiac → Cube analysis, since $\text{exch} = \text{cube}_0$.

For the shuffle: in [18] an algorithm for the Illiac to simulate the shuffle function using $4n - 4$ interprocessor data transfers is presented. The algorithm presented here, which uses only $3n - 4$ interprocessor data transfers, is based on the PM2I → shuffle algorithm. The key is that the instruction "$\text{PM}_{+i}\ [X^{m-(i+2)}01X^i]$" in the PM2I → shuffle algorithm is equivalent to the sequence of instructions:

(S1) $A \leftarrow \text{DTR}\ [X^m]$
: all PE's save original DTR contents in A

(S2) simulation of "$\text{PM}_{+i}\ [X^m]$" by Illiac
(see Illiac → PM2I)

(S3) $A \leftrightarrow \text{DTR}\ [X^m]$
: all PE's save DTR, reload original DTR

(S4) $\text{DTR} \leftarrow A\ [X^{m-(i+2)}10X^i]$
: active PE's reload DTR saved in S3

The number of interconnection functions executed in the Illiac → shuffle algorithm for the equivalent of each of Step 2 and Step 4 in the PM2I → shuffle algorithm is

$$\sum_{i=m/2}^{m-2} 2^i/n + \sum_{i=0}^{(m/2)-1} 2^i = (3n/2) - 2,$$

and the number of register-to-register operations is $3(m-1)$.

Illiac → WPM2I (Upper Bound):

Use the algorithm for PM2I → WPM2I, substituting Π_{+1} for PM_{+0}, and using the Illiac → PM2I algorithm for PM_{+i}, $0 < i < m$.

PS → PM2I (Upper Bound):

For PM_{+i}, $0 \le i < m$:

(S1) *for* $j = i$ *until* $m-1$ *do*
: execute S2 and S3 $m-i$ times for "carry"

(S2) shuf $[X^m]$
: all PE's execute shuffle

(S3) exch $[1^{m-(j+1)}X^{j+1}]$
: active PE's execute exchange

(S4) *for* $j = 1$ *until* i *do* shuf $[X^m]$
: all PE's execute i shuffles

For example, when $i = 1$ and $N = 8$, the DTR contents of PE_6 is moved to PE_5 by Step 2 and to PE_4 by Step 3 when $j = 1$, to PE_1 by Step 2 and to PE_0 by Step 3 when $j = 2$, and stays in PE_0 as a result of Step 4. Step 4 is never executed if $i = 0$. The algorithm and correctness proof are similar to that in the Cube → PM2I analysis.

PS → Cube (Upper Bound):

For $cube_0$ use the exchange.

For $cube_i$, $0 < i < m$:

(S1) *for* $j = 1$ *until* $m-i$ *do* shuf $[X^m]$
: all PE's execute $m-i$ shuffles

(S2) exch $[X^m]$: all PE's execute exchange

(S3) *for* $j = 1$ *until* i *do* shuf $[X^m]$
: all PE's execute i shuffles

For example, when $i = 1$ and $N = 8$, the data from the DTR of PE_6 is moved to 5 and then to 3 by Step 1, then to 2 by Step 2, and finally to 4 by Step 3. By definition

$$(\text{shuf}^i(\text{exch}(\text{shuf}^{m-i}(p_{m-1}\cdots p_1 p_0)))) =$$

$$p_{m-1}\cdots p_{i+1}\bar{p}_i p_{i-1}\cdots p_0.$$

PS → Illiac (Upper Bound):

Follows from the PS → PM2I and PM2I → Illiac analyses.

PS → WPM2I (Upper Bound):

For WPM_{+0} see the PS → PM2I analysis, since $PM_{+0} = WPM_{+0}$.

For WPM_{+i}, $0 < i < m$:

(S1) *for* $j = i$ *until* $m-1$ *do*
: do S2 and S3 $m-i$ times for "carry"

(S2) exch $[1^{m-j}X^j]$
: active PE's execute exchange

(S3) shuf $[X^m]$
: all PE's execute shuffle

(S4) exch $[X^m]$: all PE's execute exchange

(S5) shuf $[X^m]$: all PE's execute shuffle

(S6) *for* $j = 2$ *until* i *do*
: do S7 and S8 $i-1$ times for "wrap" carry

(S7) exch $[1^{i-j}0^{(m-i)+1}X^{j-1}]$
: active PE's execute exchange

(S8) shuf $[X^m]$
: all PE's execute shuffle

For example, when $i = 2$ and $N = 8$, the data in the DTR of PE_6 is moved to PE_7 by Step 2, to PE_7 by Step 3, to PE_6 by Step 4, to PE_5 by Step 5 and then to PE_3 by Step 8. Step 7 and Step 8 are never executed if $i = 1$. The algorithm and correctness proof are similar to the PS → PM2I analysis.

WPM2I → PM2I (Lower Bound):

For PM_{+i}, $1 < i < m$, consider $PM_{+i}(1^m) = 0^{m-i}1^i$ and $PM_{+i}(1^{m-i}0^i) = 0^m$. The only way WPM2I can move data from PE 1^m to $0^{m-i}1^i$ in two steps is WPM_{-0} followed by WPM_{+i}. The only way WPM2I can map $1^{m-i}0^i$ to 0^m in two steps is WPM_{+i} followed by WPM_{-0}. Thus, at least three steps are required.

WPM2I → PM2I (Upper Bound):

For PM_{+0} use WPM_{+0}.

For PM_{+i}, $0 < i < m$:

(S1) $A \leftarrow$ DTR $[1^m]$
: save PE 1^m DTR data in its A register

(S2) WPM_{+i} $[X^m]$
: all PE's execute WPM_{+i}

(S3) $B \leftarrow$ DTR $[X^m]$
: all PE's save DTR data in B register

(S4) DTR $\leftarrow A$ $[1^m]$
: PE 1^m reloads original DTR data

(S5) WPM_{-0} $[X^m]$
: all PE's execute WPM_{-0}

(S6) $B \leftarrow$ DTR $[0^{m-i}X^i]$
: active PE's save DTR data in B register

(S7) WPM_{+i} $[1^{m-1}0]$
: PE $1^{m-1}0$ executes WPM_{+i}

(S8) $B \leftarrow$ DTR $[0^{m-i}1^i]$
: PE $0^{m-i}1^i$ saves DTR data in B register

(S9) DTR $\leftarrow B$ $[X^m]$
: all PE's load DTR with contents of B register

For example, when $i = 2$ and $N = 8$, the data in the DTR of PE_6 is moved to PE_3 by Step 2, to PE_2 by Step 5, to the B register by Step 6, and back to the DTR by Step 9. For all PE addresses P not of the form $1^{m-i}p_{i-1}\cdots p_1p_0$, Step 2 performs the simulation, since $WPM_{+i}(P) = PM_{+i}(P)$. For PE addresses of the form $1^{m-i}p_{i-1}\cdots p_1p_0$, except for 1^m, Step 2 maps them to $(0^{m-i}p_{i-1}\cdots p_1p_0)+1$ and Step 5 subtracts the "+1." Step 5 maps 1^m to $1^{m-1}0$ and Step 7 maps this to $0^{m-i}1^i$. Step 7 is not a bijection on the addresses, but the only data destroyed, the DTR contents of $0^{m-i}1^i$, is not needed. This algorithm uses six register-to-register operations.

WPM2I → Cube (Lower Bound):

No single WPM2I function is equivalent to a Cube function.

WPM2I → Cube (Upper Bound):

For cube_i, $0 \leq i < m$: the algorithm and proof are the same as for Illiac → cube_i, $m/2 \leq i \leq m-2$, substituting "$\text{WPM}_{+i}[X^m]$" for Step 2 and "$\text{WPM}_{-i}[X^m]$" for Step 4.

WPM2I → Illiac (Lower and Upper Bound):

Follows from the WPM2I → PM2I and PM2I → Illiac analyses.

WPM2I → PS (Upper Bound):

For the exchange: see WPM2I → Cube analysis, since exchange = cube_0.

For the shuffle: same as PM2I → PS, using WPM_{+i} in the place of PM_{+i} and WPM_{-i} in the place of PM_{-i}. □

These simulation algorithms were based on the premise that the interconnection function being simulated was executed with all PE's active. This will not be true if the function is broadcast with a mask other than $[X^m]$ or inside a conditional.

If network k can simulate interconnection function f executed with mask $[X^m]$, then it can simulate f executed with arbitrary PE address mask $[R]$ using no additional interprocessor data transfers, but requiring $0(m)$ machine operations and extra space. This is done by having each PE save a copy of its DTR data, simulating "f" with all PE's active, and then having those PE's where f^{-1} (f inverse) of their address does not match $[R]$ restore their original DTR data. The value of f^{-1} for each PE could be stored or easily computed [20].

Consider the case where the interconnection function to be simulated is executed with a PE address mask inside a where statement. If network k can simulate interconnection function f executed with PE address mask $[X^m]$ using y interprocessor data transfers, then k can simulate the effect of f executed with arbitrary PE address mask $[R]$ within a where statement using at most $2y$ interprocessor data transfers. To implement conditional masks each PE must have a flag (or stack of flags if nested where statements are allowed [26]) to indicate whether or not the PE is active for the current "do" or "elsewhere" block. The simulation is done by having an instruction which allows all PE's (active or not) to save their status flag. All PE's save their status flag, all are activated, and all save their DTR contents; only PE's which were active and match $[R]$ set a tag to 1, and f is simulated, moving the tag with the data. Where tag is not 1, the original DTR contents are restored. If there is a reserved bit position which can be used for the tag, then the simulation can be done using y interprocessor data transfers. This reserved bit could be one of the W bits of the data word that is to be transferred or a $W+1$st bit position added to the network, DTR, and fast access registers.

IV. Conclusions

Current technology has created the "age of the microprocessor." This has made feasible the future construction of computer systems with 2^{10}–2^{14} processors. In order to design a multimicroprocessor system capable of operating in the SIMD mode of parallelism, a model of SIMD machines and an analysis of SIMD machine interconnection networks is necessary.

To find the lower bounds of these simulation algorithms, one must consider which operations occur in parallel and find ways to describe these actions mathematically. To construct the simulation algorithms, the parallel flow of N data words through N processing elements must be understood. It must be determined which data may get destroyed by a data transfer that is not representable as a bijection on the PE addresses and save that data in such a way that it can later be identified and used. Furthermore, special actions must be taken when the interconnection to be simulated is executed with some PE's disabled. To prove that the algorithms are correct, standard mathematical techniques, such as induction and case analysis, were adapted for use in parallel program analysis. In addition, the approach of considering the simulation problem as a task to map one integer (or class of integers) to another integer (or another class of integers) was taken, as opposed to viewing the process strictly as one of transferring data among PE's.

The methods presented in this paper may be generalized and used to compare other networks. These techniques were demonstrated by examples since there are currently no good "algorithms" for generating such lower bound proofs, simulation algorithms, or SIMD algorithm correctness proofs for arbitrary networks.

No attempt is made to claim that any one network is "best" as a result of the analysis in this paper. One factor which will influence the decision of which network to implement in the system is the types of computations for which the system will be primarily used. For example, for simple pattern recognition smoothing algorithms, the nearest neighbor interconnection scheme, with only eight interconnection functions, may be the most efficient. The number of processing elements in the system is another important factor. For small N, a cross bar switch may be acceptable. Other factors include computational speed and cost requirements. Assuming a "general purpose" SIMD machine, where N is large, some comments about the advantages and disadvantages of the various networks, independent of the factors above, can be made.

The Illiac network is much more limited with respect to simulation capabilities than its superset the PM2I, although it has the advantage of having only four interconnection functions, as compared to the $2m-1$ functions of PM2I. However, if a designer wants the capabilities of the PM2I, but cannot afford $2m-1$ interconnection functions, a compromise can be made. Any number of PM2I functions can be eliminated and simulated by the remaining functions. For example, the number of functions may be reduced from $2m-1$ to m (assuming m is even) by eliminating $\text{PM}_{\pm i}$ for all odd i. The functions $\text{PM}_{\pm(i-1)}$ could be executed twice when one of the eliminated functions is needed. Thus, the PM2I network presents more design alternatives than the Illiac and can be reduced to the Illiac.

The WPM2I network results were quite similar to those of the PM2I network. The number of interconnection functions may be reduced, as in the case of the PM2I, but with a

little more difficulty (i.e.,

$$\mathrm{WPM}_{+i}(x) \neq \mathrm{WPM}_{+(i-1)}(\mathrm{WPM}_{+(i-1)}(x))$$

for all $0 \leq x < N$). WPM2I does have certain interesting group theoretic properties, mentioned earlier, that PM2I does not have, but the WPM2I network is significantly more difficult for the programmer of an SIMD machine to use. The PM2I is based on simple mod N arithmetic, while the data transfer patterns established by the WPM2I network are complicated by the "wrap-around." Thus, the PM2I network is preferable where the particular group theoretic properties of WPM2I are not required.

The Cube and PM2I networks are conceptually similar, with the Cube connection pattern based on a "logical neighborhood" and the PM2I pattern based on a "modulo N addition/subtraction" neighborhood. The Cube uses almost half as many interconnection functions as PM2I, and the algorithm presented for simulating the shuffle is almost twice as fast as the algorithm using the PM2I (although the lower bounds on the two tasks are the same). However, the Cube requires m steps to simulate the PM2I while the PM2I requires only two steps to simulate the Cube, and the number of connections in the Cube cannot be reduced as it can with the PM2I. Due to these considerations, the PM2I may be preferable in the general case.

The PS network has the advantage of requiring only two interconnection functions, with which it can simulate the other networks discussed using at most $2m$ steps. If the system architect is concerned with minimizing hardware costs and is willing to use $2m$ transfers for the various interconnection patterns which were shown would require that amount, then PS is an excellent choice. If the main concern of the system architect is computational speed, without "unreasonable" expense, then a good choice is a PM2I-shuffle hybrid, consisting of the $2m - 1$ PM2I functions and a shuffle function, which could simulate any of the functions discussed in at most two steps. Such a hybrid network would offer great flexibility and speed, while being a relatively small portion of the total cost of a system when N is in the 2^{10}–2^{14} range.

In conclusion, the results of this paper provide comparison information to aid the SIMD machine architect in choosing an interconnection network which will be best suited to the needs of the system. The methods presented provide tools for the designer to use to evaluate and compare other networks and hybrids of networks.

Appendix I

Proof of induction hypothesis that after "$\mathrm{PM}_{+i}\ [X^{m-(i+2)}01X^{i}]$" in Step 2 of the PM2I → PS simulation algorithm is executed the data that was originally in the DTR of PE $0p_{m-2}\cdots p_1p_0$ is in the DTR of PE $p_{m-2}p_{m-3}\cdots p_i0p_{i-1}\cdots p_1p_0$.

Basis: $i = m - 2$. PE $01p_{m-3}\cdots p_1p_0$ matches $[01X^{m-2}]$ and executes PM_{+m-2}, transferring the data to PE $10p_{m-3}\cdots p_1p_0$. PE $00p_{m-3}\cdots p_1p_0$ does not match the mask, so no action is taken, which gives the desired results.

Induction Step: Assume the hypothesis is true for $m - 2 \leq i \leq j$, and consider $i = j - 1$. From the induction hypothesis, the address of the PE containing the data that was originally in the DTR of PE $0p_{m-2}\cdots p_1p_0$ is $p_{m-2}\cdots p_{j+1}p_j0p_{j-1}\cdots p_1p_0$. PE $p_{m-2}\cdots p_j01p_{j-2}\cdots p_0$ matches $[X^{m-(j+1)}01X^{j-1}]$ and executes $\mathrm{PM}_{+(j-1)}$, transferring the data to PE $p_{m-2}\cdots p_j10p_{j-2}\cdots p_0 = p_{m-2}\cdots p_jp_{j-1}0p_{j-2}\cdots p_0$. PE $p_{m-1}\cdots p_{j+1}00p_{j-2}\cdots p_0$ does not match the mask, so no action will be taken, which gives the desired results.

Appendix II

Proof of induction hypothesis that in PE $p_{m-1}\cdots p_1p_0$, where $p_{m-1} = p_0$, after cube_i in the Cube → PS simulation is executed, the DTR will contain the data that was originally in PE $p_0p_{m-2}\cdots p_{i+1}\bar{p}_ip_ip_{i-1}\cdots p_1$ and the A register will contain the data that was originally in PE $p_0p_{m-2}\cdots p_{i+1}p_ip_ip_{i-1}\cdots p_1$.

Basis: $i = 0$. After Step 1 is executed the A register contains the data that was originally in PE $p_{m-1}\cdots p_1p_0 = p_0p_{m-2}\cdots p_1p_0$. The DTR will receive the data sent by PE $p_{m-1}\cdots p_1\bar{p}_0 = p_0p_{m-2}\cdots p_1\bar{p}_0$.

Induction Step: Assume the hypothesis true for $i = j - 1$, and consider $i = j$.

Case 1: $p_j \neq p_{j-1}$. After Step 4 is executed A will contain the DTR contents of that PE before Step 3 and Step 4 were executed. From the induction hypothesis, A contains the data originally in $p_0p_{m-2}\cdots p_{j+1}p_j\bar{p}_{j-1}p_{j-1}\cdots p_1 = p_0p_{m-2}\cdots p_{j+1}p_jp_jp_{j-1}\cdots p_1$, since $p_j = \bar{p}_{j-1}$. The DTR receives the DTR contents of $p_{m-1}\cdots p_{j+1}\bar{p}_jp_{j-1}\cdots p_0$ as they were before Step 4 was executed. Thus, from the induction hypothesis, the DTR, after Step 4 is executed, contains the data originally in $p_0p_{m-2}\cdots p_{j+1}\bar{p}_j\bar{p}_{j-1}p_{j-1}\cdots p_1 = p_0p_{m-2}\cdots p_{j+1}\bar{p}_jp_jp_{j-1}\cdots p_1$.

Case 2: $p_j = p_{j-1}$. After Step 4 is executed A will contain what it did before Step 3 and Step 4 were executed. From the induction hypothesis, A contains the data originally in $p_0p_{m-2}\cdots p_{j+1}p_jp_{j-1}p_{j-1}\cdots p_1 = p_0p_{m-2}\cdots p_{j+1}p_jp_jp_{j-1}\cdots p_1$, since $p_j = p_{j-1}$. The DTR receives the contents of the A register of $p_{m-1}\cdots p_{j+1}\bar{p}_jp_{j-1}\cdots p_0$ as they were before Step 3 and Step 4 were executed. Thus, from the induction hypothesis, the DTR, after Step 4 is executed, contains the data originally in $p_0p_{m-2}\cdots p_{j+1}\bar{p}_jp_{j-1}p_{j-1}\cdots p_1 = p_0p_{m-2}\cdots p_{j+1}\bar{p}_jp_jp_{j-1}\cdots p_1$.

Acknowledgment

The author wishes to thank J. D. Ullman for his guidance and suggestions and L. J. Siegel for her many comments. The author also gratefully acknowledges the suggestions of the referees.

References

[1] G. Barnes *et al.*, "The Illiac IV computer," *IEEE Trans. Comput.*, vol. C-17, pp. 746–757, Aug. 1968.

[2] K. E. Batcher, "The flip network in STARAN," in *1976 Int. Conf. Parallel Processing*, Aug. 1976, pp. 65–71.

[3] ——, "The multidimensional access memory in STARAN," *IEEE Trans. Comput.*, vol. C-26, pp. 174–177, Feb. 1977.

[4] L. H. Bauer, "Implementation of data manipulating functions on the STARAN associative processor," in *1974 Sagamore Comput. Conf. Parallel Processing*, Aug. 1974, pp. 209–227.

[5] V. E. Benes, *Mathematical Theory of Connecting Networks and Telephone Traffic*. New York: Academic, 1965.

[6] W. J. Bouknight *et al.*, "The Illiac IV system," *Proc. IEEE*, vol. 60, pp. 369–388, Apr. 1972.
[7] G. R. Couranz, M. S. Gerhardt, and C. J. Young, "Programmable RADAR signal processing using the RAP," in *1974 Sagamore Comput. Conf. Parallel Processing*, Aug. 1974, pp. 37–52.
[8] B. A. Crane *et al.*, "PEPE computer architecture," in *Compcon 72, IEEE Comput. Soc. Conf.*, Sept. 1972, pp. 57–60.
[9] G. Feierbach and D. Stevenson, "A feasibility study of programmable switching networks for data routing," Inst. Advanced Computation, Sunnyvale, CA, Phoenix Project Memo 003, May 1977.
[10] T. Y. Feng, "Data manipulating functions in parallel processors and their implementations," *IEEE Trans. Comput.*, vol. C-23, pp. 309–318, Mar. 1974.
[11] ——, "Parallel processors and processing," *IEEE Trans. Comput.*, vol. C-26, p. 97, Feb. 1977.
[12] P. M. Flanders *et al.*, "Efficient high speed computing with the distributed array processor," in *Symp. High Speed Comput. Algorithm Organization*, Apr. 1977, pp. 113–128.
[13] M. J. Flynn, "Very high-speed computing systems," *Proc. IEEE*, vol. 54, pp. 1901–1909, Dec. 1966.
[14] L. C. Higbie, "The Omen computer: Associative array processor," in *Compcon 72, IEEE Comput. Soc. Conf. Proc.*, Sept. 1972, pp. 287–290.
[15] T. Lang, "Interconnections between processors and memory modules using the shuffle-exchange network," *IEEE Trans. Comput.*, vol. C-25, pp. 496–503, May 1976.
[16] T. Lang and H. S. Stone, "A shuffle-exchange network with simplified control," *IEEE Trans. Comput.*, vol. C-25, pp. 55–66, Jan. 1976.
[17] D. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.*, vol. C-24, pp. 1145–1155, Dec. 1975.
[18] S. E. Orcutt, "Implementation of permutation functions in Illiac IV-type computers," *IEEE Trans. Comput.*, vol. C-25, pp. 929–936, Sept. 1976.
[19] M. C. Pease, "The indirect binary n-cube microprocessor array," *IEEE Trans. Comput.*, vol. C-26, pp. 458–473, May 1977.
[20] H. J. Siegel, "SIMD machine interconnection network design," Dep. Elec. Eng., Princeton Univ., Princeton, NJ, Comp. Sci. Lab. Tech. Rep. 198, Jan. 1976.
[21] ——, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," *IEEE Trans. Comput.*, vol. C-26, pp. 153–161, Feb. 1977.
[22] ——, "Controlling the active/inactive status of SIMD machine processors," in *1977 Int. Conf. Parallel Processing*, Aug. 1977, p. 183.
[23] ——, "Partitionable SIMD computer system interconnection network universality," in *16th Annu. Allerton Conf. Commun. Control, Computing*, Oct. 1978, pp. 586–595.
[24] ——, "Interconnection networks for SIMD machines," *Computer*, vol. 12, pp. 57–65, June 1979.
[25] ——, "Partitioning permutation networks: The underlying theory," in *1979 Int. Conf. Parallel Processing*, Aug. 1979, pp. 175–184.
[26] H. J. Siegel and P. T. Mueller, Jr., "The organization and language design of microprocessors for an SIMD/MIMD system," in *2nd Rocky Mt. Symp. Microcomput.*, Aug. 1978, pp. 311–340.
[27] H. J. Siegel and S. D. Smith, "Study of multistage SIMD interconnection networks," in *5th Annu. Symp. Comp. Arch.*, Apr. 1978, pp. 223–229.
[28] S. D. Smith and H. J. Siegel, "Recirculating, pipelined, and multistage SIMD interconnection networks," in *1978 Int. Conf. Parallel Processing*, Aug. 1978, pp. 206–214.
[29] ——, "An emulator network for SIMD machine interconnection networks," in *6th Int. Symp. Comp. Arch.*, Apr. 1979, pp. 232–241.
[30] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. C-20, pp. 153–161, Feb. 1971.
[31] R. C. Swanson, "Interconnections for parallel memories to unscramble p-ordered vectors," *IEEE Trans. Comput.*, vol. C-23, pp. 1105–1115, Nov. 1974.
[32] A. H. Wester, "Special features in SIMDA," in *1972 Sagamore Comput. Conf.*, Aug. 1972, pp. 29–40.
[33] D. E. Wilson, "The PEPE support software system," in *Compcon 72, IEEE Comput. Soc. Conf.*, Sept. 1972, pp. 61–64.

Howard Jay Siegel (M'77) was born in New Jersey on January 16, 1950. He received the S.B. degree in electrical engineering and the S.B. degree in management, both from the Massachusetts Institute of Technology, Cambridge, in 1972, the M.A. and M.S.E., degrees in 1974, and the Ph.D. degree in electrical engineering and computer science in 1977, all from Princeton University, Princeton, NJ.

In 1976 he joined the School of Electrical Engineering, Purdue University, West Lafayette, IN, where he is currently an Assistant Professor. Since January 1979 he has also been affiliated with Purdue's Laboratory for Applications of Remote Sensing. His research interests include parallel processing, multimicroprocessor systems, image processing, and speech processing. His current activities include designing a partitionable SIMD/MIMD multimicroprocessor system for image processing.

Dr. Siegel is a member of the Association of Computing Machinery, Eta Kappa Nu, and Sigma Xi. He has served as Chairman of the Central Indiana Chapter of the IEEE Computer Society. He is currently Vice-Chairman of the Association for Computing Machinery Special Interest Group on Computer Architecture and an IEEE Computer Society Distinguished Visitor.