I/O and computation overlap on SIMD systolic arrays

Dominique Lavenier, Frédéric Raimbault, Patrice Frison

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

I/O and Computation Overlap on SIMD Systolic Arrays

Dominique Lavenier, Frédéric Raimbault, Patrice Frison

N° 2096

Novembre 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

Rapport de recherche

1993

ISSN 0249-6399

I/O and Computation Overlap on SIMD Systolic Arrays

Dominique Lavenier*, Frédéric Raimbault, Patrice Frison

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet API

Rapport de recherche n ° 2096 — Novembre 1993 — 21 pages

Abstract: A mechanism for overlapped I/O management operations and computation on a SIMD linear systolic array is presented. This mechanism is based on two synchronized controllers allowing a speedup factor of 2 over SIMD machines without overlapped facility. Code generation is achieved using the C-stolic language, specifically designed for the architectural features of overlapped SIMD systolic arrays.

Key-words: SIMD, systolic arrays, computation overlap

(Résumé : tsvp)

submitted to the *Journal of VLSI Signal Processing*

*lavenier@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Recouvrement des calculs et des entrées/sorties sur un réseau systolique de type SIMD

Résumé : Un mécanisme pour la gestion du recouvrement des calculs et des entrées/sorties sur un réseau systolique linéaire de type SIMD est présenté. Ce mécanisme, basé sur une architecture à deux séquenceurs, autorise un facteur d'accélération de 2 par rapport à une architecture sans possibilité de recouvrement. La génération de code est assuré par le langage C-stolic, développé pour ce style d'architecture.

Mots-clé : SIMD, réseaux systoliques, recouvrement des calculs

1 Introduction

One approach to exploiting data parallelism is based on the SIMD (Single-Instruction-Multiple-Data) architectural concept. Systolic machines are a special case of this model. As characterized by Kung [8], a systolic network is composed of identical simple cells that are locally and regularly connected. Data move through the network at constant speed and interact where they meet.

Typically, a complete systolic system consists of an host computer, an array of processors and an interface. Many machines based on a linear array of processors have been designed using this architecture scheme [10] [13] [9] [6] [5]. The host computer runs a main program and requests the systolic array for specialized tasks on specific data. The array of processors performs intensive computation on these data and delivers results. Because of the high data bandwidth required by a systolic array, the host computer cannot be directly connected to the array. An interface is needed.

The performance of the whole system depends greatly on the way the systolic array is linked to the host and, consequently, on the interface. For SIMD systolic arrays, the interface is in charge of both broadcasting instructions to the processors and sending and receiving data to and from the array. As a result, the total amount of information which is transferred between the array and the interface is very high: on each machine cycle, an instruction must be delivered and, on each systolic cycle (generally composed of a few instructions), several data are sent and received.

Managing efficiently such a quantity of data is not an easy task and must receive close attention to preserve the computation power obtained by parallel architectures [2]. This paper focuses on two complementary aspects of this data bandwidth problem. We first discuss how to manage high data bandwidth on a SIMD linear systolic network and propose a hardware mechanism based on two synchronized controllers. We approach this problem from the aspect of programming such a machine, using a specially developped language: C-stolic [11].

The next section presents different approaches of managing computation and I/O operation on a SIMD linear systolic array and shows how overlap between computation and data transfer can be achieved. Section 3 introduces the basis of C-stolic, which allows efficient parallelization of algorithms on a systolic structure. Finally, section 4 presents some experiments we have done to validate our model of architecture.

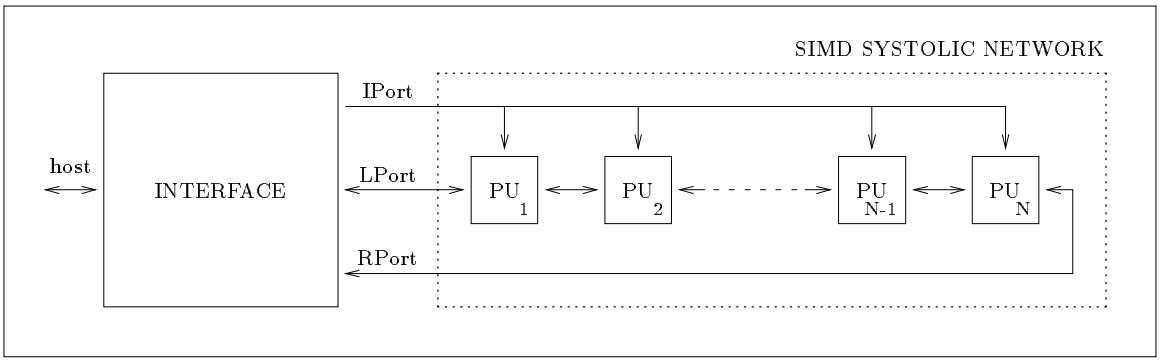


Figure 1: SIMD linear systolic array

2 Input/Output management

A SIMD systolic array must be fed with two kinds of information: the instructions (broadcasted to all the processors) and the data (sent to one extremity and received from the other). In a linear array, three ports are considered (cf figure 1): one input port (instructions) and two bidirectional ports (data). They are respectively referenced as `IPort` (instruction port), `LPort` (left port) and `RPort` (right port). The role of the interface, as represented in figure 1 includes three different task:

- broadcasting instructions to the processing units (PU),
- generating and receiving data to/from the network,
- communicating with the host computer.

We will temporarily set aside the third point and return to it later (section 2.3). The problem we now focus on is how to handle simultaneous data and instruction transfer in a SIMD systolic array. This section presents first a simple approach which uses a single control unit for both transfers. After discussing advantages and drawbacks we introduce a more sophisticated I/O management mechanism based on two controllers.

2.1 Single control unit

Parallelization of algorithms on systolic structures consists of splitting the computation among all cells in a very regular fashion. Generally this computation is repeated

many times and is called a *systolic cycle*. As an example, consider the following cycle executed on a systolic network:

```
SystolicCycle
 {
   Shift_R(Dout,Din);          /* computation done on each cell */
   Dout=F(Din);                /* of the network */
 }
```

The instruction `Shift_R` expresses a synchronous left to right transfer between the processors of the array. Each processor outputs a value and simultaneously inputs another one. The `F()` function represents the computation done on each cell. Suppose now that the network is fed with data stored linearly in an array (T) and that data which are sent have to be processed (P() function). The systolic cycle representing the I/O management process can be expressed as:

```
i=0;
while(i<SIZE_T)
  {
    y=P(T[i]);
    write(LPORT,y);             /* - computation done by the interface */
    T[i]=read(RPORT);           /* - the body of the loop corresponds  */
    i=i+1;                      /*   to a systolic cycle               */
  }
```

A first approach would be to feed the network both with instructions and data. Consider a single control unit which executes sequentially the following program:

```
i=0;
while (i<SIZE_T)
  {
    y=P(T[i]);
    write(LPort,y);
    write(IPort,"Shift_R(Dout,Din)");
    T[i]=read(RPort);
    write(IPort,"INST1_F");             /* instructions representing */
    ...                                 /* F() the function          */
    write(IPort,"INSTn_F");
    i=i+1;
  }
```

We suppose that the execution of an instruction on the array is triggered with the loading of the port `IPort`. The instructions `INST1_F` to `INSTn_F` represent the

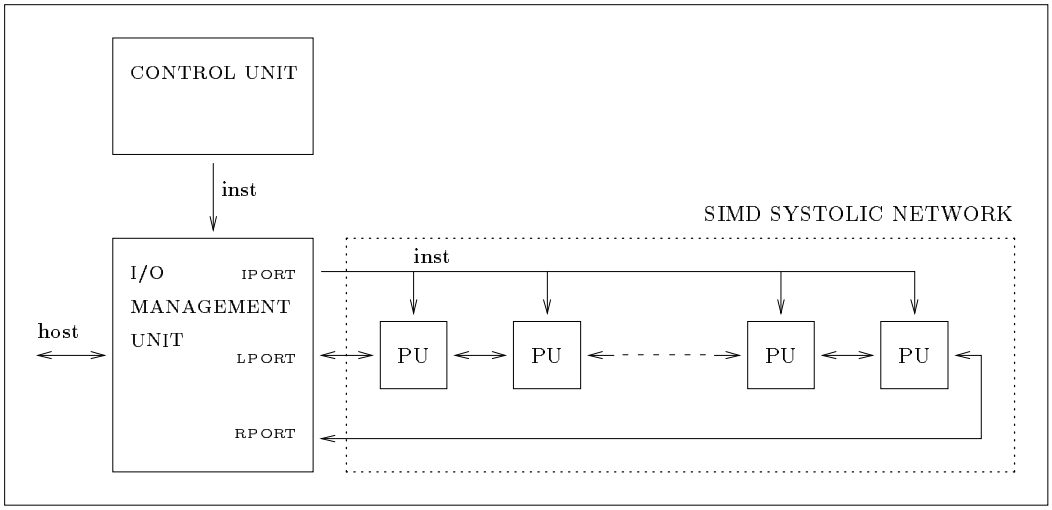


Figure 2: single control unit

instructions executed by the `F()` function. On the above program, the I/O management is done sequentially with the computation process. This implies that when input/output data are processed, the network is idle.

The advantage of this solution is the simple hardware mechanism required for controlling the network. On the other hand the performance of the network depends heavily on the complexity of the I/O management process. In the worst case, if the complexity of both I/O management and computation are similar, the performance is reduced to 50 % of the theoretical maximum [7].

Figure 2 shows the control of the array. The control unit delivers instructions to the I/O management unit (section 2.3). One approach for improving the efficiency of the machine is to parallelize the external I/O management with the computation done on the array. Still using a single control unit (cf figure 3) gives:

```
i=0;
while (i<SIZE_T)
  {
    y=P(T[i]);
    write(LPort,y);              ||    Shift_R(Dout,Din);
    T[i]=read(RPort);
    i=i+1;                       ||    INST1_F;
                                       ...
                                       INSTn_F;
  }
```

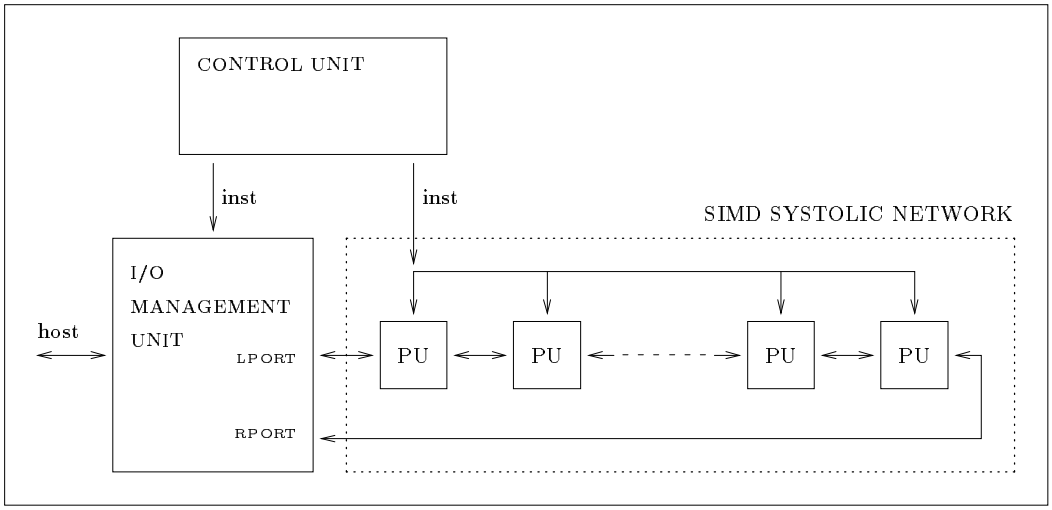


Figure 3: single control unit

When possible, 2 instructions are associated. The notation `||` means a concurrent execution of 2 instructions. For simple cases this can lead to a shorter program, and thus a better efficiency. Generally, difficulties increase when programs become more complex and when they include conditional statements inside the systolic cycle. Suppose, for instance, that the procedure `P` is simply composed of a `while` loop depending on `x` (read from the network) such as:

```
i=0;
while (i<x)
  { ...
    i=i+1;
  }
```

As we cannot predict the number of times the loop will be executed, no instruction can be associated with the instructions inside the body of the loop. In that case, parallelizing the computation process with the I/O management process does not provide good speed-up.

2.2 Two control units

This section presents another solution for parallelizing the computation and the I/O management process. Consider again the example introduced in the previous section. It can be split into 2 separate processes:

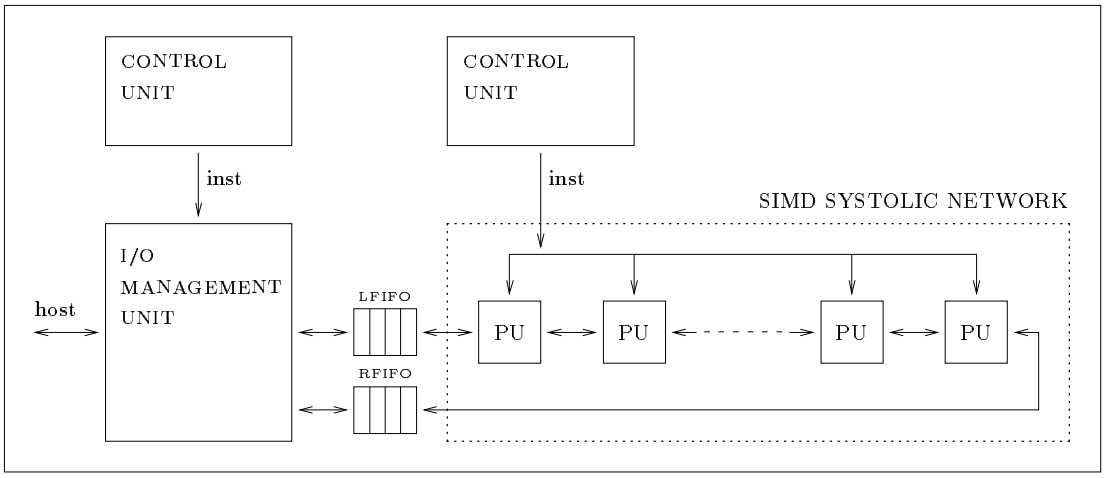


Figure 4: separate control units

```
I/O management process

i=0;
while (i<SIZE_T)
  {
    y=P(T[i]);
    write(LPort,y);
    T[i]=read(RPort);
    i=i+1;
  }
```

```
computation process

i=0;
 while(i<SIZE_T)
   {
     Shift_R(Dout,Din);
     Dout=F(Din);
     i=i+1;
   }
```

2.2.1 Data synchronization

Each process runs its own program independently but data synchronization must be ensured. The computation process can execute the instruction `Shift_R` only if the `LPort` port is provided with data ; similarly, the I/O management process can obtain a data from the network only if the instruction `Shift_R` has been performed. Synchronization between the 2 processes may be achieved by connecting two FIFOs (`LFIFO` and `RFIFO`) as represented on figure 4.

Before generating a `Shift_R` instruction to the network, the computation process has to check the status of the 2 FIFOs (is `LFIFO` empty or `RFIFO` full?). In the same way, the I/O management process checks before sending data if the `LFIFO` is not

full and before reading data if the `RFIFO` is not empty. The two processes can be rewriten as:

```
I/O management process

i=0;
while (i<SIZE_T)
  {
    y=P(T[i]);
    write(LFIFO,y);
    T[i]=read(RFIFO);
    i=i+1;
  }
```

```
computation process

i=0;
 while(i<SIZE_T)
   {
     Shift_R(Dout,Din);
     Dout=F(Din);
     i=i+1;
   }
```

The `read` and `write` instructions of the I/O management process are blocking instructions. For example, if the FIFO is full, a `write` instruction will wait until it is not. On the same way, the `Shift_R(Dout,Din)` instruction can be executed only if the `RFIFO` is not full **and** the `LFIFO` is not empty. If this condition is false, the process will wait until it is true.

2.2.2 Control synchronization

Instructions for controlling the loop are present in the two processes. A first drawback is that the same computation is done twice. A second drawback is that sometimes the loop condition is data-dependent: in such a case, the I/O management process has to communicate essential information to the other process.

To coordinate the two processes, one might compute the loop control decision in one controller and synchronize the other controller on that decision. The synchronization can be achieved using the two functions `set_sync()` and `is_sync()` as follows:

```
I/O management process

i=0;
while (set_sync(i<SIZE_T))
  {
    y=P(T[i]);
    write(LFIFO,y);
    T[i]=read(RFIFO);
    i=i+1;
  }
```

```
computation process


while (is_sync())
    {
       Shift_R(Dout,Din);
       Dout=F(Din);
    }
```

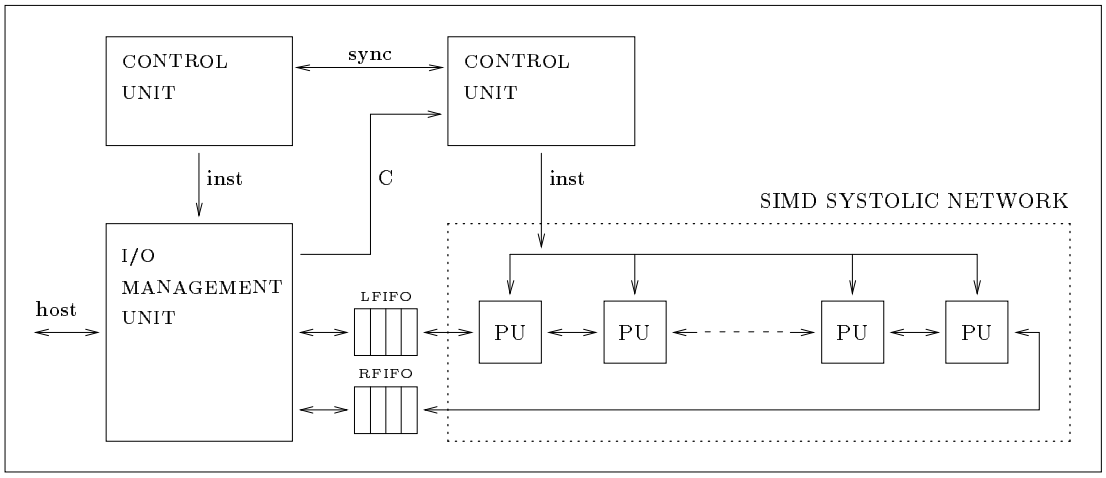


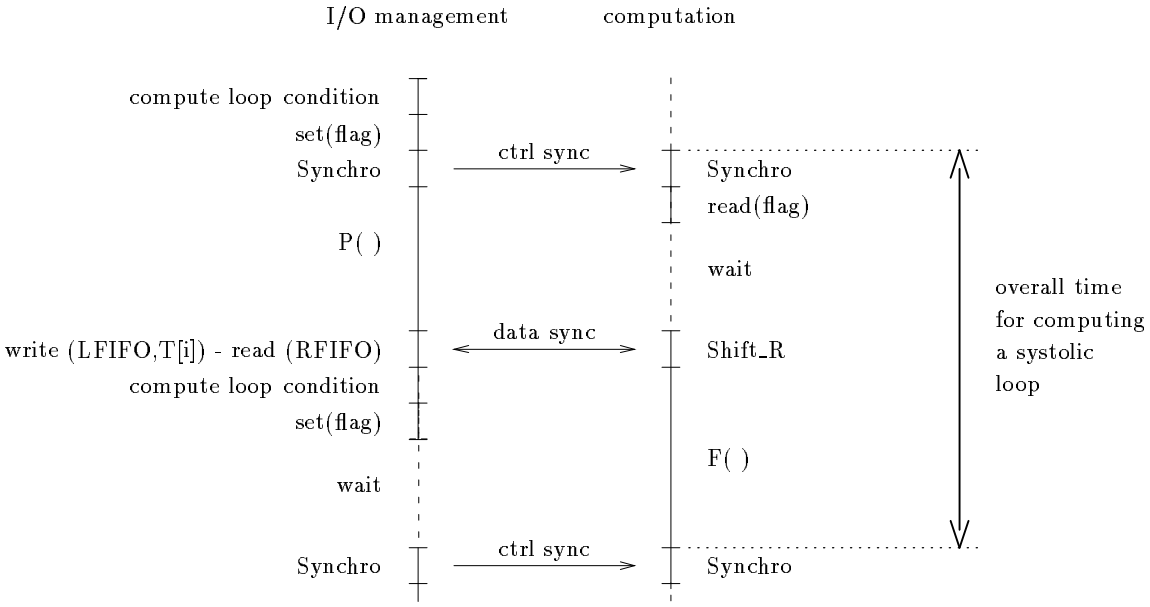
Figure 5: RDV-control architecture

The next two sections describe two different hardware implementations based on this synchronization scheme. The first one use a "rendez-vous" (RDV) mechanisme while the second one use a "fifo" mechanisme.

RDV synchronization

The "rendez-vous" synchronization works as follows: the I/O management process calculates the loop condition and sets a flag accordingly ; then the two processes synchronize each other (the first process which has reached the synchronization point waits the other) ; finally, based on the value of the condition, the loop is (re)executed or not in both processes.

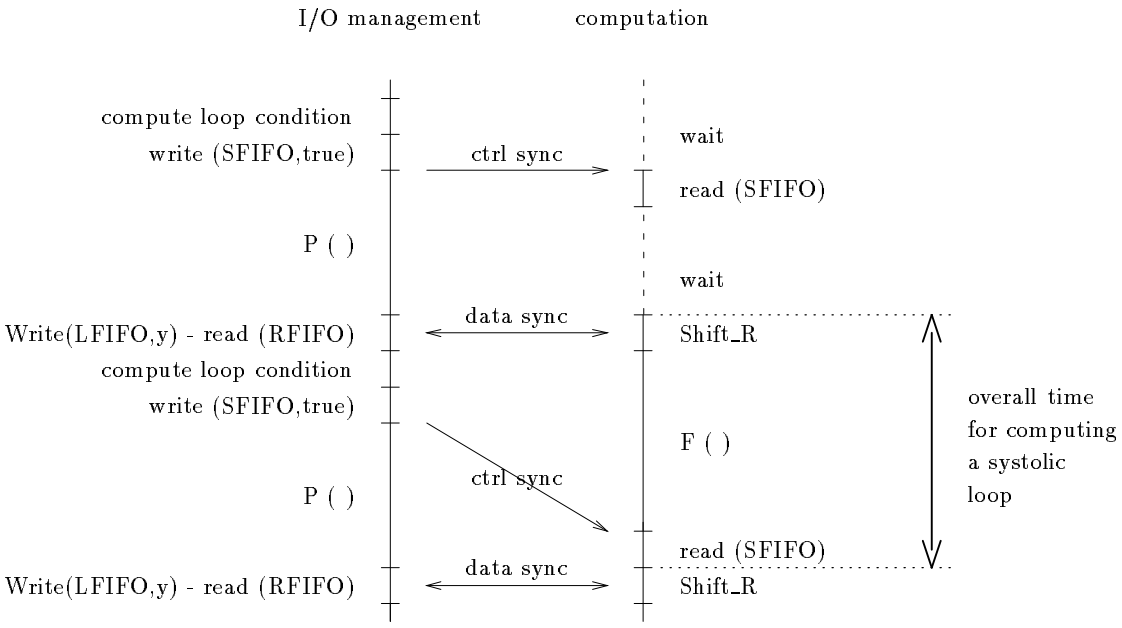
This type of synchronization is efficient if the body of the loops overlapped well. There are cases where it does not happen, as in the above example. This leads to the following timing diagram where the overall computation time is nearly equal to the sequential computation time of the two processes:



FIFO synchronization

To remedy the RDV synchronization problem, we introduce a boolean FIFO (called `SFIFO`) which, according to its value, allows the `while` statement to execute the loop or not.

In that case, the `set_sync()` function evaluates the condition and pushes the result in the `SFIFO`. Since they communicate only by fifos, the two processes are now completely independent. They can even be driven by different clocks. The figure 6 indicates the hardware implementation for supporting the two processes (one boolean fifo is added). The timing diagram associated with the execution of the two processes becomes:



This synchronization mechanisme provides the capability for the I/O management process to overlap its operation with that of the computation process. As defined by [7], such a machine is called an *overlapped* SIMD machine (OSIMD) and employs the concurrent power of N+1 processing units as compared to nonoverlapped SIMD machines that employ the computing power of N processing units. If a good balance can be achieved between the two processes, an OSIMD machine with N+1 processors may achieve a speed-up factor of two over a conventional SIMD machine with N processing units.

2.3 The I/O management unit

Until now, the I/O management unit has been expected to perform elaborate computation tasks. No assumptions nor restrictions have been made. We now detail the role and the features of this unit.

The I/O management unit receives instructions from one controller for performing the following:

- generating data for the network,
- receiving data from the network,

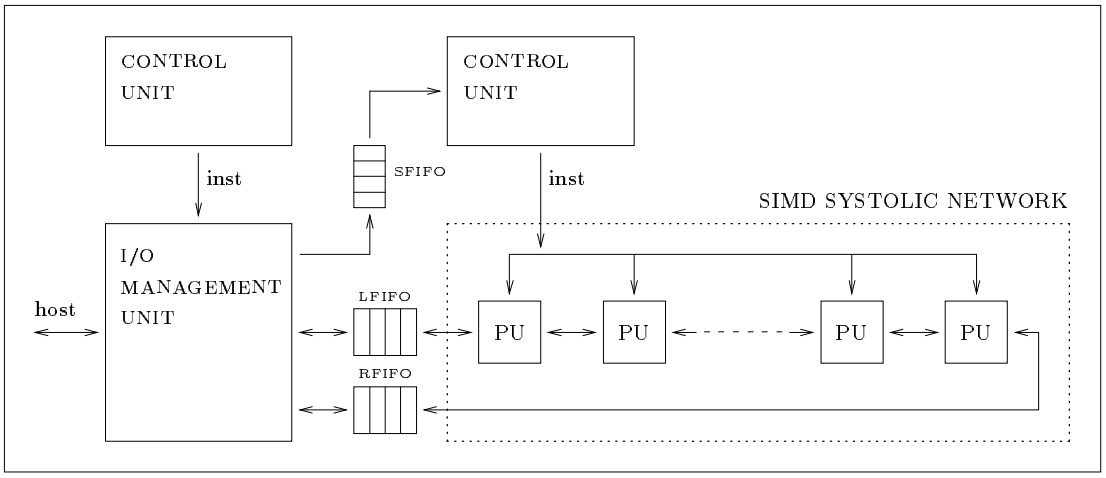


Figure 6: FIFO-control architecture

- communicating with the host computer.

One of the main characteristics of systolic arrays is the large amount of data they consume. Two kinds of data may be distinguished when feeding a network: initialization data and application data.

Generating initialization data requires arithmetic and logic operations. Application data come directly from the host and do not need to be processed. At first glance, a simple ALU could be used. However, when one has to deal with algorithm partitionning, I/O data management may become complex and generally requires a large memory (as well as addressing facilities) to store intermediate results.

Furthermore, the data collected from the network are not sent directly to the host. Most of the time, they are sorted to collect only significant results.

We believe that an I/O management unit with elaborate computation capabilities and a large memory with addressing facilities is the key to input/output efficiency [3] [2]. One of the major problems of specific hardware remains the programming flexibility: assembly language provides good performance but requires too much programming effort and thus limits the number of programmers capable of implementing new applications. On the other hand, a high level language may be easily used by many programmers but may restrict the performance. The next section addresses the problem of programming such a machine. We present a language tailored to our machine model which simplifies programming while maintaining efficiency.

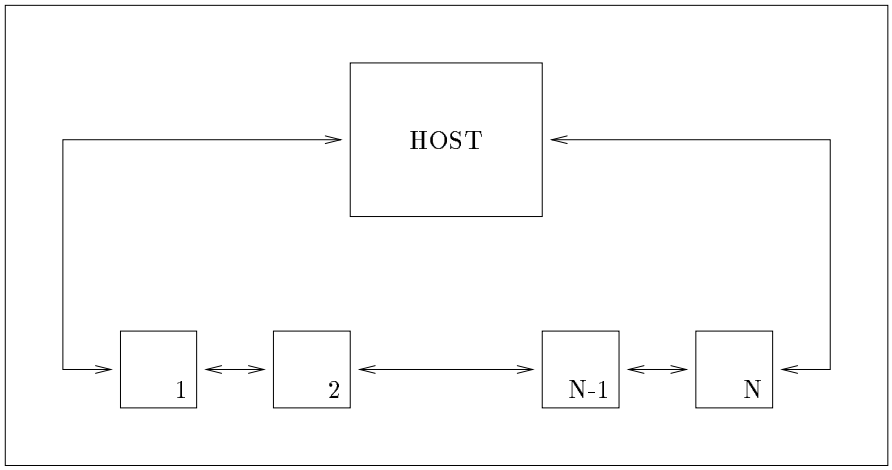


Figure 7: the programmer view

3 Programming the machine

Parallelizing an algorithm on the SIMD systolic machine presented in the previous section requires three distinct programs: the host program for managing the overall application, the I/O management process and the computation process as described previously. We now extend the role of the I/O management process for communication with the host workstation.

Our experiments have shown that writing three programs which communicate together is not an easy task and requires non-negligible efforts from programmers: much time is often spent in debugging and bugs are generally difficult to localize on a parallel machine. The C-stolic language has been designed to avoid such difficulties.

The user views the system as a programmable accelerator connected directly to a general purpose host workstation. This accelerator appears as a SIMD network composed of a linear array of identical, conventional processors that communicate synchronously with their nearest neighbors. Traditionally only the two end-cells are linked to the host (see figure 7). Note that the *interface* is hidden from the programmer.

The main application is written using the C language. When intensive calculation is requested, one has to write (in a different file) a specific function using the C-stolic language. When this function is called from the main application, the systolic array is activated and the computation is performed on the network. The input

parameters of the function indicate the data which must be processed, and the output parameters the result of the computation.

3.1 The C-stolic language

The C-stolic language has been designed to benefit from the architectural features of two synchronized controllers. We wanted this new language to be independent of the processor cell and to provide a simple way of programming data movements across a systolic architecture. These goals led to the choice of an explicit form of parallel programming to reach maximal efficiency. On the other hand, as the control of a systolic computer is not radically different from that of a conventional computer, it would be wasted effort and a source of confusion to design a completely new language. So we chose to extend the C language, a well known and efficient compiled language. The extensions provide specific ways of expressing algorithms for systolic computers, without imposing a programming style different from that used for the host. The next subsections details the specific extensions introduced in the C-stolic language.

3.1.1 Variable declaration

The programmer of the systolic machine needs a way to mark the difference between host variables, and objects which are located on the systolic array. The C-stolic language defines a new storage class specifier: `systolic`.

We mention that storage class specifiers serve, in the C language, to define variables, in that they cause an appropriate type of storage to be reserved. A `systolic` object is allocated in each cell of the systolic network. The `static` class (which is the default class) and `auto` class specify host objects. This mechanism provides the programmer with an explicit way to map the algorithm onto the systolic array, that is consistent with the C language.

3.1.2 Parallel execution

The C-stolic language makes the assumption of a purely SIMD control flow in the model. As a result, variables used for controlling the execution sequence, such as loop counters and boolean conditions, should exist only once on the host computer (`static` or `auto` storage class)

Statements operating on cell variables (`systolic` storage class) perform parallel execution in a *data parallel* fashion [4]. Mixed expressions, which attempt to combine

host and network variables, are rejected by the compiler according to the *locality principle*.

Because the cells do not have independent control, `while` loops, `if` tests and computed loop bounds on the cells are not allowed. This is a restriction on the class of typical programs executable on SIMD machine. However, a specific conditional cell instruction offers the possibility of reducing such constraints by providing some autonomy to the processor. It allows the programmer to deal with boundary conditions without requiring one sequencer per cell [1]. The conditional expression of C-stolic supports this local test. The expression has the same syntax as its C counterpart. As an example, one can write:

```
systolic int k,x,y;
...
k = (x>y) ? x : y;
```

to express a conditional assignment representing the computation of a maximum value in systolic cells.

3.1.3 Systolic communication

In systolic architectures, data transfers between processors are very important. Special care is devoted to this I/O mechanism in the C-stolic language. New operators match the hardware architecture and express the tight coupling between neighboring cells.

The data transfer can be viewed as a global shift operating on `systolic` variables and `static` variables implied in the communication sequence. In C-stolic, this overall operation is expressed by the left (`=<`) and right (`=>`) systolic assignment. An optional systolic assignment parameter handles the boundary conditions, i.e. assigns the data which is input to the array. As an example, the figure 3.1.3 represents respectively the transfer b =< a : A[i] (figure A) and the transfer a : A[i] => b : B[j] (figure B).

3.2 Programming example

In order to illustrate the principles of C-stolic, a simple example of data convolution is taken. For sake of simplicity, we suppose that the window size is equal to the number of cells available (i.e predefined variable N_CELLS). The C-stolic source code is:

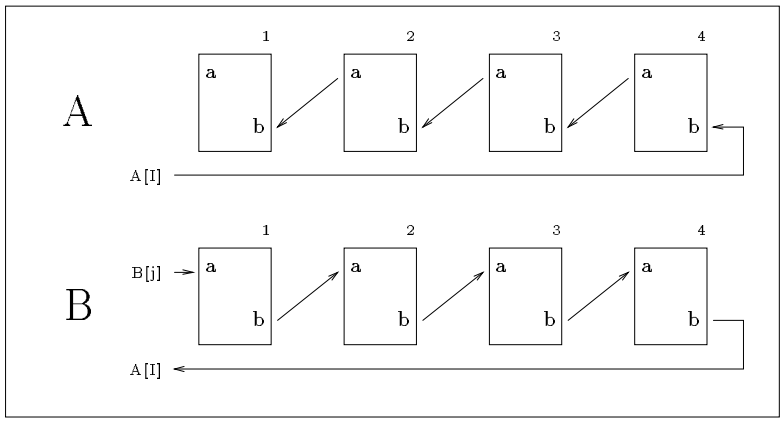


Figure 8: transfer on C-stolic

```
 1  #include "c1D.h"
 2  convol1D(A,X,Y)
 3  int A[N_CELLS],X[N_SIZE],Y[N_SIZE];     /* N_CELLS coefficients */
 4    {
 5      systolic int a,x,y1,y2,r=0;
 6      int I;
 7      I=0;
 8      while (I<N_CELLS)                   /* coef initializations */
 9        { a =< a : A[I]; I=I+1; }
10      I=0;
11      while (I<N_CELLS) {                       /* start-up phase */
13          y1 => r : 0;
14          r = y2;
15          x => x : X[i];
16          y2 = y1 + a * x;
17          I=I+1;
18      }
19      I=N_CELLS;
20      while(I<N_SIZE) {                         /* cruising phase */
22          y1 : Y[I-N_CELLS] => r : 0;
23          r = y2;
24          x => x : X[i];
25          y2  = y1 + a * x;
26          I=I+1;
27      }
28      I=0;
29      while(I<N_CELLS) {                       /* shut-down phase */
```

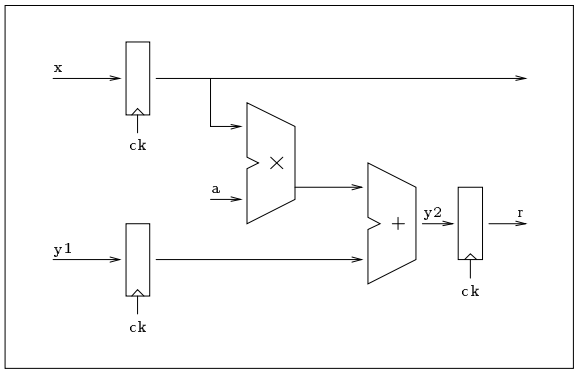


Figure 9: convolution product

```
31          y1 : Y[N_SIZE-N_CELLS+I] => r;
32          r = y2;
33          x => x;
34          y2 = y1 + a * x;
35          I=I+1;
36       }
37     }
```

This program represents the computation done on a global systolic system where I/O operations are taken into consideration. The figure 9 details the calculation done by each cell. The main application will simply perform a call to the `convol1D()` procedure when a one dimensional convolution is required.

Lines 5 and 6 are variable declarations. The objects `a`, `x`, `y1`, `y2` and `r` are variables which exist in each cell of the systolic array and are declared as `systolic`. On the other hand, the index `I`, is external to the array and is just declared as integer.

The first `for` loop initializes the coefficients inside each cell. The next three loops have the same structure. They differ on their transfer instructions due to the network latency: on the first phase, no results are collected and on the last phase, the network becomes empty and no data are sent. Note that this program could have been written in a more concise way using the conditional statement notation.

From this single program, the C-stolic compiler generates two processes, one for performing the computation on the array and one for managing external input/output communications. C code is first generated before being compiled on the target architecture. This intermediate step permits implementation of C-stolic on different machines including sequential machines for simulation and debugging purpose [12].

```
/* I/O Management Process */

I=0;
while(set_sync(I<N_CELLS)) {
  { io_send(RPort,A[I]);
    I=I+1; }
I=0;
while(set_sync(I<N_CELLS))
  { io_send(LPort,0);
    io_send(LPort,X[i]);
    I=I+1; }

I=N_CELLS;
while(set_sync(I<N_SIZE))
  { io_send(LPort,0);
    io_rcv (RPort,Y[I-N_CELLS]);
    io_send(LPort,X[i]);
    I=I+1; }
I=0;
while(set_sync(I<N_CELLS))
  { io_rcv (RPort,Y[N_SIZE-N_CELLS+I]);
    I=I+1; }
```

```
/* Computation Process */

while (is_sync())
  { io_shift_RL(a,dummy,a,RPort); }

while (is_sync())
  { io_shift_LR(y1,dummy,r,LPort);
    r = y2;
    io_shift_LR(x,dummy,x,LPort);
    y2 = y1 + a * x; }

while (is_sync())
  { io_shift_LR(y1,RPort,r,LPort);
    r = y2;
    io_shift_LR(x,dummy,x,LPort);
    y2 = y1 + a * x; }

while (is_sync())
  { io_shift_LR(y1,RPort,r,dummy);
    r = y2;
    io_shift_LR(x,dummy,x,dummy);
    y2 = y1 + a * x; }
```

The `set_sync()` function sets the boolean fifo according to the evaluation of the condition given in parameter and the `is_sync()` function corresponds to the `read(SFIFO)` expression introduced in the previous section. The `io_shift_RL()` and `io_shift_LR()` functions determine the direction of the data transfer and the boundary data management. As an example, `io_shift_RL(a,dummy,a,RPort)` represents the C-stolic `a =< a : A[I]` statement.

With a large array of data, the second `for` loop will dominate the total execution time since the other loops participate only for initialization and termination The C-stolic instruction `y1 : Y[I-N_CELLS] => r : 0;` implies a synchronization point due to data exchange: the faster process will have to wait for the slower. Experiments have shown that generally the I/O management process is faster than the computation process and, consequently, provides an optimal use of the systolic array.

4 Experiments

In order to study the behavior of the OSIMD architecture, we analyzed a set of programs written in C-stolic. These programs come from real applications and have not been tuned nor modified to fit a particular architecture. The list of programs is as follows:

- C1D: 1 D convolution,
- C2D: 2 D convolution,
- LEV: Levenshtein algorithm,
- DCT: Discrete Cosine Transform (8x8 block),
- KNA: Knapsack problem,
- MAT: Matrix product,
- SCA: Scan of biologic sequence data base,
- SRT: Sorting.

From these C-stolic programs, code was generated for the three distinct target architectures presented in section 2:

- SEQ: one control-unit (cf figure 2),
- RDV: two control-units with RDV synchronization (cf figure 5),
- FIFO: two control-units with FIFO synchronization (cf figure 6).

The following table summarizes the results. It gives the speed-up obtained by the two control-unit architectures (RDV and FIFO) versus the one control-unit architecture (SEQ):

	C1D	C2D	LEV	DCT	KNA	MAT	SCA	SRT	average
RDV	1.34	1.43	1.34	1.15	1.54	1.49	1.65	1.45	1.42
FIFO	1.34	1.66	1.57	1.29	1.64	1.73	1.67	1.70	1.57

A first observation is that in every cases we get a speed-up greater than one. This means that in spite of the overhead introduced by the synchronization, it is worthwhile to add a second control-unit to increase the performance of a systolic array.

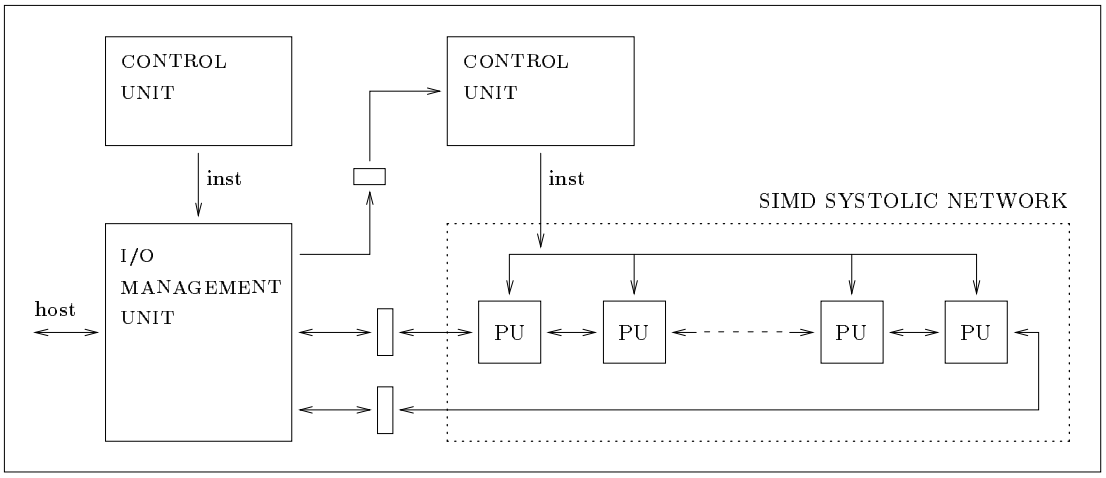


Figure 10: register-control architecture

A second observation is that the FIFO architecture is better than the RDV architecture, but the gap between the two is not very important. We may suppose that, by their nature, the processes are more or less balanced.

Another point has been the study of the optimal size for the FIFOs of the FIFO architecture. The programs were emulated with different sizes of FIFO; we notice that the speed-ups were not significantly modified (in the best case, the speed-up was increased of 1% !). This result can be explained by the systolic nature of the algorithms: there always exists a main loop, *the systolic cycle,* in which at least one datum is sent to the array and at least one datum is collected from the array. This implies that the I/O data management process cannot get ahead of the calculation process since it must wait the delivery of one or more data produced every systolic cycle.

A very reasonable size for the FIFOs is one since larger size does not produce significant improvement. In other words, the FIFOs can be advantageously replaced by simple registers (with flags associated with the register states) as shown in figure 10.

5 Conclusion

A model for overlapped I/O management operations and computation on a SIMD linear array has been presented. This model is based on two synchronized sequencers,

one in charge of the broadcast of the instructions to the network and the other with sending and receiving data.

From a programming point of view, the model is supported by the C-stolic language. This language allows the programmer to describe explicitly the data transfer between the processors of the network and the interface. The C-stolic compiler generates code for both the controller which provides instructions for the network, and for the controller which manages I/O operations.

If the work load between the two processes is perfectly balanced and overlapped, the computation can achieve a speed-up factor of two over a SIMD machine that does not support any overlapped operation. Such speed-ups are obviously program dependent and the limit bound is difficult to achieve [7].

Experiments have shown that a two unit architecture has an average speed-up of 1.5 with respect to a single unit architecture. Furthermore, a synchronization mechanism based on simple registers provides quasi-similar performance compared to a more sophisticated one based on FIFOs.

References

[1] P. Frison, E. Gautrin, D. Lavenier, and JL. Scharbarg. Designing Specific Systolic Array with the API15C chip. In S.Y Kung E. Swartzlander J.A.B Fortes K.W. Przytula, editor, *ASAP 90*, pages 505–517, IEEE Computer Society Press, Sep 1990.

[2] P. Frison and D. Lavenier. Experience in the Design of Parallel Processor Arrays. In *International Workshop on Algorithms and Parallel VLSI Architectures II*, pages 233–242, Elsevier Science Publishers B.V., Jun 1991.

[3] P. Frison, D. Lavenier, H. Leverge, and P. Quinton. A VLSI Programmable Systolic Architecture. In *International Conference on Systolic Array*, IEEE Computer Society Press, Jun 1989.

[4] W. Hillis and G. Steele. Data Parallel Algorithms. *ACM*, 29(12):1170–1183, dec 1986.

[5] Richard Paul Hughey. *Programmable Systolic Arrays*. PhD thesis, Brown University, may 1991.

[6] D. Juvin, J.L. Basille, H. Essafi, and J.Y. Latil. Sympathi2 : a 1.5d processor array for image applications. In *EUSIPCO Signal Processing IV : theories and application*, pages 311–314, North Holland, 1988.

[7] S. Kim, M.A. Nichols, and H.J. Siegel. Modeling Overlapped Operation between the Control Unit and Processing Elements in an SIMD Machine. *Journal of Parallel and Distributed Computing*, 12:329–342, 1992.

[8] H.T. Kung. Why Systolic Architectures? *Computer*, 15(1):37–46, 1982.

[9] Dominique Lavenier. *MicMacs : un réseau systolique linéaire programmable pour le traitement des chaines de caractères.* PhD thesis, Université de Rennes 1, juin 1989.

[10] Annaratone M., Arnould E., Gross T., Kung H. T., Lam M., Menzilcioglu O., and Webb J. A. The Warp Computer: Architecture, Implementation, and Performance. *IEEE Transactions on Computer*, C-36(12):1523–1538, dec 1987.

[11] F. Raimbault and D. Lavenier. Relacs for Systolic Programming. In *ASAP-93*, pages 132–135, IEEE Computer Society Press, Oct 1993.

[12] Frederic Raimbault. *Etude et réalisation d'un environnement de simulation parallèle pour les algorithmes systoliques.* PhD thesis, Université de Rennes 1, jan 1994.

[13] Borkar S., Cohn R., Cox G., Gleason S., Gross T., Kung H.T., Lam M., Moore B., Peterson C., Pieper J., Rankind L., Tseng P. S., Sutton J., Urbanski J., and Webb J. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of Supercomputing '88*, pages 330–339, IEEE Computer Society and ACM SIGARCH, nov 1988.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399