A Transformational Model of VLSI Systolic Design

Monica S. Lam, Carnegie-Mellon University
Jack Mostow, Information Sciences Institute, University of Southern California

Using transformations that formalize the systolic designer's "bag of tricks," a prototype system converts nested-loop algorithms into efficient functional-level systolic designs.

Systolic arrays have been proposed as a cost-effective solution to many computation-intensive problems. They consist of simple cells operating synchronously, each communicating only with nearby cells. Their applications range from numeric tasks, such as signal and image processing and matrix arithmetic, to symbolic tasks, such as searching and sorting, graph algorithms, and relational databases.

The design process for systolic arrays is modeled here as a series of transformations, expressed in a language devised for clearly and concisely describing systolic arrays. A prototype program called Sys (for systolic design system) accepts a software algorithm and some advice and applies a series of transformations to produce a functional-level circuit description of a systolic design.

Systolic arrays

Systolic architectures are typically large, regular arrays of processor elements (Figure 1). Such an architecture is called systolic because data is pumped steadily through the array of cells, much like blood in the body. Data items, including inputs and partial and completed results, flow through the structure synchronously in a fixed, regular pattern; all operations involving an item are applied to it as it passes through. This method of computation eliminates the need to retrieve the item from external memory every time it is used, a process that typifies von Neumann machines. Consequently, a systolic array can be expanded to provide the increased capacity required by a computation-intensive application without imposing a corresponding increase in external memory bandwidth. This property gives systolic architectures a major advantage over traditional architectures, which are limited by the von Neumann bottleneck.

Besides their low external memory bandwidth requirements, systolic architectures offer other advantages as well. The simplicity of systolic VLSI designs is especially important for special-purpose applications, where design cost must be amortized over a small production volume. A systolic array is typically made up of a few simple cell types and is therefore cheaper to design than a circuit containing a variety of complex cells. Moreover, the regular pattern of local interconnection between cells simplifies the layout problem.

Unlike architectures that broadcast data to many points, systolic architecture can easily be scaled up to handle large problems. Their local interconnections schemes avoid the clock skew that arises when data is broadcast over paths of differing lengths. Also, the low fanout in a systolic array allows the signal drivers to be independent of the number of cells in the array. Thus the size of a systolic array can be increased without altering other design parameters.

These properties—low external memory bandwidth, simplicity, regularity, and local communication—make systolic architectures especially well-suited to implementation in VLSI. Systolic design is providing new or improved hardware solutions to many computation-intensive problems. Two solutions to the simple problem of polynomial evaluation will serve as examples of systolic arrays.

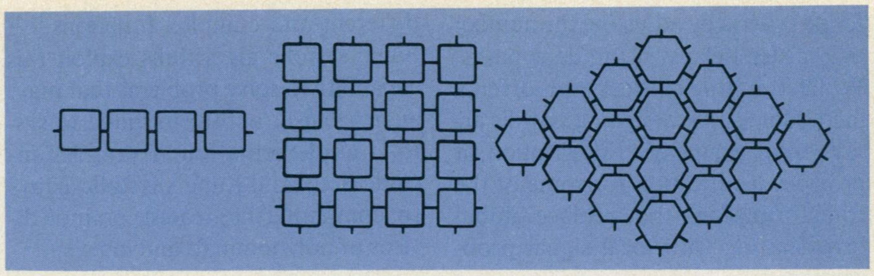


Figure 1. Some common configurations of systolic arrays.

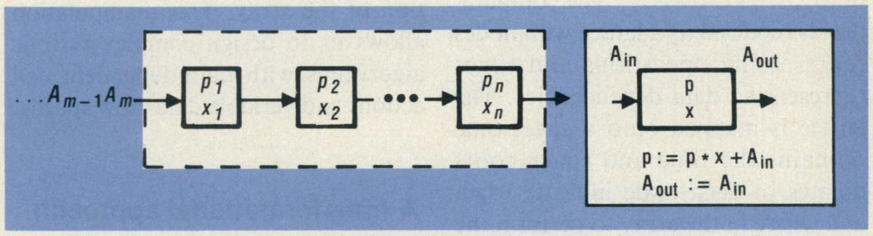


Figure 2. Poly-I: Hold x_i in register x; accumulate p_i in register p; pass A_j.

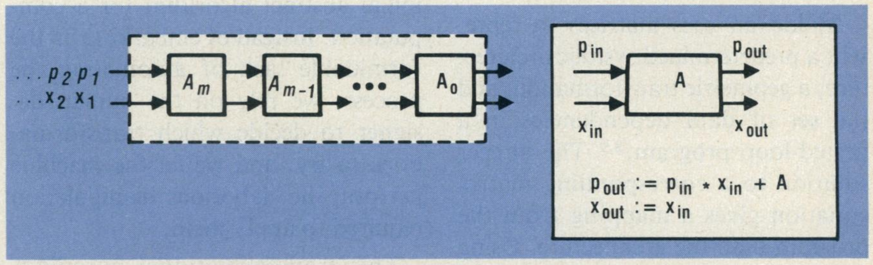


Figure 3. Poly-II: Hold A_j in register A, pass x_i and p_i.

Two systolic designs for polynomial evaluation. Suppose we have the following polynomial:

$$P(x)=A_m x^m + A_{m-1}x^{m-1} + \ldots + A_0$$

We wish to evaluate $P(x_i)$ at points x_i, $1 \leq i \leq n$. By Horner's rule, the polynomial can be reformulated from a sum of powers into an alternating sequence of multiplications and additions:

$$P(x)=(((A_m x + A_{m-1})x + \ldots + A_1)\,x + A_0$$

The value of $P(x_i)$ for each x_i, p_i, is computed by an algorithm whose inner loop is

```
for i from 1 to n do
    for j from m to 0 do
        $p_i := p_i * x_i + A_j$;
```

Here $p_1, \ldots, p_n$ are initialized to 0.

One systolic implementation is shown in Figure 2. In this design, referred to as Poly-I, there is one cell for each x_i. Each cell holds its x_i value in one register, x, and accumulates the value of the polynomial in another. The coefficients $A_m, \ldots, A_0$ are passed from one cell to another so that each cell sees the same sequence of coefficients. On each clock cycle, every cell multiplies its partial sum by the value stored in its x register, adds the coefficient it receives, and stores the result into its p register. When all the coefficients have flowed through, the results will be in the p registers.

An alternative systolic implementation is shown in Figure 3. In this design, Poly-II, the coefficients are held in cells through which the x_i and p_i data flow. On every clock cycle, each cell inputs x_i and p_i, multiplies them, adds its stored coefficient, and outputs the result as p_{out} while passing on x_i unchanged. The result $P(x_i)$ appears at the output of the rightmost cell m clock cycles after x_i is input to the leftmost cell.

Methodologies for systolic design. Although their characteristics (regular data flow, local communication, and simple replicated components) make systolic arrays easy to implement in VLSI, mapping a computation onto an array with these characteristics can be difficult. Several attempts have been made to simplify various aspects of this design process.

An appoach adopted from z-transforms in signal processing was first suggested by Cohen[1] and later elaborated by Johnsson and Cohen[2] and Weiser and Davis.[3] Delay registers are modeled with a special mathematical operator Z, which is defined by

$$z[x(i)]=x(i-1)$$

Mathematical formulas can be manipulated to obtain different expressions that correspond to different computational networks. The drawback in developing a methodology based on this representation is that the notation is unwieldy for complex computations.

Leiserson and Saxe[4] developed a general theory that allows us to optimize a synchronous circuit under differing criteria by adjusting the number of register delays in the data paths. While this theory does not offer a methodology for designing systolic arrays from a high-level description, it provides justification for some of the transformations that replace global broadcasting with local signal propagation.

A methodology based on geometric transformations has been proposed for mapping nested-loop algorithms onto systolic arrays.[5-7] The computation is modeled as a lattice with nodes representing operations and edges representing data dependencies. The lattice is mapped onto a space-time domain; the time and space coordinates of each node indicate when and where in the array to perform the operation. We can derive different systolic designs by applying geometric transformations to the lattice.

Moldovan uses matrices to represent a predetermined systolic architecture, a geometric transformation, and the set of data dependencies in a nested-loop program.[8,9] The integer solution to a corresponding matrix equation gives a mapping from the program onto the architecture. Using a similar approach, Quinton[10] demonstrates a constructive method to generate all possible systolic designs for a certain class of recurrence equation under some mapping restrictions. This approach is very powerful, but works only for highly regular recurrences.

The approaches described above abstract a computation in terms of its data dependencies; they assume that the computation has already been decomposed into operations corresponding to the behavior of individual cells. This assumption is impractical for complex computations where the design process may depend on details of internal cell behavior; for example, it prevents the use of certain design transformations that redefine this behavior.

The behavior of the cells in a systolic array can be quite complex. In particular several recent implementations of systolic arrays have programmable cells that can perform many different and complex functions.[11,12] Some systolic algorithms exploit this capability to solve problems that may, at first blush, appear unsuited to systolic implementation. Examples include Brent and Kung's systolic arrays for computing the greatest common divisor of polynominals and integers.[13]

Our methodology models cell behavior explicitly. We can manipulate the cell programs at the same time we design the data flow and interconnection of the array. This manipulation allows us to design complex systolic algorithms with data-dependent cell actions and variable data-flow speeds.

A transformational approach

We model the design process as a series of transformations applied to an initial abstract algorithm for a computation. Instead of embarking on the formidable task of automating the process, we rely on the human designer to decide which transformations to try, and we let the machine perform the laborious manipulation required to apply them.

This transformational paradigm was first suggested as a solution to the high cost of software maintenance.[14] By using a computer-aided transformational process to map a high-level specification into an implementation, we can capture all the design decisions in the machine. Changes in the design strategy or in the specification itself can be implemented simply by editing the recorded history of design decisions and replaying the modified history.[15]

Our motivation for using the transformational paradigm is to help people create systolic designs. The "bag of tricks" produced by research in systolic design can be encapsulated as machine-applicable transformations. Moreover, the transformational approach offers a solution to the verification problem. A systolic design is typically remote from the abstract computation it implements, so its verification can be quite complicated. However, if the design has been derived by applying a sequence of transformations known to preserve correctness, the derivation itself serves as a constructive proof of correctness with respect to the initial specification.

Transformational model

Our model of the systolic design process can be summarized as

```
Begin
Perform software transformations;
For block from inner to outer
Do  begin
    Allocate hardware units;
    Schedule the computation;
    Apply optimizing transformations
    end;
end;
```

Part of this process has been implemented in a prototype system, Sys. The process starts with an abstract algorithm and designs a functional-level systolic circuit. The design describes a set of functional units, their interconnections, and the input and output data streams.

The algorithm must first undergo high-level transformations to be prepared for systolic implementation. For example, Horner's rule was used to transform polynomial evaluation into a series of multiply-and-add operations, which can be performed by the same type of cell. Some of these transformations are similar to those that adapt sequential programs for execution on general multiprocessor architectures.[16,17] In systolic designs, further consideration must be given to the unique characteristic of data flowing through a regular interconnect rather than residing in a global memory. Failure to anticipate such concerns before translating the algorithm into a lower level design can force the designer to backtrack and revise the high-level algorithm. While the transformational paradigm cannot substitute for design insight in selecting a good algorithm initially, it can facilitate design evolution by assisting the transformation of successive versions of an algorithm. Sys assumes that the initial algorithm has undergone the necessary software transformations.

An algorithm ready for systolic implementation typically consists of highly repetitive computations, expressed in terms of nested loops and begin-end blocks. Sys knows how to map such constructs onto hardware; some transformations for mapping other constructs are described elsewhere.[18]

Sys takes the algorithm and goes through a bottom-up process to design a systolic array. For each loop, it allocates hardware, schedules the computation, and optimizes.

The allocation phase generates a description of the necessary primitive functional units and allocates them to operations in the algorithm. The user guides this phase by annotating each begin-end block and loop. The annotation *in place* tells Sys to use the same hardware for all the statements in the block or iterations of the loop, while the annotation *in parallel* tells Sys to use separate hardware for each one.

The scheduling phase decides when each operation can safely be performed.

The optimization phase makes the design truly systolic by inserting registers, adding data connections, and adjusting the schedule. The user can guide this phase by selecting which optimizing transformations to apply.

Description of systolic circuits. To facilitate machine transformations, we divide the description of a systolic circuit into its *structure* and its *driver*. The structure describes the hardware cells and how they interconnect; the driver describes the format of the data streams to and from the circuit.

Structure. The notation for describing the structure is similar to a programming language. A hardware module corresponds to a procedure. The description is hierarchical; composite modules are made up of submodules, which, can themselves, be composite modules. The submodules are analogous to local procedures. Local registers correspond to "own" variables, which retain their states between calls.

A module's interface to the external world consists of input and output ports. A scalar port inputs or outputs a single value at a time, while an array port transmits a vector of values simultaneously.

```
Module Poly-II;
    In-Port x_in {Sequence[i from 1 to n] of x_i};
    In-Port p_in {Sequence[i from 1 to n] of p_i};
    Out-Port p_out {Sequence[i from 1 to n delayed m] of p_i};

    Module Multiply-Add[j∈0..m];
      In-Port x_in, p_in;
      Out-Port x_out, p_out;
      Register A {preloaded with A_j};

      p_out := p_in * x_in + A;
      x_out := x_in;
    end Multiply-Add;

    p_out := Multiply-Add[0]·p_out;
    Parallel begin
      Multiply-Add[m](x_in, p_in);
      Multiply-Add[j](Multiply-Add[j+1]·x_out, Multiply-Add[j+1] ·p_out)
        for 0≤j≤m-1;
    end;
  end Poly-II;
```

Figure 4. Poly-II description.

Consider, for example, the specification of the Poly-II module, as shown in Figure 4. (The italicized information is the driver specification.) This figure shows that the Poly-II module has two input ports, x_{in} and p_{in}; an output port, p_{out}; and an array of $m+1$ primitive Multiply-Add modules, each of which has two input ports, x_{in} and p_{in}, two output ports x_{out} and p_{out}, and one local register A.

The body of the module description specifies the behavior of the module in each clock cycle. For a primitive module, the implementation of its behavior is not important to the aspects of systolic design addressed here. Therefore, the computation performed in each clock cycle is defined abstractly by an Algol-like notation relating the new values (outputs and new register contents) to the inputs and old register contents. We assume a two-phase clocking scheme such that, in phase 1, a module reads from its registers and input ports and, in phase 2, it updates its registers and writes into its output ports. Each call to the module corresponds to a clock cycle.

In the example of Poly-II, Multiply-Add is a primitive module. In phase 1 of each clock cycle or procedure call, $p_{in} * x_{in} + A$ is computed and the result is written into p_{out} in phase 2. The value of x_{in} is also read in phase 1 and written into x_{out} in phase 2. All output ports retain their values until they are written into again in the second phase of the next clock cycle when the next procedure call occurs.

A composite module is an encapsulation of a group of submodules. The function of the entire module can be modeled as invoking, in parallel, all the local procedures corresponding to the submodules. In each clock cycle, all submodules receive data from their input ports simultaneously in phase 1 of the common clock and write data into their output ports in phase 2. The interconnections among submodules are modeled as parameter bindings. Each input port of a submodule corresponds to a formal parameter of a local procedure; whatever is connected to the input port corresponds to the actual parameter.

The specification of Poly-II serves as an example of the syntax for describing composite modules. Parallel invocation of submodules is specified by putting the corresponding procedure calls inside a "Parallel begin . . . end" construct. The specification states that the inputs of the leftmost module, Multiply-Add[m], come directly from the outside and that the input ports of every other submodule are connected to the output ports of the submodule to its left.

The notation used in the definition of a module follows standard programming language conventions. Array elements are designated by indexing. The mapping between actual and formal parameters is positional. For example, the order of the actual parameters in the statement Multiply-Add[m](x_{in}, p_{in}) indicates that the Poly-II module input p_{in} is connected to the p_{in} port of the mth Multiply-Add submodule, since that is the second In-Port in the submodule definition. A module definition can refer to a submodule's ports by prefixing them with the submodule name. For example, Multiply-Add[0].p_{out} refers to the p_{out} port of the 0th Multiply-Add module.

Each level in the hierarchical description of the hardware hides the details of lower levels; the names of submodules are not visible outside the module. This restriction eliminates potential name conflicts between ports of different submodules. When we need to overcome it, we use the equate (=) construct to identify a submodule port with an externally visible port. For example, the statement p_{out} = Multiply-Add[0].p_{out} defines the p_{out} port of the Poly-II module to be the same as the p_{out} port of the rightmost Multiply-Add submodule.

Driver. The driver represents the external calling discipline for the top-level module at each stage of the design. It relates a hardware structure to the part of the abstract algorithm it implements by specifying when it inputs and outputs the abstract variables. Although the driver is not part of the structure, for brevity we show it inside the module definition. To invoke the primitive module Multiply-Add with the input A_j and obtain an updated value as output, the driver is

In-Port A_{in} {A_j};
Out-Port A_{out} {A_j};

This means that the value of A_j is input to the Multiply-Add cell in phase 1 and the updated value of A_j is written to the output port A_{out} in phase 2. Inputting a stream of values to the cell is represented as

In-Port A_{in}
{ *Sequence[j from m to 0]* of A_j };

This means that the values $A_m, \ldots, A_0$ are input to the A_{in} port in successive clock cycles, as shown in Figure 5a.

To describe the data input to or output from an array port, we introduce the *Wave* construct:

In-Port A_{in}[1..n]
{Wave[*i from 1 to n*] *of* A_j};

This example represents n elements simultaneously input into the array port A_{in}. Here all the elements have the same value A_j.

A *Wave* can be nested inside a sequence:

In-Port A_{in}[1..n]
{*Sequence*[*j from m to 0*] *of*
Wave[*i from 1 to n*] *of* A_j};

As shown in Figure 5b, this represents a two-dimensional array input a row at a time into the array port A_{in}, starting with row m. Each row is n elements wide, and all the elements in the jth row have the value A_j.

Two attributes, *delayed* and *skewed,* capture common concepts in systolic data flow. The values in a stream that is *delayed k* are unspecified for the first k cycles; the stream description applies starting with the $(k+1)$th cycle:

In-Port A_{in}
{*Sequence[j from m to 0 delayed 1]*
of A_j};

Here an unspecified value indicated by × in Figure 5c is input into A_{in} in the first clock cycle, followed by the values $A_m, \ldots, A_0$ in sequence starting at the second cycle.

If a *Wave* is *skewed k,* the values of the wave elements are not presented simultaneously; instead, each element is presented k cycles after the element preceding it:

In-Port A_{in}[1..n]
{*Sequence* [*j from m to 0*] *of*
Wave [*i from 1 to n skewed 1*] *of* A_j};

As shown in Figure 5d, all the ports input the values $A_m, \ldots, A_0$ in sequence, with the ith port starting at the ith clock cycle $(1 \leq i \leq n)$.

Sometimes a module uses the same data value repeatedly, or computes a series of intermediate values culminating in a desired result. In such cases, Sys allocates registers to hold data loaded before the computation or accumulate results to be unloaded afterwards. Sys includes annotations like {*preloaded with x*} and {*unloaded as y*} in the driver information for such registers, but does not design the loading and unloading hardware. Standard

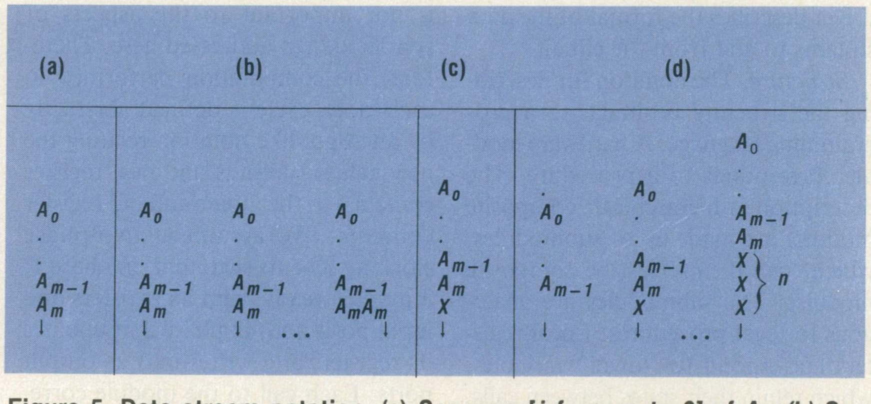


Figure 5. Data stream notation. (a) *Sequence[j from m to 0] of* A_j; (b) *Sequence[j from m to 0] of Wave[i from 1 to n] of* A_j; (c) *Sequence[j from m to 0 delayed 1] of* A_j; and (d) *Sequence[j from m to 0] of Wave[i from 1 to n skewed 1] of* A_j. An unspecified value is indicated by the symbol ×.

techniques exist for this purpose, such as loading the data sequentially with a tag bit attached to signal the loading phase or adding an extra data path to unload the results.

Scheduling. To map an algorithm, annotated with allocation advice, such as *in place* or *in parallel,* into hardware, Sys works bottom-up, starting with the innermost loop. For each loop, it designs the structure indicated by the annotation, then *schedules* the computation by constructing a simple driver that satisfies the following constraints:

(1) *Physical Constraint: No Data Collision.* A module cannot simultaneously input or output different values through the same port and can therefore compute a function of only one set of inputs at a time.

(2) *Logical Constraint: Data Dependency.* A function cannot be computed until the values of all the inputs are ready.

In the scheduling phase, Sys assumes that all the input and output values are stored in external memory. However, external I/O bandwidth is constrained by pin count, a limiting factor in VLSI design.

If a series of computations is scheduled on the same module, the data collision constraint requires that they be spaced far enough apart to avoid interfering with each other. Sys computes the number of clock cycles required between the start of successive iterations on the same module. Iterations can overlap in time without violating the data collision constraint, provided they do not use the same components simultaneously.

The data dependency constraint dictates that every input stream of a module must be delayed by at least one clock cycle relative to any output stream that supplies it, because data written into an output port in one clock cycle cannot by used until the next cycle. When scheduling a module, Sys delays all its streams by the minimum number of cycles required to satisfy this constraint. Sequential, parallel, and pipelined scheduling, illustrated in Figure 6, are the three most common schemes in systolic design.

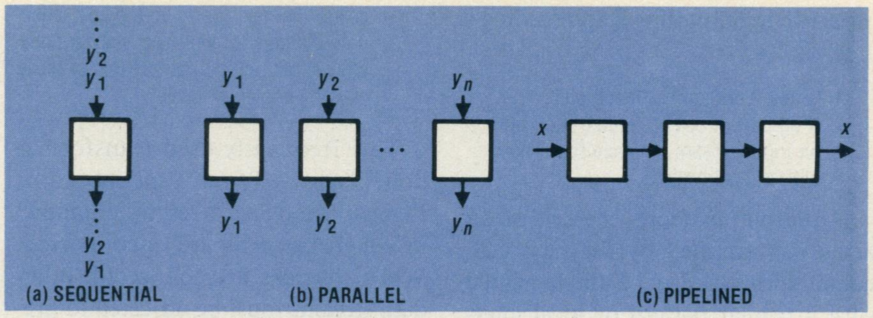


Figure 6. Three scheduling schemes.

Sequential. Consider the following loop:

For *i* from 1 to *n in place* do $y_i := F(y_i)$;

The annotation *in place* indicates that this loop is to be implemented on a single hardware cell that computes function F. The data collision constraint dicates that the loop must be computed sequentially, since the cell can compute F for only one input value at a time. As Figure 6a shows, the cell inputs the elements $y_1, \ldots, y_n$ in successive clock cycles and outputs the updated value for each element in the second phase of the same clock cycle.

Parallel. If we change the *in place* annotation to *in parallel*, Sys generates *n* modules, each of which computes one iteration of the loop. The data collision constraint is trivially satisfied. Since the iterations are independent of each other, they can be scheduled simultaneously without violating the data dependency constraint (Figure 6b). The *i*th cell inputs y_i and outputs its updated value.

Pipelined. Consider the following loop:

For *i* from 1 to *n in parallel* do $x := F(x)$;

Although *n* modules are allocated for computing this loop, the data dependency constraint precludes computing the *n* loop iterations simultaneously, since the value of *x* at the beginning of each iteration depends on the value computed during the previous iteration. However, a pipelined implementation is possible, as illustrated in Figure 6c. The initial value of *x* is input to the leftmost cell. Each cell applies the function F to its input and outputs the result to its right. After *n* steps, the rightmost cell outputs the final value of *x*. The delay is the same as for the sequential implementation; however, the pipelined structure permits overlapping execution of the loop for different initial values of *x,* accepting a new input as often as it can compute the function F. Pipelining, therefore, increases throughput by a factor of *n* relative to the sequential *in place* implementation.

Optimization. After scheduling, Sys has a correct, but inefficient, functional-level implementation of the original algorithm. It next tries to improve the design by applying correctness-preserving transformations suggested by the user.

In the optimization phase, Sys tries to reduce external memory accesses and, in general, to minimize communication between each module and its environment. Sometimes external data access can be replaced with local storage; for example, if an input value is used in many operations, it can be input into the system once and kept until it is no longer needed.

```
Preload-repeated-value:
    If an input stream consists of a
    repeated value, replace it with a
    preloaded register.
```

Similarly, if a computed value is only a temporary result to be used in some other computation, the system can save the value until it is used, rather than outputting it and reading it back in later.

```
Replace-feedback-with-register:
  If an output stream feeds back into
  an input stream, replace them both
  with a local register.
```

If the output from one cell of an array is consumed by the next cell, communication can be implemented with a regular pattern of local interconnections.

```
Internalize-data-flow:
  If the output of one cell in an array
  is input to the next cell after k steps,
  connect them through k-1 buffer
  registers.
```

Routing can take up much of the area required to lay out a VLSI circuit. Unnecessary interconnections can be eliminated by merging redundant data streams.

```
Broadcast-common-input:
  If the input streams to several ports
  are identical, replace them with one
  stream and broadcast its values to
  the old ports.
Propagate-common-input:
  If an array of cells input the same
  value with a skew of k, input it once
  and pass it from one cell to the next
  through k buffer registers.
```

Global broadcasting is usually undesirable in VLSI because the propagation delay along long wires introduces clock skew (slowing the allowable clock rate) and because signals with large fanout require powerful signal drivers that take up a large area. Replacing broadcast with local propagation allows a faster clock rate and generally saves space. This trick may not increase the speed of an individual computation because the data previously broadcast takes several clock cycles to propagate across the array. However, it increases the overall throughput by overlapping successive computations. Because cells that received the broadcast data in the same clock cycle will now receive it in successive clock cycles, the circuit must be retimed to delay the action of each cell until the data arrives.

```
Retime-to-eliminate-broadcasting:
  If an input stream is broadcast to
  several unconnected cells, propagate
  each data item in the stream from one
  cell to the next, delaying the cell action
  until the item arrives, and skewing any
  other input and output streams of those
  cells by one clock cycle.
```

This frequently-used transformation[19] is derived from a special case of Leiserson and Saxe's retiming lemma[4] in which the cells are unconnected. When the cells are connected, other data streams must be adjusted to preserve consistency. A common technique derivable from the lemma is to slow down one data stream with respect to another by inserting delay registers in each cell. Another technique is to pipe two data streams through an array in opposite directions.

Transformational derivation of two systolic designs

To generate the systolic design Poly-I and Poly-II the user annotates the algorithm with allocation advice and selects the transformations to apply in each optimization phase as Sys works its way up from the innermost loop.

Derivation of Poly-I. Sys produces the design shown in Figure 2 from the annotated algorithm

```
For i from 1 to n in parallel do
  for j from m to 0 in place do
    $p_i := p_i \times x_i + A_j$;
```

The initial values of $p_1, \ldots, p_n$ are 0.

Implement loop body. The first step in deriving Poly-I implements the inner loop body, as shown in Figure 7a:

```
Module Multiply-Add;
  In-Port $A_{in}$ $\{A_j\}$;
  In-Port $x_{in}$ $\{x_i\}$;
  In-Port $p_{in}$ $\{p_i\}$;
  Out-Port $p_{out}$ $\{p_i\}$;

  $p_{out} := p_{in} * x_{in} + A_{in}$
end Multiply-Add;
```

Allocate inner loop. Next, Sys implements the inner loop:

```
For j from m to 0 in place do
  $p_i := p_i * x_i + A_j$;
```

The annotation *in place* tells Sys to allocate a single Multiply-Add module for this computation.

Schedule inner loop. The data collision constraint requires that the loop be executed sequentially. This is done by extending the driver of the Multiply-Add module to process sequential streams of data:

```
In-Port $A_{in}$ {Sequence[j from m to 0] of $A_j$};
In-Port $x_{in}$ {Sequence[j from m to 0] of $x_i$};
In-Port $p_{in}$ {Sequence[j from m to 0] of $p_i$};
Out-Port $p_{out}$ {Sequence[j from m to 0] of $p_i$};
```

This driver specifies that the external environment first supplies Multiply-Add with the initial value of p_i. The updated value is then output at the p_{out} port and input back into the cell in the following clock cycle, as shown in Figure 7b. This process is then repeated for a total of $m+1$ cycles.

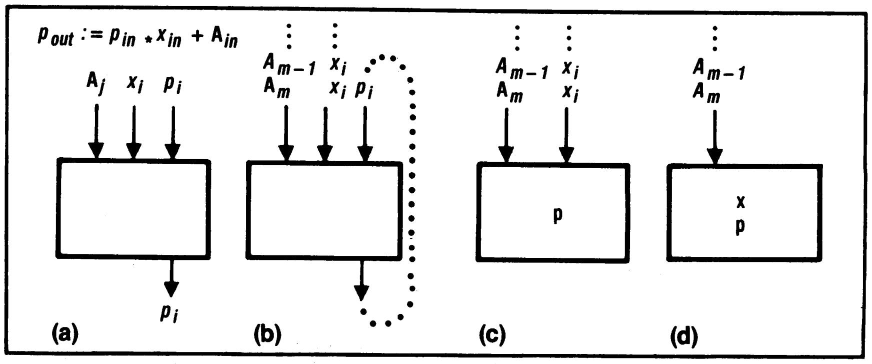


Figure 7. Implementation of inner loop for Poly-I: (a) implementation of inner loop body; (b) initial sequential implementation of inner loop; (c) result of holding p_i in register p; (d) result of replacing repeated input x_i with preloaded register x.

Optimize inner loop. Since the output from p_{out} is fed back into p_{in}, the Replace-feedback-with-register transformation applies here. As Figure 7c shows, this rule modifies Multiply-Add by replacing the p_{in} and p_{out} ports with an internal register, p, and by deleting the streams for the eliminated ports.

Next, Sys identifies the data stream *Sequence* [*j from m to 0*] *of* x_i as a repeated value by noticing that x_i is independent of the loop index j. Therefore, the Preload-repeated-value transformation applies. This rule modifies the structure by replacing the input port x_{in} with an internal register x, and deletes the input stream for x_{in} from the driver. Figure 7d shows the optimized implementation of the inner loop. At this point, the design description is

```
Module Multiply-Add;
  In-Port A_in {Sequence[i from m to 0] of
    A_j};
  Register x {preloaded with x_i};
  Register p {preloaded with 0,
              unloaded as p_i};
   p := p*x+A_in;
end Multiply-Add;
```

Allocate outer loop. The annotation *in parallel* tells Sys to implement the outer loop of the algorithm by replicating the Multiply-Add module it has constructed for the inner loop.

Schedule outer loop. Since the iterations of the outer loop are allocated to different Multiply-Add modules and can be executed independently, they can be scheduled simultaneously without violating either the data collision or the data dependency constraint. Sys groups the A_{in} ports of these modules into a single array port. It constructs the driver for this array port by wrapping a *Wave* quantifier around the driver for the original scalar port. The resulting design, shown in Figure 8a, is described as

```
Module Poly-I;
   InPort A_in[1..n]
      {Sequence]j from m to 0[ of
        Wave[i from 1 to n] of A_j};
   Module Multiply-Add[i∈1..n];
     In-Port A_in;
     Register x {preloaded with x_i};
     Register p {preloaded with 0,
                 unloaded as p_i};
   p := p * x+A_in;
  end Multiply-Add;
  Parallel begin
    Multiply-Add[i] (A_in[i]) for 1≤i≤n;
  end;
end Poly-I;
```

Optimize outer loop. Sys now realizes that the data input to the constituent ports of the array port A_{in} are identical, since A_j is independent of i, so it applies the Broadcast-common-input rule. As Figure 8b shows, Sys reduces A_{in} to a scalar port, simplifies its input stream from a sequence of waves to a scalar sequence, and broadcasts it to all the cells in the array:

```
In-Port A_in {Sequence[j from m to 0] of
A_j};
Multiply-Add[i] (A_in) for 1≤i≤n;
```

Finally, Sys applies the Retime-to-eliminate-broadcasting rule. The input stream is unchanged, but it now goes only to the leftmost cell instead of being broadcast to all the cells. Figure 2 shows the final Poly-I design:

```
Module Poly-I;
   In-Port A_in
      {Sequence[j from m to 0] of A_j};
   Module Multiply-Add[i∈1..n];
     In-Port A_in;
     Out-Port A_out;
     Register x {preloaded with x_i};
     Register p {preloaded with 0,
                 unloaded as p_i};
     p := p*x+A_in;
     A_out := A_in;
   end Multiply-Add;
   Parallel begin
     Multiply-Add[1] (A_in);
     Multiply-Add[i] (Multiply-Add
        [i-1].A_out)
          for 2≤i≤n;
   end;
end Poly-I;
```

Derivation of Poly-II. The Poly-II design shown in Figure 3 is the result of starting with different annotations in the same algorithm:

```
For i from 1 to n in place do
   for j from m to 0 in parallel do
      p_i := p_i * x_i + A_j;
```

Implement loop body. The first step, implementing the body of the inner loop, is the same as for Poly-I. The resulting Multiply-Add module is shown in Figure 7a.

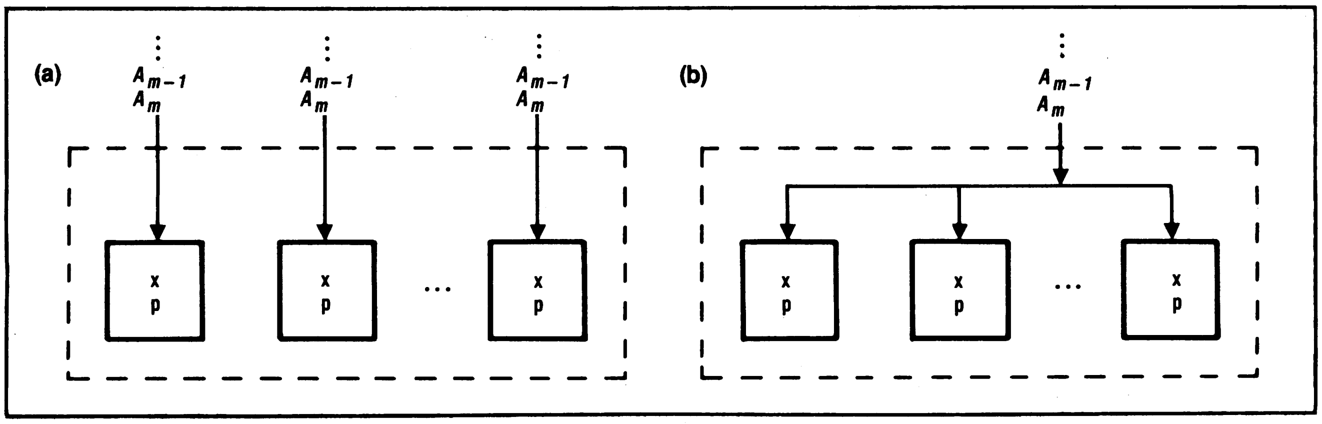


Figure 8. Outer loop implementation: for Poly-I: (a) initial parallel implementation; (b) identical streams collapsed into broadcast input A_j.

Allocate inner loop. The next step implements the inner loop. The annotation *in parallel* tells Sys to replicate Multiply-Add for each *j* from *m* to 0. The resulting module has three array inputs—p_{in}, x_{in}, and a_{in}—and an array output, p_{out}.

Schedule inner loop. Because the loop iterations update the same variable, they are not independent. To satisfy the data dependency constraint, Sys employs the pipelined scheme, scheduling the iterations successively (rather than simultaneously) in successive cells. The data streams must be delayed accordingly. This timing information is represented by the *skewed* attribute.

```
Module Poly-II;
  In-Port $A_{in}$[m..0] {Wave[j from m to
    0 skewed 1] of $A_j$};
  In-Port $Ax_{in}$[m..0] {Wave[j from m
    to 0 skewed 1] of $x_i$};
  In-Port $p_{in}$[m..0] {Wave[j from m to
    0 skewed 1] of $p_i$};
  Out-Port $p_{out}$[m..0] {Wave[j from m
    to 0 skewed 1] of $A_{pi}$};
  Module Multiply-Add[m..0];
    In-Port $A_{in}$, $x_{in}$, $p_{in}$;
    Out-Port $p_{out}$;

    $p_{out}$ := $p_{in}$*$x_{in}$ + $A_{in}$;
  end Multiply-Add;

  $p_{out}$[j] Multiply-Add[j]·$p_{out}$ for
    0≤j≤m;
  Parallel begin
    Multiply-Add[j] ($A_{in}$[j], $x_{in}$[j],
    $p_{in}$[j]
      for 0≤j≤m;
  end;
end Poly-II;
```

As Figure 9a shows, the driver specifies that the value p_i is input to the system at p_{in}[m] in the first cycle and an updated value is returned at p_{out} [m] at the end of the cycle. This value is fed in turn to p_{in} [m − 1] in the next cycle, and so forth, for a total of m + 1 cycles.

Optimize inner loop. Sys now reduces p_{in} and p_{out} to scalar ports by applying the Internalize-data-flow rule. The result is shown in Figure 9b, where p_{in} is now an input of the leftmost cell and p_{out} is an output of the rightmost cell. Sys further simplifies the circuit by applying the Propagate-common-input rule, which applies to the input stream *Wave* [*j from m to 0 skewed 1*] *of* x_i because x_i is independent of *j* and, unlike p_i, is not updated by Multiply-Add. In the simplified circuit, shown in Figure 9c, x_{in} is reduced to a scalar input port connected only to the leftmost cell. An output port x_{out} is added to each Multiply-Add cell and connected to the x_{in} port of the cell to its right. The definition of Multiply-Add is modified to store the input from x_{in} into x_{out} at each clock cycle, so as to pass x_i from one cell to the next. The design is now

```
Module Poly-II;
  In-Port $A_{in}$ {Wave[j from m to 0
    skewed 1] of $A_j$};
  In-Port $x_{in}$ {$x_i$};
  In-Port $p_{in}$ {$p_i$};
  Out-Port $p_{out}$ {$p_i$ delayed m};
  Module Multiply-Add[m..0];
    In-Port $A_{in}$, $x_{in}$, $p_{in}$;
    Out-Port $p_{out}$, $x_{out}$;

    $p_{out}$ := $p_{in}$*$x_{in}$ + $A_{in}$;
    $x_{out}$ := $x_{in}$;
  end Multiply-Add;

  $p_{out}$ = Multiply-Add[0]·$p_{out}$;
  Parallel begin
    Multiply-Add[m] ($A_{in}$[m], $x_{in}$,
    $p_{in}$);
    Multiply-Add[j] ($A_{in}$[j], Multiply-
      Add[j+1]·$x_{out}$,
      Multiply-Add[j+1]·$p_{out}$) for
    0≤j≤m−1;
  end;
end Poly-II;
```

Allocate outer loop. The annotation *in place* in the outer loop of the algorithm tells Sys to implement it on the hardware structure it has constructed for the inner loop.

Schedule outer loop. Although the iterations of the outer loop are independent, the data collision constraint requires using the sequential scheme of scheduling. Sys analyzes the module description and determines that each iteration can start one cycle after the previous one. As Figure 10 shows, only the streams are changed:

```
In-Port $A_{in}$[m..0]
  {Sequence[i from 1 to n] of Wave[j from
    m to 0 skewed 1] of $A_j$};
In-Port $x_{in}$
  {Sequence[i from 1 to n] of $x_i$};
In-Port $p_{in}$
  {Sequence[i from 1 to n] of $p_i$};
Out-Port $p_{out}$
  {Sequence[i from 1 to n delayed m] of
    $p_i$};
```

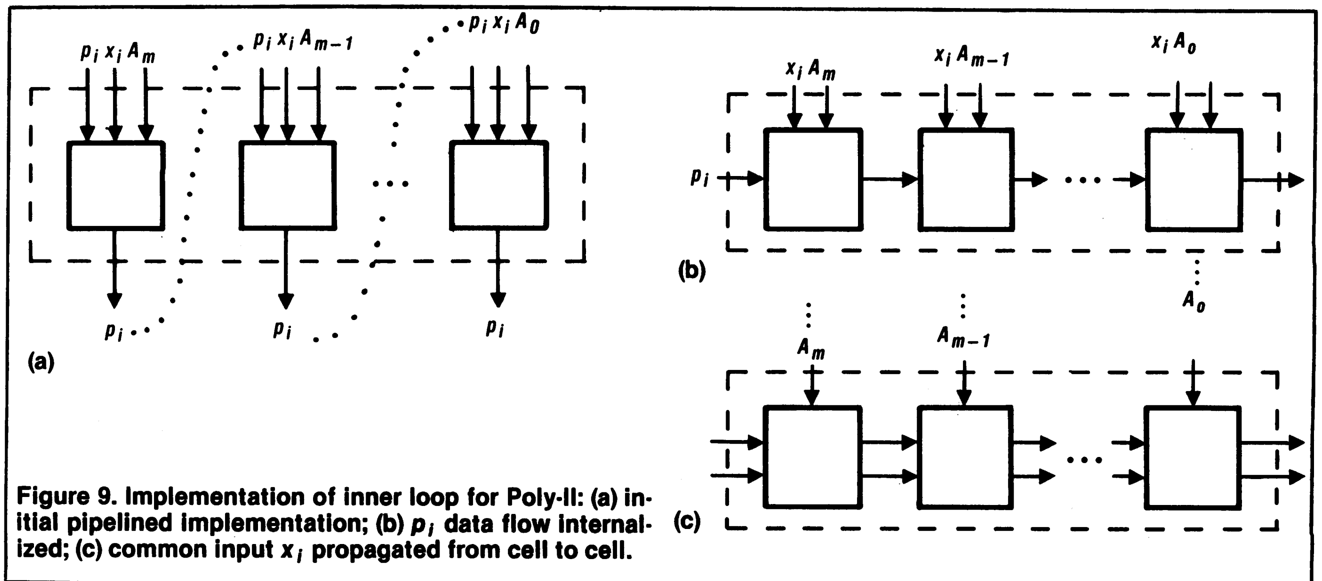


Figure 9. Implementation of inner loop for Poly-II: (a) initial pipelined implementation; (b) p_i data flow internalized; (c) common input x_i propagated from cell to cell.

Optimize outer loop. A_{in} is eliminated by applying the Preload-repeated-value rule. Sys identifies the input stream for A_{in} as a repeated value by noticing that the expression A_j is independent of the sequence index i. This step completes the Poly-II design illustrated in Figure 3 and described in Figure 4.

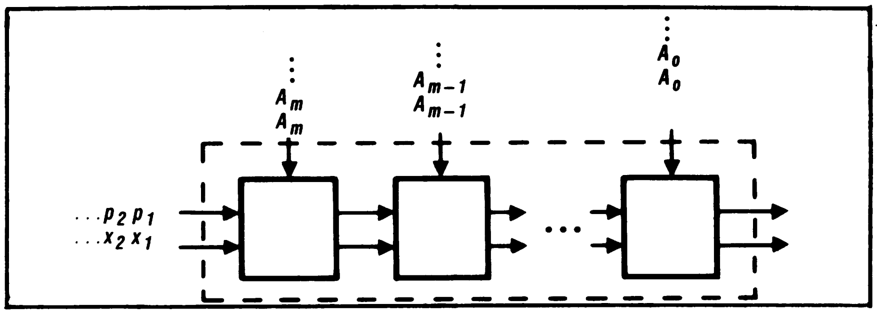


Figure 10. Initial sequential implementation of outer loop for Poly-II.

Remarks on Sys

Some algorithms implemented on systolic arrays involve not only nested loops, but additional statements before and after an inner loop. For example, an algorithm for finding averages performs a series of additions followed by a single division. These algorithms are sometimes implemented as systolic arrays with special boundary cells.[20,21] The bottom-up approach we have described allows Sys to generate such compound designs by implementing different parts of the algorithm as separate circuits and then "glueing" them together. Each part of the algorithm is first mapped onto a driver and structure. Sys then determines the connections between ports, the insertion of delay registers, and the scheduling of the composite circuit by analyzing dependencies among the data streams of the different circuits.

Sys is an experimental prototype that was developed rapidly in the sophisticated programming environment of Interlisp.[22] It has some knowledge about how to map loops and begin-end blocks onto systolic arrays, and it has a very small database of transformations. Despite these limitations, Sys generates efficient systolic designs, including a rectanglar array for matrix multiplication, several designs for convolution,[23] and various circuits for string pattern-matching. We have extended the transformational model to derive systolic designs for matrix multiplication on a hexagonal array,[20] dynamic programming, [24] and computing the greatest common divisor of polynomials.[13] However, Sys lacks some of the features required to handle these examples. For instance, Sys uses data dependency information for scheduling and optimization, but its dependency analysis is very limited. To identify a stream of replicated data, it simply tests whether the variable inside the *Wave* or *Sequence* quantifier is independent of the quantifier index. To test if one data stream is a delayed version of another, Sys checks if they use the same variable; thus Sys does not recognize that *Sequence*[*i from 0 to n−1*] *of* x_{i+1} and equivalent to *Sequence*[*i from 1 to n*] *of* x_i *are equivalent.* This limitation could be overcome by adding an algebraic manipulation capability.

The major contribution of this work is a transformational model of systolic design. In our design model, software transformations are first applied to put the algorithm to be implemented into a regular form conducive to systolic implementation. The steps of allocating operations to hardware, scheduling their execution, and optimizing the design are then performed bottom-up, starting with the innermost blocks of the algorithm. We have successfully used this model to rederive several published designs, and it appears suitable for designing complex systolic arrays. This model may help guide manual design, explain systolic algorithms, or capture the design process in the machine where it can benefit from effective automated support.

Transformations offer a convenient way to formalize systolic design expertise. They are typically very simple, and their effects are close to the designer's intent. New ones can readily be added to incorporate new design techniques.

An equally important contribution of this research is a natural notation for describing systolic designs. It preserves the structure of the original algorithm by showing clearly which operations are algorithmically related, even though they may be radically redistributed in time and space in the final design. Together, the *Wave, Sequence, skewed,* and *delayed* constructs make it easy to express the standard systolic communication patterns. Similarly, the quantified notation for arrays of ports and modules allows succinct specification of replicated structure.

Splitting the description into structure and driver has proved advantageous. Factoring transformations into their effects on structure and driver makes the transformations easy to represent and implement. Describing the hardware separately from how it is to be used makes structure descriptions context-independent and hence easier to combine into composite structures. The hierarchical representation of structure makes complex designs more manageable.

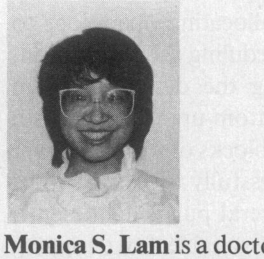
Our transformational model and notation are the basis of our prototype system, Sys. While Sys itself does not purport to be a practical tool for systolic design, it demonstrates the feasibility of a transformational design process that combines human creativity with the capability of machines to perform detailed manipulations. □

Acknowledgments

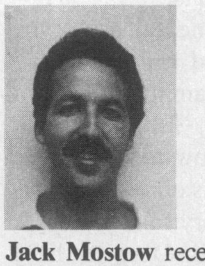
The authors are grateful for the support of H. T. Kung, Bob Balzer, and their colleagues at CMU and ISI and to Mario Barbacci, Allan Fisher, and Peter Highnam for reading early drafts. This research was supported in part by DARPA Contract MDA-903-81-C-0335.

References

1. D. Cohen, "Mathematical Approach to Iterative Computational Networks," *Proc. Fourth Symp. Computer Arithmetic,* Oct. 1978, pp. 226-238.
2. L. Johnsson and D. Cohen, "A Mathematical Approach to Modelling the Flow of Data and Control in Computational Networks," *VLSI Systems and Computations,* H. T. Kung, R. F. Sproull, and G. L. Steele, Jr., eds., Computer Science Press, Inc., Carnegie-Mellon University, Pittsburgh, Penn., Oct. 1981, pp. 213-225.
3. U. Weiser and A. Davis, "A Wavefront Notation Tool for VLSI Array Design," *VLSI Systems and Computations,* H. T. Kung, R. F. Sproull, and G. L. Steele, Jr., eds., Computer Science Press, Inc., Carnegie-Mellon University, Pittsburgh, Penn., Oct. 1981, pp. 226-234.
4. C. E. Leiserson and J. B. Saxe, "Optimizing Synchronous Systems," *J. VLSI and Computer Systems,* Vol. 1, No. 1, 1983, pp. 41-68.
5. P. R. Cappello and K. Steiglitz, "Unifying VLSI Array Designs with Geometric Transformations," *Proc. 1983 Int'l Conf. Parallel Processing,* Aug. 1983, pp. 448-457.
6. R. M. King and T. C. Brown, "Research on Synthesis of Concurrent Computing Systems," *Proc. 10th Ann. Symp. Computer Architecture,* June 1983, pp. 39-46.
7. R. H. Kuhn, "Transforming Algorithms for Single-Stage and VLSI Architectures," *Proc. Workshop on Interconnection Networks for Parallel and Distributed Processing,* Apr. 1980, pp. 11-19.
8. D. I. Moldovan, "On the Analysis and Synthesis of VLSI Algorithms," *Transactions on Computers,* Nov. 1982, Vol. C-31, No. 11, pp. 1121-1125.
9. D. I. Moldovan, "On the Design of Algorithms for VLSI Systems," *Proc. IEEE,* Jan. 1983, pp. 113-120.
10. P. Quinton, "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations," *Proc. 11th Ann. Symp. Computer Architecture,* 1984, pp. 208-222.
11. A. L. Fisher et al., "Architecture of the PSC: A Programmable Systolic Chip," *Proc. 10th Ann. Symp. Computer Architecture,* June 1983, pp. 48-53.
12. J. J. Symanski, "NOSC Systolic Processor Testbed," Tech. report NOSC TD 588, Naval Ocean Systems Center, June 1983.
13. R. P. Brent and H. T. Kung, "Systolic VLSI Arrays for Linear-Time GCD Computation," *VLSI 83,* F. Anceau and E. J. Aas, eds., North Holland, New York, Aug. 1983, pp. 145-154.
14. R. Balzer, N. Goldman, and D. Wile, "On the Transformational Implementation Approach to Programming," *Proc. Second Int'l Conf. Software Engineering,* IEEE, 1976, pp. 337-343.
15. D. S. Wile, "Program Developments: Formal Explanations of Implementations," *Comm. ACM,* Vol. 26, No. 11, Nov. 1983, pp. 902-911.
16. D. J. Kuck, *The Structure of Computers and Computations,* John Wiley & Sons, New York, Vol. 1, 1978.
17. D. J. Kuck et al., "Dependence Graphs and Compiler Optimizations," *Proc. ACM Symp. Principles of Programming Languages,* Jan. 1981, pp. 207-218.
18. J. Mostow and B. Balzer, "Application of a Transformational Software Development Methodology to VLSI Design," *J. Systems and Software,* Vol. 14, 1984, pp. 51-61.
19. R. H. Kuhn, "Efficient Mapping of Algorithms to Single-Stage Interconnections," *Proc. Seventh Ann. Symp. Computer Architecture,* 1980, pp. 182-189.
20. H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," *Introduction to VLSI Systems,* C. A. Mead and L. A. Conway, Addison-Wesley, Reading, Mass., 1980, pp. 271-292.
21. M. J. Foster and H. T. Kung, "The Design of Special-Purpose VLSI Chips," *Computer,* Vol. 13, No. 1, Jan. 1980, pp. 26-40.
22. W. Teitelman, *Interlisp Reference Manual,* Xerox Palo Alto Research Center, Palo Alto, Calif., 1978.
23. H. T. Kung, "Why Systolic Architectures?," *Computer,* Vol. 15, No. 1, Jan. 1982, pp. 37-46.
24. L. J. Guibas, H. T. Kung, and C.D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," *Proc. Conf. VLSI: Architecture, Design, Fabrication,* California Institute of Technology, Pasadena, Calif., 1979, pp. 509-525.



Monica S. Lam is a doctoral student in computer science at Carnegie-Mellon University. Her research interests include parallel computer architecture and software language support for parallel processing. She received a BS from University of British Columbia in 1980 and an MS from Carnegie-Mellon University in 1983, both in computer science.



Jack Mostow received his AB in applied mathematics from Harvard in 1974 and his PhD in computer science from Carnegie-Mellon University in 1981. He has taken a transformational approach to a variety of problems in hardware and software design. His research interests lie in the area of AI, especially in machine learning.

Questions concerning this article can be addressed to Lam at the Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213.