Mapping a Single-Assignment Language onto the Warp Systolic Array

Thomas Gross and Alan Sussman

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Single-assignment languages offer the potential to efficiently program parallel processors. This paper discusses issues that arise in mapping SISAL programs onto the Warp^{SM} array, a linear systolic array in use at Carnegie Mellon. A Warp machine with ten cells can deliver up to 100 million floating point operations per second.

The paper begins with a discussion of systolic arrays as targets for single-assignment languages and the suitability of the Warp machine for this purpose. Systolic arrays can take advantage of both large-grain parallelism and fine-grain parallelism. The communication bandwidth of the systolic array gives the translator great flexibility in mapping a SISAL program onto the linear array.

We present two principal methods to exploit parallelism on Warp, data partitioning and pipelining. Data partitioning is effective for local computations that depend on only a small neighborhood of values. Since SISAL allows the specification of array sizes at run-time, we have to provide static and dynamic methods for data partitioning. Many operations on the SISAL stream data type can be parallelized as a special case of dynamic data partitioning. Pipelining allows the overlapping of different stages of a computation or of function invocations. This method is well suited for Warp since the systolic array has high inter-cell communication bandwidth. This makes it possible to send large data sets to the next processor in a computation pipeline without performance degradation.

We use matrix multiplication and a relaxation algorithm, respectively, as examples to illustrate the data partitioning and pipeline models for mapping SISAL programs onto the Warp array.

1. Introduction

Single-assignment languages offer an elegant way to program parallel computers. There is no need for the compiler to "extract" parallelism, and users do not get involved in the explicit management of parallelism in a program. The challenge for the compiler writer and computer architect is to devise an efficient architecture that can exploit this implicit parallelism in practice.

To date, there have been two major thrusts toward implementing single-assignment languages. Since single-assignment languages like VAL or SISAL are geared towards execution in a graph-oriented processing environment, some researchers have concentrated on building hardware that directly interprets a program graph. A program is translated into a graph representation; the nodes in this graph represent operations (or function invocations), and the arcs specify data dependencies between the nodes. Such an architecture is capable of exploiting fine-grain parallelism since there is the potential for a large number of elementary nodes to be

executed in parallel. The Manchester dataflow prototype is an example of this class of machines [6]. However, efficient hardware realizations of this processor model are still the topic of ongoing research. The other approach is to implement a single-assignment language for conventional parallel computers such as vector processors or processor arrays [10]. This approach is workable, but faces the problems inherent in the organization of this class of architectures. These architectures either do not scale well (vector processors, multiprocessors based on a shared bus or shared memory) or are restricted to large-grain parallelism (bus-based multiprocessors or loosely coupled processor arrays).

Systolic processors have efficient implementations. Arrays built from systolic processors are highly parallel computers that provide sufficient inter-processor communication bandwidth for large-grain as well as fine-grain parallelism. Programmable systolic processors allow the same hardware structure to be used for a wide variety of systolic algorithms. When equipped with a local memory, these processors are powerful computing engines. Research in systolic systems has demonstrated that an array built from high-performance programmable processors can deliver high computation bandwidth [1].

This paper discusses issues arising from mapping applicative programs onto systolic arrays. To investigate the relationship between systolic arrays and functional languages, we map SISAL programs onto the Warp systolic array. SISAL is a derivative of VAL and has been used by several groups [8, 9]. SISAL is an applicative language and provides a good vehicle to investigate two important issues common to single-assignment and functional languages: the management of parallelism and the management of data. Issues raised by high-order functions, which are present in functional languages like ML or FP, are outside of the scope of this study. Our target is a specific array, namely the Warp systolic array that is in use at Carnegie Mellon. Section 2 gives a brief description of those details of the Warp computer that are relevant for this work.

2. Warp system overview

The Warp machine is a high-performance systolic array computer designed for computation-intensive applications. In a typical configuration, a Warp computer consists of a linear systolic array of 10 or more identical cells, each of which is a 10 MFLOPS programmable processor. Thus the Warp machine has a peak performance of 100 MFLOPS. The Warp machine is integrated as an attached processor into a general-purpose host running UNIX™ [4].

2.1. Architecture

There are three major components in the system—the Warp processor array (*Warp array*), the interface unit (*IU*), and the *host*, as depicted in Figure 2-1. The Warp array performs computation-intensive routines such as low-level vision routines or matrix operations. The IU handles the input/output between the array and the host, and generates control signals for the Warp array. The host supplies data to and receives results from the array, in addition to executing the parts of the application programs that are not mapped onto the Warp array. For example, the host performs those parts of an application that invoke the SISAL program for the array.

The host consists of a SUN-3 workstation connected to a VME-based multi-processor. The workstation acts as the master controller and provides a UNIX environment for running application programs. The VME-based multi-processor controls peripherals, such as graphics boards or cameras, and contains a large amount of memory for storing data to be processed by the Warp array. Its dedicated processors transfer data to and from the Warp array and perform simple reordering operations on the data.

The Warp array is a one-dimensional systolic array with identical cells called Warp cells. A linear array is easy to implement in hardware, and requires a lower external I/O bandwidth than other array shapes, since only the two end-cells communicate with the outside world. Data flow through the array on two data paths (X and

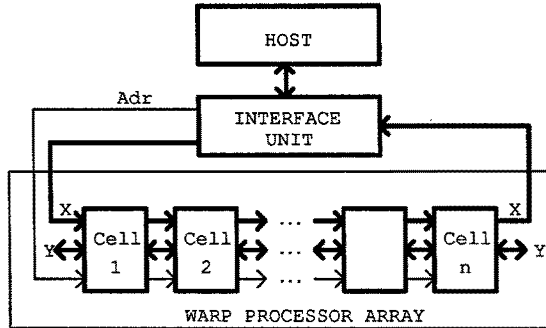

Figure 2-1: Warp system overview

Y), while systolic control signals and addresses (for local cell memories) can travel on the Adr path (as shown in Figure 2-1). The direction of the Y path is statically configurable. This feature is important in algorithms that either require accumulated results in the last cell to be sent back to the other cells (e.g., in back-solvers), or require local exchange of data between adjacent cells (e.g., in some implementations of numerical relaxation methods). Each cell is capable of transmitting 40 Mbytes/second as well as receiving the same amount for a total aggregate bandwidth of 80 Mbytes/second. This high communication bandwidth makes the Warp cells systolic processors.

Each Warp cell is implemented as a programmable horizontal microengine, with its own microsequencer and program memory for 8K instructions. Each Warp cell contains two functional units: a 32-bit floating-point multiplier and a 32-bit floating-point adder. In addition, there is a a local memory of 32K words for resident and temporary data. Program and data memory are separate on the cell.

The cells operate on a 200ns cycle time and are highly parallel internally. In a single cycle, each cell can execute two floating point operations (one addition, one multiplication), read one 32-bit word from memory, and write one 32-bit word to memory. In addition, I/O operations with the neighboring cells proceed in parallel. Each cell can input two 32-bit data words and output two 32-bit data words per cycle.

The Warp machine can be used for both fine-grain and large-grain parallelism. Warp is efficient for the fine-grain parallelism needed for systolic processing, because of its high inter-cell bandwidth. The I/O bandwidth of each cell is higher than that of other processors with similar computation power. Each cell can transfer up to 20 million 32-bit words per second to and from its neighboring cells, in addition to 20 million 16-bit addresses. High inter-cell communication bandwidth allows fast transfers of large volumes of intermediate data between neighboring cells.

The Warp machine is efficient for large-grain parallelism because it is composed of powerful cells. Each cell is capable of operating independently; it has its own program sequencer and program memory. Moreover, each cell has 32 Kwords of local data memory; this memory size is large compared to other systolic array designs. With a large data memory, high computation bandwidth can be sustained without imposing increased demand on the I/O bandwidth [7].

2.2. Programming

Two characteristics of the Warp computer are chiefly responsible for its high performance: the Warp machine employs multiple cells in the array, and within each cell, a high degree of parallelism and pipelining is available. In the current environment, the parallelism across cells is managed by the user, and the details of parallelism and pipelining on the cell are handled by an optimizing compiler [5].

The user specifies a program in a high-level language called W2; this program consists of a description for each cell in the system. This description is translated by the compiler into microcode for the cells and interface unit as well as code for the host to pump data to and from the array. This arrangement leaves the responsibility for computation decomposition with the user; in turn it allows the user to write highly efficient programs since W2 is restricted enough to be translated efficiently. Data transfers between cells are made explicit in the program text by "**send**" and "**receive**" statements. They specify the direction (*r*ight or *l*eft) as well as the pathway to be used (*X* or *Y*). Figure 2-2 sketches the W2 description of a simple program to evaluate a polynomial according to Horner's rule.

$p(x)=(((a_0 \times x+a_1)\times x+a_2)\times ....)\times x+a_9$; each cell computes $t_{new}=(x_{in}\times t_{old})+a_j$.

```
receive(L, X, xin);
receive(L, Y, told);
tnew:= xin * told + a[j];
send(R, X, xin);
send(R, Y, tnew);
```

Figure 2-2: W2 program

W2 has been used for a large group of application programs in the areas of low-level vision and scientific computing. W2 is the target language for our SISAL compiler. There are several benefits gained from such an arrangement: for example, the SISAL compiler does not have to re-implement the optimizations of the W2 compiler. Furthermore, we can compare the quality of the W2 code generated from SISAL programs with our large library of hand-written W2 programs.

The work on the SISAL translator addresses the problem of programming the Warp array without the need to individually program the cells. We want to show that, for a large class of applications, functional programs can run as efficiently on the Warp machine as programs written at a lower level (i.e., the cell programs written in W2), if the programmer chooses the appropriate SISAL constructs. In addition, we can evaluate the efficiency of different SISAL constructs executing on the Warp array. This evaluation provides feedback to the programmer as well as to the designers of future systolic arrays.

3. Outline of our approach

Our mapping starts with a data flow graph generated from the SISAL program. We use the flow graph produced by the SISAL to IF1 translator developed at Lawrence Livermore National Laboratory (LLNL) [12]. This flow graph closely resembles in structure the original program and reflects the choices of language constructs made by the programmer. Transformations that alter this flow graph to take advantage of several optimizations have been discussed elsewhere [11], and we expect that our system will be improved by including such optimizations.

The IF1 flow graph consists of a set of nodes, and these nodes are mapped onto the systolic array. Section 3.1 describes briefly the nodes of an IF1 flow graph; more information can be found in the IF1 manual [12]. If the computation described by a node can be executed in parallel, it is mapped to more than one cell. If execution has to occur sequentially, only a single cell is used. Therefore, all data must travel back to the first cell after evaluation of each node, unless the compiler can establish at compile time that the data will be needed at the current cells again. This optimization reduces data traffic and is discussed again in Section 4.1.3. For example, consider the sequence

```
let
  x := a + b;
  ...
  y := for i in 1, N
         temp:= row[i] * col[i] + f(x)
       returns value of sum temp
       end for;
  ...
  z := y / N;
```

x, **y** and **z** must be evaluated in sequential order because of data dependencies, so we cannot evaluate the expressions **a+b** and **y/N** in parallel. Instead, they will be evaluated by the first cell in the array. However, evaluation of the "**for**" construct can proceed in parallel and may therefore be mapped onto the array using the techniques described in Sections 4 and 5.

For a linear array such as the Warp array, there are two ways to get data from a cell in the array to the first cell. One way is for data to travel along the backward (Y) path (see Figure 2-1). Alternatively, data can travel in a ring-like fashion via the interface unit and the host system back to the first cell. This second route is not economical at this time for the Warp implementation since the transfer rates to and from the host are not as high as the internal transfer rates. It will become attractive once the Warp Boundary Processor is in place; this processor is a special cell that occupies the first position of the array and implements a ring-like structure in the Warp machine without sacrificing speed. The boundary processor is currently under construction [3].

In our model, only top level expressions are distributed across the array. This allows us to use static partitioning methods. That is, the method is static, but the number of data items allocated to each cell can be determined at run-time. Another advantage is that any interior node parallelism can be used by the W2 compiler to exploit cell level pipelining and parallelism. Figure 3-1 shows the principal phases of our translation system.

Source	Module	Output	Status
SISAL	Parser	IF1	Developed at LLNL
IF1	Mapper	W2	Under development at CMU
W2	Compiler	μcode	Developed at CMU

Figure 3-1: Mapping SISAL onto the Warp machine

3.1. IF1

IF1 is a graph language intended to be the target of several compilers for functional languages [12]. IF1 is strongly oriented towards the features of SISAL [9] and VAL [8].

The IF1 translation of a SISAL function is a fairly straightforward translation into data flow graph form. Each program is represented by an acyclic graph. Graph nodes denote operations, such as add or multiply, and graph edges denote values that are passed between nodes. Graphs are surrounded by boundaries that denote the input and output characteristics of a graph.

There are two kinds of nodes, simple and compound. Compound nodes contain subgraphs, while simple nodes only describe the functional relationship between their inputs and outputs. Compound nodes are defined hierarchically; the subgraphs of a compound IF1 node (such as a loop) are one level down in the graph hierarchy from the complex graph node. Graph boundaries delimit the subgraphs within a compound node. For example, a parallel loop node contains three subgraphs. The three subgraphs are:

- the generator (or header) of the loop, which produces the data values to iterate over (array or range elements),
- the body of the loop, which may contain any set of IF1 nodes and edges,
- and the result(s) of the loop, which generate the return value(s) for each iteration of the loop.

The edges in the graphs provide an explicit representation of data dependence between operations and are the only ordering constraints necessary for correct execution of programs. IF1 contains distinct graph nodes to distinguish between parallel and non-parallel loop forms. For a more in-depth discussion of the correspondence between SISAL and IF1 constructs see the IF1 manual [12]. A front-end compiler from SISAL to IF1 and an IF1 interpreter (both developed at LLNL) are currently being used to test SISAL programs.

4. Data partitioning

Data partitioning is an effective method for exploiting the parallelism in SISAL programs. The key idea is to divide up the input data for a node across the cells of the Warp array and then to execute the same function on each cell to produce the corresponding output data. Data partitioning is a variant of the algorithm input partitioning technique that was used when coding several Warp applications in W2 [2].

Data partitioning is the basis for the implementation of the SISAL parallel loop construct. This construct specifies the function to be performed over a set of data and indicates that each iteration of the loop is independent of all other iterations. There are two forms of data partitioning that we have found to be necessary in compiling SISAL programs. If the compiler knows the size N of the data set to be partitioned, the set can be mapped onto the k-cell Warp array in chunks of size N/k. Since N must be known at compile-time, we call this *static data partitioning*.

In general, data set sizes are unknown at compile-time and must be mapped by *dynamic data partitioning*. In this case, the compiler does not know the size of the input data; therefore it must generate code to partition data at run-time. This is achieved by generating code so that each cell in the k-cell array selects every k^{th} element of the input data stream to operate on, while passing on the rest of the data to the other cells in the array.

Data partitioning can only be applied to the outermost of a set of nested loops, so either the programmer or the compiler (in an optimization phase) must ensure that nested loops are ordered such that maximum data-level parallelism is exposed. We now describe our data partitioning methods more fully and illustrate them with an example of an application that can be mapped efficiently onto the Warp array using these techniques.

4.1. Static data partitioning

Static data partitioning attempts to solve the problem of executing the iterations of a loop in parallel. The IF1 construct indicates that each iteration contains no dependencies on other iterations. Therefore, the major difficulty is to map the data elements specified in the generator of the loop (the loop header) onto the cells of the Warp array. The compiler knows the size of the data set, either because the data set is a constant-bounded integer range specified by the programmer or because the programmer has given the compiler a hint as to the size of a particular input array (in the form of a pragma). In SISAL the programmer does not declare the size of arrays explicitly, because arrays are dynamic data structures that can grow and shrink at run-time. However, in the case of input data, the programmer can supply the size of an array in a pragma so that the compiler can perform static data partitioning. A pragma can be either an upper bound or an exact specification of the size of the actual input data. This allows the programmer to make a choice between a SISAL program that can be applied to different sets of input data and a SISAL program that is tailored towards one type of data set but results in more efficient W2 code.

The method for generating static data partitioning is fairly straightforward. For a Warp array of ten cells with a data set of size *N*, static data partitioning assigns the first *N*/10 data items to the first Warp cell, the second *N*/10 data items to the second cell, etc. Figure 4-1 shows the flow of data between the cells.

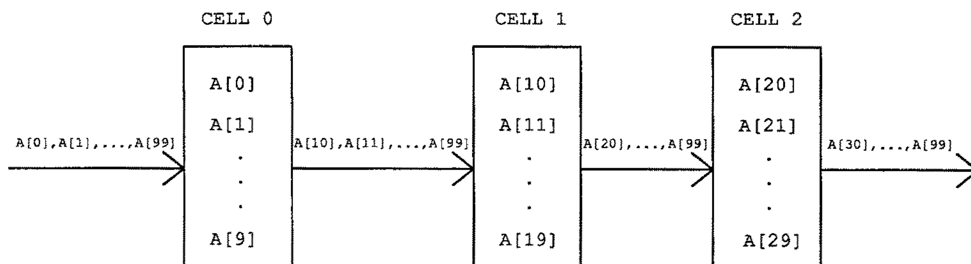

Figure 4-1: Static partitioning

4.1.1. Example: matrix multiplication

Matrix multiplication is an example of a computation that can be easily mapped onto the Warp array automatically using static data partitioning. The SISAL program in Figure 4-2 multiplies two matrices *A* and *B* by first transposing *B* so that its columns can be accessed in parallel (SISAL two-dimensional arrays are accessed in row-major order) and then performing the inner-products of the rows of *A* with the columns of *B*.

4.1.2. Translation

The principle problem for translating parallel loops from SISAL to W2 is to generate the correct communication between Warp cells. These communication operations distribute the data set across the array, and later collect the results. The transposition of *B* in the SISAL program is translated into distributing *B* over the cells of the Warp array by a sequence of W2 "`send`" and "`receive`" statements.

The actual computation is easy to translate; the outer loop of the inner product is performed in parallel on the cells. After the columns of *B* are distributed onto the Warp cells, the program passes the rows of *A* through the array. Each cell accumulates the results and sends the results back to the first cell. Each cell in the array maintains a set of partial sums, one for each column of *B* that has been allocated to the cell. This implements the sum reduction operator in the result section of the inner product inner loop.

As an example, Figure 4-3 shows the result of the translation for the SISAL program in Figure 4-2.

4.1.3. Optimizations

An important optimization can be performed if the results of a computation on statically partitioned data are to be used as inputs to another computation. Since the results of a computation on a statically partitioned data set are already partitioned, we can use this static partitioning to perform further computations. In other words, we do not have to collect the results of a statically partitioned computation until the partitioned data set is completely processed. This optimization saves both computation cycles and communication bandwidth.

4.2. Dynamic data partitioning

Static data partitioning is an effective method for exploiting parallelism from a SISAL program. However, it requires that the compiler determine the exact size of the data set at compile-time. There are many applications for which the data set size cannot be known at compile-time, for example in applications which use SISAL

```
% -------- Matrix Multiplication --------
% Function matmul multiplies NxN matrices A and B and returns
% the resulting matrix. The method is to transpose matrix B,
% so that each column can be accessed as a row vector in a
% for statement, and then do the inner products of each row
% of A with every column of B.

type Mat = array[array[real]] % size = NxN

function Matmul (A, B: Mat returns Mat)
  let
    % transpose B into B_prime
    B_prime :=  for column in 1, array_size(B[1])
                  B_row := for row in B
                             temp := row[column]
                           returns array of temp
                           end for
                returns array of B_row
                end for
  in
    % do inner products of each row of A with all rows of
    % B_prime to compute each row of the result
    for row in A
      result_row := for col in B_prime
                      elem := for j in 1, array_size(row)
                                temp := row[j] * col[j]
                              returns value of sum temp
                              end for
                    returns array of elem
                    end for
    returns array of result_row
    end for
  end let
end function % Matmul
```

Figure 4-2: SISAL matrix multiplication function

streams of indeterminate length. The problem again is to execute the iterations of a parallel loop on the cells of the Warp array in parallel, where there are no dependencies between iterations of the loop. However, we must partition the data set specified in the loop generator at run-time, rather than at compile time. The solution to the problem is what we call *dynamic data partitioning*.

Dynamic data partitioning is similar to static partitioning in that each Warp cell is assigned a set of data items to operate on and then each cell computes in parallel on its local data set. However, the assignment must be done at run-time, which means that the partitioning algorithm must be included in the W2 code compiled from the SISAL program. The method for performing dynamic partitioning works as follows. For a Warp array consisting of k cells, dynamic data partitioning assigns the first cell data items $1, k+1, 2k+1, \ldots$, the second cell is assigned data items $2, k+2, 2k+2, \ldots$, etc. Figure 4-4 shows the flow of data through the cells.

4.2.1. Example

If the programmer removes the pragma declaring that the input arrays A and B are of size $N \times N$ from the SISAL program in Figure 4-2, the compiler cannot partition the data statically. Dynamic data partitioning is necessary to distribute the elements of A and B over the cells. The result of the translation of the matrix multiplication function into W2 is shown in Figure 4-5.

The inner product step of the W2 matrix multiplication program using dynamic data partitioning is exactly the

```
function mult;
begin
  float B_prime[N/10, N], sum[N/10], a_temp;
  int i, k, temp;

  /* up here would be code to divide up the columns of B
       across the cells of the Warp array */

  /* Note that the inner j and k loops have been transposed
       from the usual way of doing matrix multiplication. */
  for i := 0 to N - 1 do begin
    /* initialize row sums */
    for j := 0 to N/10 - 1 do begin
      sum[j] := 0;
    end;
    /* use A value for each element in row */
    for k := 0 to N - 1 do begin
      receive(L, X, a_temp); send(R, X, a_temp);
      for j := 0 to N/10 - 1 do begin
        sum[j] := sum[j] + a_temp * B_prime[j, k];
      end;
    end;
    /* send results of row from cells to left */
    for j := 0 to cid * N/10 - 1 do begin
      receive(L, Y, temp); send(R, Y, temp);
    end;
    /* send results of this cell (row elements for columns
         of B stored in this cell) */
    for j := 0 to N/10 - 1 do begin
      send(R, Y, sum[j]);
    end;
  end;
end   /* function mult */
```

Figure 4-3: W2 matrix multiplication program - static partitioning

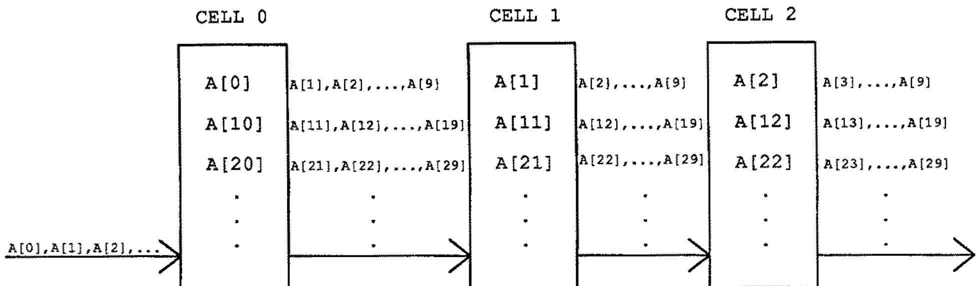

Figure 4-4: Dynamic partitioning

same as for static partitioning (modulo the actual size of the matrices). Dynamic partitioning is used to distribute the *B* matrix data values across the cells of the Warp array. Again, partial sums are accumulated. The inner product step works just as for the static partitioning example, but the loop bounds are determined at run-time by the size of the input arrays. The W2 program uses *while* loops that test both for an *end_of_row* condition and an *end_of_data* condition.

```
function mult;
begin
  /* make N large enough for any desired input matrices */
  float B_prime[N/10, N], sum[N/10], b_temp, a_temp;
  int i, j, k, x, y;

  /* up here would be code to distribute columns of B */
  /* B has x rows and y columns, determined at run-time */

  i := 0; receive(L, X, a_temp);
  while not end_of_data() do begin /* for each row of A */
    for j := 0 to y - 1 do begin   /* initialize row sums */
      sum[j] := 0;
    end;
    /* use A value for each element in row */
    for k := 0 to x - 1 do begin
      receive(L, X, a_temp); send(R, X, a_temp);
      for j := 0 to y - 1 do begin
        sum[j] := sum[j] + a_temp * B_prime[j, k];
      end;
    end;
    /* send results of row from cells to left */
    for j := 0 to cid * y - 1 do begin
      receive(L, Y, temp); send(R, Y, temp);
    end;
    /* send results of this cell (row elements for columns
       of B stored in this cell) */
    for j := 0 to y - 1 do begin
      send(R, Y, sum[j]);
    end;
  end;
end   /* function mult */
```

Figure 4-5: W2 matrix multiplication program - dynamic partitioning

Data are distributed at run-time so that each cell operates on the same number of columns to compute on (or the number of columns differs by a small amount if the number of columns is not an integral multiple of the number of cells). All cells perform the same amount of computation; this arrangement minimizes the total time for executing the entire program. Spreading the computational load across the cells of the Warp array is the key feature of dynamic data partitioning and provides effective utilization of the Warp machine when input data set sizes are not known to the compiler.

4.2.2. Optimizations

There are two optimizations that are required to generate efficient W2 code using dynamic data partitioning. First, we can fold input data distribution, computation, and result collection into a single loop body so that the highly pipelined nature of a single Warp cell can be utilized effectively by the W2 compiler.

Second, since we do not know the size of the input data set at compile-time, we may not be able to store the entire data set for a Warp cell (since it may be larger than the size of the cell local memory). In many programs cells do not have to store the entire data set, since if the W2 send and receive operations are folded into the function computation, each cell can immediately send the result *before* receiving the next data item. This optimization can be used for a SISAL parallel loop only if the dynamically partitioned data is needed for a single loop iteration. For example, in the matrix multiplication example in Figure 4-5, the dynamically partitioned data must all be stored in the cells because each data item is used multiple times in different loop iterations.

4.3. Streams

For the dynamic data partitioning approach, the major difference between streams and arrays is that streams do not allow random access. The basic operations on a stream are reading the first element, removing the first element, and appending an element to the end. This limited set of stream operations can cause some difficulties for a programmer who wishes to use a parallel loop. In many applications it is possible to use the SISAL parallel loop construct to iterate over the elements of a stream, so that the elements of a stream may be dynamically data partitioned and the SISAL program can be executed in parallel on the Warp array. However, some applications require that a loop iterating over the values in a stream use other values in the stream, for example the previous and/or succeeding elements.

Even in applications which iterate over the values in a stream using the SISAL non-parallel (*while*) loop construct, it is often possible to employ dynamic data partitioning. The compiler must note that the only operations on the stream are reading and removing the first element from the stream, and that the loop terminates when the stream ends. If these conditions are met, the compiler may apply dynamic data partitioning to the *while* loop by having the cells in the Warp array select stream elements to operate on as was described earlier. The key is that the program must explicitly save stream elements that it wishes to re-use (for example, with the SISAL **old** directive). This compiler technique will be necessary in performing data partitioning for many applications, since streams are a natural data structure for mapping applications onto a linear array of processors. The ability to detect parallelism from SISAL programs that use streams in the described manner will allow programmers to use streams outside of parallel loops without losing the performance they want to obtain from the Warp machine.

5. Pipelining

In those cases where data partitioning cannot be used, pipelining is another effective way of utilizing the parallelism in SISAL programs. Pipelining partitions the computation so that different stages are done on different cells in the array. Pipelining at this level is unique to Warp-style machines because it requires a high inter-cell bandwidth to transmit large data sets from one cell to another.

Pipelining is effective for programs which require several different stages of computation, where each stage operates on the output of the previous stage. It is also possible to allocate more than one cell of the Warp array to a single stage of a computation, if that is appropriate for the particular application program.

Pipelining attains high performance by overlapping the execution of different stages of an algorithm on the cells of the Warp array. The expectation is that the computation to be performed is large enough (both in terms of data size and computational complexity) so that pipelining stages of the computation is faster than sequentially performing the stages on a single processor. For many applications this expectation is valid, and pipelining the application program onto the Warp machine provides essentially linear speedup.

In addition to being useful in programs which require several stages of computation, where each stage executes a different function, pipelining can also be used to distribute an iterative algorithm across the cells of the Warp array. In this instance, each cell performs one pass of the iterative algorithm on the input data it receives from the cell to its left. To implement more than k iterations (where k is the number of cells in the array), the results are fed back to the first cell. In this case, each cell must test dynamically the termination condition for the iterative algorithm by comparing its input data to its output data.

5.1. An example of pipelining an iterative algorithm

For a simple implementation, it is often adequate for the programmer to fix the number of iterations of an algorithm. Then the compiler can either use the ring configuration described earlier to perform the correct number of iterations, or the compiler can allocate multiple consecutive iterations to each cell (i.e., for *N* iterations in a ten-cell array, each cell performs *N*/10 iterations and then passes data to the cell to the right). The SISAL program for a relaxation algorithm that iterates ten times is shown in Figure 5-1.

```
% --------Relaxation--------
% Function relax applies a relaxation step 10 times to the
% input stream. The relaxation step performs a weighted
% average calculation of each value in the input stream with
% the values preceding and following it. The boundary
% condition is that the left end of the data stream is padded
% with zeroes (by the program, not in the data stream). The
% function returns the stream generated by the last
% relaxation step.

function Relax (a1, a2, a3: real; indata: stream[real]
                            returns stream[real])
  for initial
    data := indata;    i := 1;
  while i <= 10 repeat
    data := for initial
        x_old := 0.0;   x := 0.0;
        x_new := 0.0;   new_val := 0.0;
        data1 := stream_rest(old data);
      while ~stream_empty(data1) repeat
        x_old := old x;   x := old x_new;
        x_new := stream_first(old data1);
        data1 := stream_rest(old data1);
        new_val := a1 * x_old + a2 * x + a3 * x_new;
      returns stream of new_val
      end for;
    i := old i + 1;
  returns value of data
  end for
end function % Relax
```

Figure 5-1: SISAL relaxation function

The pipelined nature of the SISAL relaxation function is easy for the compiler to recognize. The outer loop applies the inner loop function ten times, each time using the result of the previous inner loop. The inner loop applies a simple function to each element of the input data stream, saving the values of previous stream elements needed to compute the output stream. The W2 translation of the SISAL relaxation function is relatively straightforward using the pipelining techniques we have described, and the result is shown in Figure 5-2.

Each cell in the Warp array performs the exact same computation for this particular application, but this is not a requirement for using pipelining to map a computation onto the array.

5.2. An algorithm for applying pipelining

The IF1 dataflow graph provides a convenient form for detecting opportunities to pipeline programs onto a systolic array. Such graphs can be partitioned so that each subgraph maps onto a cell in the array. The major constraint for such a partition is that a node *N* that produces a value for a node *M* must be assigned to either the

```
function mult;
begin
  float a1, a2, a3, x_old, x, x_new, new_val;
  int j;

  /* distribute weights to all cells */
  receive(L, X, a1);  send(R, X, a1);
  receive(L, X, a2);  send(R, X, a2);
  receive(L, X, a3);  send(R, X, a3);
  x_old := 0;  x := 0;  x_new := 0;  j := 0;
  /* perform relaxation step on input stream */
  while not end_of_data() do begin
    x_old := x;  x := x_new;
    receive(L, X, x_new);
    if not end_of_data() do begin
      new_val := a1 * x_old + a2 * x + a3 * x_new;
      send(R, X, new_val);
    end
    else begin
      /* send end_of_data to cells to right */
      send(R, X, x_new);
    end;
    j := j + 1;
  end;
end /* function Relax */
```

Figure 5-2: W2 relaxation program

same processor as node *M* or to a lower numbered processor in the systolic array (i.e., to a processor logically to its left in a left to right array). A partition that satisfies this constraint is called a "legal" partition.

There are two aspects to pipelining a SISAL program: finding a partition of the dataflow graph, and mapping the partition onto the linear array. The mapping step includes generating instructions to transfer data between processors. We have developed a cost model for IF1 dataflow graphs on the Warp machine that allows us to estimate the performance of various partitions on the dataflow graphs. The model takes into account the relative costs of various types of operations on a Warp cell, the presence of loops, and also the benefits of parallel execution of the dataflow graph by overlapping operations on adjacent Warp cells. The following algorithm to find a good partition of an IF1 dataflow graph is based on this cost model.

Algorithm: IF1 graph partitioning

Input: IF1 dataflow graph

Output: Set of IF1 dataflow graphs, with each graph in the set to be executed on a distinct processor in the Warp array.

I. Preprocessing

In a single graph traversal, assign costs to all nodes and edges (recursively for complex nodes) according to the cost model.

For some sample node types *V*, cost(*V*) =

- If *V* is a simple arithmetic operation or simple array/stream operation, 1.
- If *V* is a complex array operation, number of elements of *V*.
- If *V* is an "if" construct, (maximum of costs of alternative subgraphs) + (cost of test subgraph).

- If V is a loop (forall or while), (cost of loop body) × (number of loop iterations).

For an edge E, cost(E) = number of elements being sent on the edge (e.g., 1 for scalars, number of elements for arrays/streams).

II. Partitioning by divide and conquer

- Initialization
    - Set cost_1 = cost of entire graph (sum costs of all nodes in graph, no communication costs since all on one processor).
    - Set cost_2 = $+\infty$.
- Main loop - For all legal partitions of the graph into two subgraphs N_1 and N_2, where nodes in N_1 depend only on nodes in N_1 while nodes in N_2 may depend on nodes in both N_1 and N_2, do
    - Compute cost of subgraph N_1 and of subgraph N_2 (again just by adding up the costs of the nodes in the subgraphs).
    - Find the set of edges crossing from subgraph N_1 into subgraph N_2.
    - Compute cost cost_new of pipelining N_1 and N_2 on two adjacent processors.
    - If cost_new < cost_2, set cost_2 = new_cost and save partition N_1, N_2 (it is the new best partition).
- If cost_2 < cost_1 then recursively partition N_1 and N_2 and return the resulting subgraphs.

  Otherwise, return the unpartitioned graph since executing the graph on one processor has a lower cost.

Unfortunately, the cost estimates cannot simply be applied to *all* possible partitions of an IF1 graph because there are far too many such partitions, even for a graph with only a few nodes. Even if we restrict an algorithm to partition a graph with N nodes into only two subgraphs, there are $2^{N-1}-1$ different partitions to examine. We have therefore chosen a divide and conquer algorithm that attempts to partition a graph into two subgraphs recursively, at each step trying to find a good legal partition into two subgraphs. Note that even only looking at 2-partitions becomes an intractable problem for large graphs, so that it is important to use heuristics to compute the set of legal partitions.

5.3. Example

To illustrate the actions of the partitioning algorithm, we will apply the algorithm to the SISAL program shown in Figure 5-3. The program processes a sequence of input data (for example, samples from a sensor). The program contains three steps: a convolution to compute the average of three consecutive input values, a convolution with a different kernel and a threshold is applied. The program appears to be a good candidate for pipelining onto the Warp array, because of the limited interactions between the data used at each step. The IF1 dataflow graph for this program is shown in Figure 5-4; the nodes are numbered for easy reference. Note that the algorithm will only partition the top-level IF1 dataflow graph.

We trace the behavior of the partitioning algorithm on the IF1 dataflow graph for this SISAL program. First the algorithm assigns fixed costs to all the nodes and edges in the graph. Let N be the number of elements in the input data stream and K the size of the convolution kernel, with $K \ll N$. Then node 1 has cost $6 \cdot N$, node 3 has cost K, nodes 2 and 4 have cost 1, node 5 has cost $4 \cdot K \cdot N + 7 \cdot N$, and node 6 has cost $2 \cdot N$. The costs of the loop nodes 1, 5 and 6 were obtained by computing the costs of the bodies of the loops (IF1 graphs for the bodies are not shown) and multiplying by the number of iterations of the loop. The costs of the edges depend on their types and the node that produced them. All scalar edges (marked with type integer or real) have cost 1. All array edges have cost K, since all are the same size as the convolution kernel. Finally, all the stream edges have cost N, since each loop produces exactly one output data item for each input data item.

```
function Sensor_Data(indata: stream[real]; kernel: array[real];
                     threshold: real returns stream[real])
let
    % first do averaging
    d1 := for initial
        x_old := 0.0; x := 0.0; x_new := 0.0; new_val := 0.0;
        data1 := indata
    while ~stream_empty(data1) repeat
        x_old := old x; x := old x_new;
        x_new := stream_first(old data1);
        data1 := stream_rest(old data1);
        new_val := (x_old + x + x_new) / 3.0
    returns stream of new_val
    end for;
    % then convolve result with kernel
    d2 := for initial
      data1 := d1;
      sum1 := array_fill(1, array_size(kernel), 0.0)
    while ~stream_empty(data1) repeat
      t := stream_first(old data1);
      data1 := stream_rest(old data1);
      t_sum := array_setl(array_addl(array_remh(old sum1),0.0),1);
      sum1 := for i in 1, array_size(kernel)
                conv := t_sum[i] + t * kernel[i]
              returns array of conv
              end for;
    returns stream of sum1[array_size(kernel)]
    end for
in
    % last step is to apply threshold to result of convolution
    for val in d2
        t := if val < threshold then 0.0 else val end if
    returns stream of t
    end for
end let
end function % Sensor_Data
```

Figure 5-3: SISAL program for filtering sensor data

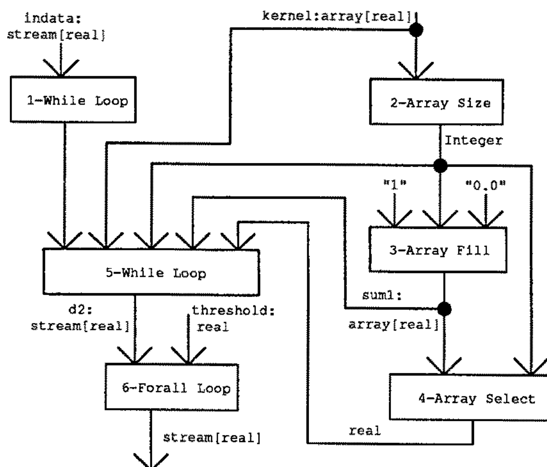

Figure 5-4: IF1 top-level dataflow graph for sensor program

The partitioning algorithm is called first for the complete IF1 graph, with all six nodes. For this graph, the

total cost, obtained by summing up the node costs, is $15 \cdot N + K + 4 \cdot K \cdot N + 2$. Of the legal partitions, the best one puts nodes 1 through 5 in one subgraph and node 6 in the other subgraph. This partition allows the pipelining of stream **d2**; this decreases the cost estimate by N, and adds only a cost of 1 for passing the threshold scalar value.

The next step of the algorithm recursively calls the partitioning algorithm on the subgraph of nodes 1 through 5, and finds seven legal partitions into two subgraphs. Of these, the best one puts node 1 in one subgraph and nodes 2 through 5 in the other subgraph. This partition pipelines stream **d1**, which decreases the cost estimate by N and increases the cost by K to pass the kernel values. Since $N \gg K$, the partitioning costs less than executing the entire subgraph on one processor.

Finally, the algorithm recursively attempts to partition the subgraph consisting of nodes 2 through 5, but all partitions have cost estimates greater than that of the complete subgraph. The resulting partitioning therefore assigns node 1 to the first processor, nodes 2 through 5 to the second processor, and node 6 to the third processor in the array. This partitioning is exactly what a user should expect from a pipelining algorithm, because it clusters local operations together.

6. Concluding remarks

Our research so far has addressed the issue of how to handle the parallelism implicit in SISAL programs. The main sources of this parallelism are the SISAL parallel loop construct, stream operations and pipelining of computation stages. Other issues in compiling SISAL programs for the Warp array require further research. These include dynamic memory management for the Warp cells, graph optimization transformations, automatic transposition of nested loops, and dealing efficiently with recursive functions.

Mapping SISAL programs onto the linear Warp array is an interesting experiment. The high computation throughput of Warp makes it an attractive host for scientific computing. Our goal is to exploit this computation power for the efficient execution of SISAL programs containing sufficient implicit parallelism. The high communication bandwidth between the cells in the linear array allows the translator to use data partitioning and pipelining. The large local memory provides additional flexibility in selecting an efficient mapping. With a complete implementation, we will be able to investigate the efficiency of applicative programs on the Warp machine for a wide range of application domains.

Acknowledgements

We appreciate the assistance of Steven Skedzielewski and the SISAL group at LLNL in providing the SISAL to IF1 translator.

The research was supported in part by Defense Advanced Research Projects Agency (DOD), monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, and Naval Electronic Systems Command under Contract N00039-85-C-0134, and in part by the Office of Naval Research under Contracts N00014-80-C-0236, NR 048-659, and N00014-85-K-0152, NR SDRJ-007.

Warp is a servicemark of Carnegie Mellon University; UNIX is a trademark of AT&T Bell Laboratories.

References

1. Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M. S., Menzilcioglu, O., Sarocky, K., and Webb, J. A. Warp Architecture and Implementation. Conference Proceedings of the 13th Annual International Symposium on Computer Architecture, IEEE/ACM, June, 1986, pp. 346 - 356.

2. Annaratone, M., Bitz, F., Clune E., Kung H. T., Maulik, P., Ribas, H., Tseng, P., and Webb, J. Applications and Algorithm Partitioning on Warp. Proc. Compcon Spring 87, San Francisco, February, 1987, pp. 272-275.

3. Annaratone, M., Arnould, E., Hsiung, P.K. and Kung, H.T. Extending the CMU Warp Machine with a Boundary Processor. Proceedings of SPIE Symposium, Vol. 564, Real-Time Signal Processing VIII, Society of Photo-Optical Instrumentation Engineers, August, 1985.

4. Bruegge, B., Chang, C., Cohn, R., Gross, T., Lam, M., Lieu, P., Noaman, A. and Yam, D. Programming Warp. Proc. Compcon Spring 87, San Francisco, February, 1987.

5. Gross, T. and Lam, M. Compilation for a High-performance Systolic Array. Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, ACM SIGPLAN, June, 1986, pp. 27-38.

6. Gurd, J. R., Kirkham, C. C., and Watson, I. "The Manchester Prototype Dataflow Computer ". *CACM 28*, 1 (Jan 1985), 34 - 52.

7. Kung, H. T. "Memory Requirements for Balanced Computer Architectures". *Journal of Complexity 1*, 1 (1985), 147-157.

8. McGraw, J. R. "The VAL Language: Description and Analysis". *ACM Trans. on Programming Lang. and Systems 4*, 1 (Jan. 1982), 44 - 82.

9. McGraw, James, Skedzielewski, Stephen, Allan, Stephen, Oldehoeft, Rod, Glauert, John, Kirkham, Chris, Noyce, Bill and Thomas, Robert. SISAL Streams and Iterations in a Single Assignement Language. Tech. Rept. M-146 (Rev.1), Lawrence Livermore National Laboratory, March, 1985.

10. Oldehoeft, R.R., Cann, D.C. and Allan, S.J. SISAL: Initial MIMD Performance Results. Proceedings of CONPAR 86 (Conference on Algorithms and Hardware for Parallel Processing), September, 1986, pp. 120-127.

11. Skedzielewski, S. K., and Welcome, M. L. Data Flow Graph Optimization in IF1. Proc. 1985 Conference on Functional Programming Languages and Computer Architecture, Nancy, Sept., 1985, pp. 17 - 34.

12. Skedzielewski, Stephen and Glauert, John. *IF1: An Intermediate Form for Applicative Languages*. Lawrence Livermore National Laboratory, 1985.