Static Mapping of Functional Programs: An Example in Signal Processing

Jack B. Dennis
MIT Laboratory for Computer Science
Cambridge, MA, 02139

Abstract

Complex signal processing problems are naturally described by compositions of program modules that process streams of data. In this paper we discuss how such compositions may be analyzed and mapped onto multiprocessor computers to effectively exploit the massive parallelism of these applications. The methods are illustrated with an example of signal processing for an optical surveillance problem.

The method illustrated involves program transformation and analysis to construct a program description tree *that represents the given computation as an acyclic interconnection of stream-processing modules. Each module may be mapped to a set of threads run on a group of processing elements of a target multiprocessor. We estimate the parameters of performance that could be realized for two forms of multiprocessor architecture, one based on conventional DSP technology and one based on a multithreaded processing element architecture. For our example, we conclude that the multithreaded architecture offers advantages in latency of results and in memory requirements, as well as in throughput.*

1 Introduction

An important goal toward making parallel computers more useable for practical computations is to provide compiling technology that is able to convert algorithms expressed directly and simply in a high level language into efficient machine code. The parallelism implicit in the expression of the algorithm must be identified and exploited by the compiler. Complex signal processing problems are naturally described by compositions of program modules that process streams of data. We use an example to illustrate how such compositions may be analyzed and mapped onto multiprocessor computers using extensions of methods used in the Paradigm compiler designed and implemented by the author [Dennis 1989].

We begin by discussing how the mapping of compositions of stream-processing modules differs from the application of data parallel principles in mapping scientific computations onto massively parallel computers. In the domain of high performance signal and image processing, applications can exploit massively parallel computation, but the form of parallelism present is not the data parallel form encountered in scientific computations: (1) Program modules often work together in producer/consumer relationships, allowing concurrent operation; and (2) All modules of the program are continuously active processing streams of data. The use of stream data types plays a central role in expressing such computation in a high level form that permits automatic analysis. We illustrate an approach to mapping such computation by analyzing a typical processing computation for image data arriving from a sensor array. We indicate how the computation (when expressed in the Sisal functional programming language) may be analyzed and its structure represented in a *program description tree*, and used to guide the construction of code for a target multiprocessor. We discuss the problem of finding an optimal mapping, and discuss the structure and performance of constructed code for two choices of multiprocessor architecture.

2 Static Mapping

The problem of implementing programs written in high level languages on parallel computers may be approached in two fundamental ways according to the philosophy of managing processing and memory resources. One may strive to implement a very general model of parallel computing and implement it by a suitable combination of architectural features and run-time services so that all scheduling and memory allocation decisions are performed during program execution. This general approach is exemplified by the Monsoon mutiprocessor [Papadopoulos/Culler 1990], but the mechanisms have not evolved to the level of efficiency required to attract practical usage. The second approach is based on making most memory management decisions at compile time. This can yield very efficient exploitation of multiprocessors built of conventional processors for computations having a suitable regular structure. This second approach has been the basis for the development of the data parallel model and its implementation in such work as the Thinking Machines Fortran compiler, [Sabot 1992] the definition of High Performance Fortran and advanced work in Prof. Kennedy's group at Rice University.

The data parallel approach can also be followed for programs expressed in functional programming languages with significant advantages. It is simpler to identify the program blocks that are suitable for data parallel implementation, and the global program analysis needed to determine optimum alignment and mapping for the arrays of a program is more readily accomplished. This is because functional language programs do not make use of side effects, and each use of any data definition is readily identified. This has been done in the Paradigm compiler [Dennis 1989] designed and built by the author for the Sisal language and targeted for the CM-2 Connection Machine.

2.1 Compiler Structure

The Paradigm compiler was designed to identify the principal data structures constructed by a program through global compile-time analysis, and to map these structures onto the processing elements of the target machine. The structure of the compiler is shown in Figure 1. It consists of a conventional Front End that parses and checks source language modules, an Analyze module that identifies code blocks in the program, and a Code Constructor that implements each code block on the basis of mapping specifications derived with optional advice from the user [Dennis 1988, Dennis 1989].

Our goal requires some departure from the typical structure of programming language support systems. Efficient machine code programs for large-scale parallel computers can be generated only if the compiler is able to consider the entire collection of program modules involved in a job in making decisions regarding how the computation should be mapped onto the target machine. This implies that the linking of program modules should be accomplished prior to the compiler's analysis and optimization decisions. A second change is more fundamental: instead of carrying

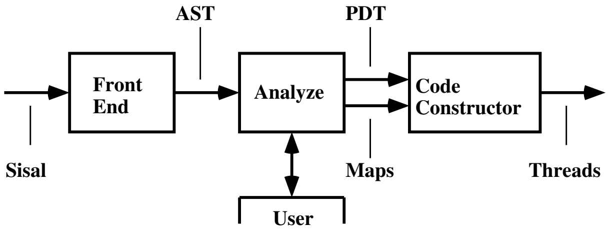

Figure 1: Structure of the Paradigm compiler.

out optimization as a sequence of independent steps, each of which supposedly leads to an "improvement" of the code, we perform an analysis of the given code, determine the best mapping strategy, then synthesize machine code according to the specified mapping.

The Sisal functional programming language [McGraw 1985] is particularly attractive for implementing this approach. The absence of global variables and the clear differentiation of arguments and results of function modules in the Sisal language make it easy for a compiler to analyze source programs and identify the parts of the code that define the major data structures. We call these parts of the source language program *code blocks*.

The data structures appropriate for scientific computation are large multi-dimensional arrays of numerical data. Each code block defines an array value and represents a computation that may be spread over the processing elements of the machine according to a chosen assignment (or mapping) of array elements to processing elements. This is the essence of data parallel computation. The parallel iteration expression of the Sisal language provides a convenient high level notation for writing data parallel algorithms.

2.2 Application to High Performance Signal/Image Processing

Another area that can exploit massively parallel computation is high performance signal and image processing. In these applications large amounts of parallelism exist, but it takes different forms: (1) producer/consumer concurrency: the possibility of executing two program modules concurrently when one (the *consumer*) processes a stream of data generated by the other (the *producer*); and (2) Simultaneous application of several instances of functions. The use of stream data types plays a central role in expressing such computations in a high level form that permits automatic analysis. The rest of the paper is devoted to describing this process, illustrating its application to a practical signal processing problem, and studying the performance achievable for two multiprocessor architectures.

3 An Example: Optical Surveillance

The computation we have chosen to illustrate the proposed mapping strategy is derived from a collection of procedures for processing information from a sky-scanning optical surveillance device and detecting objects in its field of view. The application has similarities to radar signal processing. There are many sensors, several for each line of the scanned image. These signals are conditioned, smoothed and down-sampled before a two-dimensional filter is used to suppress unimportant detail. A peak detection algorithm identifies points in the image that should be analyzed further as potential objects to be reported. A bock diagram of the computation is shown in Figure 2.

Each module in the diagram may be characterized as a function that transforms a stream of input data into a stream of output data. Hence it is natural to specify them using a language (Sisal) that includes streams as standard data types and supports analytic and constructive operations on streams.

In Sisal a *stream* is a sequence of values which may be infinite (unending). A stream of integers is a natural representation for a signal that has been converted into digital form. Interconnecting modules that process streams of data is a powerful means for combining program parts to build larger modules and is well matched to the needs of signal processing tasks. Thus the combination of processing modules shown in Figure 2 may be expressed in Sisal as the composition of functions in Figure 3. The Sisal code for the five component functions is given in the appendix of this paper.

This use of function composition for signal processing has been discussed in [Dennis 1995], where we showed how to transform tail-recursive functions on streams into non-recursive dataflow graphs that may be executed efficiently by suitable fine-grain parallel computers [Dennis/Gao 1994, Dennis 1991] The use of dataflow graphs as a natural means for specifying signal processing applications has also been studied in [Ho/Lee/Messerschmitt 1988], and the idea of compiling signal processing programs from block diagrams was described as early as [Kelly/Lochbaum/Vyssotsky 1961].

From this example we see that complete signal processing tasks may take the form of a set of processing

```
%Type Declarations

type Signal = stream [integer];
type ImageStream = array [ stream [integer] ];
type DataStream = stream [ array [ integer ] ];
type MarkStream = array [ stream [boolean] ];

%The Top-Level Function

function Process (
   D: DataStream;
   w: integer;
   n: integer
   returns MarkStream )

   let
      R := for i in 1, w
      returns array of
         let S := for j in 1,n
            returns array of
               BaseRemove (D[i,j])
         Nyquist ( SpikeAdapt ( S[i], n ) )
      end for
   in
      PeakDetect ( TwoDimFilter ( R, w ) )
   end let

end function
```

Figure 3: Type declarations and the principal function `Process` for the optical surveillance computation, written in Sisal as a composition of stream-processing functions.

modules, each generating a stream of values that is passed to other modules for further processing. Thus the overall computation may be described by a directed acyclic graph in which the nodes are stream processing modules such as those we have presented, and each link indicates a producer/consumer relationship between a pair of modules. It is well-known that such interconnections of modules may lead to deadlock if the graph contains (undirected) cycles, and the temporary storage for stream elements in each link is bounded in capacity. Given the structure of stream processing programs expressible in Sisal, a compiler can detect these situations and warn the user of the deadlock possibility.

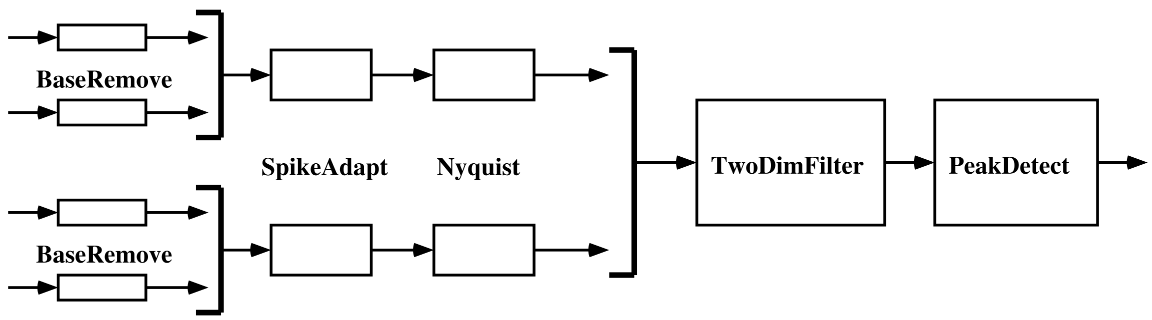

Figure 2: Structure of the optical surveillance data processing algorithms.

4 Program Analysis

In this paper we consider only programs having an overall structure that supports the continuous processing of streams of data. In these programs each module operates on data streams, produces a data stream, and runs continuously during program execution. The overall structure of such programs is an acyclic interconnection of such modules. This is in contrast to data parallel scientific codes for which the original Paradigm compiler was designed. There the top level program structure is a main loop in which the loop body is an acyclic combination of code blocks that define array values, as in the following program segment

```
Z: array [real] :=
  for i in 1, n
    Y: real :=
      if i = 1|i = n
      then X[i]
      else 0.5 * ( X[i-1] + X[1+1] )
      end if
  returns array of Y
  end for
```

This parallel expression in Sisal defines an array value `Z`, each (internal) element of which is the average of the two adjacent elements of a given vector `X`. The conditional expression provides special treatment of the end elements of `Z`. All instances of the body expression may be evaluated concurrently. In general, data parallel code blocks may be nested "for" expressions that define multidimensional arrays, and may include reduction operations that apply associative operators over specified dimensions of the defined array. In the Paradigm Compiler [Dennis 1989], programs having this structure are analyzed and transformed into data parallel programs for the CM-2 Connection Machine.

4.1 Stream-Processing Programs

The present study explores the prospects for static analysis and mapping of continuous stream-processing computations such as the optical surveillance problem. Thus we envision a new version of the Paradigm Compiler that will transform, and analyze such programs and generate machine code for multiprocessor computers. Given a program that is amenable to static resource management, the Analyze module of the rebuilt Paradigm Compiler will provide program descriptions that may be used to plan the mapping of the program onto a parallel computer and to construct code in the target machine language.

The job of program analysis has several parts:

1. Identify the program modules (code blocks)
2. Check the conditions that permit static mapping to be used
3. Extract parameters for each program module for use in performance estimation
4. Determine the relative computation rate for each module
5. Construct a program description tree containing the results of analysis

4.2 Program Transformation

The identification step includes examining recursive function definitions to determine whether they are tail-recursions and have equivalent iterative dataflow graphs. A method for doing this has been given in [Dennis 1995].

In the absence of conditional expressions in their bodies, the tail-recursive function definitions express functions that process input streams into output streams where the numbers of output elements emitted is related to the number of input elements absorbed as a ratio of integers, a rational number. If the bodies of these function definitions contain conditional expressions, it may be that the module does not have a fixed ratio of output elements emitted to input elements absorbed, and only a range of values for the ratio can be determined through static analysis. Such situations appear to be rare in practical signal processing computations, for their existence would imply a non-uniform sampling rate. In our example, the `BaseRemove` function contains a conditional, but yet has a fixed input/output ratio of unity. Given these ratios (or bounds on relative rates) a rate (or range of rates) can be calculated for every module.

Figure 4 illustrates the results of program transformation performed by the compiler. It is a dataflow graph that represents the continuous iterative processing of a stream of data arrays by the `TwoDimFilter` module. The **Group** operator at the top of the figure extracts successive groups of three elements from the input stream. Each of these elements is a w-element array. On the next cycle of operation, the selected group starts with the element one position later in the input stream. Between the square brackets is the conventional dataflow graph of body of the **ArrayProducer** code block. It is not presented in fine-grain form to avoid confusing detail. The brackets themselves represent the **ForAll** code block that defines each array element of the output stream. The opening (top) bracket is labeled with the range of indices for elements in the generated array. The **EmitStreamElement** operator simply appends its input array onto the output stream.

We assume that the usual architecture-independent optimization steps—constant folding, common subexpression elimination,etc.—have been performed. Also, and this is especially important in signal processing applications, shift operations are used to implement multiplication by known constants whenever this is more efficient (although this depends on the detailed design of the arithmetic units of the processor).

4.3 The Program Description Tree

The objective of program transformation and analysis is to provide information on which the choice of a good mapping of program modules and data structures onto a target multiprocessor may be made. We represent the results of analysis as a *program description tree* or PDT. Each node of the PDT corresponds to a syntactic element of the program analyzed. The node types include **Graph** nodes that represent acyclic interconnections of primitive operators and code blocks, and node types that represent the several kinds of code blocks. A **ForAll** node describes a code block that determines an array value each time it is executed. A **StreamProducer** node describes a code block that produces an endless stream of simple scalar elements, and an **ArrayProducer** node describes a code block that produces an endless stream of array values. The nodes of the PDT contain data type information and the index ranges for constructed arrays. **Graph** nodes contain counts of operators of various kinds, and could also contain information about degree of parallelism such as the critical path length of the graph.

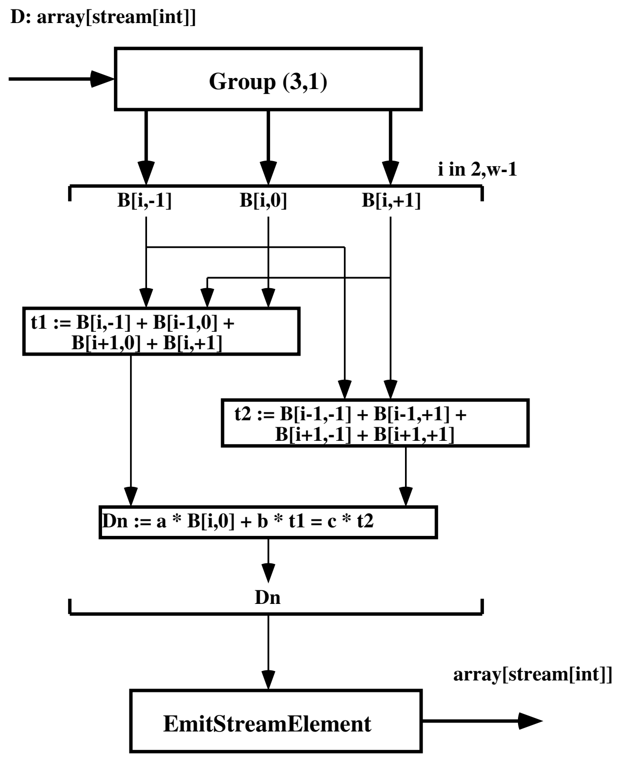

Figure 4: Transformed program module for the `TwoDimFilter` function.

4.4 Analysis of the Example

The optical surveillance program satisfies the conditions for static mapping. Each recursive function definition is tail-recursive and may be transformed into an iterative dataflow graph with a fixed memory requirement for each top-level invocation. Moreover, each of the resulting transformed modules maps one or more input streams into an output data stream, and the program in entirety is an acyclic composition of these modules, as specified by the top-level function `Process`. The program description tree for the optical surveillance example is shown in Figure 5.

4.5 Computation Rate and Load Estimate

The program description tree contains sufficient information to determine the relative computation rate for each program module, and the approximate fraction of the total computation load each module is responsible for. These data are calculated for the optical surveillance problem in Table 1. We (arbitrarily) take the processing of one array of data by `TwoDimFilter` (or by `PeakDetect`) as the basic *compute cycle* of the computation. For each program module, the table shows the number of operations performed in each execution of a module, the width of the data stream processed by the module, and the number of instances of execution of the module for one compute cycle. These yield the operation count per cycle and the load fraction for each module. These data are used in Section 6 to estimate the computation rate and latency for selected mappings of the example.

5 Mapping Plans and Strategies

In this section we discuss reasonable choices for mapping continuous processing applications to multiprocessor computers, and discuss the problem of finding the best mapping plan.

5.1 Mapping Plans

In contrast to data parallel scientific computing, the strategy of letting one program code block at a time utilize the whole machine appears to be a poor choice for continuous processing applications. Rather, in many cases the best approach is to structure the machine code so all program modules are executing concurrently at a rate the meets exactly the computation requirement[1].

[1] In current practice, a single processor is often multiplexed among program modules for different stages of processing, but because coarse-grain processing must be used to attain economical performance with conventional processors, large buffers for intermediate data must be used and high latency of results occurs. With multiprocessing, assigning different modules to distinct processors will usually yield better resource utilization.

Given an estimate of the load to be handled by each code module, we must decide how many processing elements should be actively executing each module. For each program module, two reasonable mapping possibilities are apparent:

1. **Allocate:** Assign to the program module the exact number of processing elements needed to achieve the overall computation rate the module requires.

2. **Distribute:** Spread the computation load of the module uniformly over all processing elements

The choice between the **allocate** and **distribute** strategies may be made independently for each module, but those processing elements dedicated to program modules for which the **allocate** strategy is chosen are not in the set over which the work of the remaining modules may be spread. Which choices lead to better performance depends on the relative amounts of inter-module communication and intra-module communication, and on how well the loads match up with processing element capacities.

Note that if the computation rate demanded by some program module requires the performance of several processing elements, then the program module must offer sufficient opportunities for concurrency that the processing elements can be fully utilized. Otherwise the computation is not feasible on the target multiprocessor.

Where the input of a module is an array of streams, a plan in which the modules producing the individual streams are executed by the same processing element as that assigned to the corresponding part of the array-processing module is likely to perform better by avoiding some communication cost.

An advantage of assigning a limited number of processors to chosen modules is that it is then not necessary to load all program modules into (or make them accessible from) every processor.

In data parallel scientific computation, a major issue is aligning the distribution of various data arrays so as to minimize communication. In continuous processing computations, this issue has less impact. On the other hand, it is beneficial to distribute the work of the `TwoDimFilter` and `PeakDetect` modules over processing elements in aligned fashion.

5.2 Mapping Plans

On the basis of the above considerations we propose the following class of mapping plans for continuous

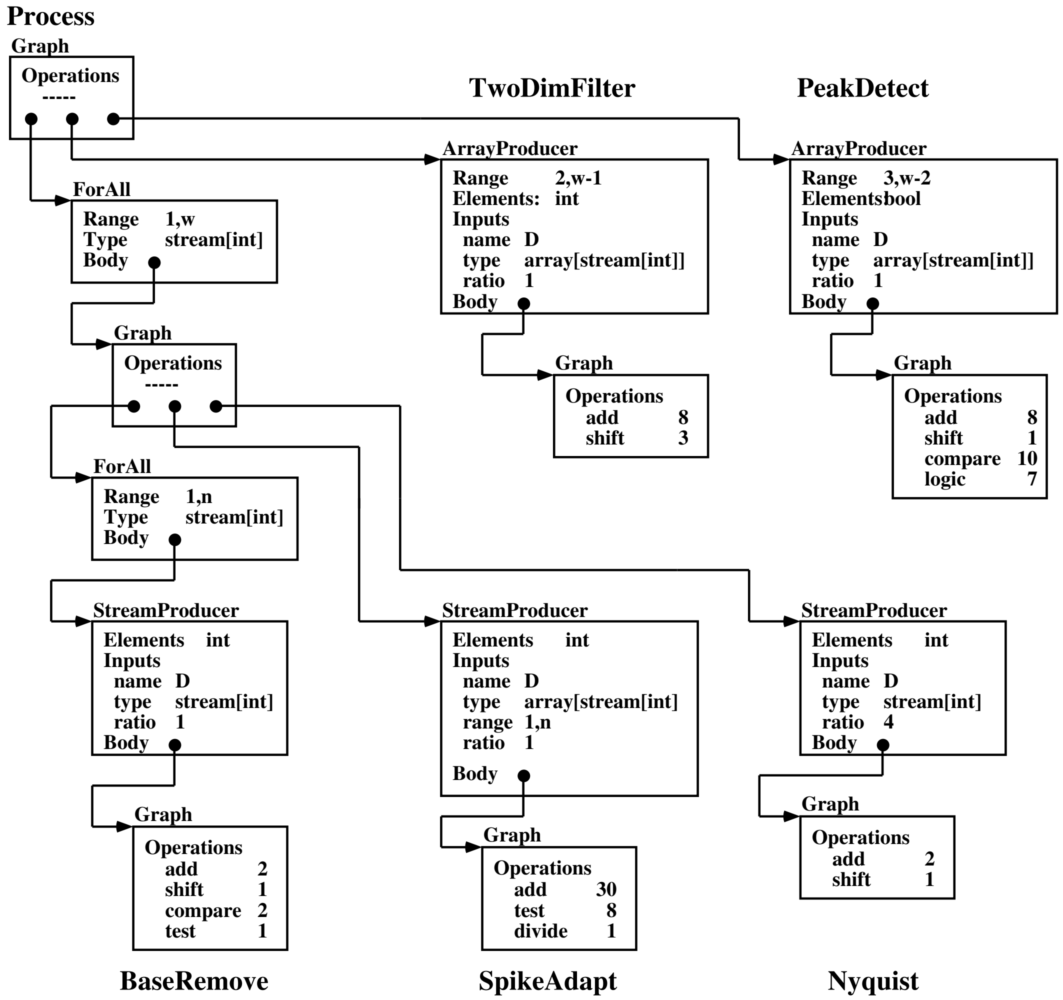

Figure 5: Program description tree for the signal processing example.

processing programs: A *mapping plan* is a specification for each node of the program description tree as to whether execution of the program section described by the subtree is to be evenly distributed over all processing elements (**distribute**), or is to be executed by a group of dedicated processing elements (**allocate**) sized to accommodate the estimated load of that program section.

Table 2 gives three reasonable proposals for mapping the optical surveillance computation. Under Plan A, the work of each of the five modules is distributed over all processing elements. This choice is attractive because it eliminates all inter-processor communication except that due to boundary exchange in algorithms `TwoDimFilter` and `PeakDetect`. In Plan B, a group of processors is dedicated to the work of algorithms `TwoDimFilter` and `PeakDetect`, but these are both distributed across the group to avoid communication costs for passing data from `TwoDimFilter` to `PeakDetect`. The relative merits of these plans are discussed in the following section.

5.3 Finding the Optimal Mapping Plan

Given a mapping plan and characteristics of the target multiprocessor including the number of processing elements, it is straightforward to estimate performance parameters for a mapping plan. Given the total operation count, the number of processors, and their speed, the rate of computation may be estimated. The costs of process synchronization may be approximated from characteristics of the target architecture

Module	Rate	Operations	Width	Count	Total	Fraction
BaseRemove	4	6	1	nw	49152	0.353
SpikeAdapt	1	39	n	w	79872	0.574
Nyquist	1	3	0	w	768	0.006
TwoDimFilter	1	11	w-2	1	2794	0.020
PeakDetect	1	26	w-4	1	6552	0.047
Total					139138	1.000

Table 1: Calculation of load fraction for each program module from data in the program description tree. The total operations and load fractions are calculated for $w = 256$ and $n = 8$.

and program structure information in the description tree. The mapping plan induces a communication load from which it can be estimated whether the computation is compute bound or communication bound.

Thus the following approach should help find the optimum mapping plan:

1. Determine computation rates and load parameters for each node of the graph.
2. Generate several plausible mapping plans based on the given program description tree and estimates of performance parameters.
3. Evaluate each proposed mapping plan by constructing target machine code and determining accurate processor and communication loads
4. Select the best mapping plan for the user's objective.
5. Construct the final machine code.

6 Multiprocessor Performance

Given the three mapping plans proposed above, let us consider their use in code construction for multiprocessor computers. First we introduce the two processor architectures we use as contrasting targets for parallel computing. Then we discuss the machine code structures appropriate to implementing continuous processing applcations such as the optical surveillance computation. We discuss the performance differences among the three plans and point out the tradeoffs possible among throughput, memory, and latency of output data.

6.1 Architectures

We consider two contrasting multiprocessor architectures. One uses processing elements of conventional architecture with features intended to support efficient multiprocessor computation. We designate this the CVA (ConVentional Architecture) machine. A commercial example of such a processing element is the Texas Instruments TMS320C3x digital signal processor. This machine has high single-thread performance, but makes only modest concessions to supporting fine-grain synchronization and communication for efficient parallel computing. In this architecture, a thread is created by a `fork` command or a parallel function call interpreted by run-time software, and may terminate at a `join` command or by execution of a `quit` command. A thread may be suspended to wait for some event to occur, and it may be preempted to allow the processor to handle interrupt events or to schedule a thread having a higher priority.

The second architecture uses a hypothetical processing element having an interleaved multithreading architecture, as proposed in [Dennis/Gao 1994, Dennis 1989]. We designate this one the MTA (MultiThreaded Architecture) machine. In this processor there may be four active threads that share resources (functional units, registers, and access to local memory). Threads are non-preemptible, so execution of a ready thread is delayed until one of the four pipeline slots is released by termination of an active thread. A thread becomes ready for execution when it is signaled from other threads, or when a message arrives from another processor. A thread uses a small number of registers to pass results from one instruction to a later instruction of the thread; registers are undefined when a thread becomes active and are not saved at thread termination. A typical thread will either send signals to activate other threads, or send an interprocessor message just before terminating by executing a `quit` instruction. In the MTA a thread is a sequence of instructions fixed at compile time and is short enough that other threads may be executed soon enough to meet performance requirements.

<table>
<tr><td>Module</td><td>BaseRemove</td><td>SpikeAdapt</td><td>Nyquist</td><td>TwoDimFilter</td><td>PeakDetect</td></tr>
<tr><td>Plan A</td><td colspan="5">distribute</td></tr>
<tr><td rowspan="2">Plan B</td><td colspan="5">allocate</td></tr>
<tr><td colspan="3">distribute</td><td colspan="2">distribute</td></tr>
<tr><td rowspan="2">Plan C</td><td colspan="5">allocate</td></tr>
<tr><td colspan="3">distribute</td><td>distribute</td><td>distribute</td></tr>
</table>

Table 2: Three mapping plans for the optical surveillance problem.

Other multithreaded architectures have supported eight active threads to allow tolerance of long memory accesses. For the present discussion we assume that multiplexing four threads in the computation pipeline is sufficient to tolerate the latency of accesses to local memory and to fill pipeline gaps due to intra-thread dependences.

To compare the two architectures for our example, we will assume that both are able to achieve the same total instruction processing rate. This means that one thread on the MTA will run one fourth as fast as a single thread on the CVA. (This is unfair to the MTA because the CVA will be slowed more by pipeline hazards.)

With respect to implementing the mapping plans for the optical surveillance problem, the differences that affect the code structure needed to get best performance are the following:

1. A CVA processor can be fully utilized by a single thread. For the proposed MTA machine, four threads are needed to fully utilize the processor.

2. Switching between threads is more expensive for the CVA, so long threads are favored. The fast switching of the MTA processor allows short threads to be used, permitting more parallelism to be exploited.

3. Sending and receiving overhead is very low in the MTA machine, so very short messages may be handled efficiently.

6.2 Machine Code Structure

In the MTA the low cost of threads allows the machine-level program structure to reflect the concurrency structure of the algorithm being implemented. In the case of the CVA, it will be advantageous for high throughput to unroll loops to obtain long threads, and to block data into long messages to amortize message-passing overhead.

For both architectures, it better to run long threads because starting and terminating threads has a non-zero cost in both processors. Since we are assuming that local memory accesses do not cause pipeline gaps, the only events that benefit from thread switching are synchronizations with data arriving in messages from other processors, and to provide sufficient multiplexing of module operation to meet latency and throughput requirements of the application.

6.3 The Example

First we determine the number of processors needed to perform the computation at the desired rate. To do this we estimate the number of instructions needed to perform one compute cycle, multiply by the desired rate and divide by the performance of the processing element.

From Table 1 we see that 139,138 operations per compute cycle are needed. Allowing an equal number of data movement and miscellaneous instructions, and allowing and additional 25 percent for overhead of scheduling and communication, we find that the total instructions per cycle will be about[2].

$$I_{cycle} = 139138 \times 2 \times 1.25 = 347,845$$

using the desired rate of 2.5 kHz., we find that the total instruction rate must be at least

$$R_{instr} = 869\ MIPS$$

This rate could be met by 18 processors at 50 MIPS each, so to be generous, let us assume a machine with 20 processors.

Of the proposed mapping plans, Plan C has the greatest communication load because communication is needed to pass the entire data stream between each of two pairs of program modules (as well as a small

[2] These are estimates made without careful analysis. We plan to carry out a more detailed evaluation.

amount of intra-module communication). The communication rate required will be $2w = 512$ words per compute cycle, or 1.28 million words per second. This is less than 0.2 percent of the required instruction rate and is far below the capacity of typical interconnection networks. This is indeed an embarrassingly parallel computation, and is compute bound for all three mapping plans.

There is one more issue to discuss before we consider the mapping plans separately. A major challenge for this computation is handling the large number of high volume input data streams. In each case we assume that input data from the sensors is made available to the multiprocessor in blocks of eight values for each "channel" of data processing. This means processing one interrupt by the CVA or one synchronization event for the MTA for each channel on every minor cycle of operation. Similarly, the results of processing are delivered to the user as blocks of 16 16-bit words containing the (Boolean) peak data for one cycle of processing. Handling the output stream is a very minor problem, but the rate of input data stream events is

$$R_{inp} = 4 \times w \times 2.5 = 2.56\ MHz$$

or 128,000 input event per second for each processing element. If each input event is handled in the CVA machine by processing and interrupt and scheduling a thread, the overhead cost will be high. In the MTA machine, the corresponding cost is that of synchronizing thread initiation with an input event, which amounts to just a few processor cycles.

Plan A:

Under Plan A, computation by each of the five program modules is distributed over all 20 processing elements. If each processor performs the work associated with $256/20 = 13$ channels of data, the only interprocessor communication will be to support the boundary references in the `TwoDimFilter` and `PeakDetect` modules. As we have already noted, this communication load is very small.

In the CVA machine, a single high-priority thread may perform the 32 executions of `BaseRemove`, four executions of `SpikeAdapt`, and one execution of `NyquistFilter` for each data stream in each compute cycle. There will be a substantial cost associated with synchronizing the start of this thread with the arrival of $4 \times 13 = 52$ blocks of sensor data for each compute cycle. Separate lower priority threads may be used to perform the `TwoDimFilter` and `PeakDetect` computations when signaled by arrival of messages containing boundary data.

In the MTA machine, many threads may be employed without thread switching overhead becoming significant. One attractive structure is to use a separate thread to perform the work of the three front-end modules for each data channel. Each of these threads would contain 351 operations, which is sufficiently short that responsiveness of processing will not be affected. One thread apiece will serve to perform the `TwoDimFilter` and `PeakDetect` computations after synchronizing with interprocessor messages.

The *latency* of processing is the time interval between arrival of input data and the availability of output data that depends on it. Some of the processing steps of the optical surveillance example have a built-in delay of from one to three operation cycles. Additional latency is introduced in the machine program by overhead costs and because once operations are performed additional work is done before the consumer of results is scheduled or signaled to begin operation. In this respect, the MTA machine has the advantage because its finer granularity of processing allows successor threads to be signaled sooner than is feasible to schedule them in the CVA machine. This is partially compensated by the property that threads execute four times faster in the CVA.

Plan B:

In Plan B a two processors would perform all computation for `TwoDimFilter` and `PeakDetect`. Because there would be only two sections of the data stream, message traffic for intra-module communication would be smaller. Instead, the entire data stream passing from Nyquist to TwoDimFilter would have to be carried in interprocessor messages. Handling this data stream on a word by word basis would involve a large overhead for the CVA machine (more cycles than needed to execute the `TwoDimFilter` algorithm), but would be a relatively minor amount for the MTA (ten percent or less). Although the higher communication load for this plan would not overload a typical network, there is no compensating saving because the intra-module communication need is so low, and the plan has the disadvantage of introducing unbalanced use of parts of the network. Under this mapping plan, there would be good opportunity to improve performance of the CVA by passing data in large blocks between stages of the computation, however this would increase the latency of results and require large data buffers.

Plan C:

Plan C takes the further step of executing `TwoDimFilter` and `PeakDetect` algorithms on separate groups of processors, further increasing the communication load without compensating benefits.

6.4 Discussion

The principal difference between the two architectures is in the cost of synchronization, which also reflects a difference in the handling of global memory access. (One may regard the communication performed to implement access to boundary values of the data array in TwoDimFilter and PeakDetect as instances of a general mechanism for global memory access.) The effect is greater in computations that can benefit from short threads.

The impact of this cost on performance of the CVA may be mitigated by several standard techniques, namely breaking the data stream up into blocks of sufficient length that the start-up cost for sending and receiving messages is acceptably small. The penalty is longer latency of results and increased amounts of memory needed to buffer blocks of data between processing stages.

In the calculation of performance it is also necessary to check that the performance is actually achievable, that is, that there is sufficient parallelism that no processing element is ever starved for work. This may be done using Petri nets to represent the dynamic behavior of the scheduling of threads, but is beyond the scope of this paper.

7 Conclusion

We have discussed how signal/image processing programs written in the Sisal functional programming language can be transformed and mapped onto multiprocessor computers. Our approach to program analysis and mapping involves the following steps:

1. Transform the program into an acyclic graph of stream-processing program modules
2. Determine relative computation rates and load parameters for each program module.
3. Choose plausible mapping plans
4. Determine performance characteristics for each mapping plan and select the best for the user's objective.
5. Construct the machine code.

We have discussed application of the method to an optical surveillance problem, and discussed program mapping plans suitable for two target multiprocessor architectures: a multiprocessor built of conventional processing elements and a hypothetical multiprocessor built of multithreaded processing elements. We suggest that, by offering lower scheduling and synchronization costs, the multithreaded architecture has the ability to support efficient fine-grain computation, leading to lower end-to-end latency and decreased memory requirement for intermediate data for the studied application. Another architectural variant that offers an intermediate choice for multiprocessing between conventional processors and the MTA machine discussed here is the *threaded abstract machine* [Culler 1991].

In the computation studied in this paper, there is plenty of parallelism to be exploited. Hence there would be no benefit to increase processing element cost by adding features designed only to increase single-thread performance.

Writing a program as a collection of stream processing functions permits easy characterization of the modules and exploration of a variety of choices for mapping the modules onto a parallel processing computer. Other work relating to static mapping of programs for multiprocessor execution includes many published results on resource management for real-time computation. A summary of work in that area appears in [Chaudhary/Aggarwal 1993]. Others have also noted the tradeoff between throughput and latency. The work presented here is distinctive in relating the mapping problem to program structure characteristic within a functional programming framework, and in dealing with multi-rate signal processing problems. The work closest in spirit to ours is the Ptolemy Project of Prof. Lee at Berkeley.

Our work indicates that the combination of functional programming with multithreaded processing elements can lead to significantly easier programming of applications in the domains of signal/image processing. Similar results are anticipated for other application areas that can benefit from use of stream data types, such as real-time embedded systems and certain industrial process control problems. For applications that require dynamic management of resources during program execution, further development of methods of scheduling and load balancing are needed together with architectural features that permit their efficient implementation. We look forward to further developments in this exciting area.

Acknowledgments

The work reported here applies the results of research conducted by the Computation Structures Group of the MIT Laboratory for Computer Science to practical signal processing algorithms. The work is an extension of work done for the Paradigm compiler. The initial version of the Paradigm Compiler was developed by the author during his appointment as Visiting Scientist at RIACS from May 1988 through April 1989.

The optical surveillance example is based on simplified algorithms taken from a large-scale defense application studied by the Boeing Company. The complete original algorithms were expressed in a variant of the Val language [Ackerman/Dennis 1979] in a study performed by Dataflow Computer Corporation under contract to Boeing. The report of this work [Dataflow 1990] included a suggested multithreaded processor design, manually derived machine code, and performance calculations for the Boeing application.

Sisal is a functional programming language developed at the Lawrence Livermore Laboratory for use in high performance scientific applications [McGraw 1985]. Sisal evolved from the Val language developed by the Computation Structures Group at the M.I.T. Laboratory for Computer Science [Ackerman/Dennis 1979].

This paper has been prepared using facilities of the M.I.T. Laboratory for Computer Science.

A Appendix: Sisal Functions for the Optical Surveillance Example

As presented in the text, the overall computation of this illustration is structured as the composition of functions given in Figure 6. The overall computation processes signals from a collection of sensors that are swept over the region under surveillance. These signal are conditioned, averaged, and filtered before a peak detection criterion is applied.

A.1 Baseline Removal

The first step is a procedure designed to ignore a slowly-varying base component of the signal from each sensor. This is defined by the program `BaseRemove` shown in Figure 7.

```
type Signal = stream [integer];
type ImageStream = array [ Signal ];
type DataStream = array [ array [Signal] ];
type MarkStream = array [ stream [boolean] ];

function Process (
   D: DataStream;
   w: integer;
   n: integer
   returns MarkStream )

   let
      R := for i in 1, w
      returns array of
         let S := for j in 1,n
            returns array of
               BaseRemove (D[i,j])
         Nyquist ( SpikeAdapt ( S[i], n ) )
      end for
   in
      PeakDetect ( TwoDimFilter ( R, w ) )
   end let

end function
```

Figure 6: The surveillance process as a composition of stream processing functions in Sisal.

A.2 Spike Adaptive Averaging

The second module (Figure 8) combines signals from groups of n sensors, rejecting data that exceeds a threshhold.

A.3 Nyquist Filter

This module (Figure 9) reduces the sampling rate of the data stream by combining groups of four samples using weights designed to provide a good approximation to the input.

A.4 Two-Dimension Filter

The function `TwoDimFilter` shown in Figure 10 represents a two-dimension filter by a single Sisal function. The filter is defined by the three coefficients, `a`, `b`, and `c`, which are the center, side and corner elements of a three-by-three array. The filter is applied at each position in the image data for which an output value is desired. The input is an array of streams indexed from 1 to w. The output is an array of streams indexed from 2 to $w-1$. (The boundary elements are

```
function BaseRemove (
  D: stream [integer]
  returns stream [integer] )

  AdaptiveBase (
    D, stream_first(D), false, stream_first(D) )

end function

function AdaptiveBase (
  D: stream [integer];
  Th: integer;
  Sw: boolean;
  B: integer
  returns stream [integer] )

  let
    Th0 := 32;
    Df := stream_first(D);
    Dr := stream_rest(D);
    B_new :=
      if Sw then Df else B end if;
    Th_new := max ( Th0, Th +
      if Sw then ( Th / 4)
      else - ( Th / 16 )
      end if );
    Sw_new := ( Df - B_new > Th0 );
  in
    stream [ Df - B_new ] ||
      AdaptiveBase (Dr, Th_new, Sw_new, B_new)
  end let
end function
```

Figure 7: The `BaseRemove` function.

omitted from the result data to avoid applying the filter function to non-existing array positions.)

A.5 A Peak Detector Algorithm

Figure 11 shows a `PeakDetect` function that identifies all elements of the (image) data that have a value that is at least equal to the values of all immediate neighbors and exceeds their average by a given threshhold `Th`. The two conditions are tested separately and combined to determine the result. The input is an array of integer streams indexed from 2 to $w-1$. The output stream is an array of boolean streams indexed from 3 to $w-2$. The peak detection function is similar in structure to the filter function; each element of the result is true if and only if the data surrounding the corresponding input pixel satisfies the specified conditions.

```
function SpikeAdapt (
   D: stream [ array [integer] ];
   n: integer
   returns stream [integer] )

   let
      Th2 := 24;

      E := stream_first( D );

      L := for i in 1, n
         returns value of least ( E[i] )
         end for;

      Sm := for i in 1, n
         returns value of sum
            if E[i] < ( L + Th2 ) then E[i]
            else 0
            end if
         end for;

      Nc := for i in 1, n
         returns value of sum
            if E[i] < ( L + Th2 ) then 1
            else 0
            end if
         end for;

   in
      stream [ Sm / Nc ] ||
          SpikeAdapt ( stream_rest(D), n )

   end let
end function
```

Figure 8: The `SpikeAdapt` function.

References

[Ackerman/Dennis 1979] W. B. Ackerman and J. B. Dennis. VAL—A Value-Oriented Algorithmic Language. Technical Report 218, Laboratory for Computer Science, MIT, 1979.

[Chaudhary/Aggarwal 1993] Vipin Chaudhary and J. K. Aggarwal. A generalized scheme for mapping parallel algorithms. *Ieee Trans on Parallel and Distributed Systems*, Vol. 4, No. 3, pages 328–346. March 1993.

[Culler 1991] David E. Culler, Anurag Sah, Klaus Eric Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded

```
function
   NyquistFilter ( D: stream [integer]
   returns stream [integer])

   let
      Df, Dfr, Dfrr, Drrrr :=
         stream_first(D),
         stream_first( stream_rest(D) ),
         stream_first( stream_rest
             (stream_rest(D) ) ),
         stream_rest(stream_rest
             (stream_rest(stream_rest(D) ) ) );
      D_new :=
         ( 2 * Dfr + Df + Dfrr );
   in
      stream [ D_new ] ||
         NyquistFilter ( Drrrr )
   end let
end function
```

Figure 9: The `Nyquist` function.

abstract machine. In *Proc. of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, pages 164–175, April 1991.

[Dataflow 1990] Dataflow Computer Corporation. Time Dependent Signal Processing Algorithms for Optical Surveillance. Final Report for Contract HA4176 to Boeing Aerospace, Dataflow Computer Corporation, Belmont, MA 02178, November 1988.

[Dennis 1987] Jack B. Dennis. Dataflow Computation: A Case Study. In V. Milutinovic, editor, *Computer Architecture: Concepts and Systems*, chapter 9. New York: Elsevier, 1987.

[Dennis 1988] Jack B. Dennis. Mapping array computations for a dataflow multiprocessor. In *Proceedings of Mapcon IV: Multiprocessor and Array Processor Conference*, pages 71-76, Society for Computer Simulation, 1988.

[Dennis 1989] Jack B. Dennis The Paradigm Compiler: Mapping a Functional Language for the Connection Machine. In *Scientific Applications of the Connection Machine*, Horst Simon, editor, pages 301-315, Singapore: World Scientific Publishing Company, 1989.

[Dennis 1991] Jack B. Dennis. The evolution of 'static' data-flow architecture. In J.-L. Gaudiot

```
function TwoDimFilter (
      D: ImageStream; w: integer
      returns ImageStream )

   let
      a := 4; b := 2; c := 1;

      Bf := for i in 1, w
         returns array of array [-1:
            stream_first(D[i]),
            stream_first(stream_rest(D[i])),
            stream_first(stream_rest
               (stream_rest(D[i])))]
         end for;

      Dt := for i in 1, w
         returns array of
            stream_rest( D[i] )
         end for;

      Dn := for i in 2, w-1
         returns array of
            a * Bf[i, 0] +
            b * ( Bf[i, -1] + Bf[i-1, 0] +
                Bf[i+1, 0] + Bf[i, +1] ) +
            c * ( Bf[i-1, -1] + Bf[i-1, +1] +
                Bf[i+1, -1] + Bf[i+1, +1] )
         end for;

      Dr := TwoDimFilter ( Dt, w )

   in
      for i in 2, w-1
      returns array of
         stream [ Dn[i] ] || Dr[i]
      end for
   end let

end function
```

Figure 10: The `TwoDimFilter` function.

and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 2. Prentice-Hall, 1991.

[Dennis 1994] Jack B. Dennis. Machines and models for parallel computing. *International Journal of Parallel Programming*, Vol. 22, No. 1, pages 47–77, February 1994.

[Dennis 1995] Jack B. Dennis. Stream Data Types for Signal Processing. In *Advances in Dataflow Architecture and Multithreading*, J.-L. Gaudiot and L. Bic, editors. IEEE Computer Society Press. To be published.

[Dennis/Gao 1994] Jack B. Dennis and Guang R. Gao. Multithreaded architectures: principles, projects, and issues. In Robert A. Ianucci, editor, *Advances in Multithreaded Computer Archtecture.* Kluwer, 1994.

[Sabot 1992] Gary Sabot. A compiler for a massively parallel distributed memory MIMD computer. In *Proceedings of the Fourth Symp. on the Frontiers of Massively Parallel Computation*, pages 4–11. ACM and IEEE, 1992.

[Ho/Lee/Messerschmitt 1988] W. H. Ho, Edward A. Lee, and D. G. Messerschmitt. High level data flow programming for signal processing. In R. W. Broderson and H. S. Muscovitz, editors, *VLSI Signal Processing, III*, pages 385–395, New York: IEEE Press, 1988.

[Kelly/Lochbaum/Vyssotsky 1961] John Kelly, C. Lochbaum, and Victor Vyssotsky. A block diagram compiler. *Bell System Technical Journal*, 40(3), May 1961.

[Lee 1991] Edward A. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, April 1991.

[McGraw 1985] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2. Technical Report M-146, Rev. 1, Lawrence Livermore National Laboratory, 1985.

[Oldehoeft/Bohm/Feo 1992] R. R. Oldehoeft, A. P. W. Bohm, D. C. Cann, and John T. Feo. SISAL Reference Manual: Language Version 2.0. Technical report, Lawrence Livermore National Laboratory and Colorado State University, 1992.

[Papadopoulos/Culler 1990] Gregory M. Papadopoulos and David E. Culler. Monsoon: an explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, Seattle, Washington, pages 82–91, May 1990.

```
function PeakDetect (
    D: ImageStream; w: integer
    returns MarkStream )

  let
    Th := 0;

    B := for i in 2, w-1
      returns array of array [-1:
         stream_first(D[i]),
         stream_first(stream_rest(D[i])),
         stream_first(stream_rest
                 (stream_rest(D[i])))]
      end for;

    Dt := for i in 2, w-1
      returns array of
        stream_rest( D[i] )
      end for;

    Pk := for i in 3, w-2

      P := B[i, 0];

      C := B[i-1, -1] <= P &
          B[i-1, 0] <= P &
          B[i-1, +1] <= P &
          B[i, -1] <= P &
          B[i, +1] <= P &
          B[i+1, -1] <= P &
          B[i+1, 0] <= P &
          B[i+1, +1] <= P;

      S := B[i-1, -1] + B[i-1, 0] + B[i-1, +1] +
          B[i, -1] + B[i, +1] +
          B[i+1, -1] + B[i+1, 0] + B[i+1, +1];

      returns array of C & ( 8 * P > S + 8 * Th )
      end for;

    Dr := PeakDetect ( Dt, w );

  in

    for i in 3, w-2
    returns array of
      stream [ Pk[i] ] || Dr[i]
    end for

  end let

end function
```

Figure 11: The `PeakDetect` algorithm.