Compilation of Narrowband Spectral Detection Systems for Linear MIMD Machines

Harry Prints
Digital Equipment Corporation
Paris Research Laboratory
85, avenue Victor Hugo
92583 Rueil-Malmaison France
prints@prl.dec.com
May 22, 1992

Abstract
In this paper, we discuss the design of a program that maps a class of digital signal processing systems, called narrowband spectral detection systems, to linear MIMD machines. Such systems contain a mixture of data-parallel, systolic and purely serial computations. We describe a new technique, called geometric scheduling, that exploits the special features of the first two styles of computation, and that can also incorporate tasks that are neither data-parallel nor systolic. The resulting schedules contain all necessary communication code, which is automatically generated. This paper includes performance figures for this method on a typical narrowband spectral detection system.

1 Introduction
A narrowband spectral detection system is a digital signal processing system that takes a time-sampled data stream as input, and generates periodic estimates of the stream's power spectrum—the distribution of signal power by frequency—as output. Such systems can be used to detect narrowband sinewaves in a background of broadband noise, and form the basis of many passive sonar systems. Efficient parallel implementations of such systems are needed to detect ever-fainter signals in the background of ocean noise, within constraints on processing latency, and on the hardware's volume and power consumption [5].

In this paper, we describe the architecture and performance of an application-specific compiler that maps such systems, represented as directed graphs of signal processing operations, to linearly-connected MIMD computers. We identify key characteristics of the application domain, describe how the signal processing operations that are the system building blocks may be parallelized, and discuss the impact of these different parallel styles on communication and scheduling. In particular, we present our new geometric scheduling technique. This technique attempts to incorporate higher-level knowledge about the structure of the computations in the scheduling process. It combines different parallel styles, and
communication code, into a single schedule. We compare the performance of this technique with some other well-known scheduling methods.

The plan of this paper is as follows. Section 2 reviews the application domain and the target machine. This section includes a brief discussion of the signal processing operations that appear in such systems, and identifies the characteristics of these operations that influence scheduling. Section 3 is an overview of the compilation process. Section 4 describes our scheduling method and gives performance figures. Section 5 treats interprocessor communication. Section 6 is a summary and conclusion.

2 Application Domain and Target Machine

In this section, we review the essential characteristics of the application domain and the target machine. On the application side, we examine the control structure of narrowband spectral detection systems, and the computations that are performed within this control structure. On the target machine side, we look at both the hardware, which determines what interprocessor communication patterns will be fast (or even possible), and the software, which defines the abstract machine that is the target of our compiler.

2.1 Application Domain

2.1.1 Control Structure

Figure 1 depicts a typical narrowband signal processing system. The input is a single discrete-time stream of samples, which enter the system at the upper left corner. The output is an estimate of the power spectral density of this input signal, presented as a horizontal line on a computer display—bright spots for regions of high power, dark spots for those of low power. Successive lines are stacked on a display, and a sinusoidal signal appears as a bright vertical streak.
As Figure 1 suggests, the systems we consider can be represented as directed graphs (not necessarily acyclic), in which the nodes stand for signal processing operations, and the edges indicate the flow of data. When invoked, an operation consumes a fixed, known amount of data from each of its input arcs, and generates a fixed, known amount of data on each of its output arcs. Every operation comes with a data-independent bound on its execution time.

In the parlance of reference [7], these are synchronous signal flow graphs. The adjective "synchronous" appears because the annotations of input and output sizes, and execution times, permit us to determine at compile time a fixed schedule of node invocations that can be iterated indefinitely to implement the desired computation. Again, following [7], we will call this a periodic schedule. The goal of our compiler is to transform such a graph into a periodic schedule, which includes code for all necessary computation and communication, of the shortest possible execution time. This is a classic NP-complete problem; this paper introduces a new heuristic for solving it.

2.1.2 Computation

Now we investigate the types of signal processing operations that appear in narrowband spectral detection systems. We identify three execution styles for these operations—data-parallel, systolic, and serial.

In the data-parallel style [3], the computation consists of a collection of identical actions, distributed as uniformly as possible over a set of processors. Each processor operates on its slice of the data, to produce some section of the output. For instance, real-to-complex bandshifting, defined by the equations

\[
\begin{align*}
v_k &= Re(e^{-j\omega_{0}k}) \cdot x_k \\
v_k &= Im(e^{-j\omega_{0}k}) \cdot x_k
\end{align*}
\]

has an obvious data-parallel implementation, as two pointwise vector products.

Much of the computation in narrowband spectral detection systems appears in operations that have data-parallel implementations. Moreover, most of these operate on data sets that are large compared to the number of processors in a typical SIMD machine. Table 1 gives the distribution of MFLOPs by data set size for the system of Figure 1. This is necessarily because the size of input and output data sets determines how many processors can simultaneously work on a single data-parallel task. The computation can be subdivided up to the point where each processor generates only a single output, or uses only a single input. Table 1 tells us that in this application, a data-parallel approach—for those tasks that can be programmed in this way—will efficiently use a machine with a moderate number of processors, since there is enough data in these tasks to occupy most of the machine.

One advantage of data-parallel operations is that providing there is enough data to occupy the whole machine, the scheduler does not have to look for additional sources of parallelism. The programmer, by writing in a data-parallel style, has identified a collection of computations that can be executed together. Exploiting this kind of programmer-identified parallelism—rather than forcing the compiler to try to discover it—is key to our approach.

Our scheduler represents a data-parallel operation as a rectangle of width equal to the number of processors the operation uses, and of height equal to the execution time of a single processor on its slice of the data (see Figure 2). We call this object a slab. Note that among any collection of slabs, all of which are ready for execution, the order in which they are scheduled is irrelevant—the total execution time is the same.
Table 1: Distribution of Data Parallel MFLOPs. This table gives the distribution, by data set size, of the MFLOPs appearing in data-parallel tasks in one iteration of the narrow signal processing system. The remaining MFLOPs are not in data-parallel operations.

<table>
<thead>
<tr>
<th>Data Set Size</th>
<th>MFLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-16</td>
<td>0.9</td>
</tr>
<tr>
<td>17-64</td>
<td>10.7</td>
</tr>
<tr>
<td>65-256</td>
<td>55.3</td>
</tr>
<tr>
<td>257-512</td>
<td>3.4</td>
</tr>
</tbody>
</table>

In the systolic style [4], a computation is divided into stages, akin to the combinatorial logic sections of a pipelined circuit; these stages are distributed among adjacent processors. Data are introduced at one boundary of this array, intermediate results propagate through the stages, and the desired values emerge at another boundary. In the systems that we consider, only the FFT is implemented systolically, but this one operation accounts for 30% of the MFLOPs in a typical system. Thus it is important to perform this computation efficiently.

The latency of a computation distributed in this fashion can be the same as for a uniprocessor implementation. The advantage of this style comes from having many different instances of the same computation executing at once, in various stages of completion; this gives an improvement in throughput.  

Thus to realise a performance gain through this approach, it is essential for the scheduler to collect and schedule together many instances of the same systolic operation. We explain below how the scheduler achieves this, even when the instances lie in widely separated portions of the adg. We call the systolic objects that the scheduler deals with stairs, since they consist of adjacent rectangles, each representing one stage of the computation, offset from one another by the skew $\sigma$ required by the particular systolic algorithm (see Figure 2).

Finally, those operations that cannot be parallelised at all must be performed in the serial style. A typical serial operation is noise mean estimation, a recursive operation in which a running average of the noise background is developed in a scan across an array of inputs. For each succeeding input point, the result of all previous computations is used to determine the next output point.

The serial objects that the scheduler handles are called strips. Each represents a computation executing on a single processor for a given period of time.

2.1.3 Summary

The systems we consider can be described by synchronous signal flow graphs. They process unbounded streams of input, periodically generating some output. Though the computation runs indefinitely, it is specified by a single finite schedule that is executed repeatedly.

The signal processing operations that appear in these systems come in three different styles: data-parallel, systolic and serial. Almost all of the computation can be fit into the first two styles; this means that in many cases there is some high-level structure to the computation that we would like to exploit. Moreover, most data-parallel operations process large data sets, compared to the number of processors in the machine. Thus data-parallel implementations of these operations will waste few machine cycles.

Systolic operations often read some of their operands from interprocessor communication queues, and this can also give an advantage, by reducing the number of local memory accesses.
2.2 Target Machine

The target architecture of our compiler is the Warp computer [1], designed at Carnegie Mellon and fabricated by the General Electric Company. Warp is a linear, SIMD array of 10 custom processors, each capable of delivering 10 MFLOPs. There are bidirectional links between adjacent processors, but there is no direct connection from the last processor to the first. (However, there is an indirect connection through an external data path.) The array runs as an attached processor of the host. Only the first processor in the array can read operands from the host, and only the last one can return results.

In a single machine cycle, each processor is capable of performing all of the following operations: send a word to each neighbor, receive a word from each neighbor, read and write memory, increment and test a loop counter, perform two address generations, and emit a floating point product and sum.

Warp is programmable in two high-level languages, Lam's W2 [6] and Tseng's AL [10], that shield us from details of managing and scheduling each processor's individual functional units. These are both algorithmic languages, with the usual constructs for expressing conditional execution, iteration, and procedure call. The key differences between them are the way they handle communication, and the facilities they provide for specifying and operating upon parallel data objects. We make use of the advantages of both.

W2 makes the send and receive communication primitives available to the programmer. In fact, the programmer is responsible for generating all communication code. This close relation with the hardware makes it possible to specify systolic algorithms, which synchronise on the receipt of individual words.

In AL, communication instructions are not directly available to the programmer. Instead, the language includes primitives for converting among local, shared and distributed
objects; these are ultimately compiled into sequences of W2 communication statements. However, we tend to make little use of this feature. Most signal processing operations do not require it; moreover, we want the control of communication to be in the hands of our compiler, so that we can implement it in the most efficient possible way, and in some cases eliminate it.

For our purposes, the key feature of AL is the support it provides for operations that are to be performed in parallel over all parts of a distributed data object. The key construct from the programmer's point of view is the do*, which is used to specify a loop over a distributed object. The AL compiler automatically distributes the loop iterations appropriately.

AL and W2 form a tower of languages, in the following sense. The AL compiler is a program that generates W2. Its output must then be run through the W2 compiler, which produces a loadable binary for the Warp machine. Our compiler generates interprocessor communication code in W2. Code for the actual signal processing operations is drawn from a handcrafted library, written in AL or W2 as appropriate to the operation.

3 Overview of the Compilation Process

In this section, we sketch the operation of the compiler, from start to finish. The input to our compiler is a directed graph, hereafter the signal flow graph or sfg. The nodes of the sfg stand for computations, which we call tasks, and the arcs indicate the flow of data. Note that two different tasks may perform the same underlying sequence of arithmetic operations—say, a 7-point FIR filter. We call this underlying definition an operation.

An sfg represents a computation that continues without bound—each task is a finite computation, taking a fixed amount of input and generating a fixed amount of output, but we think of the graph as processing an unbounded stream of input, and generating an unbounded stream of output. Our compiler generates a static schedule for this graph, which can be iterated indefinitely. This is called a periodic schedule. We now review the five phases of compilation for generating this schedule, and ultimately executable code.

In phase 1, the compiler translates the signal flow graph into an acyclic dependency graph, or adg. This means that it determines how many times a task will be executed in the periodic schedule; each distinct invocation of a task is called a task instance. Task instances are the nodes of the adg; they reflect the tasks of the sfg. Likewise, the arcs of the adg are instances of the arcs of the sfg, except that they indicate which instances of one task feed which instances of another. The adg arcs are also labeled with numbers that identify which items flow over them; these are called universal sequence numbers, or usns.

The compiler now has a representation of what computations must be performed in one period of the schedule. It also has a partial ordering among task instances, given by the arcs of the adg. In phase 2, the compiler assigns the nodes of the adg to individual processors. This phase is the first pass of the scheduler. In this pass, the compiler determines which processors will execute each task instance, and in what order. If the instance is a slab or stairs, it will occupy more than one processor. The result is a periodic schedule that obeys the partial ordering of the adg, but which lacks interprocessor communication code.

In phase 3, the compiler traverses the adg, inspects the identity of the processor(s) that execute each task instance, and inserts additional adg nodes to perform any required interprocessor communication.

In phase 4, the compiler invokes the scheduler for a second time on the adg. In this
pass, the communication nodes that were created in phase 3 are assigned to processors, and to specific execution cycles. The processor assignment of nodes visited in phase 2 remains unchanged, but their assignment to execution cycles can (and in general will) be moved to accommodate communication code. This is done in a way that continues to obey the adg's partial order. The result is a complete execution schedule, including both interprocessor communication and computation.

In phase 5, the final phase, the compiler traverses each processor's execution schedule, and emits W2 code in the proper order for the desired computation and communication. The code for computations comes from a library of AL and W2 procedures; if necessary the AL compiler can be invoked to produce the required W2 code. The code for communication is generated directly in W2 by the compiler itself.

4 Scheduling

To exploit the capabilities of a parallel machine, the scheduler must find computations that can be executed at the same time. In the case of the data-parallel and systolic styles, this problem has already been solved to a certain extent by the programmer, who specified the operation in a parallel idiom. The problem facing us is to devise a scheduling method that can take advantage of the parallelism that the programmer has built into the data-parallel and systolic tasks, while at the same time finding and exploiting parallelism among serial operations. We begin with a review of classical methods, and then explain our approach.

4.1 Classical Methods

Consider a set of $P$ identical processors, and a directed acyclic graph $G$, where $G$'s nodes, $\{T_1, T_2, \ldots, T_N\}$, stand for computations, each with a fixed, known execution time. We will denote this set by the letter $T$. The classical multiprocessor scheduling problem is to find an assignment of computations to processors, consistent with the partial order on $T$ given by the arcs of $G$, so that they can all be completed in the shortest possible time.

This problem is well-known to be NP-complete, as established by Ullman in [11]. This has led to the study of efficient heuristics for approximate solutions of the problem.

Many heuristics for approaching this problem can be cast into the following general framework. As depicted in Figure 3, the program divides $T$ into three disjoint sets: waiting, ready, and scheduled. Initially, all of $T$ is in waiting. The action begins when a node with no predecessors is moved from waiting to ready. Thereafter the program runs in the following loop. First, it uses some rule to extract a node from ready and assign it to a processor. Then this node is adjoined to scheduled, and any nodes which now have no unscheduled predecessors are moved from waiting to ready. This process continues until all of $T$ is in scheduled.

There are numerous variations of this scheme, which differ in the rules for selecting a node from ready and for assigning the selected node to a processor, in the direction for traversing the graph (top-down or bottom-up), and the way in which interprocessor communication costs are handled, if at all. For a discussion of some of these variations, see [8, Section 2.3].

The methods reviewed in [8] that fit this framework all share two characteristics. First, each scheduled object occupies only one processor when it executes. In the terminology

---

Footnote: This step is currently done by hand.
introduced earlier, they are all strips. Second, some sort of numerical score, usually derived from the longest path to the graph's starting or finishing node (the "critical path"), and possibly combined with some measure of communication overhead as well, is used to select a node from ready to schedule, and to assign this node to a processor.

4.2 Higher-Level Structure

Our scheduling technique fits this framework, but does not share these characteristics. The objects that we schedule may occupy one, several or all processors in the machine. And although a numerical score will play a role in our scheduler, the primary criterion for selecting the next object for assignment is the geometry of the node (slab, strip or stairs).

Our approach is motivated by a desire to exploit the natural parallel structure that many signal processing operations exhibit, and which in fact is easy for the programmer to express. To understand the role this plays, consider a 7-point FIR filter, defined by the equation

\[ y_k = \sum_{l=0}^{6} a_l x_{k+l}. \]

Some systems treat the computation of a single output point \( y_k \) as a complete operation. This increases the scheduler's flexibility, since the objects it deals with are all quick to execute. However, the responsibility for finding parallelism among these objects now lies exclusively with the scheduler.

But if we are processing a large block of \( N \) consecutive data points, this filter has the natural data-parallel implementation

\[
\text{DO}\ i = 0 \text{ to } N-1 \{ \quad y[i] = a0 \ast x[i] + \ldots + a6 \ast x[i+6]; \}
\]

The \( \text{DO} \) in this pseudo-code indicates that the loop is data-parallel—the computation can be spread out over the processors in the machine. Viewed another way, the programmer has defined a schedule for the \( N \) micro-operations that compute these outputs—one that load-balances the computation well, provided \( N \gg P \), where \( P \) is the number of processors in the machine.\(^3\)

This is the sort of high-level structure that we wish to take advantage of in the scheduler. But different parallel styles must be exploited in different ways. For data-parallel routines, the computation must be spread as evenly as possible across some contiguous set of processors. For systolic routines, the different stages of the computation must be scheduled onto topologically adjacent processors, taking into account the required skew between

\(^3\)It can be shown that the number of wasted cycles in such a slab, as a fraction of computation, is bounded by \( P/N \). See [8, Section 4.4].
their initiation times. Moreover, for efficiency, as many systolic operations as possible must be grouped together. Our geometric scheduling technique does all this, and also gracefully includes the scheduling of strips (operations that the programmer could not parallelize).

4.3 Geometric Scheduling

In geometric scheduling, the ready set is subdivided by node geometry into the disjoint sets slabs, stairs, and strips. This is illustrated in Figure 4. As before, the scheduler works by selecting an object from one of these sets, assigning it to one (or more) processors, and then moving any newly-executable nodes from waiting into one of slabs, stairs or strips, according to its geometry. The key to this scheme is that the program draws nodes to schedule first exclusively from slabs, then exclusively from stairs, and finally exclusively from strips, always exhausting each subset before moving on to the next one. Then it starts again with slabs, and continues in round-robin. As a result, objects are grouped together by geometry in the schedule: first we find a collection of slabs, then a collection of stairs, and finally a collection of strips.

![Figure 4: Subdivision of Ready Nodes During Scheduling.](image)

The intuition behind this control structure is as follows. In preliminary experiments, we found that schedulers that were based exclusively upon the critical path length would often select a slab, then a strip, then a few more slabs, then another strip, and so on. Each time a slab is placed, though, it becomes an impenetrable barrier in time in the schedule, leaving a large hunk of idle processor-cycles.

It is not at all surprising that such algorithms yield poor results, since they were formulated on the assumption that the only impediment to executing a task at some time \( t \) is that all its predecessors must have completed before \( t \). But if we admit data parallel and systolic tasks, and interprocessor communication tasks, this is no longer true. To achieve good performance, it is necessary to come to grips with the "barrier" effect of slabs and stairs.

Figure 5 illustrates the key features of our technique. (This figure shows the results of the scheduler's first pass; the second pass incorporates the automatically-generated communication tasks.) The top portion of the figure displays a fragment of a larger acyclic dependency graph to be scheduled. Each object is a task instance; the width of an object shows how many processors it must occupy, the height shows how long it will run. The arcs indicate the partial ordering among these objects in the graph, and the numbers on the nodes give their critical path priorities, for a traditional scheduler.

The bottom portion of the figure shows two possible schedules for this fragment. The left-hand one is the result of traditional critical path scheduling. The right-hand one is the result of geometric scheduling. Note how the geometric scheduler groups systolic tasks together, and likewise for strips. The former is a necessity to exploit the systolic approach;
the latter eliminates the waste left by scheduling a single strip between two slabs, and attempts to maximise parallelism among strips.

One optimisation we do not perform is overlapping successive invocations of the periodic schedule, as done in Schwartz's cyclo-static scheduling [6], or by building an acyclic dependency graph that implicitly contains multiple instances of the schedule \( J > 1 \), in the notation of reference [7]. The cyclo-static approach is inappropriate here because there is little waste at the ends of the schedule that could be recovered by overlapping instances. The multiple-instance approach could yield somewhat higher processor utilisation, but at a cost of higher latency. In this application domain, low latency is paramount.

4.4 Performance

We will report two kinds of performance figures for our scheduler. Naturally, we will give the total execution time, in clock cycles, of one iteration of the periodic schedule. However, we will also give the total number of computation \( (C) \), alignment \( (A) \) and wasted \( (W) \) processor-cycles in a schedule. The unit processor-cycle means one processor, occupied for one clock cycle. Thus if \( P \) is the number of processors in the machine, and \( \omega \) is the total execution time in clock cycles of one iteration of the periodic schedule, we have the relation \( P\omega = C + A + W \). Hence the execution time of the schedule is

\[
\omega = \frac{C + A + W}{P}
\]

Table 2: Scheduler Performance. TD means "top-down," BU means "bottom-up." pcycles means "processor-cycles."

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>( C ) pcycles</th>
<th>( A ) pcycles</th>
<th>( W ) pcycles</th>
<th>( \omega ) cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD, Geometric</td>
<td>7,731,061 68</td>
<td>2,177,649 19</td>
<td>1,452,460 13</td>
<td>1,137,116</td>
</tr>
<tr>
<td>TD, Hu-level</td>
<td>7,731,061 13</td>
<td>3,865,729 5</td>
<td>49,289,270 82</td>
<td>6,047,507</td>
</tr>
<tr>
<td>BU, Geometric</td>
<td>7,731,061 69</td>
<td>2,177,649 19</td>
<td>1,539,000 12</td>
<td>1,124,861</td>
</tr>
<tr>
<td>BU, Hu-level</td>
<td>7,731,061 25</td>
<td>3,876,466 13</td>
<td>17,876,294 61</td>
<td>2,948,282</td>
</tr>
</tbody>
</table>

Table 2 gives the execution time and processor-cycle breakdown for the system of Figure 1, scheduled by four different methods: top-down versions of the geometric and Hu-level algorithms, and bottom-up versions of the same. In all cases, the target machine was a 10-processor Warp. In the bottom-up comparison, the schedule produced by the geometric algorithm is 2.6 times faster, in the top-down comparison, it is 5.3 times faster.

Returning to equation (2), note that the quantity \( \omega_{\text{min}} = C/P \) is a lower bound on the execution time of any schedule for the system. This time is probably not attainable, since it would require perfect load-balancing of the computation, with no alignment or waste cycles. However, it is definitely a lower bound on the best attainable execution time, denoted by \( \omega_{\text{opt}} \). Thus we have \( \omega_{\text{min}} \leq \omega_{\text{opt}} \).

Now note that since \( \omega = (C + A + W)/P \), we have

\[
\frac{\omega}{\omega_{\text{opt}}} \leq \frac{\omega_{\text{min}}}{\omega_{\text{opt}}} = \frac{C + A + W}{C} = 1 + \frac{A}{C} + \frac{W}{C}.
\]

These figures are estimated from timings of program segments; the Warp processor does not have enough memory to hold a complete system.
all legal permutations identical

systolic operations batched

strips are independent

Figure 8: The intuition behind Geometric Scheduling. Refer to Section 4.3 of the text for an explanation of this figure.
Under suitable conditions on the signal flow graph supplied as input to our compiler, which are given in [8, Chapter 7], it is possible to obtain bounds on the quantities $A/C$ and $W/C$ as functions of $P$. We call these bounding functions $\Lambda_A(P)$ and $\Lambda_W(P)$; then we have

$$\frac{\omega}{\omega_{opt}} \leq 1 + \Lambda_A(P) + \Lambda_W(P).$$

Figures 6 and 7 are graphs of these functions. Within the $P$ range shown, geometric scheduling is guaranteed to produce results that are within a factor of $1 + \Lambda_A + \Lambda_W$ of optimum.

Unfortunately, the exact functional forms of $\Lambda_A$ and $\Lambda_W$ are specialized to the class of narrowband spectral detection systems defined in [8], and are not particularly illuminating to inspect. But we will sketch the general way they were obtained.

We start by defining a class $S$ of signal processing systems, in terms of the operations that appear within it, and the structure of the graphs built from these operations. For each operation, we also supply the parameters $\gamma$, which is the number of processor-cycles to compute a single output, and $N$, the number of outputs. Then it is possible to show (see [8]) that for any collection of slabs,

$$\frac{W}{C} \leq P \cdot \frac{\sum \gamma_i}{\sum N_i \gamma_i},$$

that for any collection of $D$ identical stairs,

$$\frac{W}{C} \leq \frac{P}{D},$$

and that for any collection of strips,

$$\frac{W}{C} \leq \left(2 - \frac{2}{P+1}\right) \max\left\{ \frac{P \max\{N_i\}}{C}, 1 \right\} - 1.$$

By summing these expressions, and then maximizing the sum over the class $S$, we obtain the function $\Lambda_W$. To obtain the function $\Lambda_A$, we add up the worst-case alignment costs along each graph edge, and likewise maximize this sum over $S$.
5 Alignment

In this section we sketch our approach to interprocessor communication, which we call alignment. First we explain how our system detects the need for alignment code and generates it, then we discuss some consequences of this approach.

5.1 Detecting and Generating Alignments

The compiler labels each task instance's input and output arcs with its data organisation. For inputs, a data organisation describes where the operation expects to find its input distributed within the machine; for outputs, it describes where the operation leaves its results. There are four types of data organisation:

- spread across contiguous processors, no data overlap between processors
- spread across contiguous processors, partial data overlap between processors
- data fully shared among contiguous processors
- data present on the input or output queues between the array and the host.

A data organisation is specified by giving its type, the starting and finishing universal sequence numbers (usns) of the data item, the processor(s) or queue where it resides, its density (number of items per processor) and its window size (amount of interprocessor overlap).

After the first pass of the scheduler, the compiler knows which processors each task will occupy, and hence the data organisation of each input and output. It traverses the acyclic dependency graph and inspects the data organisations at either end of each arc. If these are compatible, no action is necessary. Otherwise it breaks the arc in two, and inserts an interprocessor communication task, or alignment. The W2 code for this alignment is automatically generated, by decomposing it into loops that pass blocks of data between adjacent processors.

5.2 Consequences

There are two important consequences of this approach. First, each alignment will be treated as a slab by the scheduler. This is really due to the Warp hardware—passing data from processor \( p_k \) to \( p_{k+1} \) in the machine requires each intervening processor to touch the data, under explicit program control. Thus code must be generated and scheduled for every processor along the way. (More advanced architectures decouple computation and communication; see for instance reference [2].)

Second, this approach, combined with the data-parallel style, yields alignments that exhibit high coherence. We cannot yet give a mathematical definition of this notion, but we will try to present the idea with an example. Let us return to the 7-point FIR filter. Consider the generation of 256 output points; this requires \( 256 + 6 = 262 \) input points. Output \( y_0 \) requires inputs \( z_2 \) through \( z_4 \) inclusive, \( y_1 \) requires \( z_3 \) through \( z_5 \), and so on.

Suppose for a moment that the generation of each individual output were treated as a separate task instance. Then in general, this could require moving inputs \( z_0 \) through \( z_6 \) to the processor that computes \( y_0 \), then \( z_1 \) to \( z_7 \) to the processor that computes \( y_1 \), and so
on up to $\Pi_{256}$ to $\Pi_{256}$. In the worst case, this could mean moving $256 \times 7$ items through the processors, at a cost of up to $256 \times 7 \times P$ processor-cycles.

However, if the computation runs data-parallel, then the required data organisation is to spread 262 data items among the processors, with an overlap of 6 items between adjacent processors. Because the computation pattern is regular, the required distribution of inputs is regular. This alignment can be implemented efficiently, requiring no more than $262 \times P$ processor-cycles. Indeed, if the task that generates the filter's input (that is, values $\Pi_0$ through $\Pi_{261}$) has left its output suitably spread through the machine, then the only required alignment is to pass shared data from each processor to its neighbors, at a cost of $6 \times P$ processor-cycles.

To put this another way, the high-level structure of the computation induces a high-level structure to the required pattern of input data distribution. The regularity of this pattern permits an especially efficient implementation of the necessary interprocessor communication steps.

6 Conclusions

In this paper, we have presented the design and performance of an application-specific compiler for linear SIMD machines, tailored to narrowband spectral detection systems. These systems are described by synchronous signal flow graphs, and can be scheduled at compile time. The individual operations that appear in these flow graphs may themselves contain substantial amounts of parallelism, which is natural for a programmer to express in a data-parallel or systolic style.

To exploit the parallelism present in these operations, we have devised a new scheduling heuristic, called geometric scheduling. The essence of this technique is to select objects to schedule so that those of like geometry are grouped together. This permits them to be packed efficiently into the schedule. In the case of a real narrowband spectral detection system, we found that this technique improved upon the classical HN method by factors of 2 to 5. Moreover, this method gives results that are within a certain fixed factor of optimal, on a suitably restricted class of systems.

We also sketched the way our system generates interprocessor communication code, or alignments. We argued that exploiting the high-level structure of operations yields more regular patterns of data distribution. The structure in these patterns permits us to implement the required alignments more efficiently.

Acknowledgements

Thanks to the entire Warp team of Carnegie Mellon, especially Todd Mummert, who wrote much of the AL code, to P. S. Tseng, who wrote the AL compiler and Mike MacPherson, who maintained it, and to Monica Lam and Thomas Gross, who led the development of the W2 compiler. Thanks also to Tom Stover and Paul Scherer of the Naval Air Development Center, who introduced me to spectral analysis systems, and to Hervé Touati, who commented upon earlier drafts of this paper.
References


