DISTRIBUTED SIGNAL PROCESSING

Li Lee and Alan V. Oppenheim

Research Laboratory of Electronics
MIT, Cambridge, MA 02139

ABSTRACT

This paper explores issues arising from designing digital signal processing algorithms for dynamically-varying computing environments such as an unreliable network of processors. We present a language for specifying signal processing algorithms which permits the execution path of the algorithm to be dynamically chosen. The language leads naturally to a graphical representation of the algorithm with interesting interpretations. Finally, we formulate and characterize the solution for the problem of dynamically and optimally choosing the execution path of algorithms to minimize a system-wide cost function such as expected congestion.

1. INTRODUCTION

With the growing prevalence of computer networks, the ability to efficiently exploit the computing power of the entire network becomes increasingly attractive. While distributed networks offer advantages of resource sharing and fault tolerance, they are more dynamic than traditional computing environments such as a microprocessor chip. In this paper, we explore some issues arising from designing digital signal processing algorithms with the assumption that the computing environment is dynamically varying. We are particularly interested in dynamically-varying, heterogeneous computing networks, in which the device capabilities and link capacities vary throughout the network, and the configuration of the network can change unpredictably.

Such a computing environment is fundamentally different from the traditional signal processing platform of single chips, each individually designed and optimized to run specific algorithms on a specific architecture. When the computation model is variable, algorithms should be designed to *adapt* to the current state of the network. There are two important implications of this requirement. First, algorithms should not be designed with stringent assumptions about the architecture of devices or even the availability of certain devices. Second, their specification must allow them to be dynamically "optimized" to take advantage of the current conditions of the network. In this paper, we present a language for specifying signal processing algorithms which permits the execution path of the algorithm to be dynamically altered. We then present and extend a graphical representation of the language to a method of choosing the execution paths to minimize the total expected processing time experienced by any piece of data.

This research was supported in part by the U.S. Air Force Office of Scientific Research under Grant AFOSR-F49620-96-1-0072 and in part through collaborative participation in the Advanced Sensors Consortium sponsored by the U.S. Army Research Laboratory under Cooperative Agreement DAAL01-96-2-0001. Li Lee is supported by an AT&T Graduate Research Fellowship for Women.

2. SIGNAL PROCESSING ALGORITHM SPECIFICATION

Due to the mathematical framework inherent in their design, digital signal processing algorithms have the desirable property that many common operations can be executed in a variety of ways, all leading to the same final result. For example,

- Convolution in the time domain is equivalent to multiplication in the frequency domain.
- High-order filters can be implemented as a cascade of lower-order filters or as a sum of parallel lower-order filters.
- Filtering operations preceded or followed by up- or down-samplers can be implemented using a variety of polyphase structures.

Here we present a simple language for specifying signal processing algorithms with a variety of different execution paths. We then show that the representation of the algorithm as a directed graph leads to an intuitive interpretation of the load-balancing problem when the execution paths of algorithms are allowed to vary.

2.1. Algorithm Specification

We consider algorithms which can be implemented as a combination of primitive operations. For now we assume that the set of available primitives in the system is prespecified and denoted by $R = \{r_1, r_2, \cdots, r_N\}$. The tradeoffs involved in choosing a complete and "interesting" set of r_i's will be discussed later. Three operators in prefix notation are used to specify the algorithms:

- $>$ Sequential operations. For example, $(> \ r_1 \ r_2)$ means that the only permitted execution sequence is operation r_1, then operation r_2. Expressions of this type will also be referred to as *Then* clauses.
- $*$ Parallel operations. For example, $(* \ r_1 \ r_2)$ means that r_1 and r_2 can operate on the data concurrently. Expressions of this type will also be referred to as *And* clauses.

- + Alternative operations. For example, (+ r_1 r_2) means that operations r_1 and r_2 are alternatives for each other. Expressions of this type will also be referred to as *Or* clauses.

These operations can be nested and combined to form more complex expressions. For example, the following expression gives four different alternatives for implementing a modulated filter bank [4]:

```
(+  (*  (>  Filter Down-sample)
        (>  Filter Down-sample)
        ... (>  Filter Down-sample))
    (>  (+  (*  Filter Filter ... Filter)
            (>  Serial->parallel
                (* Filter Filter ... Filter))
            (>  Window Time-alias
                    Serial->parallel))
        Discrete-Fourier-Transform))
```

Note that the filtering operations in the above expression refer to using the filtering primitive with different filter coefficients. Depending on the chosen set of primitive operations, the above expression can contain many more choices of implementation. For example, filters and discrete Fourier transforms (DFTs) can be implemented in a variety of ways.

The following "algebraic" equivalences are rules of commutativity, associativity, and distributivity which relate to the composition rules of these algorithm operators:

- **Commutative equivalences:**
  - (+ r_1 r_2) = (+ r_2 r_1)
  - (* r_1 r_2) = (* r_2 r_1)
- **Associative equivalences:**
  - (+ r_1 r_2 r_3) = (+ (+ r_1 r_2) r_3) = (+ r_1 (+ r_2 r_3))
  - (* r_1 r_2 r_3) = (* (* r_1 r_2) r_3) = (* r_1 (* r_2 r_3))
  - (> r_1 r_2 r_3) = (> (> r_1 r_2) r_3) = (> r_1 (> r_2 r_3))
- **Distributive equivalences:**
  - (+ (> r_1 r_2) (> r_1 r_3)) = (> r_1 (+ r_2 r_3))
  - (+ (* r_1 r_2) (* r_1 r_3)) = (* r_1 (+ r_2 r_3))

Note that the distributive equivalences require that the common operation r_1 refers to the same primitive applied with the same arguments. For example, if filtering is a primitive in the system, then the distributive equivalences apply only if the common operation refers to filtering with the same filter coefficients.

2.2. Algorithm graphs

The language gives rise to a graphical representation, which leads to a simple conceptualization of load balancing in distributed networks. In particular, we can represent the algorithm as a *pyramid* of directed graphs with two types of edges. The first type is a "simple" edge, which corresponds directly to a single primitive operation in the algorithm description. The second type ("compound" edge) corresponds to an And-clause, and is further associated with directed graphs which represent the arguments of the And-clause.

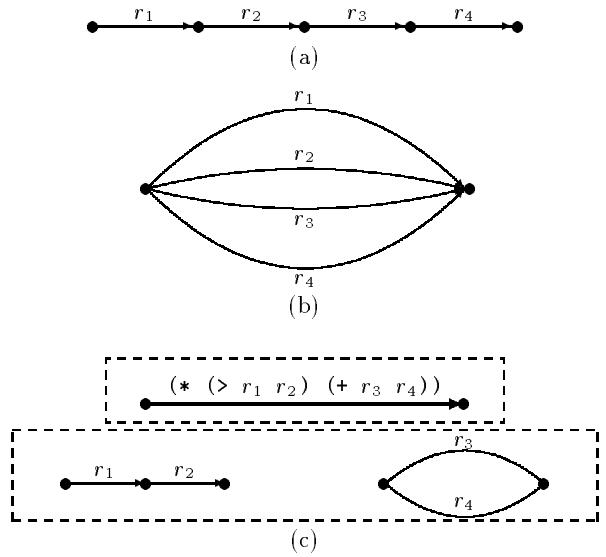

Figure 1: (a) Directed graph representing (> r_1 r_2 r_3 r_4). (b) Directed graph representing (+ r_1 r_2 r_3 r_4). (c) Directed graphs representing (* (> r_1 r_2) (+ r_3 r_4)).

At the top level of the graph pyramid is a graph of the algorithm which represents any And-clause with a compound edge. Each lower level of the pyramid expands the And-clauses of the previous level, so that at the bottom of the pyramid is a collection of directed graphs with only edges of the first type. We refer to the top-level graph of this pyramid as the "algorithm graph".

To construct the graph pyramid based on the algorithm expression, we first apply the equivalence rules presented above to reduce the algorithm to a minimal form using as few operators as possible. The graph for a Then-clause is constructed following the rule that the origination node corresponding to each argument is simply the destination node of the previous argument. As a example, Figure 1(a) shows the graph for the expression (> r_1 r_2 r_3 r_4). In contrast, the origin and destination nodes are the same for all arguments of an Or-clause. As a example, Figure 1(b) shows the graph for the expression (+ r_1 r_2 r_3 r_4). An And-clause is represented by a single edge between its origin and destination nodes, and a collection of lower-level graphs of each of its arguments. Figure 1(c) shows the graph for the expression (* (> r_1 r_2) (+ r_3 r_4)), where we have represented the And-clause edge using a bold line. These rules apply recursively to all sub-clauses.

In the algorithm graph, each edge represents one transformation of the data, and each node represents one state of the data. Hence, the different paths connecting two different nodes in the graph correspond to different sequences of operations which have exactly the same effect on the data. Every path from the origin to the destination node of the algorithm graph corresponds to one possible execution path of the algorithm. (Notice that traversing a compound edge involves traversing all of the lower-level graphs associated with that edge.)

2.3. Choice of Primitives

The choice of the primitives is important since they are the building blocks of the algorithms implementable in the system. First, the set of all functions provided in the system must be complete, so that a large number of algorithms can be specified using these basic functions. Secondly, the granularity of this set of functions must be appropriate. If all of the primitives perform high-level functions, alternative implementations of algorithms become difficult to specify, and the algorithm graph becomes uninteresting single-edge graphs. In the other extreme, if the primitivies perform only bit-wise multiplication or addition, algorithm specifications become impossibly cumbersome, and the cost of communications and control overhead incurred from passing the data from point to point in the network dominate the processing costs.

3. STATISTICAL COMPUTATIONAL MODEL

The algorithm specification presented above is useful in describing how the system can "optimize" the processing using information on current system conditions such as load and availability of processors and links of the system. In this section, we present a framework to minimize the average processing time experienced by data in the network. This framework draws heavily on the representation of the algorithm as a directed graph.

We first consider a single stream of data blocks which all require processing by the same algorithm. The system objective is to choose the algorithm execution path for each data block so that the average processing time experienced by each block is minimized. We conceptualize this as a routing problem through the algorithm graph. In this conceptualization, each directed edge of the algorithm graph is analogous to a link in an "algorithm network". Since each edge corresponds to a processing primitive, crossing a link is analogous to transforming the data with the associated processing primitive. Data blocks are analogous to data packets which need to be routed from an origin to a destination. With multiple data streams, the analogy can be extended to routing through a highly complex network with many different origin-destination pairs. The problem of dynamically choosing an execution path for each data block is conceptually similar to the problem of dynamic routing encountered in data networking and multi-commodity flow literatures.

However, there are important differences between the execution path assignment problem here and the routing path selection problem in data networks. First, the presence of And-clauses in the algorithm graph has no direct equivalence in data networks. An And-clause in an execution path implies that the data is processed by all of the arguments of the clause before the next processing operation starts. An analogous requirement in the data network sense would be highly unusual. Second, in a data network, sending a packet of data across a link generally does not change the size of the packet significantly. In the algorithm graph, however, since each link corresponds to processing the data with a specific primitive, the resulting packet size can change dramatically. Simple examples of primitives which would change the data size dramatically include up- or down-samplers, multiplexers, and de-multiplexers.

Despite these differences, we show here that, with some modifications, existing networking literature on characterizations of optimal routing can be used to specify how execution paths can be chosen to minimize the expected processing time of each packet.

3.1. Optimal Execution Path Assignment

In this subsection, we formulate the problem of choosing the execution path for each data block as a contrained optimization problem where the goal is to minimize the expected system congestion in steady-state. Notice that this is a system optimization problem, rather than an individual optimization problem where the execution path of each packet of data is chosen to minimize the delay for that packet without regard to the rest of the system. We choose this system-wide optimization problem since individual-based optimization often suffers from instabilities in the system when the link rates change, and our system of processors are assumed to be unreliable.

Consider a network of processors, each implementing a primitive operation $r \in R$, where R is the set of all primitive operations. (Note that there may be more than one processor which implements any particular function.) A denotes the set of algorithms $\{a_1, a_2, \cdots, a_N\}$ implementable using these primitives. r_a denotes the rate (in data units/s) at which data requiring processing with algorithm $a \in A$ enters the network. P_a denotes the set of all execution paths for algorithm a. x_ρ denotes the rate (in data units/s) at which data is processed through execution path ρ, hereafter also referred to as the flow of data through path ρ. Finally, $X = \{x_\rho | \rho \in P_a, a \in A\}$ denotes the vector of path flows x_{ρ_i} through all paths ρ in A.

Under this setup, the load, or combined input rate, on processors implementing primitive r can be expressed as

$$L_r = \sum_\rho M_\rho(r) x_\rho, \tag{1}$$

where $M_\rho(r)$ is a multiplier relating the total data input rate to primitive r to the execution path flow x_ρ. As an example, in the path expressed by `(> filter downsample-2 filter downsample-2 filter)`, the multiplier associated with the filter primitive is 1.75, since the input rate to the second filter is at half the original input rate, and the input rate to the third filter is at a quarter of the original input rate. Of course, $M_\rho(r) = 0$ if the primitive r is not used on path ρ.

The delay, or general cost, associated with processing with primitive r is assumed to be a function of only the load on that class of processors, so that the total cost function is

$$D(x) = \sum_r D_r(L_r) = \sum_r D_r \left[\sum_\rho M_\rho(r) x_\rho \right], \tag{2}$$

where $D_r(\cdot)$ is the cost function associated with primitive r.

The optimal execution path assignment problem can

now be formulated as follows:

$$\text{minimize} \quad \sum_r D_r \left[\sum_\rho M_\rho(r) x_\rho \right]$$

$$\text{subject to} \sum_{\rho \in P_a} x_\rho = r_a, \qquad \text{for all } a \in A \qquad (3)$$

$$x_\rho \geq 0, \qquad \text{for all } \rho \in P_a, a \in A$$

This problem, which minimizes the cost function (2) in terms of the unknown vector of path flows X, is the optimal routing problem encountered in data networks and multi-commodity flow problems. From [1], the optimality condition can be derived in terms of the first derivative of the cost function with respect to path flows:

$$\frac{\partial D(x)}{\partial x_\rho} = \sum_{r \in x_\rho} \frac{dD_r}{dL_r} M_\rho(r).$$

In particular, restricting $D_r(L_r)$ to be convex and monotonically increasing with L_r, the path flow vector $X^* = \{x^*_\rho\}$ is optimal if and only if

$$x^*_\rho > 0 \rightarrow \frac{\partial D(x^*)}{\partial x_{\rho'}} \geq \frac{\partial D(x^*)}{\partial x_\rho}, \qquad \text{for all } \rho' \in P_a. \qquad (4)$$

In words, data is routed on the execution paths with the smallest first derivative costs, and all execution paths with positive flow have the same first derivative.

Numerical algorithms to solve this problem in the data network scenarios have been developed [2][3]. Many of these solutions are based on using gradient projection methods, and have been extended and shown to converge for distributed and asynchronous implementations. Since the nodes in the algorithm graph do not have a physical interpretation, however, a direct application of these algorithms to the execution path selection problem will require that every processor in the system knows about the state of all the other processors. An efficient algorithm which requires a minimal amount of inter-processor communication overhead needs to be developed.

3.2. Cost Function

The optimality condition in Eqn. (4) was derived under loose conditions that the cost functions D_r are convex and monotonically increasing. In this subsection, we further our description of the computation model by specifying D_r in terms of statistics such as the speed of the processors and the operating load on them.

Suppose that the network contains N_r processors implementing primitive operation r, and each processor r_i has the processing rate of μ_{r_i} data units/s. We assume the load L_r is distributed among these N_r processors according to their processing rates, so that the probability p_i that a data block is processed by processor r_i, given that it requires processing by primitive r, is

$$p_i = \frac{\mu_{r_i}}{\sum_{i=1}^{N_r} \mu_{r_i}}.$$

Physically, a data block requiring processing by r is simply routed to r_i with probability p_i, regardless of the current load on r_i. While in principle, the system could keep track of the load on each processor, the overhead incurred from implementing such a control strategy is generally prohibitively high.

We now characterize the expected number of jobs in the system of type-r processors as seen by a data block waiting for service from a type-r processor. We model the waiting system associated with each processor as an $M/M/1$ queue. That is, we assume that data blocks arrive for processing according to a Poisson process, and that the probability distribution of the service time is exponential. Each processor has an infinite queue, so that no data is rejected or dropped by any processor. With these assumptions, the expected number of jobs in the system of any processor of type r (i.e., the number in the queue, plus the one currently being serviced) is

$$\sum_{i=1}^{N_r} p_i \frac{p_i L_r}{\mu_{r_i} - p_i L_r} = \frac{L_r}{\sum_{i=1}^{N_r} \mu_{r_i} - L_r} = \frac{L_r}{C_r - L_r}. \qquad (5)$$

Using C_r to denote the total processing capacity of type r processors, and d_r to denote the average processing and communication delay (in seconds/data unit), one possible form for the cost function D_r can then be defined as ([1])

$$D_r(L_r) = \frac{L_r}{C_r - L_r} + d_r L_r. \qquad (6)$$

Notice that D_r is convex and monotonically increasing for $L_r \in [0, C_r)$. Qualitatively, under light load conditions ($L_r \ll C_r$), the processing cost is dominated by communication and actual processing time. Under congested conditions ($L_r \rightarrow C_r$), however, the processing delay is dominated by queuing time. In particular, as the load L_r nears the total capacity C_r, the cost incurred by processing with primitive r will grow unbounded.

4. CONCLUSION

This paper describes a framework and formulation for dynamically adapting signal processing algorithms to processor conditions in distributed processing environments. By conceptualizing the algorithm graph as a network through which data is routed, we formulate the problem of dynamic execution path selection as a constrained optimization problem, similar to those seen in data network and multi-commodity flow literature. Currently, we are implementing a simulation to test the applicability of our models and algorithms.

5. REFERENCES

[1] D.P. Bertsekas and R.G. Gallager. *Data Networks, 2nd Edition*, Prentice-Hall, 1992.

[2] R.G. Gallager. "A Minimum Delay Routing Algorithm Using Distributed Computation," *IEEE Transactions on Communications,* Vol. 23, pp. 73-85.

[3] J.N. Tsitsiklis and D.P. Bertsekas. "Distributed Asynchronous Optimal Routing in Data Networks," *IEEE Transactions on Automatic Control,* Vol. 31, pp. 325-331.

[4] P.P. Vaidyanathan. *Multirate Systems and Filter Banks* Prentice Hall, 1993.