Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

ASPEN: A STREAM PROCESSING ENVIRONMENT

Brian K. Livezey
Richard R. Muntz

October 1988
CSD-880080

ASPEN: A Stream Processing Environment[†]

Brian K. Livezey

Richard R. Muntz

Computer Science Department
University of California
Los Angeles, CA 90024-1596

ABSTRACT

In this paper, we describe ASPEN, a concurrent stream processing system. ASPEN is novel in that it provides a programming model in which programmers use simple annotations to exploit varying degrees and types of concurrency. The degree of concurrency to be exploited is not fixed by the program specification or by the underlying system. Increasing or decreasing the degree of concurrency to be exploited during execution does not require rewriting the entire program, but rather, simply re-annotating it.

Examples are given to illustrate the varying types of concurrency inherent in programs written within the stream processing paradigm. We show how programs may be annotated to exploit these varying degrees of concurrency. We briefly describe our implementation of ASPEN.

[†]This work was done within the Tangram project, supported by DARPA contract F29601-87-C-0072.

ASPEN: A Stream Processing Environment

Brian K. Livezey

Richard R. Muntz

Computer Science Department
University of California
Los Angeles, CA 90024-1596

1 INTRODUCTION

Stream processing is an ideal paradigm for data-intensive applications. In addition to allowing programmers to elegantly solve a rich and varied set of problems that are, at best, awkward to express in other paradigms, stream processing presents an execution model in which such problems can be solved efficiently.

In this paper, we discuss three forms of parallelism which are inherent to the stream processing paradigm. We describe ASPEN, a stream processing environment in which each of these forms of concurrency may be exploited.

Existing implementations of concurrent stream processing systems [2, 6] require programmers to structure their programs so as to reflect the degree of concurrency that is to be exploited during execution. Our goal is to allow programmers to write their programs first, and later insert annotations which indicate the concurrency to be exploited. These practices allow the same program to be executed in different configurations with only minor changes to the annotations, thus exhibiting different performance characteristics, but producing identical results. The execution characteristics of a program can be changed without altering the program itself. Only the annotations need to be changed.

This research is part of the Tangram project [10, 13] at UCLA, whose goal is to develop a Prolog-based distributed modeling environment which combines DBMS and KBMS technologies with a variety of modeling tools. We see the work described in this paper as the foundation upon which much of the Tangram system will be built.

The remainder of this paper is organized as follows. Section 2 provides necessary background information including a characterization of streams and stream processing. The third section describes the types of concurrency inherent in a stream-based language and explains how each can be exploited. Section 4 briefly describes the implementation. Finally, section 5 discusses related work and contains concluding remarks.

2 STREAMS

Streams, at the highest level of abstraction, can be viewed merely as ordered sequences of data objects that are accessed in a sequential manner. Thus, they could be implemented as lists in Prolog [15]. However, implementing streams as lists severely limits the power inherent in stream processing. Performing successive transformations on a list produces intermediate lists, and a complete copy of the transformed list must be stored for each step. This storage is expensive or intractable for large streams.

We avoid such storage problems by providing an implementation which permits *lazy evaluation* within a single process. When lazy evaluation is used, an element of the stream is produced only when it is needed. Transformations can be coroutined such that one element of the input stream can be moved through a succession of transformations until it is completely transformed. Only then is the next element of the input stream requested. Thus, one can avoid storing intermediate streams and one

can write programs that manipulate potentially infinite streams.

2.1 Transducers

The elementary unit of computation in stream processing languages is the *transducer*. Abelson and Sussman [1] recognize four basic forms of transducers. *Enumerators* are stream sources. They take zero or more parameters and generate a stream of output. A *mapping* takes as input one or more streams and performs some sort of transformation on its input to produce an output stream. *Filters* are used to reduce the amount of data on a stream. They remove elements that fail to satisfy certain specified properties. *Accumulators* perform aggregate functions on streams. They take a stream as input and generally produce a single output value.

Hybrid transducers are quite common. Hybrid transducers are often formed by composing two or more elementary transducers. This mode of programming is common in ASPEN. Programmers define elementary transducers or obtain existing transducers from libraries and use them to compose more complex transducers.

2.2 Log(F)

In this section, we briefly review Log(F) [11, 12], a rewrite rule language developed by Sanjai Narain at UCLA. As we shall see, Log(F) provides lazy evaluation. With the extensions described in the next chapter, Log(F) is an excellent language for expressing ASPEN programs. Log(F) allows programmers to express computations with a very fine degree of potential parallelism where each transducer potentially represents a process. ASPEN, as described in the next section, allows programmers to annotate their programs to exploit the desired amount of concurrency.

The following example illustrates how one composes Log(F) rules and how they are translated into Prolog for interpretation by a standard Prolog engine. The rules below describe how to append two streams.

```
append([ ], C)   =>  C.
append([A | B], C)   =>  [A | append(B, C)].
```

We use the Prolog list notation to represent streams. In the term, `[A | B]`, `A` represents the first element of the stream, and `B` represents the rest of the stream.

Log(F) rules are easily translated into Prolog *reduce* rules. The corresponding reduce rules for the above Log(F) rules are:

```
reduce(append(A, B), E)   :-
      reduce(A, D),
      append(D, B) => C,
      reduce(C, E).

append([ ], C)   =>  C.
append([A | B], C)   =>  [A | append(B, C)].

reduce([ ], [ ]).
reduce([H | T], [H | T]).
```

The symbols `[]` and `[H|T]` have reflexive reductions; that is, they reduce to themselves. Such symbols are referred to as *constructor symbols*.

When executed by a standard Prolog engine, reduce rules can be made to behave in a lazy fashion. If we were to reduce `append([a,b,c], [d,e])` once, we would obtain the result `[a|append([b,c], [d,e])]`. We could further reduce the tail, or *continuation*, to obtain the result `[b|append([c], [d,e])]`. Thus, we see that the computation is demand-driven; that is, no result is computed until it is needed.

3 CONCURRENCY IN ASPEN

This section describes the types of concurrency achievable in ASPEN programs. A single mechanism for achieving all types of concurrency is proposed. We describe situations in which the performance of this mechanism is unacceptable and provide additional mechanisms to handle such cases.

3.1 Types of Concurrency

We recognize three basic types of potential concurrency in ASPEN programs. They are stream parallelism, AND-parallelism, and merge parallelism. In this section, we describe each type of concurrency and give examples in which they arise.

3.1.1 Stream Parallelism

Stream parallelism is equivalent to pipelining. The potential for stream parallelism arises in ASPEN programs when transducers are nested. It is illustrated by a transducer which takes a stream of numbers as input, filters out the odd numbers and produces a stream consisting of the squares of the even numbers. The transducer is specified as `sq(even(S))`, where `S` represents a stream of numbers. This transducer can be represented as the pipeline composition of two transducers as in Figure 1.

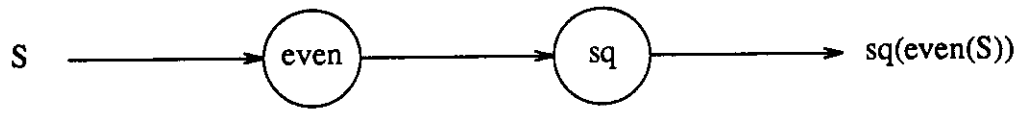

Figure 1 Pipeline Parallelism

Concurrent execution of both stages of the pipeline active at all times could poten-

tially double the throughput of this transducer. Obviously, as the number of stages in the pipeline increases, the potential parallelism also increases.

Note that no parallelism is achieved if strictly lazy evaluation is used. However, if each of the transducers behave eagerly and run concurrently, we do achieve parallelism. In the absence of multiple processes, lazy evaluation in this case yields an efficient use of storage as there is no need to store intermediate values.

3.1.2 AND-Parallelism

The potential for AND-parallelism arises in ASPEN programs whenever a transducer has multiple inputs which are streams. An example of this is a transducer which computes the sum of the squares of two streams, expressed as **`add(sq(A), sq(B))`**. Decomposition of this transducer yields the graph in Figure 2.

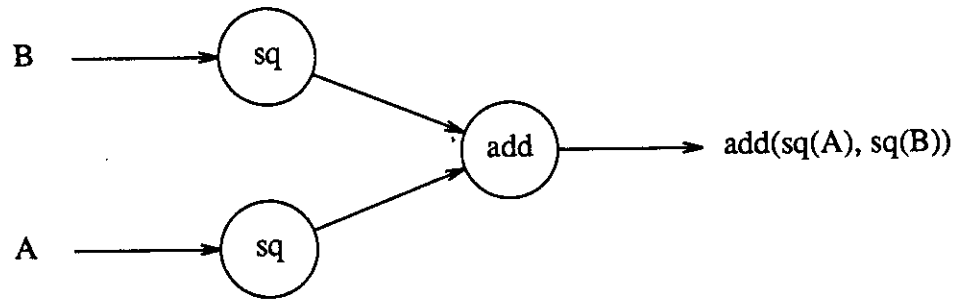

Figure 2 AND-Parallelism

Since the two inputs to **`add`** are independent, they can be produced concurrently.

In this case, we can achieve concurrency even when lazy evaluation is used. Multiple inputs to a transducer can be produced in parallel while remaining within the framework of lazy evaluation.

3.1.3 Merge Parallelism

Merge parallelism can be achieved if the task of producing a single input stream to a transducer can be shared by multiple transducers. Consider a transducer which produces a stream that represents the relation, **R**, which is comprised of fragments, **R1**,...,**Rn**. We can express this as **select([tuples(R1),...,tuples(Rn)])**. Decomposing this transducer into its component transducers, we get the graph in Figure 3. **select**/1 is a transducer which accepts as input a list of transducers whose outputs are to be interleaved in some unspecified manner to form one stream. **tuples**/1 is a transducer which produces a stream of output tuples corresponding to the relation name that is given as an argument.

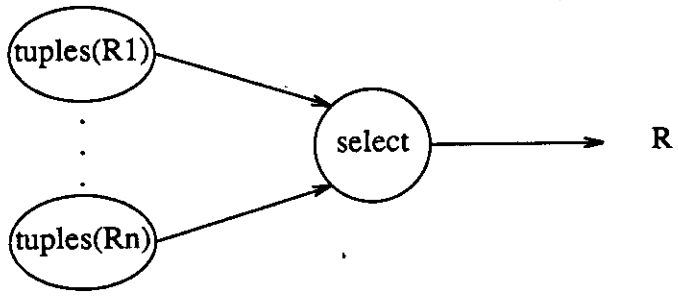

Figure 3 Merge Parallelism

3.2 Achieving Concurrency Through Annotations

Parallelism is achieved when different portions of a computation are performed concurrently on different processors. However, having every node in a transducer network represented by a different processor could result in too much overhead; the communication overhead could far outweigh the concurrency gained. Instead, a computation must be judiciously partitioned over the available processors. One must consider not only the potential concurrency in a given program, but also the overhead introduced by exploiting that concurrency. One must consider the computation costs

of various portions of the program as well as the communication costs involved. Such factors are often determined by the nature of the input data as well as the structure of the program itself.

Automated optimization in a distributed environment is an open problem. We are currently considering how one could assign weightings to individual transducers so that an optimizer could automatically make decisions about how to distribute a given program. The output of such an optimizer would be an annotated program. At present, we require programmers to provide such annotations. The prefix annotation, #, allows programmers to indicate which transductions it would be cost-effective to perform concurrently. All three types of parallelism mentioned in the previous section can be achieved through the use of the # annotation.

Stream parallelism is achieved in the example of Figure 1 by annotating it as follows: `sq(#even(S))`. Even integers are filtered out of the stream `S` on some remote site and streamed to the local site where they are squared as they arrive. Alternatively, the program could be annotated as `sq(even(S) @ ipswich)` to specify that reduction of the term `even(S)` is to take place on the site whose name is `ipswich`.

AND-parallelism is achieved in a similar manner. The transducer `add(sq(A), sq(B))` is annotated as `add(#sq(A), #sq(B))` to indicate that the two input streams are to be produced concurrently.

Finally, merge parallelism is achieved in the example of Figure 3 by annotating it as follows: `select([#tuples(R1), ... , #tuples(Rn)])`. The exact interleaving of elements on the stream produced by `select`/1 is determined by the availability of data elements from each of the concurrent processes. The imple-

mentation of **select**/1 is discussed fully in [9].

Decisions about how to annotate an ASPEN program may be guided by several different factors. The programmer may have knowledge of the expected data and how it might affect computation costs. Some transductions may be known to be computationally expensive, while others are comparatively inexpensive. Many other factors may determine the best way to annotate a program, but regardless of how the program is annotated, the underlying program is not changed. Only the annotations are changed. Thus, the original specification of the transducer is the same whether it will be executed concurrently or sequentially.

We support the annotation of rule invocations rather than the annotation of rule definitions for several reasons. Some invocations of a rule warrant the use of a separate process while other invocations of the same rule do not. The same rule may be invoked once to solve a problem which the programmer feels is likely to be expensive, while another invocation might be to solve a rather trivial problem. Secondly, transducers are quite often defined in an iterative manner. For example, consider the transducer **intsfrom(N)**, which generates a stream of integers starting from **N**.

```
intsfrom(N)  =>  [N | intsfrom(N + 1)].
```

If the definition of **intsfrom**/1 were annotated, a new process would be created for every iteration, or for every output element produced. By annotating invocations, we can specify that the entire stream represented by **intsfrom**/1 is to be produced by a single process. Finally, the effect of annotating definitions can be achieved easily by annotating invocations.

3.3 Extending Log(F)

While the # annotation is sufficient for expressing all three types of concurrency, there are a number of important cases in which it is not entirely appropriate. In this section, we describe those situations in which the # annotation alone is insufficient and propose alternatives which greatly improve performance. These language extensions are necessary for the efficient execution of sequential programs as well as concurrent programs.

3.3.1 Common Subexpressions

Pure functional programming languages allow programmers to construct only programs whose data flows are trees. Tree dataflows disallow the optimization of common subexpressions; if an expression is used in several places throughout a computation, it must be recomputed each time it is used. Allowing dataflows which are directed acyclic graphs (DAGs) rather than restricting the dataflows to be trees would permit optimizations such that these common subexpressions need only be computed once.

We introduce in this section a mechanism that allows common subexpressions to be optimized. This mechanism is available to the programmer so that programs with optimized common subexpressions can be expressed. Common subexpression optimization could be done automatically by a compiler. However, programmers may not want this optimization to be applied to all common subexpressions. In some cases, for example, the common subexpression may represent computations that are intended to produce different results upon different invocations. Such situations arise, for instance, in real-time processing when accessing a clock value. In such cases, optimizing common subexpressions may alter the behavior expected by the program-

mer.

Henderson [5] describes the introduction of *local definitions* into a functional programming language to allow the optimization of common subexpressions. We have introduced into ASPEN a mechanism which appears to the programmer very similar to local definitions. However, local definitions in ASPEN programs must guarantee not only that the same initial value is assigned to all occurrences of a common subexpression; they must also guarantee that whenever one occurrence of a common subexpression is reduced, the result of that reduction will be visible to all occurrences of the common subexpression. Thus, no redundant reductions are performed.

By introducing an infix operator, **where**, we make it possible to express transducers with optimized common subexpressions. Suppose that we had the following transducer:

```
sq_plus_dbl(S)   =>  add(sq(S), dbl(S)).
```

Each instance of **S** is treated as a separate stream. So, each element of **S** is calculated twice, once for each consumer. This overhead could be quite significant if **S** represents a complex transduction. By rewriting the transducer as follows:

```
sq_plus_dbl(S)   =>  add(sq(R), dbl(R))
                 where
                 R <= S.
```

the overhead may be significantly reduced. Thus isolating common subexpressions in the **where** portion of a transducer guarantees that when any instance of the common subexpression is reduced, all instances will be reduced.

The compiler described in [9] translates rules of the above form into a form which may be compiled by the Log(F) compiler and executed. For example, the above transducer would be translated into the following.

```
sq_plus_dbl(S)  =>
     add(sq(common(T, S), dbl(common(T, S)).
```

The **common**/2 transducer guarantees that each time one instance of **S** is reduced, the reduction will be seen by all other instances. When the first occurrence of **common(T, S)** is reduced, the logical variable **T**, which is shared by all occurrences of the term, is bound to the result of the reduction. When the second occurrence of **common(T, S)** is reduced, **T** is found to be bound, so its value is returned immediately as the result of the reduction.

In order to support the optimization of common subexpressions in distributed executions, we introduce a new use of the **#** annotation. By annotating a common subexpression with a **#** annotation, the programmer can specify remote reductions whose results are consumed by multiple processes. As with the in-line use of **#**, replacing **#** annotations with **@**/2 annotations in the **where** portion of a transducer allows the user to explicitly specify the site to be used for reduction.

With this use of the **#** annotation, the above transducer can be annotated for parallelism as follows:

```
sq_plus_dbl(S)  =>  add(#sq(R), #dbl(R))
          where
          R <= #S.
```

The generation of the input stream, **R**, the **sq** mapping, the **dbl** mapping, and the addition of the resulting streams can all take place in parallel.

3.3.2 Multiple Outputs

Pure functional languages make the expression of transducers that have multiple outputs very difficult. One possible approach would be to have transducers generate streams of output structures, each with one argument for each output. This technique works well for transducers which produce all of their outputs at the same rate. However, it is awkward and inefficient for transducers that produce their different outputs at differing rates. Narain [12] proposes a method in which rewrite rules are extended with extra arguments to accumulate their outputs. When all of the streams have been collected in their entirety, a single output structure whose arguments represent the multiple outputs of the rewrite rule is generated. The major weakness of this approach is that no output is generated until all output streams have been computed in their entirety; such behavior fits very poorly into a concurrent stream processing environment.

In the ASPEN programming model, all transducers produce a single stream of output. Elements of the stream may be *typed* to indicate the logical output stream to which they belong. Consumers merely filter out the proper type. The problem of representing and implementing transducers with multiple outputs is now reduced to that of representing and implementing common subexpression optimization.

To solve the multiple output problem in sequential ASPEN programs, we make use of the common subexpression optimization techniques described previously. The transducer for quicksorting a stream illustrates their use. In the following example, `o1/1` and `o2/1` are assumed to be constructor symbols and thus are not rewrit-

ten.

```
quicksort([ ])  =>  [ ].
quicksort([H | T])  =>
     append(quicksort(first(S)),
          [H | quicksort(second(S))])
     where
     S <= partition(T, H).

partition([ ], P)  =>  [ ].
partition([H | T], P)  =>
     if(H =< P,
     [o1(H) | partition(T, P)],
     [o2(H) | partition(T, P)]).
```

Note that **partition**/2 produces a single stream of output. Each element of that output stream is specified as either the first output (**o1**) or the second output (**o2**). Each element of the output stream is made available to consumers as soon as it has been calculated.

The transducers **first**/1 and **second**/1 simply filter the appropriate output from the stream. They are defined below.

```
first([ ]) => [ ].
first([E | R]) =>
     if(E = o1(T), [T | first(R)], first(R)).

second([ ]) => [ ].
second([E | R]) =>
     if(E = o2(T), [T | second(R)], second(R)).
```

This set of filters is easily extended to allow functions with many outputs.

The quicksort transducer for distributed execution is shown below.

```
quicksort([ ])  =>  [ ].
quicksort([H | T])  =>
     append(#quicksort(first(S)),
          [H | #quicksort(second(S))])
     where
     S <= #partition(T, H).
```

The **partition** transducer need not be changed. The same filters that are used in

the sequential case can also be used here.

In the distributed execution of `quicksort`/1, `partition`/2 produces two copies of its output stream. Both consumers receive all elements on the stream, regardless of type. Each stream is filtered as it arrives at its consumer. An obvious optimization is to force the filtration to take place at the producer, thus reducing the total amount of data that must be transmitted and buffered. For context-free filters like `first`/1 and `second`/1, this is a trivial optimization. For more complex filters (e.g. filters that maintain state), the optimization may be more difficult.

3.4 Increasing Merge Parallelism

Now that we are able to deal effectively with transducers that have multiple outputs, we can exploit another form of merge parallelism. This form of merge parallelism arises when a node in a pipeline is replaced by a graph of processes which is capable of processing multiple elements of the stream concurrently. A `split` transducer at the beginning of this network splits the input stream into multiple substreams. Each of these substreams is operated upon by some number of intermediate transducers. The resulting streams are then merged back together at the end of the network by a `merge` transducer.

In [9] we describe several methods for exploiting this form of merge parallelism. We describe here one method which extracts merge parallelism through cooperation between the `split` transducer and the `merge` transducer. In addition to producing streams of data, the `split` transducer also produces a stream of control information. This control stream is used by the `merge` transducer to assure that the streams are merged in the proper order. We give the ASPEN code for such a merge operation below. The first argument to `merge` is the control stream. Each element,

E, indicates whether to accept input from the first stream (**E** = 1) or the second stream (**E** = 2).

```
merge([ ], X, Y)   =>  [ ].
merge([1 | Cs], X, Y)  =>  x_merge(Cs, X, Y).
merge([2 | Cs], X, Y)  =>  y_merge(Cs, X, Y).

x_merge(C, [X | Xs], Y)   =>  [X | merge(C, Xs, Y)].

y_merge(C, X, [Y | Ys])   =>  [Y | merge(C, X, Ys)].
```

One can now construct a transducer that splits its input stream based upon some characteristic of the elements. Such a method is very useful when intermediate transformations differ based upon the split characteristic. Each stream that is generated by the split transducer can be operated upon by a unique transducer, rather than trying to collapse the functionality of several transducers into one. Such a method is used in the Stream Machine [2] for an application in the field of oil exploration. Three streams of measurements are used to calculate the volumetric percentage of hydrocarbons in the rock formations at various depths in a borehole.

These measurements are lithology (the type of rock), electrical resistivity, and transit time (the time for a sound wave to propagate through the formation). The difference between the porosity of the rock and the percentage of water in the formation yields the volumetric percentage of hydrocarbons in the formation. The percentage of water is computed directly from the resistivity. The porosity is computed from the transit time, but different calculations are selected based upon the type of rock. It is for this calculation that the split/merge technique is applicable.

Figure 4 shows a re-implementation of their example in ASPEN. The parameter **Rw** is the resistivity of water and is constant for each borehole. The **selectModel** transducer produces three output streams, **o1**, **o2**, and **c**. **o1** and **o2** are consumed by **sPorosity** and **lPorosity**, respectively. The stream **c** is used

by **merge** to assure that the output streams of **sPorosity** and **lPorosity** are merged in the proper order.

```
hydrocarbons(Rock, Time, Resistivity, Rw) =>
     subtract(#porosity(Rock, Time),
          #water(Resistivity, Rock, Rw)).

porosity(Rock, Time) =>
     merge(control(R), #sPorosity(first(R)),
          #lPorosity(second(R)))
     where
     R <= #selectModel(Rock, Time).

selectModel([ ], [ ]) =>  [ ].
selectModel([Rock | Rs], [Time | Ts]) =>
     case(Rock, [
          sandstone: [o1(Time), c(1)
               | selectModel(Rs, Ts)],
          limestone: [o2(Time), c(2)
               | selectModel(Rs, Ts)]
          ]
     ).

water([ ], [ ], Rw) => [ ].
water([Resistivity | Rs], [Rock | R2s], Rw) =>
     case(Rock, [
          sandstone : [sqrt(0.8 * Rw/Resistivity)
               | water(Rs, R2s, Rw)]
          limestone : [sqrt(Rw/Resistivity)
               | water(Rs, R2s, Rw)]
          ]
     ).

lPorosity([ ]) => [ ].
lPorosity([Time | Ts]) =>
     [(Time - 47)/142 | lPorosity(Ts)].

sPorosity([ ]) => [ ].
sPorosity([Time | Ts]) =>
     [(5/8)*(Time - 55)/Time | sPorosity(Ts)].
```

Figure 4 Oil Exploration Example

This example introduces a new filter, `control`, which extracts control messages from a stream. The transducer `case` is also introduced to allow the programmer to construct case statements as in other high-level programming languages.

The original Stream Machine program consists of six modules of Pascal code. The ASPEN code consists of six transducers composed entirely in ASPEN. The Stream Machine code requires an additional code segment to specify how the Pascal modules were to communicate. No such module is required here; the relationships are specified in the transducers through functional composition.

The split/merge technique is very useful when the operations to be performed on the intermediate streams are complex and are intended to be executed in parallel. Perhaps even more significant is that this technique allows better software engineering of ASPEN programs. Note that the porosity models in the above example are defined independently of each other as transducers. Without using split/merge techniques, they would have had to be collapsed into one transducer or they would have had to be defined as operations on individual elements rather than streams, a severe restriction on their potential power as well as the degree of parallelism that may be exploited.

The above method has the advantage of not requiring any modifications to the transducers that process the intermediate streams. The same code is used for these transducers whether they are participating in a split/merge operation or not. However, the intermediate transducers are required to perform one-to-one mappings; otherwise, ordering cannot be maintained. In [9], we present an option which allows one-to-N mappings while still maintaining order on the stream. In this scheme, we introduce synchronization markers into the stream that is to be split. These synchronization markers are used to insure that the streams are merged in the proper order even if intermediate transducers perform mappings which add or remove elements from the

stream. The Sync Model [8], a parallel execution model for logic programming, uses a similar technique to achieve parallelism in the all-solutions evaluation of logic programs.

4 Implementation

ASPEN is implemented in terms of a server-pool model. All sites that are to provide ASPEN support allocate a pool of server processes and await requests from clients. Servers receive messages which request them to reduce a term and stream the results back to the client.

4.1 Multiple Outputs

When a server is to produce output for **N** (**N** > 1) consumers, it is informed as to the number of consumers to expect[†]. A *stream descriptor* is returned to the client. This stream descriptor may be passed to the **N** consumers. Reduction of a stream descriptor causes a connection with the server to be requested. After this connection is established, results are streamed back to the consumer. If fewer than **N** connection requests have arrived, results of the reduction are buffered until all consumers have arrived.

If a stream descriptor is passed to a transducer which never reduces that stream descriptor, the server process will block and will never be deallocated. To prevent this, we provide the ability to cancel input streams. By syntactic analysis, we can determine which arguments to a transducer will never be reduced. When such an argument is detected, the transducer is modified so that it will cancel the argument. Cancelling a term causes the term to be analyzed in search of stream descriptors.

[†]Programmers are not expected to provide this number. It is generated by a compile-time analysis of the program.

When a stream descriptor is found, a message is sent informing the associated server that the stream is no longer necessary.

4.2 Constrained Eagerness

By default, servers behave in a completely eager fashion. That is, a server continues to produce output until the end of the output stream is generated or until all of its consumers have cancelled. While such a mode of operation can result in a high degree of parallelism, it can also result in a great deal of wasted work. If a consumer decides at some point that it has seen enough of the input stream and does not wish to see any more, all of the subsequent terms that have been eagerly created by the producer represent wasted work. Creation of these extra terms may have actually resulted in the creation of additional processes, thus increasing the amount of wasted work dramatically. In such situations, much computation may be saved without sacrificing concurrency by producing the stream in a constrained fashion. We have generalized the *anticipation coefficient* [6] to allow programmers to control the eagerness with which servers execute. Specification of the production mode *lazy(N, M)* indicates that a stream is to be produced in bursts. The first burst is to contain N elements, the maximum number of terms that the programmer wishes to be outstanding on the stream at any one time. Thereafter, elements are produced in bursts of size M each time more elements are requested. All modes of production along the continuum from completely lazy to completely eager can be achieved through the use of lazy(N, M). By specifying lazy(0, 1), completely lazy behavior can be achieved. A behavior in which bindings are produced on demand, N at time, can be achieved by specifying lazy(N, N). A behavior in which the consumer is kept busy any time there are fewer than N bindings on the stream (equivalent to the anticipation coefficient above) can be achieved by specifying lazy(N, 1). By specifying lazy(∞, ∞), completely eager

behavior is achieved. Note that specification of lazy(∞, ∞) is equivalent to specifying no constraint at all.

5 CONCLUSIONS

Stream processing is a powerful programming paradigm. It allows rich and varied programs to be expressed elegantly and solved in an efficient manner. Several interesting applications of stream processing are presented in [13] and [3].

In this paper, we have presented a stream processing environment called ASPEN. Annotations allow programmers to exploit concurrency without rewriting their programs. A term may be annotated with the # annotation if the programmer feels that the performance of his program may be improved by evaluating that term in parallel with the rest of the program. The @ annotation allows the programmer to explicitly specify a site upon which the reduction of a term is to take place. This can be extremely useful when the term to be reduced requires access to data which resides only at a specific site. Programmers can express programs in ASPEN whose dataflows are DAGs as well as trees. Using simple annotations, programmers can express programs which exploit three types of parallelism, stream parallelism, AND-parallelism, and merge parallelism, making ASPEN a powerful stream processing language.

The annotations presented in this paper allow programmers to freely mix eager and lazy evaluation. Portions of an ASPEN program which run within a single process are evaluated in a lazy fashion. This is desirable for two reasons. First, there is no need to buffer intermediate results since each result is calculated only when it is needed. Second, the delay before the first result is produced is much lower than if eager evaluation were used. These factors are extremely significant for large, poten-

tially infinite, streams. By introducing eagerness at process boundaries, concurrency can be exploited. We have introduced further annotations to allow the programmer to constrain this eagerness in order to prevent one process from running too far ahead of another and requiring unbounded buffer space.

The annotations presented in this paper are intended to be, to the greatest extent possible, semantics-free. That is, their introduction into a program changes performance characteristics of the program only, without changing the semantic meaning of the program. When annotations which explicitly specify an execution site are used to gain access to a particular database relation or fragment, there is, of course, a semantic impact upon the program. Those annotations which do not explicitly specify an execution site can be taken as *hints*, to be either ignored or heeded during execution, at the discretion of the system. Since adherence, or lack of adherence, to these annotations has no impact on the semantics of the program, there is significant potential for making program distribution decisions automatically, with no need for the programmer to annotate his program.

Previous work in distributed stream processing has suffered from one of two drawbacks. First, the work by Kahn and MacQueen [6] and by Barth, et al [2] lack *uniformity*. That is, there are two different programming models. Processes were specified in one manner, and the linking together of those processes is specified in a different manner with explicit communication between the processes. In ASPEN, communication is implicit. Turning a sequential ASPEN program into a distributed program requires only introducing the appropriate annotations into the program and does not require rewriting any portion of the program. Similarly, turning a distributed program into a sequential program requires only removing annotations.

The second drawback with other distributed stream processing systems is that it is often difficult to specify the granularity of concurrency. In Flat Concurrent Prolog [14], stream processing is achieved by repeatedly refining the instantiation of a shared variable. CFL, a concurrent functional language has been proposed by Levy and Shapiro [7] as a user-level language for Flat Concurrent Prolog, relieving the programmer of the burden of specifying correct synchronization. The current evaluation technique for CFL is eager evaluation. CFL provides no mechanism for the user to control concurrency.

Parlog [4] allows programmers to annotate a group of goals to indicate that it is to be run entirely within a single process. However, Parlog does not support lazy evaluation within a single process. Thus, introduction of sequential nodes in a stream processing network will result in a disruption of flow through the network. In ASPEN, programmers specify the grain of concurrency with simple annotations. By supporting lazy evaluation within a single process, ASPEN allows the construction of efficient stream processing networks.

The proposals by Kahn and MacQueen [6] and Barth, et al [2] suffer from similar problems. Although both support the evaluation of multiple modules within a single process, explicit input/output operations and scheduling are still necessary. Demand-driven execution is the default behavior in sequential ASPEN and is achieved within a single process without incurring the overhead of IPC mechanisms or schedulers.

ASPEN has proved to be an effective environment in which to express and evaluate data-intensive applications. As ASPEN is used more extensively, its strengths and weaknesses will no doubt become more apparent. We believe that the strengths will outnumber the weaknesses and that addressing the weaknesses will

require only incremental changes to the implementation presented here.

REFERENCES

[1] Abelson, H. and G. Sussman, *The Structure and Analysis of Computer Programs,* MIT Press, Boston, MA (1985).

[2] Barth, Paul, Scott Guthery, and David Barstow, "The Stream Machine: A Data Flow Architecture for Real-Time Applications," pp. 103- 110 in *Proceedings 8th International Conference on Software Engineering*, London, England (August 1985).

[3] Chau, L., "Functional Grammars and Stream Pattern Matching," Draft, UCLA Computer Science Dept. (March 1988).

[4] Gregory, S., *Parallel Logic Programming in PARLOG: The Language and its Implementation,* Addison-Wesley, Reading, MA (1987).

[5] Henderson, Peter, *Functional Programming: Application and Implementation,* Prentice-Hall, Englewood Cliffs, New Jersey (1980).

[6] Kahn, Gilles and David B. MacQueen, "Coroutines and Networks of Parallel Processes," *Proceedings of the IFIP Congress 77,* pp.993-998 (1977).

[7] Levy, Jacob and Ehud Shapiro, "CFL - A Concurrent Functional Language Embedded in a Concurrent Logic Programming Environment," CS86-28, The Weizmann Institute of Science, Rehovot, Israel (December 1986).

[8] Li, P-Y.P. and A.J. Martin, "The Sync Model: A Parallel Execution Method for Logic Programming," *Proc. Symp. on Logic Programming*, pp.223-234, IEEE Computer Society (1986).

[9] Livezey, Brian Kevin, "A Stream-Based Language for Concurrent Programming and Distributed Database Access," M.S. Thesis, UCLA Computer Science Department, Los Angeles, California (1988).

[10] Muntz, R.R. and D.S. Parker, "Tangram: Project Overview," Technical Report CSD-880032, UCLA Computer Science Dept., Los Angeles, CA 90024-1596 (April 1988).

[11] Narain, S., "Log(F): A New Scheme for Integrating Rewrite Rules, Logic Programming and Lazy Evaluation," CSD-870027, UCLA Computer Science Dept., Los Angeles, CA (1987).

[12] Narain, S., "LOG(F): An Optimal Combination of Logic Programming, Rewrite Rules and Lazy Evaluation," Ph.D. Dissertation, UCLA Computer Science Dept., Los Angeles, CA 90024-1596 (1988).

[13] Parker, D.S., R.R. Muntz, and L. Chau, "The Tangram Stream Query Processing System," Technical Report CSD-880025, UCLA Computer Science Dept., Los Angeles, CA 90024-1596 (March 1988).

[14] Shapiro, E.Y., *Concurrent Prolog: Collected Papers,* MIT Press, Cambridge, MA (1987).

[15] Sterling, L. and E. Shapiro, *The Art of Prolog,* MIT Press, Cambridge, MA (1986).