STREAM PROCESSING

Allen Goldberg*
University of California, Santa Cruz
and
Kestrel Institute, Palo Alto, CA

Robert Paige**
Rutgers University
New Brunswick, NJ

Abstract

Stream processing is a basic method of code optimization related to loop fusion that can improve the space and speed of iterative applicative expressions by a process of loop combination. It has been studied before in applications to program improvement and batch oriented database restructuring. Previous attempts at implementation have been either highly restrictive, have required manual intervention, or have involved naive strategies resulting in impractically slow running times.

This paper defines a new model for a stream processing problem for handling a wide class of applicative expressions that can be evaluated by looping in an unordered way through a single set or tuple valued argument. This problem is formulated graph theoretically and shown to be NP-Hard, even in the presence of constraints. Two new efficient heuristic algorithms will be presented. The efficiency of these solutions allows stream processing to be applied dynamically as a database query optimization, and as an important component of a high level language optimizer.

Our method of stream processing has been implemented and used effectively within the RAPTS transformational programming system.

1. Introduction

Translation of relational database queries into efficient executable code is an interesting and challenging problem facing database, data structure, and program language researchers. The optimization techniques stressed in the database literature mainly involve algebraic manipulations that change the order of evaluation of subqueries so that selections and projections are performed before join and cartesian products [27]. Since the cost of forming a join or cartesian product of two relations is related to the product of the sizes of these relations, it makes sense to first restrict the sizes of each relation by performing selections and projections. Yet after a query is transformed in this way, a naive execution requires each subquery to be executed and each intermediate relation to be stored. Stream processing is a technique that seeks to avoid intermediate storage and to minimize sequential search.

In the next section the rudiments of our technique are explained in terms of a simple example. Section 3 presents a precise formal description of the stream processing problem. Sections 4 and 5 discuss the complexity of solving this problem. Section 6 gives relevant backround. The final section concludes with open problems.

2. Stream Processing Example

To illustrate stream processing applied to database queries, we will assume a simple database modeled after the SETL programming language [24]. Primitive data values consist of standard boolean, numeric, and string types organized within finite tuples, sets, and maps. Tuples are dynamic vectors, and are ordered from the first to the last component; sets are unordered, and cannot contain repeated elements; maps are dynamic binary relations represented by sets of pairs [x,y] each of which associates a domain value x with a corresponding range value y. The elements of tuples, sets, and maps can themselves be tuples, sets, and maps to any depth of nesting.

The SETL-like query language incorporates conventional dictions as are found in other programming languages, and with a few exceptions uses universally accepted mathematical set notations (see [11]). Table 1 below lists the main notational features of SETL used in this paper.

Operation	Remarks
s with:= x	add element x to set s
s less:= x	delete element x from set s
x ∈ s	set membership
s ∪:= delta	add set delta to set s
s −:= delta	delete set delta from set s
f(x) := y	modify function f at point x to have value y
f(x1,...,xn)	function retrieval
(For x ∈ s) Block(x) end forall	Execute Block(x) for every value x ∈ s
$\{x \in s \mid k(x)\}$	set former
$\exists\ x \in s \mid k(x)$	existential quantifier
$\forall\ x \in s \mid k(x)$	universal quantifier
s ∪ t	set union
s ∩ t	set intersection
s − t	set difference
f[s]	image set
#s	set or tuple cardinality
+/s	aggregate sum over set or tuple s
∪/s	aggregate union
min/s	aggregate minimum
f^{-1}	map inverse; i.e., {[y,x]: [x,y]∈f}
f.g	map composition; i.e., {[x,z]: [x,y] ∈ g, z ∈ f{y}}
s x t	cartesian product; i.e., {[x,y]: x ∈ s, y ∈ t}

Table 1.

The tuple, set, and map data types together with powerful set theoretic operations facilitate the modeling of high level optimizations used to improve relational queries; but more than that, we can even model the physical structure of databases conveniently so that lower level optimizations can be represented abstractly. The discussion of stream processing will benefit from this set theoretic model.

We note that database relations can be represented either as sets of tuples, or as 'entity' sets of blank atoms (i.e., Lisp gensyms) together with maps from entity sets into primitive values. The relational notion of attribute is captured in the set theoretic notion of map. All of the primitive relational operations - selection, projection, join, difference, and union - can be represented by various forms of the set former (see [20] for further discussion of this thought).

We assume that sets are implemented on secondary storage using an extensible hash file organization that supports membership tests, element additions, and element deletions in unit time, and iteration through all set elements in time proportional to the cardinality of the set [5]. The domains of maps can be hashed like sets, and range elements can be accessed rapidly using pointers.

Fagin, et. al. originally proposed that the directory portion of their extensible hashing file structure could reside in primary memory to improve the performance of the operations just mentioned [5]. We note that this arrangement also makes it possible to perform direct access to a record in the same file that is being iterated through without any degradation in speed. This remarkable feature of extensible hashing, which has not been emphasized before, is exploited in our stream processing model.

It is convenient to introduce the essential ideas behind stream processing using a simple example. Consider how to evaluate the query: 'find all employees whose salary is greater than the average employee salary'. This can be expressed

$$\{x \in emps \mid sal(x) > (+/[sal(y): y \in emps])/\#emps\} \quad (1)$$

where emps is a set of employees, sal is a function from employees to salaries, [sal(y): y ∈ emps] forms a vector of salaries for all employees, +/[sal(y): y ∈ emps] is the sum of all these salaries, and #emps is the number of employees in emps.

A standard, but naive, way of evaluating query (1) would be to compute its inner to outer subexpressions as in the following 4 assignments:

$$\begin{aligned} &c1 := [sal(y): y \in emps]; \quad (2)\\ &c2 := +/c1;\\ &c3 := \#emps;\\ &c4 := \{x \in emps \mid sal(x) > c2/c3\}\end{aligned}$$

These 4 assignments represent 4 sequential passes through possibly large sets or tuples, and 3 intermediate stored results before c4 is finally computed. As we will see, stream processing will enable c4 to be computed by a straightforward procedure using only 2 sequential passes and without having to store the costly tuple c1.

We assume that each of the subexpression calculations in the sequence (2) is computed by a single sequential pass through a set or tuple valued parameter, called a *stream* parameter.[3] Parameters that are not stream parameters are called *nonstream* parameters. Table 2 below illustrates the stream parameters for c1-c4:

```
c1 := [];               $initialize c1 to empty tuple;
(for x ∈ emps)          $perform a sequential search
  c1 with:= sal(x);     $through emps; concatenate
end;                    $the salary of x to c1

c1 := [];
(for [e,amount] ∈ sal)   $search through the salary
  c1 with:= amount;      $index; concatenate the
end;                     $salary of e to c1
```

a. emps and sal are 2 stream parameters for c1

```
c2 := 0;
(for x ∈ c1)  $search the c1 tuple from the 1st to
  c2 +:= x;   $last component; add the component
end;          $value x to c2
```

b. c1 is the stream parameter for c2

```
c3 := 0;
(for x ∈ emps)  $search through emps
  c3 +:= 1;     $increment c3
end;
```

c. emps is a stream parameter for c3

[3]For sets the sequential pass is unordered; for tuples the search proceeds from the first to last tuple component.

```
c4 := {};
(for x ∈ emps)
  if sal(x) > c2/c3 then  $if the salary of x is
    c4 with:= x;  $greater than the average
  end if;         $employee salary, add x to
end;              $the set c4

c4 := {};
(for [e,amount] ∈ sal)
  if amount > c2/c3 then
    c4 with:= e;  $add e to the set c4
  end if;
end;
```

d. the stream parameters for c4 are sal and emps; c2 and c3 are nonstream parameters

Table 2.

Note that c1 and c4 can be computed using either of two alternative stream parameters.

Table 2 exhibits two kinds of loop combination techniques. One of these techniques can be used when the value of one expression is a stream parameter for another expression. For example, c1 is the stream parameter for c2, and so c1 and c2 can be computed together in a single sequential pass through emps; i.e.,

```
c2 := 0;                                   (3)
c1 := [];
(for x ∈ emps)
  c2 +:= sal(x);
  c1 with:= sal(x);
end;
```

The kind of loop combination used in (3) is called *vertical loop fusion*. Note that the natural data dependency of c2 on c1, which is explicit within (2), is absent in (3). Consequently, the evaluation of c1 within (3) does not contribute to the calculation of c2 and can be eliminated.

Another loop combination technique, called *horizontal loop fusion*, can be used when two expressions share a common stream parameter, as in the case of c1 and c3.

```
c3 := 0;                                   (4)
c1 := [];
(for x ∈ emps)
  c3 +:= 1;
  c1 with:= sal(x);
end;
```

Vertical and horizontal fusion can also be combined. For example, since c1 is a stream parameter of c2, and since c1 is constructed within the loop (4) (which resulted from horizontal fusion of c1 and c2), we can embellish the loop (4) by applying vertical fusion of c2 with respect to c1 within (4). The result is shown just below:

```
c3 := 0;                                   (5)
c2 := 0;
c1 := [];
(for x ∈ emps)
  c3 +:= 1;
  c2 +:= sal(x);
  c1 with:= sal(x);
end;
```

One important thing to note about the code (3), (4), and (5) which results from loop fusion is that the sequence of modifications to each of the variables c1, c2, and c3 is identical to the corresponding sequence found in the unoptimized code (executed with respect to one of the stream parameters) shown in Table 2. Thus, whenever loop fusion is applied, improvement over some standard unoptimized implementation is guaranteed. In general, improved performance results from cutting down on sequential search and eliminating code made useless by vertical fusion.

Application of loop fusion is mitigated by correctness as well as performance considerations. For example, it would not be correct to apply horizontal fusion of c4 and c3 with respect to their common stream parameter emps. If this kind of fusion were applied, c3 would be constructed incrementally (so that its full value would not be available) in the same loop in which c4 is constructed incrementally. However, the code to construct c4 incrementally with respect to emps makes reference to the full value of c3, which is not available. (see Table 2 (d)) For the same reason horizontal fusion cannot be applied to c4 and c1.

Code that results from loop fusion in accordance with restrictions like those just mentioned and that evaluates several expressions in the same loop is called a *valid loop*. It is convenient to use the term 'loop' to mean 'valid loop'.

Sequences of valid loops are also restricted. For example, since c4 depends on c3 but cannot be fused with it, the loop used to construct c3 must precede the loop used to construct c4. Sequences of loops that obey such restrictions are called *schedules*. The four loops (2) used to construct c1, c2, c3, and c4 represent a naive schedule with no loop fusion. Another schedule for these expressions consists of 3 loops: the first loop results from vertical fusion of c2 and c1 with respect to sal; the next loop is formed by evaluating c3 using the code shown in Table 2 (c); the last loop is an evaluation of c4 by either of the alternatives given in Table 2. A schedule for c1, c2, c3, and c4 can even be made from only 2 loops – loop (5) followed by either of the loops shown in Table 2 (d) for evaluating c4. This two-loop schedule is optimal in the sense that there are no schedules with fewer than 2 loops using vertical and horizontal loop fusion transformations. The problem of *stream processing* is to find optimal schedules.

In this paper we only consider stream processing for *iterative expressions*; i.e., nonrecursively defined expressions that can be evaluated using an unordered sequential search with respect to one stream parameter. Thus we do not consider evaluation by use of merging two streams, or related techniques in which stream values are sorted or ordered in some way. Consequently, join operations must be performed by methods not involving sorting (e.g., by the 'inner-outer' method [25]). A relational union is handled by decomposing it into two operations (consequently, two loops), a copy and the addition of one relation to another. To perform the cartesian product of two relations, we must choose one of the relations as the stream parameter from which the outer loop is formed. For each iteration through this loop, the cartesian product is built up incrementally by peforming an inner loop through the other relation.

As we will see later on, the problem of stream processing is so hard that with these and other restrictions the problem remains NP-Hard. Yet stream processing is a fundamental optimization method essential to the efficient translation of database queries and very high level languages. As a programming language optimization, it may be viewed together with finite differencing [21] as a transformation that gives rise to variables and turns functional code into imperative code. Specifically, stream processing could establish collections of equalities indicated within functional programs; such equalities could then be maintained in program regions where they are needed by finite differencing. (See the Appendix for an example of this idea.) Stream processing is also an interesting and unusual scheduling problem involving precedence constraints between processors (i.e., subexpressions) and also between groups of processors (i.e., between loops). We feel that stream processing should be investigated, and the simple model discussed in this paper is offered as a basis that provides enough utility to be convincing, and a sufficient window of tractibility to invite further work with more complicated models.

3. A Graph Theoretic Framework for Stream Procesing

Consider the problem of computing a collection of queries. Before stream processing can be used we must decompose all these queries into the basic iterative expressions out of which they are composed.[4] Suppose that there are n such iterative expressions E1 = f1, E2 = f2,...,En = fn, ordered from inner to outer subexpression; i.e., there are no occurrences of Ej in the expressions fi, 1<=i<=j<=n. The expressions can then be computed naively by performing n assignments.

```
    E1 := f1; E2:= f2;..En := fn;
```

which amounts to n loops. The stream processing problem is to partition the computed variables E1,...En into groups of variables. The variables in each group will be constructed within the same loop by vertical and horizontal fusion so that an optimal schedule (with the fewest number of loops) is obtained.

A more formal description of the problem can be stated using a graph theoretic representation. We construct a dependency dag D = (V, E) with vertices V and edges E in the following way:

i. V = {E1,...,En,vn+1,...,vn+k}, where vn+1,...,vn+k are all the variables on which the expressions f1,...,fn depend other than E1,...,En ;

ii. E = {[Ei,x]: i=1..n, x ∊ V | x occurs within fi};

It is also useful to define the set of stream edges; i.e.,

```
  stream_edges = {[Ei,x] ∊ E | x is a stream parameter
                   of fi}
```

We assume that all iterative expressions have at least one stream edge. All edges in E that are not stream edges are called nonstream edges.

An optimal schedule can then be found by the following steps:

i. Choose one stream out-edge [E,x] for every internal vertex E. This indicates that E will be computed incrementally with respect to its stream parameter x.

ii. Partition the edges chosen in step (i.) into trees with edges directed toward the root. Each tree represents a loop involving a sequential pass through the variable associated with the root. Within each tree the sybling vertices are horizontally fused with respect to the parent vertex. Each grandchild vertex in a tree is vertically fused with its parent.

iii. Within each tree selected in step (ii), all nontree edges of E must lead to the root of the tree. Within a schedule, no edge can lead from an internal vertex of one tree t1 to an internal vertex of another tree t2 scheduled after t1. These rules ensure that no expression can be referenced until it has been fully computed. In particular, the value of the expression associated with the root of a tree t must be computed at the point when t is scheduled.

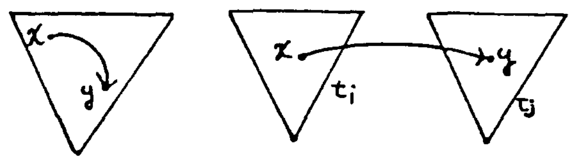
iv. Choose a partition in step (ii) with a minimum number of trees and that also satisfies the conditions of (iii).

A precise abstract but computable formulation of the steps (i)-(iv) above is expressed just below. We use the notation ∋s to denote the value of an arbitrary element selected from the set s. The expression **domain t** denotes all vertices of the tree t other than the root. (Such vertices are associated with the computable variables Ei,i=1..n.)

```
  $find a tree partition of the stream edges
  Find {t1,...,tn} partitionof
     ∋{ F ⊂ stream_edges | (∀ x ∊ domain stream_edges
        | #F{x} = 1)} |
  $ y can never be referenced before it is fully computed
          (∀ 1 <= i <= j <= n | (not∃ [x,y] ∊ E - tj |
                        x ∊ domain ti & y ∊ domain tj )
             minimizing n                $optimality condition
```

Figure 1 exhibits the two cases of prohibitted edges.

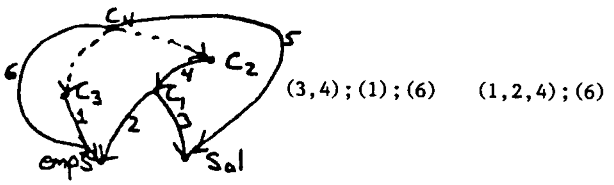
[4] Of course, such decomposition can greatly effect the success of stream processing. However, this topic will not be considered in the paper.



a. A nontree edge between 2 internal nodes of a tree b. An edge between 2 trees ti and tj,i<j

Figure 1.

Figure 2 is a graph representation of the stream processing problem discussed in the last section.



a. Problem Instance b. Suboptimal Schedule c. Optimal Schedule

Figure 2. Broken lines denote nonstream edges.

4. Complexity Results

Unfortunately, the following theorem states that the stream processing problem is intractable in general.

Theorem 1: The Stream Processing Problem (SPP) is NP hard.

proof) Consider a simple instance of the stream processing problem, E1=f1,...,En=fn,vn+1...,vn+k, where none of the fi's depend on any of the Ej's, and the dag representation only contains stream edges. Then we only have horizontal fusion. (See figure 3 below.)

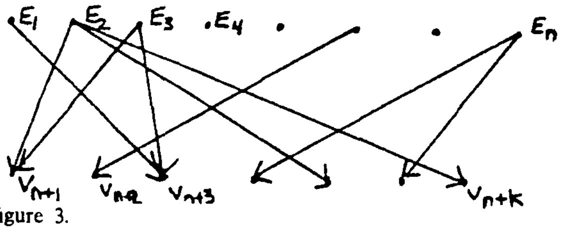


Figure 3.

An optimal schedule is equivalent to finding a minimal set of input variables vn+1,...,vn+k connected to all expressions f1,...,fn. But this is a formulation of the NP-Hard Hitting Set Problem [9]. ∎

Because of this negative result it would seem to be worthwhile to make the problem easier to solve by imposing some additional conditions. The obvious restriction is to preselect the stream edges so that each basic iterative expression has only 1 stream parameter. In terms of the dag representation this restriction makes the stream edges a forest. But surprisingly, this restricted stream processing problem (RSPP) is also NP-Hard.

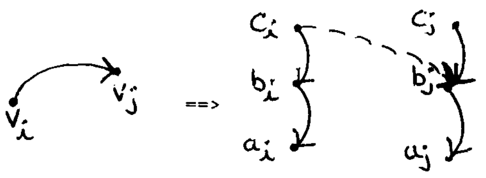
In the following discussion it is useful to distinguish between the decision and optimization problems for RSPP. The *RSPP Decision Problem* asks whether there is a schedule of length k or less for an arbitrary RSPP instance. The *RSPP Optimization Problem* is to find the optimal schedule for an arbitrary RSPP instance.

Observe that in order to produce an optimal schedule it is only necesary to compute an initial loop of an optimal schedule. If D = (V,E) is the Dag representation of an RSPP instance, let F be the initial loop of an optimal schedule, and let D' be the subgraph of D induced by removing the edges of F and all nonstream edges of D entering vertices of F. If P2,...,Pm is an optimal schedule for D', then F,P2,...,Pm is an optimal schedule for D. In the proof of Theorem 2 below, we say that D' is the graph resulting from scheduling F.

Theorem 2:. The RSPP Decision problem is NP-Complete

proof) We reduce a variation of the NP-Complete problem called CYCLEBREAK to the RSPP decision problem. CYCLEBREAK [9] is the following problem: given a directed graph G and a natural number k, is it possible to obtain an acylic graph from G by removing at most k vertices? The variation which we consider is: are there k vertices w1, w2, ... , wk of G such that removal of all edges leading into these k vertices results in an acylic graph?

Given a directed graph G = (V, E) with V = {v1, ..., vn}, let f(G) be the RSPP instance with vertex set {ai, bi, ci: i=1..n}, stream edges {[bi, ai], [ci, bi] : i=1..n}, and nonstream edges {[ci, bj] : [vi, vj] in E}.



a. edge [vi,vj] in E b. f([vi,vj]), where the broken edge is nonstream

We show that a graph G can be made acyclic by removing k vertices iff f(G) has an RSPP schedule of n+k or fewer loops. Observe that by the definition of a loop, the graph f(G) restricted to a loop must be a tree. The only possible loops are of the forms {[bi, ai]}, {[bi, ai], [ci, bi]} and {[ci, bi]}. Loops of the form {[bi, ai]} are called type 1 loops; loops of the other two forms are type 2 loops. Since the vertices bi, i = 1...n have no nonstream parameters, a type 1 loop may always be placed as the initial loop of a schedule. Let D = f(G). If D' is a subgraph of D, define C(D') to be the graph obtained by collapsing the vertices ai, bi, and ci for each i = 1...n. Thus C(f(G)) = G. If D' is the graph that results from scheduling a type 1 loop {[bi, ai]}, then C(D') is the graph formed from G by removing all edges entering vi. If D' results from scheduling a type 2 loop containing the edge [ci, bi], then C(D') is identical to G except that the vertex vi is removed. Note that such a type 2 loop can only be scheduled for a graph D when the vertex vi in C(D) has outdegree 0. Also note that any schedule for D must have exactly n type 2 loops.

Now suppose that a graph G can be made acyclic by removing k vertices. We show that f(G) has an RSPP schedule of k + n or fewer loops. Let D = f(G). There is a set W = {w1, w2, ..., wk} such that removal of all edges leading into w1,w2,...,wk results in an acyclic graph G'. Let D' be the graph resulting from scheduling k type 1 loops {[bi, ai] : vi in W}. C(D') = G'. Since G' is finite and acyclic there is a vertex vi with outdegree 0. Thus it is valid to schedule a type 2 loop corresponding to vi, i.e. either {[bi, ai], [ci, bi]} or if {[bi, ai]} has been already scheduled then {[ci, bi]}. In fact it is possible to schedule all n type 2 loops, because scheduling of a type 2 loop in D' corresponds to removing a vertex from C(D'). The resulting graph will again have a vertex with outdegree 0, which will allow another type 2 loop to be scheduled. This can be repeated until all n vertices have been removed. Thus, RSPP for the graph f(G) can be solved using k + n loops.

Conversely suppose the RSPP instance D can be solved using k + n loops. Since any schedule can be altered by moving all type 1 loops to the front, we assume our schedule has all of its k type 1 loops occurring first. Let W = {vi : {[bi, ai]} is a loop appearing in the schedule for D}. Let D' be the graph resulting from scheduling the type 1 loops. Let G' = C(D'). G' is the graph formed from G by removing all edges entering W. We claim that G' is acylic. Suppose it is not. Then it is not possible to find a sequence s1, ..., sn of vertices such that si has outdegree 0 in the graph G' with vertices s1, ..., si−1 removed. But that is precisely what is required for the RSPP instance D' to be solved with n type 2 loops■

Corollary:. The RSPP Optimization Problem is NP-Hard

Despite the preceding negative result we should understand that any feasible schedule that involves only a single instance of fusion is better than the naive unoptimized schedule. Thus, it is worthwhile to consider various efficient algorithms that give suboptimal schedules, or that give optimal schedules on significant subclasses of problem instances. In the next section we will present two such algorithms.

5. Stream Processing Algorithms

The first algorithm for solving RSPP sketched below is highly efficient, but gives suboptimal performance except for only a modest subclass of problem instances. It uses a greedy strategy in that each tree added to the schedule contains a maximum number of edges.

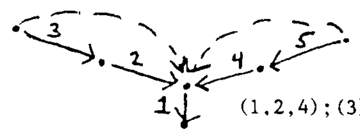
```
Greedy Algorithm

input: an instance of RSPP: D = (V,E), stream-edges

A. mark all roots in the forest
B. repeatedly find loops until all vertices are
   marked:
     i.  pick a stream edge [x,r] where r is
         marked and x is not, and assign it to
         tree; r will be the root of the tree
    ii.  until it is no longer possible,
         augment tree with a stream edge [u,v]
         for which,
           a.  v is in tree,
           b.  u is not in tree
           c.  all nonstream edges leading out
               of v must lead into marked vertices
   iii.  add tree to the end of the schedule and
         mark the vertices in tree
```

The greedy algorithm runs in O(#E) steps, and can be modified with little effort to solve the more general problem of SPP with the same time complexity.

The greedy algorithm for solving RSPP can be made even greedier. In step B (i.), the choice of root can be made based on a maximal tree over all choices of possible roots. This algorithm (called the 'greedier' algorithm) maintains a heap of tree sizes for each potential root and runs in O(#E log #V). It too can be adapted to SPP without changing the asymptotic time complexity. However, as is illustrated by Figure 4 below, the greedier algorithm can easily yield suboptimal schedules.



(1,2,4);(3);(5)   (1);(2,3,4,5)

a. RSPP dag   b. greedier schedule   c. optimal schedule

Figure 4. Broken lines are nonstream edges

Since the greedier algorithm can only yield optimal solutions on a modest class of problem instances, it is worthwhile considering other algorithms designed specifically for a wide class of instances for which optimality is achieved. If this class is contained in the class of RSPP instances, and if the algorithm will give reasonable suboptimal solutions on RSPP instances ouside of this class, then 2 approaches can be taken to solve SPP. The first approach to solve an SPP instance P would be to choose a best solution over all possible RSPP instances contained in P. The second approach is to solve an RSPP instance formed heuristically from an SPP instance.

The first approach obviously yields better results, and may be fairly efficient for small sized SPP instances, or for the case when there are only a few vertices having more than 1 stream edge leading out. Because so many of the iterative expressions given in Table 1 are either unary or have 2 stream parameters, we expect that this first approach will often be reasonable.

When the first approach is too costly, it is possible to utilize a modified version of the greedier algorithm to reduce any

SPP instance to a 'reasonable' choice of RSPP subinstance. This entails choosing a subset of the stream edges that forms a forest (i.e., a valid RSPP subinstance) with a minimal number of trees. All of the stream edges outside of this forest are considered nonstream edges.

The next algorithm will produce optimal results for a subclass of RSPP instances. The basis of our algorithm is the observation that a certain kind of cyclical structure in the dag is the basis for the NP completeness result. If this structure is not present an optimal solution can be found efficiently.

The following definitions facilitate the explanation of the algorithm.

Definition 1: A *forward edge* in an RSPP instance is a nonstream edge [x,y] for which there is a path from x to y consisting only of stream edges. All other nonstream edges are called *cross edges.*

Definition 2: The *head* vertices in an RSPP instance are the dag roots (vertices with 0 outdegree but that have at least one entering stream edge) and the heads of forward edges.

Definition 3: We define a partial ordering on the head vertices in the following way. If x and y are 2 head vertices, then $x < y$ if there exists a path from x to y consisting only of stream edges.

Definition 4: x is a minimal head if there is no head y for which $y < x$.

Definition 5: A minimal tree is a tree that has a minimal head as the root and a maximal number of stream edges.

Forward-edge Algorithm

input: an instance of RSPP, D = (E,V), stream-edges, heads;

Repeat the following steps until no stream edges are left in the dag.

A. If there exists a minimal tree (representing a loop) with no cross edges leading into any of its internal vertices, place it at the beginning of the schedule. Otherwise execute a variant of the greedy algorithm that chooses a tree t starting from the leaves (instead of from a root) of any minimal tree. Place t at the beginning of the schedule.

B. Remove the tree chosen in step A and all edges leading out of it from the dag. Also, delete from heads the head of the tree just removed and all heads in the remaining dag with no forward edges leading into them.

Theorem 3: When the greedy algorithm is never executed in step A, the Forward Edge Algorithm yields an optimal solution.

proof) The proof is by induction on the size of an optimal schedule. Let S = [T1,...,Tk] be an optimal schedule of k loops T1,...,Tk. If k=1, then the Dag representation of the RSPP instance must have only one head vertex. Hence, the forward edge algorithm will execute step A only once and form the optimal schedule from a single minimal tree. Assume that the forward edge algorithm will find optimal schedules of size less than or equal to k, and consider an RSPP instance with an optimal schedule of size k+1. Let T be the first minimal tree detected in step A of the forward edge algorithm. Let Tj be the last tree in the schedule S that contains common edges with T. Since any nonstream edge contained in T must lead from an internal vertex of T to the root of T, and since there are no stream edges in the dag representation of the RSPP instance leading into T, we know that Tj must be a subtree of T. We can then form a new optimal schedule S' from S by replacing Tk with T and removing from all other components of S all those edges belonging to T but not Tk ■

The forward-edge algorthm can be tailored to run in O(#E) steps. It can also be modified to construct parallel schedules. Parallel schedules are sequences of stages in which each stage contains a set of loops that can be executed in parallel, and all the loops at one stage must be executed before any of the loops at the next stage. A parallel schedule can be constructed by changing Step A so that the set of all minimal trees (with no cross edges leading in) are determined. This set would be a stage placed at the beginning of the schedule.

6. History

The idea of stream processing traces back to loop fusion, a program transformation described by Allen and Cocke [1] for merging two separate Fortran do loops into a single loop. Figure 5 illustrates the technique on code that computes the sum of the vector dot products a.b + c.d.

```
     do 1 i = 1,n
       sum1 = sum1 + a(i)*b(i))
  1  continue
     do 2 j = 1,n
       sum2 = sum2 + c(j)*d(j)
  2  continue
     sum = sum1 + sum2

          ==>

     do 2 i = 1,n
       sum1 = sum1 + a(i)*b(i))
       j = i
       sum2 = sum2 + c(j)*d(j)
   2 continue
     sum = sum1 + sum2
```

Figure 5. Fortran Loop Fusion

Recently, Fortran loop optimizations by fusion and more complicated loop rearangements have been developed in depth by Allen [2].

Burstall and Darlington discuss both horizontal and vertical loop fusion of recursive equations within the context of their Lisp transformational programming system [4]. Figure 6 illustrates the dot product example of Figure 5 in in this new context.

```
dot(x,y,n) = if n=0 then
                 0
             else
                 dot(x,y,n-1) + x(n)*y(n)
             end if
```

(a) Dot product definition

```
f(a,b,c,d,n) = dot(a,b,n) + dot(c,d,n)      ==>

f(a,b,c,d,n) =  if n=0 then
                  0
                else
                  f(a,b,c,d,n-1) + a(n)*b(n)
                                  + c(n)*d(n)
                end if
```

(b) Horizontal Loop Fusion

Figure 6.

Burge later reformulated the techniques of Burstall and Darlington, and he presented examples of loop fusion of recursively defined Lisp procedures [3]. Friedman and Wise investigated more complicated examples of streaming and proposed a new implementation technique called suspended evaluation [7]. They also applied their technique to file systems [8]. Some of these ideas are surveyed in [12].

Using a Lisp program development system based on Darlington's earlier transformational system, Feather [6] demonstrated the importance of fusion technique to the construction of moderate sized software systems. While Burstall and Darlington illustrated fusion for simple expressions, Feather generalized the technique effectively to interface more complicated procedures. Guibas and Wyatt illustrated fusion techniques to improve APL code, and they presented a simple algorithm that could apply fusion automatically within an optimizing compiler [10]. More recently, Reif and Scherlis stressed the importance of loop fusion in the derivation of complicated graph algorithms using a depth first search strategy [23].

While the preceding references discuss loop fusion transformations implemented manually or by highly specialized algorithms, Morgenstern explored automatic loop fusion transformations in depth, within the context of file processing systems [16]. Morgenstern used a dynamic programming algorithm to implement his transformations, but no analytic investigation of performance was made. However, his empirical studies showed that his algorithm was too slow to handle more than toy problems.

Housel [13] later considered a batch oriented data restructuring application, which was similar to Morgenstern's work, but involved a simpler model and a more naive scheduling algorithm. In particular, Housel only considered a limited number of primitives that included sorting and operations constrained to perform sequential processing on all of its arguments, the kind of model that would be useful for tape resident files.

Various algorithmic improvements in data restructuring have emerged recently from the Model programming language project [15, 22]. A nonalgorithmic approach in which loop definitions and scheduling are all specified within declarations has been proposed by Waters [28]. Overmars has applied stream processing to improve the space bounds on data structures used for solving batched search problems [17].

Based on the general transformational technique of finite differencing, Paige discovered a useful framework for implementing vertical and horizontal expression fusion efficiently [18, 21]; however, the scheduling algorithms presented there yielded solutions much worse than those presented in our paper. Weixalbaum later found a notation based on the integral calculus for describing Paige's ideas, and his unpublished manuscript refers to the fusion transformations expressed in his operator language as 'formal integration' [29]. Sharir later published similar results along with some examples [26]. Like earlier research in loop fusion, Weixalbaum and Sharir describe fusion informally and provide no implementation algorithms.

7. Conclusion

We have presented a new graph theoretic model for stream processing of iterative expressions. Within this model, optimality is investigated with respect to the number of fused loops. Although solving this problem is shown to be theoretically hard, several new efficient algorithms are presented. The greedier algorithm yields suboptimal solutions on all but a modest subclass of problem instances, while the forward-edge algorithm gives optimal solutions on a wide class of problem instances. Our initial case studies of stream processing applied to database and program optimization strongly suggest the importance of this transformation, and should encourage further investigations to explore optimal schedules with respect to other criteria; e.g., space utilization (regarding total space required after dead code elimination), access costs, criteria with respect to models involving merging and sort orders, parallel models, etc.

Appendix. Stream Processing Applied to a Useless Code Elimination Procedure

In [19], a derivation is presented of an efficient useless code elimination procedure (based on [14]) from the abstract formulation just below:

```
crit := prints;
(converge) $Repeat until crit no longer changes.
  crit ∪:= instof [usetodef [iuses [crit] ] ] ∪
                compound [crit];
end;
```

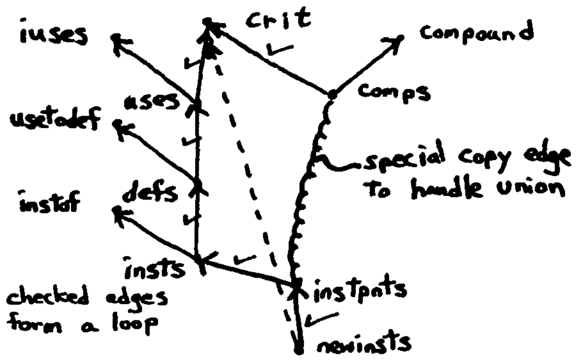
where the variables are defined as follows:

```
prints: set of print statements
iuses: maps each statement to the variable uses
       it contains
usetodef: maps each variable use to the variable
       definitions that can reach it
instof: maps each variable occurrence to the
       statement immediately containing it
compound:  maps each statement to the compound
       statement immediately containing it
```

The derivation requires collective initialization and differential maintenance of the following 14 invariants,

```
uses =  iuses [ crit ]
defs =  usetodef [ uses ]
insts = instof [ defs ]
comps = compound [ crit ]
instpnts = insts ∪ comps
newinsts =  { x ∈ instpnts | x ∉ crit }
compstream =  {[y,x]: x ∈ crit, y ∈ compound { x }}
ncompstream = {[x,# compstream{ x } ]: x ∈ domain
               compstream}
usepstream =  {[y,x]: x ∈ crit y ∈ iuses { x } }
nusepstream = {[x,# usepstream { x } ] x ∈ domain
               usepstream}
defpstream = {[y,x]: x ∈ uses, y ∈ usetodef { x }}
ndefpstream = {[x, # defpstream { x } ]: x ∈ domain
               defpstream}
instpstream =  {[y,x]: x ∈ defs, y ∈ instof { x } }
ninstpstream = {[x,# instpstream{ x }]: x ∈ domain
                instpstream}
```

whose dependency graph (with stream parameters drawn as single edges) appears below:



broken lines are nonstream edges

Application of the Greedier or the Forward-edge algorithms will collapse those 14 implicit loops into a single loop shown below:

```
newinsts := { } ;
instpnts := { } ;
insts := { } ;
ninstpstream := { } ;
instpstream := { } ;
defs := { } ;
ndefpstream := { } ;
defpstream := { } ;
uses := { } ;
nusepstream := { } ;
usepstream := { } ;
comps := { } ;
ncompstream := { } ;
compstream := { } ;
( forall x4 ∈ crit )
  ( forall x3 ∈ iuses { x4 } |
      nusepstream ( x3 ) = 0 )
    ( forall x16 ∈ usetodef { x3 } |
        ndefpstream ( x16 ) = 0 )
      ( forall x21 ∈ instof { x16 } |
          ninstpstream ( x21 ) = 0 )
        if x21 ∉ crit then
          newinsts with := x21 ;
        end if ;
        instpnts with := x21 ;
        insts with := x21 ;
      end forall ;
      ( forall x29 ∈ instof { x16 } )
        ninstpstream ( x29 ) + := 1 ;
        instpstream { x29 } with := x16 ;
      end forall ;
      defs with := x16 ;
    end forall ;
    ( forall x26 ∈ usetodef { x3 } )
      ndefpstream ( x26 ) + := 1 ;
      defpstream { x26 } with := x3 ;
    end forall ;
    uses with := x3 ;
  end forall ;
  ( forall  x38 ∈ iuses { x4 } )
    nusepstream ( x38 ) + := 1 ;
    usepstream { x38 } with := x4 ;
  end forall ;
  ( forall x8 ∈ compound { x4 } |
        ncompstream ( x8 ) = 0 )
    if x8 ∉ insts then
      if x8 ∉ crit then
        newinsts with := x8 ;
      end if ;
      instpnts with := x8 ;
    end if;
    comps with := x8 ;
  end forall ;
  ( forall x13 ∈ compound { x4 } )
    ncompstream ( x13 ) + := 1 ;
    compstream { x13 } with := x4   ;
  end forall ;
end forall ;
```

Acknowledgements

We are grateful to Martin Dowd and Ernst Mayr for important insights that contributed to our understanding of the complexity of stream processing. We are also grateful to Ron Fagin for pointing out that extensible hashing could provide a convenient physical structure for a set theoretic data model.

References

1. Allen. F. E., Cocke, John. A Catalogue of Optimizing Transformations. In *Design and Optimization of Compilers*, Randall Rustin, Ed.,Prentice Hall, 1971, pp. 1-30.

2. Allen, J. R. *Dependence Analysis for Subscripted Variables and its Application to Program Transformations.* Ph.D. Th., Rice University, 1983.

3. Burge, William. An Optimizing Technique for High Level Programming Languages. Tech. Rept. Computer Science RC5834 #25271, IBM Research Center/Yorktown Heights, 1976.

4. Burstall, R. M., and Darlington, J. . "A Transformation System for Developing Recursive Programs ." *JACM 24*, 1 (Jan 1977).

5. Fagin, R., Nievergelt, J., Pippenger, N. and Strong, J. "Extendible Hashing - A Fast Access Method for Dynamic Files." *ACM TODS 4*, 3 (Sep 1979), 315-344.

6. Feather, Martin S. *A System for Developing Programs by Transformation.* Ph.D. Th., U. of Edinburgh, Dept. of Artificial Intelligence, 1979.

7. Friedman, D., Wise, D. CONS should not evaluate its arguments. In *Automata, Languages, and Programming*. S. Michaelson and R. Milner, Ed.,Edinburgh Univ. Press, Edinburgh, 1976, pp. 257-284.

8. Friedman, D. and Wise, D. Aspects of Applicative Programming for File Systems. Proc. ACM Conf. on Language Design for Reliable Software, March, 1977, pp. 41-55. SIGPLAN Notices, Vol 12, Num. 3

9. Garey, Michael R., Johnson, David S.. *Computers and Intractability.* Freeman. 1979.

10. Guibas, L. and Wyatt, K. Compilation and delayed evaluation in APL. Proceedings 5th ACM Symposium on Principles of Programming Languages, Jan. 1978, pp. 1-8.

11. Halmos, P.R.. *Naive Set Theory.* Nan Nostrand. 1960.

12. Henderson, P.. *Functional Programming.* Prentice-Hall, 1980.

13. Housel, B. "Pipelining: A Technique for Implementing Data Restructurers." *TODS 4,* 4 (Dec 1979), 470-492.

14. Kennedy, K. A Survey of Compiler Optimization Techniques. In *Program Flow Analysis*, Muchnick, S. and Jones, N., Eds., Prentice Hall. 1981, pp. 5-54.

15. Lu, Kang-Sen. *Program Optimization Based on a Non-procedural Specification.* Ph.D. Th., U. Penn., Dec 1981.

16. Morgenstern. Matthew. *Automated Design and Optimization of Management Information System Software.* Ph.D. Th., MIT. Laboratory for Computer Science, Sep 1976.

17. Overmars, M.. *The Design of Dynamic Data Structures.* Springer-Verlag, Berlin. 1983. Lecture Notes in Computer Science #156

18. Paige, R.. *Formal Differentiation.* UMI Research Press, Ann Arbor, Mich, 1981. Revision of Ph.D. thesis, NYU, June 1979

19. Paige, R. Transformational Programming — Applications to Algorithms and Systems. Proceedings Tenth ACM Symposium on Principles of Programming Languages, Jan. 1983, pp. 73-87.

20. Paige, Robert. Applications of Finite Differencing to Database Integrity Control and Query/Transaction Optimization. In *Advances In Database Theory, Volume 2*, Gallaire, H., Minker, J., and Nicolas, J.-M., Ed.,Plenum Press, New York, 1984, pp. 171-210.

21. Paige, R., and Koenig, S. "Finite Differencing of Computable Expressions." *ACM TOPLAS 4*, 3 (July 1982), 402-454.

22. Prywes, N., Pnueli, A., and Shastry, S. "Use of a Non-Procedural Specification Language and Associated Program Generator in Software Development." *ACM TOPLAS 1*, 2 (Oct 1979), 196-217.

23. Reif, J. and Scherlis, W. Deriving Efficient Graph Algorithms. Carnegie-Mellon U., 1982. Technical Report

24. Schwartz, J. T.. *On Programming: An Interim Report on the SETL Project, Installments I and II.* CIMS, New York Univ., New York. 1974.

25. Selinger, P.G.,et.al. Access path selection in a relational database management system. SIGMOD, Boston, May, 1979, pp. 23-34.

26. Sharir, M. "Formal Integration - A Program Transformation Technique." *Compout. Lang. 6*, 1 (1982), 35-46.

27. Ullman, J.D.. *Principles of Database Systems.* Computer Science Press, 1980.

28. Waters, R. Expressional Loops. Proceedings 11th ACM Symposium on Principles of Programming Languages, Jan. 1984.

29. Weixalbaum, E. Formal Integration. unpublished manuscript

---

[1]This work describes research done in part at Kestrel Institute and is supported in part by the Defense Advanced Research Projects Agency Contract N00014-81-C-0582, monitored by the Office of Naval Research. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied of KESTREL, DARPA or the US Government.

[2]This material is based in part upon work supported by the Defense Advanced Research Projects Agency, under the KBMS project, Contract Number N00039-82-C-0250, and by the National Science Foundation under Grant No. MCS-8212936. Part of this work was done while the author was visiting Stanford University.