Array Dataflow Analysis for Explicitly Parallel Programs

Jean-François Collard[1] and Martin Griebl[2]

[1] Laboratoire PRISM, Université de Versailles St-Quentin, 45 Avenue des Etats-Unis, 78035 Versailles, FRANCE, Jean-Francois.Collard@prism.uvsq.fr
[2] Martin Griebl, FMI, Universität Passau, Innstraße 33, 94032 Passau, GERMANY, Martin.Griebl@fmi.uni-passau.de

Abstract. This paper describes a dataflow analysis of array data structures for data-parallel and/or control- (or task-) parallel imperative languages. This analysis departs from previous work because it 1) simultaneously handles both parallel programming paradigms, and 2) does not rely on the usual iterative solving process of a set of data flow equations but extends array dataflow analysis based on integer linear programming, thus improving the precision of results.

1 Introduction

After decades of parallel processing, few programming languages can claim to be used on a wide range of parallel architectures. Probably, one of the reasons lies in the difficulty of *efficiently* compiling a general language on an ever-widening spectrum of machines. Among useful analyses, *dataflow analysis* derives information about the definition and the subsequent use(s) of data values in a program. Its applications include dead-code elimination, strength reduction, array expansion [1] or equivalently conversion into single-assignment form.

Unfortunately, very few data-flow analyses have been proposed for parallel languages [7]. This paper presents an analysis for data-parallel languages, e.g. HPF [2], control-parallel (also called task-parallel) languages, such as PCF [3], or a mixture of both [4, 5, 6].

2 Motivating examples

Several languages [4, 6] have indexed parallel constructs whose semantics correspond to what we call `doall` in this paper: a statement instance is spawned for each possible value of the index variable (e.g., from 0 to $2n$ by step 1 in Program `ExD`, hence $2n+1$ instances or tasks are spawned). Each instance has its own copy of shared data structures, and all reads and writes are applied to this copy. Shared data structures are updated only when the instances of all statements in the loop body have completed. Thus, within one iteration of a `doall` loop, all statements work on the same copy.

For example in Program `ExD`, if $n = 2$, the read of `a(1)` in Instance number 1 of Statement R may return the value produced by the instance number 1 of

```
     program ExD                        program ExF
P1   doall( i = 0 : 2*n : 1 )           forall( i = 0 : 2*n : 1 )
       where ( ... )                      where ( ... )
S1 :       a(i) = ...            S1 :         a(i) = ...
S2 :   else a(2*n-i) = ...       S2 :     else a(2*n-i) = ...
       endwhere                           endwhere
R :    ... = a(i)                R :      ... = a(i)
```

Fig. 1. Two examples

S_1, but not from Instance 3 of S_2. Consequently, the information we would like to automatically derive is that the source of `a(i)` in Instance i of R is either S_1 in Instance i or undefined (written as $\perp$), if $i \neq n$ or S_1 or S_2 in Instance i, if $i = n$ (because in this case, either S_1 or S_2 writes into `a(n)`).

Notice that we consider that all instances of both arms of the parallel conditional structure `where` are executed in parallel.

Similarly, a `forall` construct spawns as many instances as there are possible values for the index variable (e.g., $2n+1$ instances are spawned in Program `ExF`). The semantics of `forall` we consider is reminiscent from the HPF semantics[3] [2]: in a multi-statement `forall`, the array assignment semantics are applied to each statement in turn. Each instance of a statement has its own copy of shared data structures, but these shared data structures are updated before the instances of following statements begin.

Consider the read of `a(1)` in Program `ExF`: it may be the case that both S_1 and S_2 simultaneously wrote into this cell. Such an *over-determined source* will be denoted by top ($\top$). To sum up, our analysis derives that the source of `a(i)` in Instance i of R in Program `ExF` is either S_1 in Instance i or S_2 in Instance $2n - i$ or $\perp$ or $\top$, if $i \neq n$, or S_1 or S_2 in Instance i, if $i = n$ (without $\perp$ nor $\top$).

3 Definitions

The input language includes the following sequential control structures: `do`, `while`, `if`, the following parallel structures: `forall`, `doall`, `where`, and parallel sections `parsection`.

The dimension of a vector $\mathbf{x}$ is denoted by $|\mathbf{x}|$. The k-th entry of vector $\mathbf{x}$, $k \geq 0$, is denoted by $\mathbf{x}[k]$. The sub-vector built from components k to l is written as: $\mathbf{x}[k..l]$. If $k > l$, then this vector is by convention the vector of dimension 0. Furthermore, $\underline{\ll}$ ($\ll$) denotes the (strict) lexicographical order on such vectors.

The *index vector* of a statement S is the vector built from the counters of surrounding `do`, `forall`, `doall` and `while` constructs. An *operation* is an instance of some Statement S, and will be denoted by $\langle S, \mathbf{x} \rangle$, where $\mathbf{x}$ is some value of the index vector of S.

[3] Note that in HPF the only parallel constructs inside `forall`s are `forall`s and `where`s (Rules H404 and H406 of [2]).

The *depth* of a statement or construct is the number of surrounding `do`, `forall`, `doall` or `while` constructs. So, the depth of S equals to $|\mathbf{x}|$. If $\mathbf{x}[p]$, $p \geq 0$, is a counter of a `do`, `forall` or `doall` construct, then lower and upper affine bounds are known: $l_p(\mathbf{x}[0..p-1]) \leq \mathbf{x}[p] \leq u_p(\mathbf{x}[0..p-1])$ where l_p and u_p are syntactically given by the loop bounds. In the case of `while`-loops, we have by convention $1 \leq \mathbf{x}[p]$. The index domain of a statement S is denoted by $\mathbf{D}(S)$ and is given by the conjunction of all inequalities on surrounding loop bounds. We define $C_p(S)$ as the iterative (`do`, `while`, `forall` or `doall`) construct surrounding S at depth p. (When clear from the context, S will be omitted.) For example, let us consider Statement S_1 in Program `ExD`: $C_0 = P_1$ (the `doall`).

We define $\mathcal{P}(S, R)$ to be the `par` construct surrounding both S and R such that S and R appear in distinct sections of the `par` construct. (Notice that there is at most one such construct.) Moreover, let M_{SR} be the depth of $\mathcal{P}(S, R)$. (If $\mathcal{P}(S, R)$ does not exist, then M_{SR} is the number of `do`, `forall`, `doall` or `while` constructs surrounding both S and R.) In Program `ExF`, $\mathcal{P}(S_1, R) = \emptyset$ and $M_{S_1R} = 1$. Predicate Δ_{SR} is true if S and R appear in opposite arms of an `if..then..else` or `where..elsewhere` construct, or in distinct sections of a `par` construct. Predicate T_{SR} is true if S textually precedes R and Δ_{SR} is false. We denote by $\mathcal{W}(u)$ (resp. $\mathcal{R}(u)$) the memory cell written into (resp. read) by operation u, so for instance $\mathcal{W}(\langle S_2, i\rangle) =$`a(` $2n - i$ `)` and $\mathcal{R}(\langle R, i\rangle) =$`a(` i `)` in Programs `ExD` and `ExF`.

4 The semantics: execution order

The purpose of this section is not to give a complete semantical description of a parallel language. As far as dataflows are concerned, we are mainly interested in the order in which computations, and their corresponding writes and reads to memory, occur. The fact that $\langle S, \mathbf{x}\rangle$ is executed before $\langle R, \mathbf{y}\rangle$ in the parallel program will be denoted by $\langle S, \mathbf{x}\rangle \prec \langle R, \mathbf{y}\rangle$. In the case of sequential programs, $\prec$ is a total order which can be expressed as the lexicographical order on index vectors. In turn, the lexicographical order can be expressed as a disjunction of linear inequalities. Expressing the execution order $\prec$ of parallel programs is more intricate. For instance, Section 2 showed that two semantics can be chosen for a data-parallel construct: the "synchronous" semantics of the construct we call `forall`, where the memory is updated between the execution of two successive statements inside a `forall`; and the "asynchronous" semantics of the construct we call `doall`, where none of the spawned tasks sees the effects produced by other tasks. However, thanks to the semantics of `forall` and `doall` constructs, $\prec$ can still be expressed in a linear way:

$$\begin{aligned}\langle S, \mathbf{x}\rangle \prec \langle R, \mathbf{y}\rangle \Leftrightarrow & \left(\bigvee_{\substack{p=0..M_{SR}-1,\\ C_p=\text{do} \vee C_p=\text{while}}} Pred(p, \mathbf{x}, \mathbf{y}) \right) \\ & \vee \left(\left(\bigwedge_{p=0..M_{SR}-1} Equ(p, C_p, \mathbf{x}, \mathbf{y}) \right) \wedge T_{SR} \right) \qquad (1)\end{aligned}$$

where:

$$Pred(p, \mathbf{x}, \mathbf{y}) \equiv \left(\bigwedge_{i=0..p-1} Equ(i, C_i, \mathbf{x}, \mathbf{y}) \right) \wedge \mathbf{x}[p] < \mathbf{y}[p] \tag{2}$$

$$Equ(p, \texttt{do}, \mathbf{x}, \mathbf{y}) \equiv Equ(p, \texttt{while}, \mathbf{x}, \mathbf{y}) \equiv \mathbf{x}[p] = \mathbf{y}[p] \tag{3}$$

$$Equ(p, \texttt{forall}, \mathbf{x}, \mathbf{y}) \equiv \mathbf{true} \tag{4}$$

$$Equ(p, \texttt{doall}, \mathbf{x}, \mathbf{y}) \equiv \mathbf{x}[p] = \mathbf{y}[p] \tag{5}$$

Obviously, (1) is a partial order on operations. (In particular, it has no cycle.) Intuitively, Predicate *Pred* in (1) formalizes the sequential order of a given `do` or `while` loop at depth p. Such a loop enforces an order up to the first `par` construct encountered at depth M_{SR} while traversing the nest of control structures, from the outermost level to the innermost. The order of sequential loops is given by the strict inequality in (2), under the condition that the two operations at hand are not ordered up to level $p-1$; hence the conjunction on the p outer predicates *Equ*. Notice that the instances of two successive statements inside a `forall` at depth p are always ordered at depth p due to (4), but are ordered inside a `doall` only if they belong to the same task (i.e., the values of the `doall` index are equal, cf (5)).

Note that $\forall P$, $\bigvee_{i \in \emptyset} P(i) = \mathbf{false}$ and $\bigwedge_{i \in \emptyset} P(i) = \mathbf{true}$. Note also that the lexicographical order on nests of `do` loops [1] and in (sequential) dynamic control programs [8] comes as a special case of (1).

Example As far as programs `ExD` and `ExF` are concerned, the order between S_1 (or S_2) and R is given by:

ExF: $\mathcal{P}(S_1, R) = \emptyset$, so $M_{SR} = 1$. $C_0 = \texttt{forall}$ and $T_{S_1 R} = \mathbf{true}$. Thus:

$$\begin{aligned} \langle S_1, i' \rangle \prec \langle R, i \rangle &\Leftrightarrow \left(\bigvee_{p \in \emptyset} Pred(p, i', i) \right) \vee (Equ(0, C_0, i', i) \wedge T_{S_1 R}) \\ &\Leftrightarrow \mathbf{false} \vee (\mathbf{true} \wedge \mathbf{true}) \;\Leftrightarrow\; \mathbf{true} \end{aligned} \tag{6}$$

This is a formal restatement of a semantical property of `forall`s, cf Section 2. Notice that $\langle S_2, i' \rangle \prec \langle R, i \rangle$ is also always true.

ExD: $\mathcal{P}(S_1, R) = \emptyset$, so $M_{SR} = 1$. $C_0 = \texttt{doall}$ and $T_{S_1 R} = \mathbf{true}$. Thus:

$$\begin{aligned} \langle S_1, i' \rangle \prec \langle R, i \rangle &\Leftrightarrow \left(\bigvee_{p \in \emptyset} Pred(p, i', i) \right) \vee (Equ(0, C_0, i', i) \wedge T_{S_1 R}) \\ &\Leftrightarrow \mathbf{false} \vee (i' = i \wedge \mathbf{true}) \;\Leftrightarrow\; i' = i \end{aligned} \tag{7}$$

5 Dataflow analysis

Our dataflow analysis first builds the set of possible sources of the flow of a given data, and then selects the latest element, i.e., the maximal element according to the original sequential order. Selecting the maximal element is then done using Integer Linear Programming techniques.

5.1 Method overview

Let us consider two statements S and R. Suppose that S writes into an array `a` and that R reads that same array:

$$\begin{array}{ll} S: & \texttt{a}(f(\mathbf{x}))\ \texttt{= ...} \\ R: & \texttt{... = a}(g(\mathbf{y})) \end{array}$$

The aim of array dataflow analysis is to find the source of the value $\texttt{a}(g(\mathbf{y}))$ read in R for a given $\mathbf{y}$. This source is denoted by $\sigma(\langle R, \mathbf{y}\rangle)$[4]. To be a source candidate, an operation $\langle S, \mathbf{x}\rangle$ has to satisfy the following constraints:

Existence predicate: $\langle S, \mathbf{x}\rangle$ is a valid operation: $e(\langle S, \mathbf{x}\rangle)$ evaluates to **true**. (See Section 5.2.)

Conflicting accesses: $\langle S, \mathbf{x}\rangle$ and $\langle R, \mathbf{y}\rangle$ access the same array cell: $f(\mathbf{x}) = g(\mathbf{y})$. f and g are possibly multi-dimensional affine functions.

Sequencing condition: $\langle S, \mathbf{x}\rangle$ is executed before $\langle R, \mathbf{y}\rangle$ in the parallel program: $\langle S, \mathbf{x}\rangle \prec \langle R, \mathbf{y}\rangle$. Notice that this leads to a formal definition of $\bot$: $\bot$ is the name of the operation that is executed once before all other operations of the program, i.e., $\forall S, \mathbf{x} : \bot \prec \langle S, \mathbf{x}\rangle$.

Environment: The set of source candidates is computed under the hypothesis that $\langle R, \mathbf{y}\rangle$ is a valid operation, i.e., $\mathbf{y} \in \mathbf{D}(R)$.

The set of candidate sources is thus:

$$\mathbf{Q}_{SR}(\mathbf{y}) = \{\langle S, \mathbf{x}\rangle \mid \begin{array}{ll} e(\langle S, \mathbf{x}\rangle), & (\textit{Existence}) \\ f(\mathbf{x}) = g(\mathbf{y}), & (\textit{Conflicting accesses}) \\ \langle S, \mathbf{x}\rangle \prec \langle R, \mathbf{y}\rangle\} & (\textit{Order}) \end{array} \tag{8}$$

The direct dependence from S to R is then $K_{SR}(\mathbf{y}) = \max_{\prec} \mathbf{Q}_{SR}(\mathbf{y})$. Obviously, S may not be the only statement writing into `a`. Let $\mathcal{S}(R)$ be the set of statements writing into the array read by R. Then, the set of candidate sources is $\mathbf{Q}_R(\mathbf{y}) = \bigcup_{S \in \mathcal{S}(R)} \mathbf{Q}_{SR}(\mathbf{y})$. The source $K_R(\mathbf{y})$ of the flow of datum $\texttt{a(}\ g(\mathbf{y})\ \texttt{)}$ is the maximal element in $\mathbf{Q}_R(\mathbf{y})$ according to $\prec$:

$$K_R(\mathbf{y}) = \max_{\prec} \mathbf{Q}_R(\mathbf{y}). \tag{9}$$

Clearly, three problems occur:

- We have to express Predicate e. This issue is addressed in Section 5.2.
- Maxima according to $\prec$ among sets of operations have to be computed. Section 5.3 explains how to compute $\max_{\prec}$ using the Omega tool.
- $K_R(\mathbf{y})$ may not be uniquely defined, since $\prec$ is not a total order. Intuitively, a non-unique maximum means that two operations wrote in an undefined order into the same memory cell. The over-determined source is denoted by $\top$ (cf Section 5.4).

[4] For the sake of clarity, we assume that an operation executes at most one read.

5.2 Existence of operations

We say that an operation *exists* if this operation executes. In the case of static-control sequential programs, the only loops are `do` loops, and the existence predicate boils down to $e(\langle S, \mathbf{x}\rangle) \Leftrightarrow \mathbf{x} \in \mathbf{D}(S)$.

When arbitrary `while`, `if` and `where` constructs appear, the control flow is dynamic, and existence of operations cannot in general be predicted. The problem then boils down to finding a suitable coding of Existence predicate e. The reader is referred to [8, 10] for details. In the sequel, we will use the Omega package[9] and phrase this paper in the corresponding framework[5]. For instance, the case of `if` and `where` constructs is handled as follows: If $\mathbf{x}$ is the index vector of a conditional statement, the execution of the `then` branch is coded by postulating that some uninterpreted function f evaluates to, say, a nonnegative value: $f(\mathbf{x}) \geq 0$. The execution of the `else` or `elsewhere` branch is then denoted by $f(\mathbf{x}') < 0$. Since both branches cannot execute in the same instance of the construct, $f(\mathbf{x}) \geq 0 \wedge f(\mathbf{x}') < 0 \Rightarrow \mathbf{x} \neq \mathbf{x}'$.

Example Let us find possible bottoms in the source of the read $\langle R, k\rangle$ in Program `ExF`. (k is here a parameter.) The set of possible writes from S_1 is:

```
# Omega Calculator [v1.00, Mar 96]:
# symbolic n, f(1), k;
# W1 := { [iw] : 0 <= iw <= 2n && f(Set) >= 0 && iw = k } ;
```

where `f(Set)` means that f is applied to the bounded variable(s) that define the set (here, `iw`). Then, for S_2:

```
# W2 := { [iw] : 0 <= iw <= 2n && f(Set) < 0  && 2n - iw = k } ;
```

We are interested in finding all reads that do not have a corresponding write from either S_1 or S_2. We thus take the union of the two sets of writes:

```
# W1 union W2 ;
{[iw]: k = iw && 0 <= iw <= 2n && 0 <= f(iw)} union
 {[iw]: k+iw = 2n && 0 <= iw <= 2n && f(iw) <= -1}
```

We then subtract the obtained set to the set `R` of all reads:

```
# R := { [ir] : 0 <= ir <= 2n && ir = k } ;
# Bottoms := R intersection complement (W1 union W2);
# Bottoms ;
{[In_1]: k = In_1 && n < In_1 <= 2n && f(In_1) <= -1} union
 {[In_1]: k = In_1 && 0 <= In_1 < n && f(In_1) <= -1}
```

Since nothing is known about f, we have to take a conservative approach and assume that any predicate involving f is true. The set of possibly undefined reads (bottoms) is thus given by $\{k : 0 \leq k < n\} \cup \{k : n < k \leq 2n\}$. Notice that, as expected, the case $k = n$ does not occur, i.e., $\langle S, n\rangle$ is always defined by S_1 and/or S_2.

[5] When called with an input file, the Omega calculator (v1.00 dated March 1996) copies the input equations on the standard output, prefixed by a hash # sign.

5.3 Computing maxima

Solving (9), i.e., computing the maximum (maxima), is equivalent to finding the element(s) $K_{SR}(\mathbf{y})$ such that $\neg\exists \mathbf{x} \in \mathbf{Q}_{SR}(\mathbf{y}), \mathbf{x} \neq K_{SR}(\mathbf{y}) : K_{SR}(\mathbf{y}) \prec \mathbf{x}$. This expression may have several solutions, since parallel programs have partial execution orders. (See Section 5.4.) Similarly, $K_R(\mathbf{y})$ in (9) can be computed by:

$$\neg\exists\langle S,\mathbf{x}\rangle, e(\langle S,\mathbf{x}\rangle), \mathcal{W}(\langle S,\mathbf{x}\rangle) = \mathcal{R}(\langle R,\mathbf{y}\rangle), \langle S,\mathbf{x}\rangle \neq K_R(\mathbf{y}), K_R(\mathbf{y}) \prec \langle S,\mathbf{x}\rangle \tag{10}$$

5.4 Detecting over-determined sources

Detection of over-determined sources is done by checking that no two distinct operations satisfy their respective existence predicates, write into the same memory cell, and are not related by $\prec$. Formally: $Error(u,v) \equiv e(u) \wedge e(v) \ \wedge \ u \neq v \ \wedge \ \mathcal{W}(u) = \mathcal{W}(v) \ \wedge \ \neg(u \prec v) \wedge \neg(v \prec u)$, where u and v are operations. Notice that, in the general case, checking that no dependence is carried by a `doall`- or a `forall`-loop is not sufficient.

$\boxed{\text{Example}}$ Let us find the over-determined cases in the source of operation $\langle R,k\rangle$ in Program `ExF`.

```
# Symbolic f(1), n , k ;
# Error :=
# { [iw1] -> [iw2] : 0 <= iw1, iw2 <= 2n && f(In) >= 0 && f(Out) < 0
#     && iw1 = 2n - iw2 = k }
# union
# { [iw1] -> [iw1'] : 0 <= iw1, iw1' <= 2n && f(In) >= 0 && f(Out) >= 0
#     && iw1 != iw1' && iw1 = iw1' = k }
# union
# { [iw2] -> [iw2'] : 0 <= iw2, iw2' <= 2n && f(In) < 0 && f(Out) < 0
#     && iw2 != iw2' && 2n - iw2 = 2n - iw2' = k } ;
```

`In` and `Out` are collective names of the input (the tuple preceding the `->` and output (the tuple following the `->`) variables of a relation, respectively.

```
# Error ;
{[In_1] -> [iw2] : In_1+iw2 = 2n && k = In_1 && n < In_1 <= 2n
                   && f(iw2) <= -1 && 0 <= f(In_1)} union
 {[In_1] -> [iw2] : In_1+iw2 = 2n && k = In_1 && 0 <= In_1 < n
                    && f(iw2) <= -1 && 0 <= f(In_1)}
```

As in the case of bottoms, f is unknown. The set `Error` of possibly over-determined reads ($\top$) is thus $\{k : 0 \leq k < n\} \cup \{k : n < k \leq 2n\}$. Notice that this is the result expected from Section 2.

6 Conclusion

This paper presented a dataflow analysis that can be applied to data- and/or task- parallel programs, with dynamic flows of control. Li and Wolfe proposed [11] a simple and precise framework to express the interaction of arbitrarily nested parallel control structures, together with an appropriate data dependences. They did not, however, extended this work to data-flow analyses. The main results of this paper are: a general affine expression for the execution order of programs written in a structured imperative parallel language (this expression subsumes the lexicographical execution order in sequential programs), and a general affine expression for the over-determined cases ($Error$).

Acknowledgments We would like to thank L. Bougé, T. Brandes, W. Pugh, Y. Robert, G. Utard and M. Wolfe for their helpful comments, and D. Wannacott for his technical help with Omega. Authors are supported by the CNRS and the DFG project RecuR, respectively, and in addition by a German-French programme Procope.

References

1. P. Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.
2. C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Stelle Jr, and M.E. Zosel. *The High Performance Fortran Handbook.* The MIT Press, 1994.
3. Parallel Computing Forum. PCF fortran extensions. *Fortran Forum*, 10(3), 1991.
4. M. Gerndt and R. Berrendorf. SVM-Fortran, Reference Manual, Version 1.4. KFA-ZAM-IB-9510, Research Center Jülich. May 1995.
5. I. Foster et al. A Compilation System That Integrates High Performance Fortran and Fortran M. In *Proc. of the Scalable High-Performance Computing Conf (SHPCC'94).* pages 293–300, Knoxville, TN, May 1994.
6. B. Chapman et al. Extending Vienna Fortran with Task Parallelism. In *Proc. of the 1994 Intl Conf on Parallel and Distributed Systems.* pages 258–263, Hsinchu, Taiwan, December 1994.
7. J. Ferrante, D. Grunwald, and H. Srinivasan. Computing communication sets for control parallel programs. In K. Pingali et al., editor, *Proc. of* 7^{th} *Int. W. on Lang. and Compilers for Parallel Comp.*, volume 892 of *LNCS*, pages 316–330, Ithaca, NY, August 1994.
8. J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *Proc. of 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 92–102, Santa Barbara, CA, July 1995.
9. W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, August 1992.
10. W. Pugh and D. Wonnacott. Nonlinear Array Dependence Analysis. In *Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, Troy, NY, May 1992.
11. J. Li and M. Wolfe. Defining, Analyzing, and Transforming Program Constructs. *IEEE Parallel and Distributed Technology*, 32–39, Spring 1994.