SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 20(2), 133–155 (FEBRUARY 1990)

Structured Dataflow Analysis for Arrays and its Use in an Optimizing Compiler

THOMAS GROSS AND PETER STEENKISTE
School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania 15213, U.S.A.

SUMMARY

We extend the well-known interval analysis method so that it can be used to gather global flow information for individual array elements. Data dependences between all array accesses in different basic blocks, different iterations of the same loop, and across different loops are computed and represented as labelled arcs in a program flow graph. This approach results in a uniform treatment of scalars and arrays in the compiler and builds a systematic basis from which the compiler can perform numerous global optimizations.

This global dataflow analysis is performed as a separate phase in the compiler. This phase only gathers the global relationships between different accesses to a variable, yet the use of this information is left to the code generator. This organization substantially simplifies the engineering of an optimizing compiler and separates the back end of the compiler (e.g. code generator and register allocator) from the flow analysis part.

The global dataflow analysis algorithm described in this paper has been implemented and used in an optimizing compiler for a processor with deep pipelines. This paper describes the algorithm and its compact implementation and evaluates it, both with respect to the accuracy of the information and to the compile-time cost of obtaining and using it.

KEY WORDS Compilation Global dataflow analysis Interval analysis Optimization Pipelining

INTRODUCTION

Many compilers for computers with multiple functional units, pipelines, or vector units face the task of extracting parallelism from a sequential description of the program. Depending on the architecture, the compiler has to change the order of execution between statements, move operations from one basic block to another, map the body of a loop into a vector instruction, or execute different loop iterations in parallel. To preserve the semantics of sequential execution, the compiler needs information about the control and data dependences in the program.

Scalar dataflow analysis

Several efficient algorithms for scalar dataflow analysis are known; they include iterative dataflow analysis[1] and interval analysis.[2–4] Compilers typically calculate the dependence information for the whole program at once and attach this information to the program representation, where it is used by the optimization phase. When depen-

Received 16 February 1989
Revised 21 June 1989

dences change because of optimizations, the dependence information is either updated incrementally or recalculated.

This organization separates a compiler into two parts, one that builds and analyses the flow graph, and one that generates and optimizes the code; this organization is almost universally accepted for all scalar compilers. However, it has the disadvantage that dataflow analysis algorithms usually treat arrays as one unit, that is, all accesses to the same array interfere with each other. This is unacceptable in scientific programs where practically all computation is done on arrays. If all references to the same array have to be executed sequentially because of overly conservative dependence information, the compiler can never extract any reasonable parallelism from the program, which is essential to use any high-performance computer effectively.

Dataflow analysis for arrays

Because of the importance of arrays in scientific code, special methods have been developed to gather dependence information regarding arrays. Most research on dataflow analysis for arrays has concentrated on analysing a single for-loop or a group of nested for-loops. The objective is to find implicit parallelism so that the program can be transformed for execution on shared-memory multiprocessors or vector processors.[5–7] Dependences between array references in FORTRAN programs have been studied extensively[8, 9] and this prior research has led to the development of tools that can transform sequential programs for multi-processors and vector processors. The Paraf-rase system can transform loops to execute in parallel on multiprocessors[10] as well as identify loops for efficient execution on vector processes.[11] The Parallel Fortran Converter (PFC) uses dependence information to determine if it is possible to interchange the nesting of loops.[5, 12] The framework for dependences that was developed for these tools forms also the basis for several commercial systems, e.g. the vectorizer for the IBM 3090[13] and the Ardent compiler system[14] are based on the earlier PFC tool.

The basic dependence analysis operation used in these compilers is the disambiguation of two references to the same array. This operation is either used to disambiguate array references whenever necessary,[15] or systematically to check for dependences by comparing every pair of statements inside a nested loop.[5] This approach to dependence analysis is different from the approach used for scalars: in scalar analysis, dataflow information is propagated through all basic blocks of the program, based on a set of flow equations that capture the behaviour of the statements.

A COMPILER FOR SCIENTIFIC CODE

In this paper we present an approach to dependence analysis that is based on collecting detailed information for scalars and arrays once during compilation in a systematic way. Then this information is re-used throughout the compiler. In the front-end of the compiler, where the structure in the program (e.g. the structure of loops) is readily available, we perform dataflow analysis. This information is stored together with the program representation, where it is available for use in the back-end of the compiler, i.e. the code generator.

Such an approach is desirable for highly optimizing compilers since all phases of the compilation can benefit from global flow information. To be a realistic approach, collecting global dataflow information for both scalars and arrays must be cheap and systematic. We developed a global dataflow analysis algorithm that calculates dependence information for both scalars and arrays in a uniform and efficient way; the information for arrays is accurate up to an individual array element. The algorithm is based on the interval analysis method for gathering scalar dataflow information. It is interesting that the proposed structure for the compiler is the same as the structure of traditional scalar compilers: build an internal representation for the program, do dataflow analysis, optimize, and finally generate code.

The outline of the paper is as follows. We first present the algorithm for gathering data dependence information. We then describe how the algorithm is used in an optimizing compiler, how the dependence information is inserted in the program representation, and how the optimizer and the scheduler use the dependence information. We conclude with an evaluation of our method.

To illustrate the compilation process, we will use the program segment in Figure 1 as an example of a program with multiple array accesses. (Information regarding scalar variables is not shown in the examples to keep the description concise. This paper, as well as our current implementation, assumes that all arrays are zero based. This limitation is not inherent in the approach but turns out to be quite convenient.)

If arrays are treated as one unit, all statements will have to be executed sequentially. In reality, a lot of parallelism is present. For example, if information is available on the individual array elements, the execution of loops 1 and 2 could be scheduled to overlap. Simple comparison of the array subscripts is not sufficient: we have to include the value of the lower and upper bounds of the loop indices in the analysis. Loop 3 must follow loops 1 and 2, but no iteration of loop 3 uses a value computed in an earlier iteration of this loop, and this observation can be exploited when scheduling this loop for a pipelined processor. In loop 4, each iteration depends on the result of a prior iteration; reference 4 restricts the parallel execution of different iterations.

A FAST ALGORITHM FOR GLOBAL DATAFLOW ANALYSIS FOR ARRAYS

In scientific programs, most computation time is spent in for-loops. As illustrated above, it is important that the values of the lower and upper bounds of loop indices, if available, are included in the analysis. Of the different scalar dataflow analysis methods, interval analysis is best suited to be generalized to arrays. In interval analysis,

```
for i := 0 to 5 do                              /* loop 1 */
     a[i] := 0;                                 /* def 1 */

for j := 6 to 10 do                             /* loop 2 */
     a[j] := j;                                 /* def 2 */

for k := 0 to 10 do                             /* loop 3 */
     a[k] := a[k+1] + c;                        /* def 3, ref 1 */

for m := 2 to 10 do                             /* loop 4 */
     if (a[m] > 0)                              /* ref 2 */
        then a[m] := (a[m] +                    /* def 4, ref 3 */
                              a[m-2])/2;        /* ref 4 */
        else a[m] := 0;                         /* def 5 */
```

Figure 1. Program example

a complicated flow graph is analysed incrementally by performing dataflow analysis one 'interval' at a time. Each loop corresponds to an interval, and it is straightforward to include information on loop counters systematically in the analysis. Interval analysis requires that the program's flow graph be reducible; this is guaranteed by the block structure of the programming language in our case and is satisfied for almost all programs in other languages.

In the next section we briefly review the classical interval analysis algorithm for scalars, and we then extend the algorithm to include arrays.

Interval analysis for scalars

The use of interval analysis to collect global dataflow information was first described in References 2 and 4 and has later been extended to deal with backward global dataflow problems.[16] Since then other forms of interval analysis have been proposed.[17–20] The methods differ mainly in the way the intervals are defined, but they are all based on the same idea, as is informally described below for the case of reaching definitions.[21] The terminology is taken from Reference 1.

Global dataflow analysis for a program with a cycle-free flow graph is simple and can be performed in a single pass. Taking a forward flow problem (such as reaching definitions) as an example, information is collected for each basic block and is then propagated 'downwards', starting with the unique start node and proceeding along the control flow graph. Since the program is cycle-free, all potential reaching definitions for any point in the program will be 'upstream' of that point in the graph, and one single pass over the graph is sufficient to find all reaching definitions for all references in the program.

If the flow graph has cycles, a single pass over the program graph cannot uncover all reaching definition: for nodes that are the target of backward arcs in (the spanning tree of) the graph, information about what happens 'downstream' in the graph is required to determine the reaching definitions. The idea behind interval analysis is to break up the analysis of a potentially very complicated flow graph into the analysis of a sequence of smaller graphs. During the first part of the analysis, the analyser isolates *intervals* from the flow graph; an interval is a subgraph with a single header node and with one or a limited number of back arcs and corresponds roughly to the body of a loop. Because of the way intervals are selected, it is possible to determine by a single pass over the interval what variables are set in the interval, and what definitions in the interval can reach exit nodes of the interval. Using the dataflow information about one interval, we can determine what definitions from inside the interval can reach the loop header through the backward arcs, and we can replace the interval by a *summary node* that has the same effect on the rest of the graph as the interval it replaces. This process is repeated until all backward arcs are eliminated; when this is done, a cycle-free flow graph remains and Phase 1 of interval analysis is completed. Phase 1 propagates the information from inner loops to the top level, and at the end, the reaching definitions for all nodes in the reduced, cycle-free flow graph can be found in a single pass.

The second phase propagates the information from the top level to the innermost loops. In this phase of the analysis, the reaching definitions for all basic blocks in the original graph are calculated by expanding the summary nodes one by one, in the reverse order of how they were created. Each time a summary block is expanded, the reaching definitions of the header node can be found as the union of the reaching definitions that reached the summary node in the flow graph before expansion, and of the definitions inside the interval that reach the header node through the backward arcs, as determined during the first phase. These reaching definitions can then be propagated in the interval to find the reaching definitions for all basic blocks in the interval.

Extending interval analysis to arrays

Only the names of the variables are propagated during scalar dataflow analysis, but when extending the analysis to arrays, it is also necessary to keep track of what elements in the array are referenced. Listing individual array elements is not an attractive solution: the sets of variables and definitions would become very big, they could not handle procedure or function arguments, and handling such sets would be very slow. For this reason, we chose to group array elements in *regions* that can be represented in a concise way. Figure 2 shows two array references inside a loop; both references access a rectangular region in the array. Since we need a representation that fosters a fast implementation of the intersection, union, and difference of sets of definitions or variables, we decided to restrict regions to have a rectangular shape as will elaborated on in the implementation section. Regions of arrays have been used in other compilers, for example to capture the extended effect of a procedure call on its actual parameters.[22, 23]

Reaching definitions

We present the interval analysis algorithm for reaching definitions. The algorithm handles both arrays and scalars at the same time; scalars are analysed as arrays of dimension zero. The term *variable* in the following means either a scalar variable or a region in an array, and the basic blocks header and exit refer to the header and exit basic blocks of intervals. Finding the intervals is usually trivial and can be done as part of building the program flow graph.

```
for i := IL to IU do
  for j := JL to JU do
    a[i,j] := ...
           := a[i+2, j-1]
```

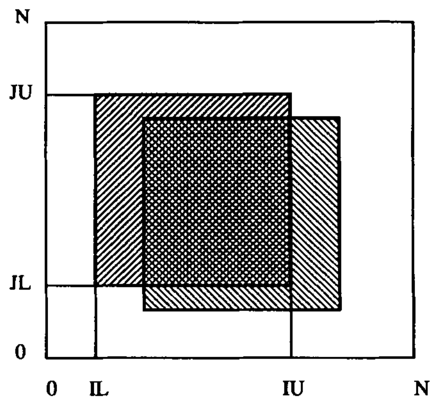


Figure 2. Two regions referenced in an array

Phase 1: propagating information from inner loops to the top level

For each basic block i in the program, calculate:

(a) MAYGEN[i]: the set of definitions in basic block i that reach the end of the block; associated with each definition is the set of variables that may be defined by that definition.
(b) DOESGEN(i]: the set of variables that are definitely defined within basic block i.
(c) KILL[i]: the set of definitions outside basic block i that are killed by the definitions in this block. This set does not have to be calculated explicitly, DOESGEN can be used instead.

Starting with the inner loops, we execute the following steps for each interval (loop body):

1. For each basic block i calculate:

   (a) $M_{in}[i]$: the reaching definitions for the entry of basic block i on any path from the loop entry.
   (b) $M_{out}[i]$: the reaching definitions for the exit of basic block i on any path from the loop entry.
   (c) $U_{in}[i]$: the set of variables that are definitely defined on any path from the loop entry to the entry of basic block i.
   (d) $U_{out}[i]$: the set of variables that are definitely defined on any path from the loop entry to the exit of basic block i.

   The sets $M_{in}[i]$, $M_{out}[i]$, $U_{in}[i]$ and $U_{out}[i]$ are found by solving the following flow equations for all basic blocks, starting with the header node, and proceeding down the flow graph of the loop body:

$$M_{in}[i] = \cup_p M_{out}[p]$$
$$M_{out}[i] = (M_{in}[i] - \mathrm{KILL}[i]) \cup \mathrm{MAYGEN}[i]$$
$$U_{in}[i] = \cap_p U_{out}[p]$$
$$U_{out}[i] = U_{in}[i] \cup \mathrm{DOESGEN}[i]$$

   where block p is a predecessor of block i in the loop. As mentioned earlier, KILL[i] is not calculated explicitly; instead, DOESGEN[i] is used to determine what definition have to be removed from M_{in}. For the header node of the interval (i.e. the first basic block in the loop body), U_{out}[header] and M_{out}[header] are DOESGEN[header] and MAYGEN[header], respectively.

The sets M_{out}[exit] and U_{out}[exit] describe the effect of a single loop iteration.

2. Replace the body of the loop by a summary basic block called loop. DOESGEN [loop] is the union over all possible values of the loop index of U_{out}[exit] of the loop body. MAYGEN[loop] is obtained by taking the union over all possible values of the loop index of M_{out}[exit], and by eliminating definitions which define variables that are overwritten in later iterations. Two cases can occur during the calculation of the union:
   (i) If an array subscript of the reference is a function of the loop index for that loop, the loop bounds determine the part of the array that is modified. The index expression and the loop bounds are used to calculate the boundaries of the region of the array.

(ii) If an array subscript expression is loop-invariant, then the last iteration of that loop defines the reaching definition of the entire loop.

Scalars or references to arrays where all array subscript are constants always fall in the second category.

The summary basic block created in the last step summarizes all actions in the loop body. After the innermost block is replaced with the summary node, its enclosing loop (if it exists) becomes the innermost loop, and at the end of Phase 1, all the loop bodies have been replaced with summary locks, and a cycle-free flow graph remains.

Phase 2: propagating reaching definitions from the top level to inner loops

We first calculate the reaching definitions for each of the basic blocks and summary blocks of the cycle-free flow graph obtained at the end of Phase 1. We then find the reaching definitions for statements inside loops by replacing the summary blocks one after another by the loops they represent, starting with the outer loops. For each loop, the reaching definitions for (the entry of) the blocks bb in the loop body fall into three classes:

1. *Definitions in the current iteration of the loop*: they correspond to the set M_{in}[bb] that was calculated during the first pass. This group of dependences corresponds to the *loop-independent dependences* described in Reference 5.
2. *Definitions from previous iterations of the loop*: these definitions reach the header node of the loop through the backward arc, as determined during the first phase. The set consists of the union of all reaching definitions of the previous iterations, minus those overwritten in earlier iterations or in the current iteration. M_{out}[loop exit], U_{out}[loop exit] and U_{in}[bb] are needed for this computation. This group of dependences corresponds to the *loop-carried dependences* described in Reference 5.
3. *Definitions from outside the current loop*: these definitions are the reaching definitions of the summary block before expansion that are not overwritten in an earlier iteration or in the current iteration. The reaching definitions for the summary block, U_{in}[bb] and U_{out}[loop exit] are required. We will call these dependences *interloop dependences*.

Figure 3 depicts the classes of reaching definitions for a Basic Block *I* inside a loop body (incated by the shaded region). Note that definitions that appear textually before the basic block in question can be either in Class 1 or Class 3. Definitions that appear textually later (and are not subsequently killed) are in Class 2.

The reaching definitions for the entry of a basic block can be propagated inside the basic block to find the reaching definitions for each statement. To find the reaching definitions for a specific reference we compare the subscript expressions of the reference with the regions of the variable in the reaching definition set. The condition that the reference has to lie inside the region results in a condition on the loop counters which determines for which iterations the definition is a reaching definition for the reference.

This algorithm visits each basic block in the program twice: once during Phase 1, and once during Phase 2. For each visit, each reference node or definition node is considered once. The run-time of the algorithm is linear in the number of basic blocks.

To catch uninitialized variables, the compiler inserts initialization definitions for all variables at the beginning of the program before starting the dataflow analysis. When

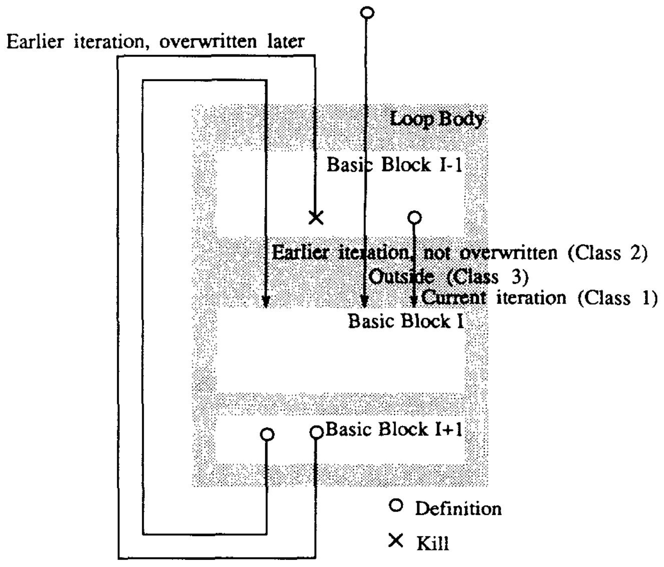

Earlier iteration, overwritten later
Loop Body
Basic Block I-1
Earlier iteration, not overwritten (Class 2)
Outside (Class 3)
Current iteration (Class 1)
Basic Block I
Basic Block I+1
O Definition
X Kill

Figure 3. Classes of reaching definitions

the set of reaching definitions for a variable includes such a 'fake' definition, there must be an execution path through the flow graph that does not initialize all elements of the array, and the compiler generates a warning. Since the complete program is analysed systematically, the compiler detects uninitialized scalars and array elements in a uniform way.

Example

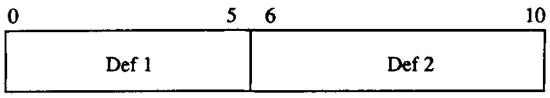
Let us look in detail at the different steps of the dataflow analysis for the program from Figure 1. For the first loop body, MAYGEN[block 1] consists solely of definition 1, and DOESGEN[block 1] contains the variable a[i]. When we replace the loop by a summary basic block during phase 1, the term a[i] is replaced by the region a[0..5]. Similar procedures on the other loops yield the following DOESGEN and MAYGEN sets at the end of Phase 1:

DOESGEN[loop 1] = { a[0..5] }	MAYGEN[loop 1] = { Def 1: a[i], i: 0..5 }
DOESGEN[loop 2] = { a[6..10] }	MAYGEN[loop 2] = { Def 2: a[j], j: 6..10 }
DOESGEN[loop 3] = { a[0..10] }	MAYGEN[loop 3] = { Def 3: a[k], k: 0..10 }
U_{out}[exit loop 4] = { a[m] }	M_{out}[exit loop 4] = { Def 4: a[m], Def 5: a[m] }
DOESGEN[loop 4] = { a[2..10] }	MAYGEN[loop 4] = { Def 4: a[m], m:2..10; Def 5: a[m], m: 2..10 }

The compiler detects that the fourth loop always defines a[2..10], even though the loop contains a conditional statement.

To illustrate how the second phase of the algorithm works, we calculate the definitions that reach the body of loop 3, for interaction K. We expand the summary nodes representing loops 1 and 2, and we calculate the definitions that reach the start of loop 3:

```
Def 1: a[0..5];
Def 2: a[6..10]
```



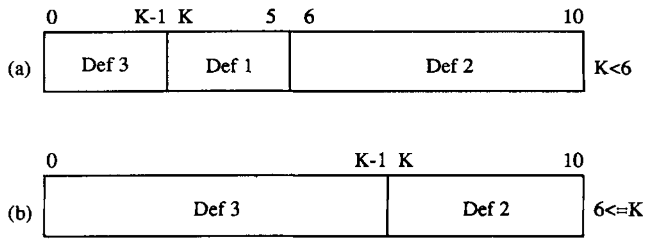
We now expand loop 3. The definitions that reach the start of iteration K include those definitions that reach the loop entry and are not overwritten in iterations 0 to K−1 (the interloop dependences of Figure 3):

```
Def 1: a[K..5];
Def 2: a[6..10], K<6;     Def 2: a[K..10], 6<=K
```

and definitions from previous iterations that are not overwritten in later iterations (the loop-carried dependences of Figure 3):

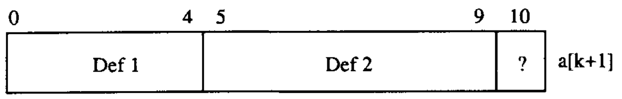
```
Def 3: a[0..K-1]
```

The whole set of definitions that reach the start of iteration K of loop 3 is as follows:



To find the reaching definitions for a[k+1] (reference 1 in loop 3), we compare its array subscript expression for the 'current' iteration, i.e. for k = K, with the array subscript ranges of all definitions that reach this block. This gives a condition on the loop index, specifying for which values of the loop index the definition is a reaching definition. For some definitions the condition will be false for all values of the loop index (for example definition 3). We find the following reaching definitions:

```
Def 1: K<=4
Def 2: 5<=K<6
Def 2: 6<=K<=9
```

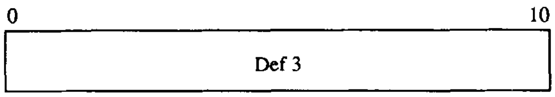


or, after merging the ranges for definition 2,

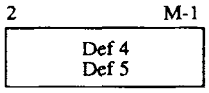
```
Def 1: K<=4
Def 2: 5<=K<=9
```

There is no reaching definition for a[k+1] for k=10.

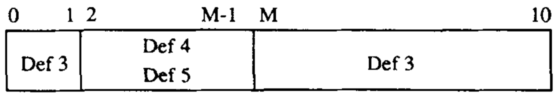
The reaching definitions for the references in loop 4 can be found in a similar way: definition 3 always reaches the loop entry:



Definitions 4 and 5 from earlier iterations reach the start of iteration M,

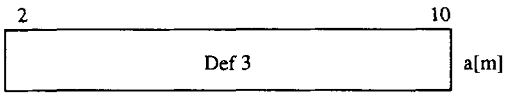


and combining this information yields the set of all the definitions that reach the start of iteration M:



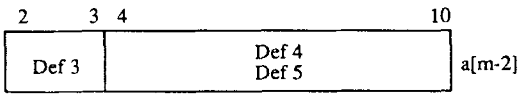
The reaching definitions for a[m], reference 3, for iteration m=M are then

```
Def 3: 2<=M<=10
```



and for reference 4 a [m−2], we have

```
Def 3: 2<=M<=3
Def 4: 4<=M<=10
Def 5: 4<=M<=10
```



There are two reaching definitions for this reference since the analyser cannot determine which branch of the if-statement will be taken.

Other types of dataflow analysis

The above algorithm calculates reaching definitions. They correspond roughly to the *flow dependences* as used by the Parafrase systems.[8] They limit the reordering of statements because, to preserve the semantics of the program, all reaching definitions of a reference have to be executed before the reference itself.

There are two additional restrictions on the reordering of statements: the order of two definitions of the same array element or scalar should not be changed, and the use of a scalar or array element should not be moved after a later redefinition of that variable. These dependences correspond to *output relation* and *antidependence*.[8] The definitions that overwrite a value set by an earlier definition, or a value used in an earlier statement, can be looked at as reaching definitions for the program in which the control flow has been inverted. They are obtained by changing the above algorithm for reaching definitions to solve the backward flow problem, in a manner similar to the extension of scalar interval analysis to deal with backward global dataflow problems.[16]

The following changes have to be made to the algorithm for backward flow analysis:

1. Redefine DOESGEN and KILL as the definitions that are visible from the entry of the basic block. In the reaching definition problem they are the definitions that reach the end of the basic block.
2. Use the following equations:

$$\begin{aligned}
\mathrm{M_{out}}[i] &= \cup_s \mathrm{M_{in}}[s] \\
\mathrm{M_{in}}[i] &= (\mathrm{M_{out}}[i] - \mathrm{KILL}[i]) \cup \mathrm{MAYGEN}[i] \\
\mathrm{U_{out}}[i] &= \cap_s \mathrm{U_{in}}[s] \\
\mathrm{U_{in}}[i] &= \mathrm{U_{out}}[i] \cup \mathrm{DOESGEN}[i]
\end{aligned}$$

   with s the successors of basic block i.

The backward and forward algorithms correspond directly to the different types of scalar dataflow anlaysis. Our reaching definitions correspond to the reaching definitions in Reference 1 (forward flow analysis with confluence operator). The backward problem corresponds to the du-problem in Reference 1 (backward flow analysis with confluence operator), but the role of uses and definitions is inverted.

USE OF THE DATAFLOW ALGORITHM IN THE WARP COMPILER

The Warp machine and compiler

The Warp machine is a systolic array of ten high-performance processors.[24] Each processor has multiple highly pipelined functional units that are controlled by a 'wide instruction word'. To use the fine-grain parallelism effectively, detailed dataflow information for both arrays and scalars is needed. For example, each processor of the Carnegie Mellon Warp machine has two arithmetic units, and each unit has a five-stage pipeline. Unless the compiler can overlap the execution of several operations, the performance of such a processor can be limited to 1/10 of its peak, or less if other resources like memory or registers are involved.

The algorithm presented in the previous section is used to collect the global dependence information. The analysis is implemented as an independent phase in the compiler; the generation of the data is separated from its use. This global information is used for the traditional global optimizations (loop invariant removal, induction variable optimizations) as well as in the code scheduler. Since the code scheduler knows the resource requirements of a code segment under consideration, it can better assess the benefits of moving operations to different basic blocks. By making the global flow information available to the code scheduler, code motion can be performed in the back-end, leading to results superior to those that could be achieved by estimates in a separate phase.

The input language for the Warp compiler (W2[25, 24]) is similar to Pascal (as far as array references and loops are concerned), but the algorithms presented in this paper apply to other languages (including FORTRAN) as well.

Program representation

The compiler represents a program as a flow graph of dags; each basic block is modelled by a node in the flow graph, and the nodes in the dag represent operations in the basic block.[1] Arcs between nodes in the flow graph denote control dependences, and arcs between dag nodes indicate data dependences and necessary sequencing constraints. The compiler normalizes loops so that the loop index is zero for the first iteration.

Since the operations represented by the nodes in a loop body are executed once for each loop iteration, a simple arc is not sufficient to capture all the dependence information, and labels have to be added to the dependence arcs to distinguish between the different instances of nodes. Each arc is labelled by a *context* and an *instance*. The context defines the instantiation of the target (use) node in the data dependence relationship, and the instance specifies the instantiation of the source (definition) node. The use arcs for the program in Figure 1 are shown in Figure 4, with the

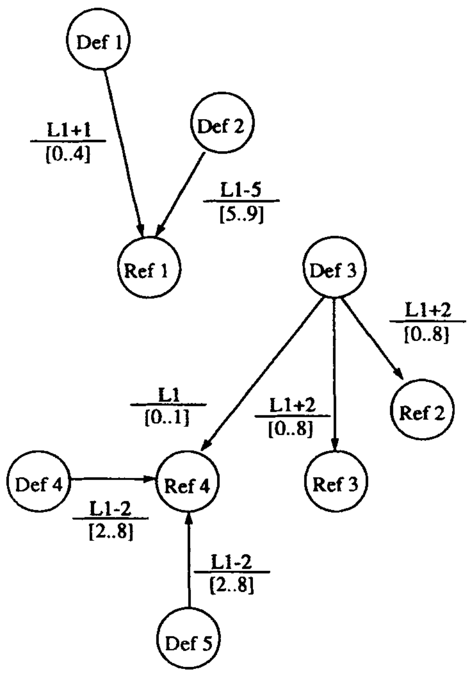


Figure 4. Dependence arcs

instance (shaded) specified above the context for each arc. For example, reference 1 uses the result of definition 1 in iterations 0 to 4, and uses the result of definition 2 for iterations 5 to 9. (We saw earlier that a [11] for iteration 10 is undefined.) The contexts for these arcs are simply 0..4 and 5..9, respectively. When two or more arcs with overlapping context fields arrive at the same node, this indicates that several definitions can generate the result. For example, either definition 4 or definition 5 can generate the value used to be reference 4. The expressions in the instance can either be constants or a functions of the normalized loop index L1 of the source node. For example, when L1 is between 5 and 9, the reference of the L1-th iteration in loop 3 depends on the (L1−5) -th iteration of loop 2.

Implementation of regions and sets

Regions play a critical role in the algorithm. We decided to restrict regions to rectangular areas in arrays, so the only array access pattern that can be represented accurately is a sum of rectangular areas. This restriction is not inherent in the algorithm, but is an implementation decision that is based on a trade-off between complexity of the implementation and accuracy of the information. The following observations motivate this decision:

(a) Loops access rectangular areas in arrays in the vast majority of the cases in our application domain.
(b) Rectangular areas in arrays can be represented concisely by recording the upper bound and lower bound for each dimension in the array.
(c) If the access pattern of an array reference in a loop is not rectangular, the opportunities for optimization in that loop are usually restricted, i.e. it would be very hard for the back-end code generator or optimizer to use information about more complex access patterns.

The access pattern of an array reference is guaranteed to be rectangular if the following conditions are fulfilled:

1. All array subscript expressions are either constant, or a linear function of a loop counter.
2. In each array reference, a loop counter can appear in only one array subscript expression at a time.
3. If there are several references to the same array in the same loop, then the same array subscript depends on the loop counter either in all the references or in none.

While building the MAYGEN and DOESGEN sets, the compiler tests whether these conditions are fulfilled. For references that violate some of the conditions, worst-case assumptions are made and these ‘worst case’ sets of array elements are placed in MAYGEN and DOESGEN. For example, in the case of reaching definitions, the region is extended to include the whole array dimension. As a result, the rest of the flow analysis only has to handle rectangular regions.

Rectangular regions are described by a list of independent *dimension records*. A dimension record describes valid values for one dimension of the array: it captures the lower bound and upper bound for that dimension, an *offset function* that describes how the loop index relates to the array subscript for that dimension, and possibly conditions on loop counters. In the remainder of this section we discuss in more detail what information has to be stored in the dimension records, and how the above conditions simplify the representation. We will represent loop counters by i, j ..., and array subscripts by α, β, ...

The elements of the sets MAYGEN and DOESGEN at the beginning of phase 1 are definitions and variables of the form A[α,β], with α and β either constant, or linear expressions that depend on a loop counter, so dimension records are of the form (type 1)

```
array subscript = offset_fct(loop index)
```

To calculate the overall effect of the loop on the surrounding program, we evaluate the array subscript expression for all values of the loop counters, possibly eliminating elements that are overwritten in later iterations. After this expansion, the dimension records are of the form (type 2)

```
array subscript <= upperbound
array subscript >= lowerbound
```

As we replace loops by summary blocks, all dimension records are gradually turned into dimension records of the second type. Constant array subscripts are represented by dimension records of the second type with both the upper and lower bound equal to the constant. When reaching definitions are propagated in the cycle-free graph in the beginning of phase 2, all dimension records are of the second type.

When propagating reaching definitions from the outer loops towards the inner loops in Phase 2, we replace the summary blocks by the loops they represent. When building the set of reaching definitions for iteration I of such a loop, we have to consider definitions from three sources:

1. *Definitions from the current iteration*: these regions are calculated in phase 1, and their dimension records are of types one and two, with loop counter equal to I.
2. *Definitions from previous iterations that were not overwritten in later iterations*: we have to consider two cases:

   (a) One of the array subscript expressions depends on the loop index: we have to evaluate the index expression for the values of the counter in the previous iteration, and the dimension record will be of the format

   ```
   array subscript <= f(I)
   array subscript >= g(I)
   ```

   with f, g, functions of the loop index. Such a dimension record is of type 3.

   (b) None of the array subscript expression depends on the loop counter: in this case the reaching definition will be the definition executed in the previous iteration, if the counter is higher than its lower bound, so the dimension record looks like

   ```
   loop counter = I-1
   loop counter > lowerbound
   ```

This is a dimension record of type 4.

3. *Definitions from outside the loop that were not overwritten in the previous iterations*: they are found by subtracting the variables set in iterations lowerbound .. (I − 1) from the definitions that reached the summary node, and the resulting dimension records are of types 3 and 4.

To find the reaching definitions for a specific array reference A[α, β, ...], the index expressions α, β, ... (e.g. α = i+3) are substituted in the conditions in the dimension records of the regions of A in the reaching definitions for the statement. This results in a boolean expression in the loop counters; this expression can be unconditionally true or false, or it can express a condition on the loop counters, indicating the conditions under which the reference falls in the region. In the first and last cases, an arc is added to the program graph with a tag that indicates for what iterations the definition is a reaching definition for the array reference.

When the array subscripts do not fulfil all the conditions listed above, it is also necessary to represent conditions between array subscripts, and between loop counters, i.e.

⟨loop counter⟩ ⟨relation⟩ ⟨loop counter ⟩

or

⟨array subscript⟩ ⟨relation⟩ ⟨array subscript⟩

For example, given the definition A[i+j,i], with i and j ranging between 0 and 10, then after replacing loops j and i by summary blocks, we have the following region:

$0 <= \alpha <= 20$
$0 <= \beta <= 10$
$\beta \leq \alpha$.

This type of information makes the set operations more complicated and rarely results in useful information, and is therefore conservatively approximated.

The current implmentation does not use information that can be derived from the test in if-then-else statements. If the test puts a condition on a loop counter, this information could be used to collect more specific information about what elements are referenced in the true and false branches of the if-statement.

Operations on sets of variables and definitions

The following operations are performed on sets of variables and sets of definitions:

(a) *union of two sets of definitions*: for each definition, take the union of their variables
(b) *difference of a set of definitions and a set of variables*: for each definition in the first set, take the difference of its variable, and of the variable in the second set with the same name
(c) *intersection of two sets of variables*: for each variable in the first set, take its

intersection with the variable in the second set with the same name

(d) *union of two sets of variables*: for each variable in the first set, take its union with the variable in the second set with the same name

(e) *expand loop index in a set of definitions, while removing variables overwritten by later iterations, given in a set of variables*: for each definition, if there is no variable with the same name, and with a smaller index in the set of variables, then place conditions on the array subscript that correspond to the loop bounds, otherwise impose a stronger condition that excludes overwritten array elements.

(f) *expand a loop index in a set of variables*: replace all array subscripts which are a function of that loop index by two conditions on the loop index.

Operations on variables and regions

A variable is represented as a boolean expression in a *normalized sum of products* form of regions, and the operations on variables can be described as boolean operations on these boolean expressions. In this context, 'normalized' means that regions do not overlap, i.e. the and of any two terms should be false. We have the following operations on variables:

(i) *union*: take the or of the two sets followed by a normalization step; the result is a sum of products

(ii) *intersection*: take the and of the two sets, and bring into sum of products form using the distributivity law; no normalization is necessary

(iii) *difference*: take the and of the first set and of the not of the second set; bring into sum of products form and normalize.

The boolean operations on individual regions are straightforward to implement. For example for the union, add the regions of the first set term by term to the second set.

Normalization compares every pair of regions in a variable, and if the regions overlap, it replaces them by a set of non-overlapping regions. Normalization in general reduces the number of regions in a variable, since it eliminates identical and containing regions; this speeds up further operations on the sets. Given two regions in an array of dimension n, each of the dimensions will fall under one of the cases in Figure 5.

The normalization algorithm for two regions is straightforward:

1. For each dimension, check whether the dimensions of the two regions are disjunct, containing or overlapping.
2. Test for the following three special cases:

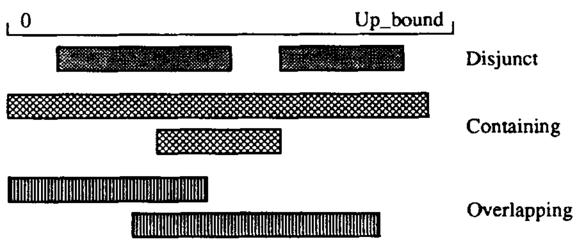


Figure 5. Cases for dependency information

   (a) If any of the dimensions are disjunct, the regions are disjunct already.
   (b) If all dimensions of one region are contained in the dimensions of the other region, the smaller region can be dropped from the set.
   (c) If the regions differ in only one dimension, they can be merged into one larger region.

3. In all other cases, the region with the largest number of dimensions containing the other region is returned as the first region of the normalized set. The second region is divided into a number of smaller regions as follows:
   (a) Select all dimensions that were overlapping, or that contained the first region.
   (b) For each of these dimensions, slice the region into two (for overlapping dimension), or three (for containing dimension) areas. Continue working with the region that is the intersection of the two regions in that dimension; the other regions are accumulated in the result.

EVALUATION

Accuracy of the information

It is difficult to compare the different algorithms that have been used for dataflow analysis on arrays. Several papers classify data dependences and describe the transformations and optimizations that can be done once the dependences are known,[8, 10, 12, 26, 27] but they rarely describe how accurate the dependence information is, or even how it is obtained. An interesting issue is also that the accuracy of the derived information is usually not determined by the dataflow analysis algorithm, but by its implementation: what algorithm is used to disambiguate two subscript equations, and how much effort was spent on including knowledge about the possible values of loop indices in the analysis.

Implementors of compilers determine the accuracy of the dataflow information they want to obtain based on how useful the information is for the types of applications and architectures the compiler is targeted for, and on how much effort is required to get the information. As a result, the dataflow algorithm used indirectly influences the accuracy of the information: a cleaner and more efficient algorithm will make it easier (or feasible) to extract more information.

Various papers describing the research performed at the University of Illinois[11] distinguish three types of dependences between two statements; the usage pattern of the variable that creates dependence determines the type of the dependence relationship:

(i) *true dependence*: a definition followed by a use
(ii) *antidependence*: a use followed by a definition
(iii) *output dependence*: two consecutive definitions.

The forward version of our algorithm detects true dependences, while antidependences and output dependences are detected by the backward version. Parafrase uses Banerjee's algorithm[9] to test two subscript expressions for dependences.

The dataflow analysis used in the PFC system is described in References 5 and 28. There, two types of dependences that can hold between two statements are introduced, and these types differ in the relative position of the iterations in which the conflicting instances of the two statements occur in the nested loop:

(a) *loop-carried dependences*: the conflicting accesses occur in different iterations
(b) *loop-independent dependences*: the conflicting accesses occur in the same interation.

Our algorithm detects both loop-carried and loop-independent dependences, and we also detect interloop dependences, dependences between accesses in loops that are not nested (or that are not in loops). The dataflow analysis algorithm in the PFC compiler at Rice tests every pair of statements inside nested loops for dependences. The GCD test and Banerjee's inequality are used to test for subscript analysis. Subscript analysis is done one subscript at the time, which is similar to what we do.

Dataflow analysis for the Multiflow compiler[29] is done on a 'demand' basis: when the scheduler wants to know whether two references can interfere, it asks a special disambiguation module. The user can give hints to help the disambiguator.

Our implementation of the dataflow algorithm presented in this paper detects data dependence information that is similar to the information derived in for example PFC and Parafrase.

In some cases our information is less accurate because our algorithm to disambiguate index expressions is very simple. For example, in the case of step sizes different from one, we make the worst-case assumption. We found that the simple algorithm was sufficient for the applications that are used in our environment (mainly low-level vision applications). Our algorithm could support a more detailed analysis, for example by supporting more general regions such as 'sparse rectangles'. We decided not to do this because it would make our data structures more complex and it would slow down the analysis, whereas the payoff would be marginal.

In other cases our information is more accurate. For example, we also detect interloop dependences, and we do not only detect the presence of a data dependence, but also have accurate information on what loop indices or parts of the array the dependence applies to. This information is probably of limited use when vectorizing, but it is useful when scheduling instructions for a pipelined or for a parallel architecture with multiple parallel functional units. For example, it makes it possible to overlap the epilogue of one loop with the prologue of the following loop.

Comparison

In addition to the benefits obtained from treating scalars and arrays uniformly for global dataflow analysis, there are other reasons that make computation of global dependencies based on program flow (as an alternative to a loop-by-loop analysis) attractive for an optimizing compiler.

First, since the dependency information is based on flow analysis, the compiler can in some cases determine that an array element is not initialized and issue a warning. This feature is probably of limited use for big programs with many paths through the source code, since a compiler must use approximations and cannot compute an exact solution to the dataflow equations. But we have experienced the usefulness of such warnings in compiling library programs of moderate size where the compiler alerted the user.

Secondly, a data dependency module that is based on flow analysis can conveniently compute interloop dependences (see Figure 3). Those dependences are important whenever the compiler attempts to overlap the epilogue of one loop with the prologue of the next. The compiler also needs to know about these dependences whenever an action outside the loop can influence the code generated for the loop body. For example, on machines with deep pipelines, an operation O that is started outside the loop affects the schedule of the body. Given interloop dependences, the compiler can try to find operations that do not use the result of operation O. If interloop dependences are not available, the compiler has to make a worst-case assumption, possibly resulting in a worse code sequence.

Furthermore, computing data dependences based on the flow graph of a program can actually require less work than comparing all definitions of an array variable with all uses of this variable in a set of nested loops. For example, there are two definitions and two references in the loop body depicted in the following code fragment:

```
⟨block i⟩
  .. := a[i]        /* ref 1 */
⟨block j⟩
  a[i] := ..        /* def 1 */
⟨block k⟩
  .. := a[i]        /* ref 2 */
⟨block l⟩
  a[i] :=           /* def 2 */
```

A straightforward comparison would compare each reference with each definition, whereas a flow-based computation of dependences eliminates half of the comparisons in this example.

Use of the information

The W2 compiler for Warp has been extensively used for three years in applications such as low-level vision for robot vehicle navigation, image and signal processing, and scientific computing.[30] Many of the programs in these applications are intrinsically parallel, and the global flow analyser can often detect the available parallelism in the code. Many of the loop optimizations such as dead code removal and loop invariant removal have been implemented. However, source-to-source transformations commonly found in vectorizing compilers, such as loop interchange or loop jamming, have not been implemented, although the information for such optimizations is available. Currently, we rely on users or high-level program generators to apply such optimizations.

Since a machine with a wide instruction word offers the code scheduler numerous choices, the code scheduler in the back-end is the principal user of the global flow information. Using this information, the scheduler often produces near-optimal, and sometimes even optimal code. The speed-up obtained by the global dataflow analyser and the scheduling techniques for a set of 72 programs, collected from the applications developed at Carnegie Mellon, has been reported elsewhere.[31, 32] The performance was compared with that obtained by local compaction only, and a speed-up of 50 to 250 per cent was observed.

For example, in loop 4 of Figure 1, with the dependence arcs as depicted in Figure 4, it is clear to the code generator that Ref 4 of iteration M must follow Def 4 and Def 5 of iteration M − 2. However, there are no restrictions regarding Ref 4 of iteration M and Def 4 and Def 5 of iteration M − 1, and a possible order of execution is:

Ref 4—iteration M
Ref 4—iteration M + 1
Def 4—iteration M
Def 5—iteration M
Def 4—iteration M + 1
Def 5—iteration M + 1

Of course, the loop body would now be executed only 10/2 times, to adjust for the fact that code for two iterations has been generated.

Cost

There are two types of cost associated with computing the global dependences: the implementation cost (to design, implement and maintain the dependency analyser module) as well as the execution cost (to obtain the dependency information for the compilation of a program).

The dataflow analysis described here has been implemented in Common Lisp. This implementation is extremely compact; the complete module is contained in about 9000 lines of source code. One of the reasons for this compactness is the nice property of interval analysis that it treats reaching and killing definitions symmetrically, and one set of routines can be used to obtain both forward and backward flow information. About 1·5 man-years were spent on the implementation. The compactness of the implementation was essential to the success of the implementation; a single person was able to handle the implementation.

Separating the analysis from the use of the flow information proved to be very beneficial for the engineering and management of the compiler. The representation described earlier is the sole interface between the code generator and the flow analysis module. This organization keeps both modules manageable. It also makes the global flow analysis machine-independent. When we retargeted the code generator for the Integrated Warp System (a single-chip VLSI implementation of the Warp cell with a completely different architecture), the flow analyser required no changes and is shared by both compilers.[33, 34]

Interval analysis is an attractive method for global flow analysis since the number of times each basic block is visited is fixed and is independent of the operations in that basic block. To assess the practical relevance of this method, we obtained compilation metrics for a collection consisting of 35 image-processing programs. These are programs from the SPIDER image processing library[35] that were recorded for the Warp machine.[36] For these programs, we measured that global flow analysis takes between 5 and 30 per cent of the total compilation time, and 16·7 per cent on average. (Compilation time exlcudes any preprocessor or macro-expander time and also excludes assembly and linking.)

To allow the user a comparison with other implementations, Table I provides information regarding the compilation of sample programs to solve some well-known problems. The second column contains the CPU time for the compilation, the third column indicates what percentage of that time is due to the computation of the global dependences. These data were obtained on a 20MHz SUN-3 workstation using the Lucid Common Lisp compiler (Release 3.0.1) and represent the average of multiple compilations. As above, the measured compilation time excludes preprocessor, assembler and linker time.

When comparing Table I with other results, three points must be kept in mind. First, the choice of implementation language (Common Lisp in our case) and the quality of its compiler significantly affect the total compilation time. Secondly, the Warp compiler was developed as part of a research project at Carnegie Mellon University and has never been rewritten or tuned for compilation speed. Thirdly, the cost of *using* the global flow information to optimize the user program is charged against the code generator in our compiler; other implementations often consider dataflow analysis and optimization together. In summary, in this implementation less than 20 per cent of the total compilation time is spent in the global dataflow module, and we consider this to be acceptable for an optimizing compiler.

CONCLUDING REMARKS

This paper has described a systematic approach to analysing the flow information on array elements. Our global flow analyser is based on an extension of interval analysis. By qualifying the use and definitions of array variables with 'regions', the information is accurate up to the array element level to the extent that the regions precisely capture the array elements affected. The global flow information is propagated over the nesting levels of the program, using the structure of the program to organize the gathering of the information. After the information has been collected, the nodes in the flow graph are annotated to present the global dependences in a form that is suitable for use by the back-end code generator.

The approach described in this paper has several advantages. Since it is based on interval analysis, the implementation of a flow analysis module is extremely compact, without sacrificing capabilities. The same flow analysis module solves both forward (reaching definitions) and backward (killing definitions) problems without increase in

Table I. Compilation cost

Program	Compilation time (s)	Global dataflow (%)
Discrete cosine transform (DCT)	476·3	11·9
Dynamic programming	256·2	19·5
Fast Fourier transform	666·1	10·8
Inverse DCT	419·4	12·7
Mandelbrot set	197·5	6·5
Matrix multiplication	114·8	6·1
Singular value decomposition	486·9	10·1
Successive overrelaxation	123·5	4·7

code size. The information collected by this flow analysis tool is sufficient for the applications typical of our environment.

The second major advantage of this approach is that comprehensive flow information is obtained and represented uniformly for both scalar and array variables. This information is critical to modern scheduling techniques, such as hierarchical reduction and software pipelining. Treating scalars and arrays uniformly simplifies the interface between flow analysis and the back-end code generator.

The separation of the generation and use of the dataflow information and the representation of the information in the general dependence graph have proven to be beneficial. The code generator is in the best position to make decisions on which optimizations to apply but is not burdened with the task of analyzing the program; this task is performed separately. This compiler structure conveniently addresses software engineering concerns; only one analyser needs to be written for different compiler back-ends (code generator), and the information generated can be used for different optimizations.

ACKNOWLEDGEMENTS

The research was supported in part by the U.S. Defense Advanced Research Projects Agency (DOD) monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251, and in part by the U.S. Office of Naval Research under Contracts N00014-87-K-0385 and N00014-87-K-0533.

REFERENCES

1. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers*, Addison-Wesley, 1986.
2. F. E. Allen, 'Control flow analysis', *ACM SIGPLAN Notices*, **5**, (7), 1–19 (1970)
3. K. Kennedy, 'A survey of data flow analysis techniques', in S. S. Muchnick and N. D. Jones (eds), *Program Flow Analysis*, Prentice-Hall, New Jersey, 1981, pp. 1–54, chap. 1.
4. J. Cocke, 'Global common subexpression elimination', *ACM SIGPLAN Notices*, **5**, (7), 20–24 (1970).
5. J. R. Allen, 'Dependence analysis for subscripted variables and its application to program transformations', *Ph.D. dissertation*, Rice University, April 1983.
6. M. J. Wolfe, 'Optimizing supercompilers for supercomputers', *Ph.D. dissertation*, University of Illinois at Urbana-Champaign, October 1982.
7. L. Lamport, 'The parallel execution of DO loops', *Comm. ACM*, **17**, (2), 83–93 (1974).
8. D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure and M. Wolfe, 'Dependence graphs and compiler optimizations', *Conf. Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, ACM, Williamsburg, January 1981, pp. 207–218.
9. U. Banerjee, 'Data dependence in ordinary programs', *Tech. report UIUCDCS-R-76-837*, University of Illinois at Urbana-Champaign, Department of Computer Science, November 1976.
10. D. A. Padua, D. A. Kuck and D. H. Lawrie, 'High-speed multiprocessors and compilation techniques', *IEEE Trans. Computers*, **C-29**, 763–776 (1980).
11. D. Kuck, R. Kuhn, B. Leasure and M. Wolfe, 'The structure of an advanced vectorizer for pipelined processors', *Proc. IEEE 4th International COMPSAC*, IEEE, Chicago, 1980, pp. 709–715.
12. J. R. Allen and K. Kennedy, 'Automatic loop interchange', *Proc. ACM SIGPLAN '84 Symposium on Compiler Construction*, ACM SIGPLAN, Montreal, June 1984, pp. 233–246.
13. R. G. Scarborough and H. G. Kolsky, 'A vectorizing FORTRAN compiler', *IBM J. Res. Develop.*, **30**, (2), 163–171 (1986).
14. Randy Allen, 'Unifying vectorization, parallelization, and optimization: the Ardent compiler', *Proc. Third Int. Conf. on Supercomputing*, International Supercomputing Institute, Inc., 1988, pp. 176–185 (Vol. II).
15. J. A. Fisher, J. R. Ellis, J. C. Ruttenberg and A. Nicolau, 'Parallel processing: a smart compiler &

a dumb machine', *Proc. ACM SIGPLAN '84 Symposium on Compiler Construction*, ACM, Montreal, June 1984, pp.37–47.
16. K. Kennedy, 'A global flow analysis algorithm', *International Journal of Computer Mathematics*, **3**, 5–15 (1971).
17. M. S. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland, Programming Languages Series, 1977.
18. J. D. Ullman, 'Fast algorithms for the elimination of common subexpressions', *Acta Informatica*, **2**, 191–213 (1973).
19. R. E. Tarjan, 'Solving path problems on directed graphs', *Tech. Report CS-528*, Stanford University, October 1975.
20. S. Graham and M. Wegman, 'A fast and usually linear algorithm for global flow analysis', *J. ACM*, **23**, (1), 172–202 (1976).
21. F. Allen and J. Cocke, 'A program data flow analysis procedure', *Comm. ACM*, **19**, (3), 137–147 (1976).
22. R. Triolet, F. Irigoin and P. Feautrier, 'Direct parallelization of call statements', *Proc. ACM SIGPLAN '86 Symposium on Compiler Construction*, ACM SIGPLAN, Palo Alto, June 1986, pp. 176–186.
23. D. Callahan and K. Kennedy, 'Analysis of inter-procedural side effects in a parallel programming environment', *Proc. First Int. Conf. on Supercomputing*, Springer, Athens, Greece, June 1987, pp. 138–171.
24. M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. S. Lam, O. Menzilcioglu and J. A. Webb, 'The Warp machine: architecture, implementation and performance', *IEEE Trans. Computers*, **C-36**, (12), 1523–1538 (1987).
25. T. Gross and M. Lam, 'Compilation for a high-performance systolic array', *Proc. ACM SIGPLAN '86 Symposium on Compiler Construction*, ACM SIGPLAN, Palo Alto, June 1986, pp. 27–38.
26. D. A. Padua and M. Wolfe, 'Advanced compiler optimizations for supercomputers', *Comm. ACM*, **29**, (12), 1184–1201 (1986).
27. U. Banerjee, S. Chen, D. Kuck and R. Towle, 'Time and parallel processor bounds for FORTRAN-like loops', *IEEE Trans. Computers*, **C-28**, (9), 660–670 (1979).
28. R. Allen and K. Kennedy, 'Automatic translation of FORTRAN programs to vector form', *ACM Trans. Programming Languages and Systems*, **9**, (4), 491–542 (1987).
29. R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth and P. K. Rodman, 'A VLIW architecture for a trace scheduling compiler', *IEEE Trans. Computers*, **C-37**, (8), 967–979 (1988).
30. M. Annaratone, F. Bitz, E. Clune, H. T. Kung, P. Maulik, H. Ribas, P. Tseng and J. Webb, 'Applications and algorithm partitioning on Warp', *Proc. Compcon Spring 87*, IEEE Computer Society, San Francisco, February 1987, pp. 272–275.
31. M. S. Lam, 'A systolic array optimizing compiler', *Ph.D. dissertation*, Carnegie Mellon University, May 1987.
32. M. Lam, 'Software pipelining: an effective scheduling technique for VLIW machines', *Proc. ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation*, June 1988, pp. 318–328.
33. S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski and J. Webb, 'iWarp: An integrated solution to high-speed parallel computing', *Proceedings Supercomputing '88*, IEEE Computer Society and ACM SIGARCH, Orlando, Florida, November 1988, pp. 330–339.
34. R. Cohn, T. Gross, M. Lam and P. S. Tseng, 'Architecture and compiler tradeoffs for a wide instruction word microprocessor', *Proc. Third. Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOSIII)*, ACM/IEEE, Boston, April 1989, pp. 2–14.
35. H. Tamura, S. Sakane, F. Tomita, N. Yokoya, K. Sakaue and N. Kaneko, Joint System Development Corp., *SPIDER Users' Manual*, Tokyo, 1983.
36. T. Kanade and J. Webb, 'End of year report for parallel vision algorithm design & implementation', *Tech. Report*, CMU, 1987.