A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimizations†

Evelyn Duesterwald Rajiv Gupta Mary Lou Soffa

Department of Computer Science
University of Pittsburgh, Pittsburgh, PA 15260
duester@cs.pitt.edu

Abstract

Data flow analysis techniques have traditionally been restricted to the analysis of scalar variables. This restriction, however, imposes a limitation on the kinds of optimizations that can be performed in loops containing array references. We present a data flow framework for array reference analysis that provides the information needed in various optimizations targeted at sequential or fine-grained parallel architectures. The framework extends the traditional scalar framework by incorporating iteration distance values into the analysis to qualify the computed data flow solution during the fixed point iteration. Analyses phrased in this framework are capable of discovering recurrent access patterns among array references that evolve during the execution of a loop. The framework is practical in that the fixed point solution requires at most three passes over the body of structured loops. Applications of our framework are discussed for register allocation, load/store optimizations, and controlled loop unrolling.

1. Introduction

Traditional program analysis and optimization techniques are typically restricted to scalar variables. Interest in the analysis of array references is growing with the recognition of its importance in detecting loop-level parallelism. The development of parallel architectures motivated the study of data dependence tests for array references to fully exploit the available parallelism [17, 25, 26]. These tests are designed to disambiguate array references, that is, to determine whether two array references may refer to the same memory location. However, sequential or fine-grained parallel architectures (e.g., pipelined, superscalar, or VLIW architectures) do not always benefit from this kind of dependence information and the complex code transformations used in parallelization. Instead, these architectures offer their own class of loop optimizations and analysis requirements.

Recurrent accesses to subscripted data are a major source in optimizing the execution of loops. For example, an important optimization of recurrent access patterns among the subscripted references in a loop is the avoidance of redundant memory accesses through appropriate register allocation strategies. When dealing with scientific programs that spend most of their time executing loops, reducing the overhead in memory accesses contributed by array references can greatly improve the overall program performance.

The information needed to support these fine-grained optimizations of individual array references differs from data dependence information in two ways. First, flow-sensitive information is needed. A potential overlap in the referenced *memory locations* as determined by conventional data dependence tests, does by itself not imply a *flow of values* between the involved references. However, optimizations such as register allocation are based on information about reuse of previously accessed values as opposed to information about re-accessed memory locations. To determine if two references that access the same memory location actually reference the same value mandates the inclusion of control flow into the analysis. Conditional control flow is often ignored when testing for data dependencies in a parallelizing compiler, as a potential (loop-carried) dependence between two arbitrary memory references may prevent parallelization. Hence, conventional data dependence information is inadequate for fine-grained optimizations.

Second, dependence information can only be exploited for fine-grained optimizations if the precise identity of the instances that are involved in the dependence is known. For example, in order to assign registers to array elements, the live range of an array reference must be determined, which includes the precise number of iterations that the computed value is live. Thus, the analysis we need should be focused on the dependencies among array references that are separated by constant iteration distances, that is, on a class of dependencies whose detection requires less complex analysis methods.

This paper presents a data flow framework that addresses these two requirements to systematically exploit fine-grained optimization opportunities for array references. The framework extends the traditional scalar framework [16] to the analysis of subscripted variables to meet the requirement of flow-sensitivity which includes the proper handling handling conditional control flow. The detection of recurrent access patterns among subscripted references that evolve during the execution of a loop is modeled in the framework as a fixed point computation. The solution to classical scalar data flow problems, such as reaching definitions, is adapted to subscripted references by qualifying the solution with iteration distance values. Specifically, the computed fixed point at a program point denotes the maximal iteration distance for which the respective data flow information holds at that point. Analyses phrased in this framework are practical in that at most three passes over a structured loop are required to find the fixed point solution. This efficiency makes the analyses attractive for use in optimizing compilers for sequential or fine-grained parallel architectures.

The optimizations supported by the framework extend traditional scalar optimizations to individual array elements. We demonstrate three classes of applications. First, we show how

† Partially supported by National Science Foundation Presidential Young Investigator Award CCR-9157371 and Grant CCR-9109089 to the University of Pittsburgh.

the computed information can be used to extend register allocation techniques to subscripted variables. We outline our technique for register pipelining [9] that presents an integrated approach to register allocation for both scalar and subscripted variables. The second class of applications is simple load/store related optimizations of array references. Finally, we show how the framework is used to efficiently provide the information needed to control loop unrolling, a loop transformation aimed at uncovering fine-grained parallelism across loop iterations.

In our analysis, we consider Fortran-like Do-loops, that may contain conditionals or nested loops, but no arbitrary gotos†. A loop is controlled by a basic induction variable, i, and we assume that no statement in the loop contains an assignment to i. Furthermore, we assume that prior to the analysis, non-basic induction variables have been identified and removed [1] and that all loops are normalized, i.e., the induction variable ranges from 1 to an upper bound UB with increment one. The array references in a loop are restricted to references whose subscripts are affine functions of the loop's induction variable, i.e., references of the form $X[f(i)]$, where $f(i) = a \times i + b$ with induction variable i and integer constants a and b.

Section 2 introduces the notion of data flow information for array references. The data flow framework to compute this information is presented in Section 3. The analysis is described in detail for one-dimensional arrays. Multi-dimensional array references are discussed in Section 3.6. Section 4 discusses applications of the framework. Related work and conclusions are given in Section 5 and Section 6.

2. Data Flow of Array References

A data flow analysis of array references in a loop provides the solution to a specific data flow problem, that is, to a specific question about the flow of array element values inside the loop. To express the access patterns of array references the computed solution is qualified with iteration distance values to denote that the solution at a point holds only up to a maximal iteration distance.

As an example, consider the computation of reaching definitions. In contrast to scalars, a single textual definition of a subscripted variable represents definitions of an entire range of memory locations. Reaching definition analysis for subscripted variables determines at each point in the loop the range of previous definition instances that reach that point (i.e, are not redefined prior to the point). Thus, instead of the binary information "reaching" and "not reaching" that is computed in scalar reaching definition analysis, array reference analysis requires more refined information. Specifically, the information "reaching" is qualified with respect to a maximal iteration distance δ.

> A definition d reaches a point p with iteration distance δ if the latest δ instances of d reach p.

The iteration distance δ is *maximal* if δ is the largest distance value for which definition d reaches point p. The upper bound for the maximal iteration distance in a loop with UB iterations is $UB-1$ denoting the complete range of iteration instances. Instances of a definition may not reach a point if there was a prior redefinition of the same array element. For example, the definition $C[i]$ in statement (3) in Fig. 1 redefines the instance of definition $C[i+2]$ from two iterations earlier. Thus, definition $C[i]$ *kills* all instances of $C[i+2]$ with an iteration distance greater than one. As a consequence, only the instances of $C[i+2]$ with a distance up to 1 reach the exit of statement (3) and in turn the entry of statement (4). Thus, the reference $C[i+1]$ in statement (4) is always a use of the value defined by statement (1) one iteration earlier.

do i=1,UB
(1) C[i+2] := C[i] $\times$ *2;*
(2) B[2 $\times$ *i] := C[i]+X;*
(3) if C[i]=0 then C[i]:=B[i-1];
(4) B[i] := C[i+1];
enddo

Fig. 1. Subscripted references inside a loop

3. A Framework for Array Reference Analysis

Our array reference analysis operates on a *loop flow graph* $FG = (N, E)$ that represents the body of a loop. The nodes in N denote statements in the loop or *summary nodes*, and the edges in E are control flow edges. A node, *exit*, is added to the graph to explicitly represent the increment operation of the loop induction variable, i.e., *exit* contains the operation $i := i+1$ for induction variable i. The loop flow graphs are constructed in a hierarchical fashion, starting with the graphs for the innermost nested loops. A single loop is analyzed at a time. When moving to the analysis of array references that are contained in an enclosing loop, the nested loops are replaced by summary nodes. Thus, no loop flow graph has nested cyclic control flow at the time it is being analyzed.

The data flow framework defines a system of equations over the flow graph of a loop body that is solved by fixed point iteration. The fixed point at each node in the loop flow graph describes the maximal iteration distance for which the solution holds at that node. Informally, instances of subscripted references are propagated throughout the loop from points where they are *generated* until points are encountered that *kill* the instances. Which subscripted references act as generating or killing references (e.g., uses and/or definitions of subscripted references) depends on the specific data flow problem, and is thus a parameter of the framework.

We describe the analysis in a *meet* data flow framework [16], which most naturally models data flow problems that provide *must*-information, i.e., an underestimate of the actual information that holds. However, as in the scalar case, problems to provide *may*-information, i.e., an overestimate of the actual information, can also be formulated in the framework. The equation system in the framework defines for each subscripted reference in the loop a lattice value that denotes a maximal iteration distance. Thus, instead of the conventional binary lattice $\{\top, \bot\}$ used in the classical scalar data flow problems, a chain lattice L of iteration distance values, as shown in Fig. 2, is used. A lattice value $x \in L$ for a subscripted reference r denotes the range of the latest x instances of r.

The meet operator $\wedge$ in this lattice is the usual minimum operator *min* defined for integers, where $\forall x \in L$: $min(x, \bot) = \bot$ and $min(x, \top) = x$. We also use a dual *max* operator which corresponds to the maximum operator for integers: $\forall x \in L : max(x, \bot) = x$ and $max(x, \top) = \top$.

† Note that only the gotos that destroy the single-entry, single-exit property of loops cause problems in our analysis. Gotos that keep control inside the loop can be handled. However, backward gotos inside a loop affect the efficiency of the analysis. Nested cycles created by backward gotos result in additional passes over the loop during the fixed point iteration.

$\top = UB-1$ top = all instances

$\cdots$

1 } instances up to max. distance

0

$\perp$ bottom = no instance

Fig. 2. Lattice L of maximal iteration distance values

3.1. The Flow Functions

To model a specific data flow question about the way the subscripted variables are referenced inside a loop, a pair of parameters (G, K) must be supplied. G is a set describing the subscripted references in a loop that *generate* instances and K is a set describing the references that *kill* instances. The subscripted references that occur in the loop are identified by integers. Thus, the sets G and K are sets of integers. We use the notation $G[n]$ and $K[n]$ to denote the generating and killing references that occur locally in node n.

The pair (G, K) uniquely determines a space of *flow functions* in the framework. Let $|G| = m$. Then a flow function $f_n : L^m \longrightarrow L^m$ that operates on tuples of iteration distance values is defined for each node n. The function f_n describes how the iteration distance values associated with the m references in G are affected when control passes through node n. For ease of presentation, we describe the definition of the flow functions using the example of determining the *must*-reaching definitions. The generalization to other must-problems is immediate.

Determining must-reaching definitions is the problem of computing the instances of subscripted variable definitions that reach a node along all paths in the loop flow graph. In this problem the set $G[n]$ contains the definitions of subscripted variables that occur in node n. Each definition may also act as a killing reference. Hence, the set $K[n]$ also consists of the definitions in node n. The flow function $f_n : L^m \longrightarrow L^m$ operates on tuples $(x_1, \ldots, x_m)$ of maximal iteration distance values. For each definition d, f_n determines a lattice element $x_d \in L$ denoting the range of previous instances of d that must reach the exit of node n. All flow functions operate independently on each component in L^m, i.e., there are functions $f_n^1, \ldots, f_n^m$ such that $\forall\ (x_1, \ldots, x_m) \in L^m$:

$$f_n(x_1, \ldots, x_m) = (\, f_n^1(x_1), \ldots, f_n^m(x_m)\,).$$

We define the component $f_n^d : L \longrightarrow L$ with respect to a single definition d. The definition of f_n for m definitions results as the Cartesian product over the m components. There are two types of flow functions:

$$f_n^d(x) = max(x, 0) \quad \text{or} \quad f_n^d(x) = min(x, p_n^d),$$

where p_n^d is a constant in L.

3.1.1. Generate Functions

If $d \in G[n]$ then node n generates definition d with distance 0. The flow function f_n^d is of the first type, i.e., $f_n^d(x) = max(x, 0)$, and is called a *generate function*.

3.1.2. Preserve Functions

Functions of the second type, i.e., $f_n^d(x) = min(x, p_n^d)$, are called *preserve functions*. The constant p_n^d denotes the maximal iteration instance of definition d that is *preserved*, i.e., not killed, by an element in $K[n]$. Let definition d be of the form $d = X[\,f_1(i)]$.

(i) If node n contains no definition to array X, then $p_n^d = \top$, i.e., all instances of d are preserved. In this case, the preserve function f_n^d is simply the identity, i.e., $f_n^d(x) = x$.

(ii) If node n contains a definition $d' = X[\,f_2(i)] \in K[n]$, a safe underestimate of the maximal number of previous iteration instances of d that are not redefined by d' must be determined. The range of previous iteration instances of $d = X[f_1(i)]$ with respect to node n is described by $X[f_1(i - \delta)]$, where $\delta \geq 1$. If definition d occurs in a predecessor node of n in the loop body, the instance of d with distance 0 is also included in range of previous instances, i.e., in this case $\delta \geq 0$. To uniformly model these two cases we define a predicate pr:

$$pr(d,n) = \begin{cases} 0 & \text{if reference } d \text{ occurs in a predecessor node of } n \\ 1 & \text{otherwise} \end{cases}$$

Let I denote the iteration range $[1, \ldots, UB]$ of induction variable i, then

$$p_n^d = \max\{\delta < UB \mid \forall\ i \in I,\ \ \forall \delta' \ni pr(d,n) \leq \delta' \leq \delta: \ f_2(i) \neq f_1(i - \delta')\ \}$$

Let the subscripts f_1 and f_2 be $f_1(i) = a_1 \times i + b_1$ and $f_2(i) = a_2 \times i + b_2$ and let

$$k(i) = \frac{a_1 - a_2}{a_1} \times i + \frac{b_1 - b_2}{a_1}$$

The equation can then be rewritten as:

$$p_n^d = \max\{\delta < UB \mid \forall\ i \in I, \forall\ \delta' \ni pr(d,n) \leq \delta' \leq \delta:\ \delta' \neq k(i)\ \}.$$

To determine the constant p_n^d we distinguish three cases for function k.

- If k is the constant $pr(d,n)$ (i.e., $\forall\ i \in I: k(i) = pr(d,n)$), then every definition generated by d is killed by d'. In this case: $p_n^d = \perp$. Examples are textually identical references for $pr(d,n) = 0$.
- If k is below $pr(d,n)$ in the iteration range I (i.e., $\forall i \in I: k(i) < pr(d,n)$), then there are no instances of definition d' that redefine a previous instance of definition d. Hence, $p_n^d = UB - 1 = \top$. An example of this case is $d = X[i]$ and $d' = X[i+2]$.
- Otherwise, k is a function that has positive values above $pr(d,n)$ in the iteration range I (i.e., $\exists i \in I: k(i) > pr(d,n)$). If there are positive integer values, then there are instances of d' that kill some earlier instances of d. An example is $d = X[\,2 \times i\,]$ and $d' = X[i]$. We obtain a safe approximation of p_n^d from the minimum (integer) value of k above $pr(d,n)$, i.e., $p_n^d = \lceil min\{\ k(i) \mid i \in I, k(i) > pr(d,n)\ \} \rceil - 1$. If k is a positive integer constant this approximation is accurate. In summary for case (ii) we obtain:

$$p_n^d = \begin{cases} \perp & \text{if } \forall\, i \in I: k(i) = pr(d,n) \\ \top & \text{if } \forall\, i \in I: k(i) < pr(d,n) \\ \lceil min\{\ k(i) \mid i \in I, k(i) > pr(d,n)\ \} \rceil - 1 & \text{otherwise} \end{cases}$$

3.1.3. Exit Function

Based on the constants p_n^d the flow functions for statement nodes are fully defined. Consider now the flow function for the additional node *exit* in the loop. Node *exit* represents the increment operation of induction variable i, which expresses the transfer to a higher iteration. Thus, propagating information across node *exit* reflects the propagation to future iterations. The increment in node *exit* affects the propagated lattice values as follows: An instance of a definition that reaches the exit of the loop body with distance δ reaches the entry of the next loop iteration with distance $\delta + 1$. Thus, the flow function f_{exit} increments the iteration distance value of each reference in preparation for the next iteration, i.e., $f^{d}_{exit}(x) = x{+}{+}$, where the increment operator ++ is defined as:

$$x{+}{+} = \begin{cases} \top & \text{if } x = \top \\ \bot & \text{if } x = \bot \\ x+1 & \text{otherwise} \end{cases}$$

For an example of the flow functions see Section 3.5 that illustrates the analysis for the loop in Fig. 1.

3.2. The Equation System

Based on the parameters (G, K) all flow functions are defined locally at each node for the m references in the loop. The global data flow solution is determined in a greatest fixed point computation that iteratively lowers lattice elements associated with each node until information stabilizes. The results are the assignments $IN[n] = (x_1, \ldots, x_m)$ and $OUT[n] = (y_1, \ldots, y_m)$ at each node n. $IN[n,d] = x_d$ describes the maximal iteration distance for which instances of definition d must reach the entry of node n. $OUT[n,d]$ describes the corresponding information at node exit.

The iteration is started with a safe initial guess, that is, one that overestimates the greatest fixed point. We overestimate the solution by assuming that $IN[n,d] = \top$ for each definition that is generated along the meet-over-all-paths in the loop body (i.e., along *all* paths in the loop body leading to n in a must-problem). The initial guess is obtained in the following initialization pass that visits the loop nodes in reverse postorder. Let $pred(n)$ denote the set predecessor nodes of node n. Then $\forall d \in \{1, \ldots, m\}$:

$$IN[n,d]^0 = \begin{cases} \bot & \text{if } n = 1 \text{ (loop entry)} \\ \bigwedge\limits_{m \in pred(n)} OUT[m,d]^0 & \text{otherwise} \end{cases}$$

$$OUT[n,d]^0 = \begin{cases} \top & \text{if } d \in G[n] \\ IN[n,d]^0 & \text{otherwise} \end{cases}$$

Based on the initial guess, the fixed point is computed iteratively over the following equation system. $\forall d \in \{1, \ldots, m\}$:

$$IN[n,d]^{i+1} = \bigwedge_{m \in pred(n)} OUT[m,d]^i$$

$$OUT[n,d]^i = f^d_n(IN[n,d]^i)$$

Note that the computed information describes the access patterns among the subscripted variables that evolve during the loop iterations. Thus, the information determined in the tuples $IN[n]$ and $OUT[n]$ holds after some initial start-up iterations. Specifically, let the lattice values for a definition d be x on entry to node n, i.e., $IN[n,d] = x$. Instances of d must reach the entry of node n up to iteration distance δ, where $pr(d,n) \le \delta \le x$, for all but the first δ iterations of the loop. Clearly, the number of required start-up iterations must be taken into account when applying optimizations to the subscripted references.

An important property of the equation system is that in addition to the initialization pass, only two passes through the loop body are needed to reach the fixed point, provided the nodes in the loop flow graph are visited during each pass in reverse postorder. This efficiency results from the fact that a loop flow graph contains no nested cycles and from properties of the flow functions. There are two types of statement node flow functions, $f(x) = min(x, c)$ and $f(x) = max(x, 0)$ with constant $c \in L$, and the exit function is $f_{exit}(x) = x{+}{+}$. As can easily be shown, the statement node flow functions are idempotent, i.e., $f \cdot f = f$ and f_{exit} is weakly idempotent, i.e., $f_{exit} \cdot f_{exit} \ge f$. Weak idempotence implies that one traversal around a cycle is sufficient to compute the contribution of the cycle [18]. Thus, in addition to the initialization pass, two passes over the loop body are sufficient for the fixed point convergence. Let N denote the number of statements in the loop. The fixed point is reached in $3 \times N$ nodes visits and the space required is $O(N^2)$ to maintain the sets IN and OUT at each node.

The overall analysis of a program is performed hierarchically starting with the innermost nested loops and working towards the outermost loops and the main program. The analysis provides information of the evolving access patterns that arise in the currently analyzed loop. If induction variables of enclosing loops occur in subscript expressions, they are treated as symbolic constants. After the analysis of a loop l_1 with induction variable i_1 is completed, the references in the enclosing loop l_2 with induction variable i_2 are analyzed. The nested loop l_1 is incorporated into the analysis of l_2 in the form of a summary node that replaces l_1 in the loop flow graph of l_2. During the analysis of l_2, a summary node is treated as any other node. Thus, sets $G[l_1]$ and $K[l_1]$ are determined for the loop l_1 and a corresponding flow function f_{l_1} is defined. The set $G[l_1]$ contains only references in l_1 whose subscripts are functions of the outer induction variable i_2, i.e., references of the form $X[a \times i_2 + b]$. Other references in l_1 do not act as generating references in the outer loop. However, all references in l_1 may act as killing references. We conservatively assume that a reference $X[a_1 \times i_1 + b_1]$ in the inner loop kills all instances of a reference $X[a_2 \times i_2 + b_2]$ in an enclosing loop. We are currently investigating how knowledge about the iteration bounds of the inner induction variable i_1 can be incorporated to provide more accurate killing information in an enclosing loop.

3.3. May-Information

The meet framework as defined in the previous section naturally models *must-problems*, i.e., all-paths problems. To formulate may-problems, i.e., any-path problems, such as conventional reaching definition analysis or live variable analysis, some modifications are needed in both the meet lattice L and the space of flow functions.

The lattice for a may-problem is simply the reverse of the lattice L, i.e., $\top$ denotes "no instance" and $\bot$ denotes "all instances". The meet in the reverse lattice is the dual operator *max*, i.e., $\wedge = max$.

Changes in the flow functions involve the definition of the constant p^d_n in a preserve function $f^d_n(x) = min(x, p^d_n)$. Unlike a must-problem where the constant p^d_n describes an underestimate of the actual information, p^d_n describes an overestimate in a may-problem, so as to ensure that no information is missed. Thus, p^d_n denotes the instances of d that *may* be preserved by elements in $K[n]$. Unless there is a definite kill that regularly kills instances of d in each iteration, we assume that all instances of d are preserved. Let $d = X[f(i)]$ and let d' be a reference in $K[n]$. A definite kill occurs if d' is of the form $d' = X[f(i) + c]$. In this case, only instances of d up to distance $c-1$ are preserved by d'. Let the function k be defined as in Section 3.1.2, then p^d_n is determined as:

$$p_n^d = \begin{cases} \top & \text{if } \forall\, i \in I:\; k(i) = pr(d,n) \\ c-1 & \text{if } \forall\, i \in I:\; k(i) = c,\, c > pr(d,n) \\ \perp & \text{otherwise} \end{cases}$$

A further modification in a may-problem concerns the estimate of an initial guess for the fixed point iteration. A safe initial guess for a may-problem would be $\top$ (no instance) for all definitions. However, when starting the iteration with this initialization, the fast fixed point convergence can no longer be guaranteed. The reason is that the function f_{exit} is no longer weakly idempotent, i.e., $f_{exit} \cdot f_{exit} \le f_{exit}$ in the reverse lattice. Starting with an initial value $\top$ may require $UB-1$ iterations to obtain the lattice value $\perp$ for a definition. As the bound UB may not be known, the iteration could continue infinitely.

Fortunately, it is possible to predict the increment effect of function f_{exit}. The initial guess in a may-problem predicts the maximal effect of the increment function, i.e., all lattice values are initialized to $\perp$[†]. It can easily be shown that the composition of any flow function with f_{exit} is idempotent with respect to the initial guess $\perp$, i.e., $(f_{exit} \cdot f) \cdot (f_{exit} \cdot f)(\perp) = (f_{exit} \cdot f)(\perp)$, which implies that information converges within two passes over the loop body. Note, that although the initialization is lower in the lattice than the fixed point, no information is lost since the preserve functions will eventually raise the initial lattice value "all instances" to the greatest fixed point. An initialization pass is not required in a may-problem and the greatest fixed point is computed in $2 \times N$ node visits.

3.4. Backward Problems

The framework models a forward flow of information. That is, information is propagated from control predecessors to control successors and from earlier iterations to later iterations. In a backward problem, such as live variable analysis, the flow of information is in the opposite direction. As usual, to phrase the backward flow from control successors to control predecessors in a forward framework, the reverse graph of the loop flow graph is used. In addition, to model the backward flow from later to earlier iterations, the role of negative and positive iteration distances is interchanged. This interchange only involves the definition of preserve functions and is modeled by redefining the function k used to determine the maximal distance of preserved instances as:

$$k(i) = \frac{a_2 - a_1}{a_1} \times i + \frac{b_2 - b_1}{a_1}$$

3.5. An Example: Must-Reaching Definitions

We illustrate the framework by an instance to compute the must-reaching definitions. Must-reaching definitions provide information about the guaranteed use of previously computed values, which can, for example, be exploited by register allocation techniques.

> A definition d at a point p must reach point p' with maximal iteration distance δ if d reaches p' along all paths starting at p that extend up to δ iterations.

As described earlier, the framework parameters $G[n]$ and $K[n]$ both contain the definitions of subscripted variables that occur in node n. The solution $IN[n,d] = x$ denotes that definition d must reach node n with iteration distance δ for $pr(d,n) \le \delta \le x$.

† The increment effect of function f_{exit} can be formally modeled using a *widening* operation [6].

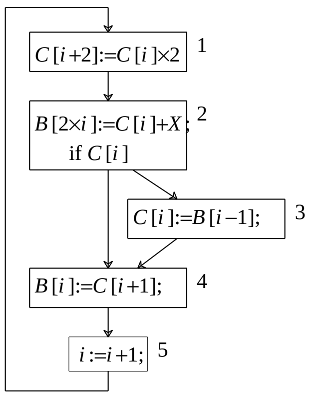

Fig. 3. Loop flow graph for the loop in Fig. 1.

The data flow analysis for must-reaching definitions is illustrated for the loop in Fig. 1 whose flow graph is depicted in Fig. 3. Each subscripted definition is numbered by the containing node, i.e., {1: $C[i+2]$, 2: $B[2\times i]$, 3: $C[i]$, 4: $B[i]$}. Thus, we obtain $G[j] = K[j] = \{\, j \,\}$. In the five node flow functions that follows the component x_i denotes the lattice element for definition i according to the above numbering.

$$f_1(x_1, x_2, x_3, x_4) = (\, max(x_1, 0),\ x_2,\ x_3,\ x_4\,)$$
$$f_2(x_1, x_2, x_3, x_4) = (\, x_1,\ max(x_2, 0),\ x_3,\ x_4\,)$$
$$f_3(x_1, x_2, x_3, x_4) = (\, min(x_1, 1),\ x_2,\ max(x_3, 0),\ x_4\,)$$
$$f_4(x_1, x_2, x_3, x_4) = (\, x_1,\ min(x_2, 0),\ x_3,\ max(x_4, 0)\,)$$
$$f_5(x_1, x_2, x_3, x_4) = (\, x_1{+}{+},\ x_2{+}{+},\ x_3{+}{+},\ x_4{+}{+}\,)$$

tuples	initialization pass (C[i+2],B[2×i],C[i],B[i])
$IN[1]$	$(\perp,\perp,\perp,\perp)$
$OUT[1]$	$(\top,\perp,\perp,\perp)$
$IN[2]$	$(\top,\perp,\perp,\perp)$
$OUT[2]$	$(\top,\top,\perp,\perp)$
$IN[3]$	$(\top,\top,\perp,\perp)$
$OUT[3]$	$(\top,\top,\top,\perp)$
$IN[4]$	$(\top,\top,\perp,\perp)$
$OUT[4]$	$(\top,\top,\perp,\top)$
$IN[5]$	$(\top,\top,\perp,\top)$
$OUT[5]$	$(\top,\top,\perp,\top)$

(i) Initialization pass.

tuples	1. pass (C[i+2],B[2×i],C[i],B[i])	2.pass (C[i+2],B[2×i],C[i],B[i])
$IN[1]$	$(\top,\top,\perp,\top)$	$(2, 1, \perp, \top)$
$OUT[1]$	$(\top,\top,\perp,\top)$	$(2, 1, \perp, \top)$
$IN[2]$	$(\top,\top,\perp,\top)$	$(2, 1, \perp, \top)$
$OUT[2]$	$(\top,\top,\perp,\top)$	$(2, 1, \perp, \top)$
$IN[3]$	$(\top,\top,\perp,\top)$	$(2, 1, \perp, \top)$
$OUT[3]$	$(1, \top, 0, \top)$	$(1, 1, 0, \top)$
$IN[4]$	$(1, \top, \perp, \top)$	$(1, 1, \perp, \top)$
$OUT[4]$	$(1, 0, \perp, \top)$	$(1, 0, \perp, \top)$
$IN[5]$	$(1, 0, \perp, \top)$	$(1, 0, \perp, \top)$
$OUT[5]$	$(2, 1, \perp, \top)$	$(2, 1, \perp, \top)$

(ii) Iteration over two passes.

Table 1. Data flow tuples for must-reaching definitions in Fig. 3.

The data flow computation is illustrated in Table 1 (i) showing the initialization pass and in Table 1 (ii) showing the iteration over two passes that visit the loop nodes in reverse postorder. To determine the guaranteed use of previously computed values based on the results of must-reaching definition analysis, the tuples $IN[n]$ are inspected. Let u be a use of a subscripted variables at node n, i.e., $u = X[f(i)]$. Use u has a must-reaching definition in the loop with iteration distance δ if the loop contains a definition $d = X[f(i-\delta)]$ and $pr(d,n) \le \delta \le IN[n,d]$, i.e., δ is within the range of previous instances of d that must reach node n.

Consider the computed must-reaching definitions for the loop in Fig. 3. The uses of $C[i]$ in nodes 1 and 2 reuse the value computed by definition $C[i+2]$ two iterations earlier since $C[i+2]$ must reach these nodes with iteration distance 2. The solution at node 3 indicates that all previous instances of definition $B[i]$ must reach node 3. Hence, the reference $B[i-1]$ uses the value computed in node 4 one iteration earlier. Finally, the definition $C[i+2]$ must reach node 4 with distance 1, implying that the reference to $C[i+1]$ uses the value computed by $C[i+2]$ one iteration earlier.

3.6. Multi-Dimensional Arrays

The framework models the analysis of recurrent access patterns that arise with respect to one enclosing loop l_i with induction variable i. Multi-dimensional array references in l_i are of the form $X[f_1(i), \ldots, f_n(i)]$, where the subscript $f_k(i)$ in each dimension k is an affine function of induction variable i. Induction variables of other enclosing loops that occur in $f_k(i)$ act as symbolic constants during the analysis of l_1. Multi-dimensional references are incorporated into the analysis by linearizing them to a single-dimensional reference $X[f(i)]$. The presence of symbolic constants in the linearized subscript $f(i)$ may complicate the determination of the preserve functions. To prevent overly conservative information, the loop bounds that restrict the range of values for these symbolic constants may be incorporated into the computation of the preserve functions.

Consider for example the recurrence among the references in statement (1) in Fig. 4. The linearized references are of the form $X[N \times i + (N+j)]$ and $X[N \times i + j]$, where N is the size of the first dimension. By symbolically evaluating the distance between the two references during reaching definition analysis of loop l_i, a reuse with distance 1 can be discovered. To discover recurrences that are due to an outer induction variable, a separate analysis of the loop body is performed with respect to the corresponding outer loop. For example, the recurrence among the references in statement (2) is determined during an analysis of loop l_j. During the analysis of l_j, induction variable i acts as a constant, i.e., the linearized subscripts are interpreted as functions of j: $Y[j + (N \times i+1)]$ and $Y[j + (N \times i-1)]$.

```
lj: do j:=1,UBj
    li: do i:=1,UBi
        (1)  X[i+1,j]:=X[i,j];
        (2)  Y[i,j+1]:=Y[i,j-1];
        (3)  Z[i+1,j]:=Z[i,j-1];
    enddo
enddo
```

Fig. 4. Multi-dimensional array references

A separate analysis of each enclosing loop in a tight loop nest is capable of discovering recurrent access patterns that are due to a single induction variable. However, a recurrence that arises with respect to multiple induction variables simultaneously, as the one among the two references in statement (3), cannot be determined. Discovering these recurrences requires a combined analysis of the loop nest. The distance information that is computed for each reference must be expanded to a vector of distance values, one for each induction variable of an enclosing loop. We are currently investigating how our framework can be extended in this way for the analysis of tight loop nests.

4. Applications

This section discusses applications of the array analysis framework that cover a number of optimization problems for array references.

4.1. Register Allocation for Subscripted Variables

Conventional register allocators typically ignore the potential benefits of keeping array elements in registers for reuse. This is in part due to the fact that standard data flow analysis techniques used in register allocation are not expressive enough to compute the live ranges of array elements. Consider, for example, the loop in Fig. 5 (i). Conventional compilers typically generate load and store instructions for each reference to an array element, as shown in the generated code in Fig. 5 (ii). However, the computed values can be preserved in registers for reuse, thereby avoiding memory load instructions, by allocating a *register pipeline*.

```
do i=1,1000
(1)  A[i+2]:=A[i]+X;
enddo
```

(i)

```
     load rX ← X
     rI ← 1
L1:  load r ← A(rI)
     r ← r+rX
     store A(rI+2) ← r
     rI ← rI+1
     if rI ≤4 1000 goto L1
```

(ii)

```
     load r1 ← A(2)
     load r2 ← A(1)
     load rX ← X
     rI ← 3
L2:  r0 ← r2+rX
     store A(rI) ← r0
     rI ← rI+1
     r2 ← r1
     r1 ← r0
     if rI ≤ 1000 goto L2
```

(iii)

Fig. 5. A sample loop (i), the conventional code generated (ii), and the improved version using a register pipeline *(r0, r1, r2)* for elements of array A (iii).

A register pipeline is a set of registers constituting the stages of the pipeline. A computed or loaded value enters the pipeline at the first stage and progresses through the pipeline, one stage per iteration. The code in Fig. 5 (iii) illustrates the use of a three-stage register pipeline *(r0, r1, r2)*. In each iteration a value computed in statement (1) enters the pipeline in stage r0 (instruction: *r0 ← r2+rX*) and at the end of each iteration the values currently in the pipeline progress one stage further. With proper initialization of the pipeline, the values of the array elements referenced by A[i] are always found in the third stage of the pipeline (i.e., in register *r2*) and memory load instructions inside the loop are entirely avoided.

The major challenge in allocating register pipelines is the construction of live ranges for subscripted variables, as the entities of the register assignment. Once live ranges for both the scalar and subscripted variables have been determined, it is possible to generalize traditional register assignment strategies [4, 5] to effectively include subscripted variables. We have developed such an integrated register assignment strategy based on a register allocation technique using priority-based coloring to enable a fair and uniform competition of both classes of variables for the available registers [9]. The overall allocation task for a loop is achieved in four phases:

(i) Live range analysis
(ii) Construction of the integrated register interference graph
(iii) Multi-coloring
(iv) Code generation

First, the loop is analyzed to determine the live ranges of the scalar and subscripted variables in the loop. The live ranges of both classes of variables are represented in a common graph: the integrated register interference graph. Based on this graph we generalize the conventional priority based coloring heuristic to a multi-coloring strategy. Finally, the code to implement register pipelines is generated.

We show how the developed framework is used for live range analysis, which presents an improvement over the analysis used in [9]. The remainder of this section overviews the remaining three phases. Details of the integrated allocation technique are found in [9].

4.1.1. Live Range Analysis

The live range of a variable value v starts at a point where v is generated (either be a definition or a use site) and extends up to the last use of v. Live ranges of scalar variables are determined using conventional methods [1]. To construct live ranges for array elements, the individual reuse points of a generated value must be identified. However, a generated value can only be preserved in a register for reuse if all instances of the generation site are available at the reuse point. This information is captured by an instance of our framework that computes a *must* problem: the δ-*available values* inside a loop. This problem is an extension of the classical scalar problem of computing available subexpression [1].

> A value v of an array element e generated at point p is δ-*available* at point p' if there is no redefinition of e along all paths leading to p' that start at p and extend up to δ iterations.

The problem of computing δ-available values is similar to the must-reaching definition analysis described in Section 3.5, except that in addition to the definition sites of subscripted variables, the use sites also act as generating references.

To specify the framework for δ-available values the parameters G and K are defined. $G[n]$ contains both the definitions and uses of subscripted variables that occur in node n and $K[n]$ contains only the definitions. Based on G and K the flow functions associated with nodes in the loop flow graph are determined as described in Section 3.1 and the fixed point is computed over the equation system IN and OUT. The solution $IN[n,r] = x$ at node n denotes that the value of the subscripted reference r is δ-available at node n for distance δ where $pr(r,n) \le \delta \le x$.

Live ranges for subscripted variables are constructed by inspecting each node that contains a use site for the δ-available values. Let $r = X[f(i)]$ be a reference that is δ-available on entry of node n. A use r' in node n *reuses* the value generated by reference r, if r' is of the form $r' = X[f(i - \delta')]$ and $pr(r,n) \le \delta' \le \delta$. Reference r' reuses the value generated by r with the constant iteration distance δ'. By collecting the individual reuse points in a live range in this fashion, the complete ranges inside the loop are determined.

4.1.2. Construction of the IRIG

Traditionally, the problem of assigning registers to live ranges is formulated as the problem of *k*-coloring the register interference graph [1], where k is the number of available registers. The nodes in this graph represent the live ranges of variables in the loop. Two nodes are connected by an edge, i.e., *interfere*, if the corresponding live ranges overlap and cannot be assigned the same register. Live ranges are assigned priorities that express the benefits of keeping the corresponding variables in registers. Registers are assigned to live ranges by coloring the corresponding nodes in the graph based on their priorities.

We extend the traditional structure of the register interference graph to represent live ranges of subscripted variables as well as scalar live ranges. The resulting graph is called the integrated register interference graph (IRIG). A priority function P is defined over the set of live ranges represented by the nodes in the IRIG. The priority of a live range l is based on the resource requirements of l, which are expressed by a second function $depth$. $Depth(l)$ is the depth of the register pipeline that is needed to preserve the generated values in l. Let $\delta_0(l)$ denote the iteration distance between the generation point and the latest reuse point in l, then

$$depth(l) = \begin{cases} 1 & \text{if } l \text{ is scalar} \\ \delta_0(l)+1 & \text{otherwise} \end{cases}$$

The priority $P(l)$ expresses a savings/costs ratio of allocating registers to live range l. The savings result from memory load instructions that are avoided, normalized with respect to the length of the live range, so as to favor live ranges that require the register resources for only short periods. Priorities are determined uniformly over the scalar and subscripted live ranges by calculating the utilization of each required register. Let C_{LD} be the average cost of executing a memory load instruction, $access(l)$ the number of reuse points in l, and let $|l|$ be the length of l. The priority $P(l)$ is computed as:

$$P(l) = \frac{[access(l) - 1] \times C_{LD}}{|l| \times depth(l)}$$

4.1.3. Multi-Coloring

Register pipelines are assigned to live ranges by multi-coloring the IRIG based on the calculated priorities. In the standard priority-based coloring strategy for scalars [5], the coloring of nodes that have fewer interferences (i.e., neighbors in the graph) than available registers is postponed knowing that these nodes can always be colored. These nodes are called the *unconstrained* nodes. A node that has more interferences than available registers is called a *constrained* node. The constrained nodes are split, creating two or more nodes with fewer neighbors. This process continues until there are no more constrained nodes in the graph and all nodes can be assigned a color.

To adapt this coloring strategy to the IRIG, it must be acknowledged that nodes in the IRIG that represent subscripted live ranges may require more than one register (i.e., more than one color). Specifically, to determine whether a node is an unconstrained node, the possibly varying register requirements of the neighbor nodes as well as the register requirements of the node itself must be taken into account. A node n in the IRIG is an unconstrained node if

$$depth(n) + \sum_{m \text{ a neighbor}} depth(m) \le k.$$

If the above inequality holds, there will always be $depth(n)$ colors to multi-color node n. The remainder of the coloring procedure is essentially unchanged and details are found in [9].

4.1.4. Code Generation

The use of a register pipeline involves three phases. First, the pipeline is initialized prior to entering the loop. Let l be a live range and let the generating reference of l be of the form $X[f(i)]$. The pipeline stages for l are initialized by generating for each stage r_j, where $j = 1, \ldots, depth(l)-1$, a load instruction: load $r_j \leftarrow X[f(1-j)]$.

The pipeline is used inside the loop by replacing each reuse point with a reference $X[f(i-\delta)]$ by an access to pipeline stage r_δ. A load may only be needed for the generating reference in l, i.e., if the generating reference is a use.

The final phase of using a pipeline consists of implementing the pipeline progression at the end of the loop body. A value located in stage r_j progresses to stage r_{j+1} for $1 \le j \le d-1$ for use in the next iteration. Note that physically moving values among the stages of the pipeline is not necessary if the loop is unrolled $depth(l)$ times. However, loop unrolling has the disadvantage of increasing the number of instruction in the loop body. If the unrolled loop body is too large for the instruction cache, more additional memory accesses may be introduced than saved by the register assignment.

One way to implement the pipeline progression at the end of each iteration of the original loop body is to generate $depth(l)-1$ register-to-register move instruction: $r_j \leftarrow r_{j-1}$. If these instructions cannot be executed concurrently with other operations in the loop, the pipeline progression has an overhead that is absent if register pipelines are not used. For large pipelines, this overhead may undo some of the savings gained by using the pipeline. This problem of potential overallocation can be avoided by incorporating the costs of the pipeline progression as a negative factor into the priority calculation.

Alternatively, hardware support may be exploited to perform the pipeline progression at constant costs. The Cydra 5 architecture [7] provides *iteration control pointer (ICP)* that is implicitly addressed by each register reference. By appropriately updating the ICP at the end of each iteration, this architecture can be used to implement the pipeline progression as a register windowing scheme. Another hardware solution provides a concurrent register-to-register move instruction, that performs the individual move instructions simultaneously provided the pipeline consists of a set of consecutive registers.

4.2. Load/Store Related Optimizations

The data flow framework supports the extension of traditional optimizations to subscripted variables. As an example, this section presents load/store optimizations in relation to subscripted references.

4.2.1. Eliminating Redundant Stores

A store (i.e., a definition site) of an array element is *redundant* if, on all paths, it is followed by another store to the same array element without an intermediate use of the stored value. Fig. 6 shows an example of a redundant store that arises across one iteration (i) and the transformed loop with the redundancy eliminated (ii).

The data flow problem of detecting redundant stores is a generalization of the scalar problem of determining very busy expressions [1] applied to definitions of subscripted variables. We refer to this problem as computing the δ-busy stores.

> A store s at point p is δ-*busy* at point p', if the store is executed without a preceding use of the stored array element along all paths leading to p' that start at p and extend up to δ iterations.

```
do i=1,1000
    A[i]:=...;   {store}
          {1-redund. store}
    if cond then A[i+1]:...;
enddo
```
(i)

```
do i=1,999
    A[i]:=...;
    if cond then ...;
enddo
A[1000]:=...;
if cond then A[1001]:=...;
```
(ii)

Fig. 6. Example of a 1-redundant store across one iteration (i) and the transformed loops with the redundancy eliminated (ii).

Like the problem of computing very busy expressions, determining δ-busy stores is a *must*-problem and a backward problem. The candidates for redundant stores are the subscript expressions that occur in definition sites, i.e., G is the set of textually distinct subscript expressions in definition sites. A busy store is killed when it meets a use of the same array element. Thus, the set K consists of the uses of subscripted variables in the loop. The fixed point solution to δ-busy stores determines at each node n for each store $s \in G$ a lattice value $IN[n,s] = x$ denoting that store s is δ-busy on exit of node n for $pr(s,n) \le \delta \le x$†.

The solution to δ-busy stores is used to detect opportunities for eliminating redundant stores. Let $s = X[f(i)]$ be a store at a node n. The store s at node n is δ-*redundant* (i.e., redundant with iteration distance δ), if there is another store $s' = X[f(i-\delta)]$ in the loop and s' is δ-busy at node n, i.e., $IN[n,s'] = \delta$. Thus, a δ-redundant store is followed by another store to the same array element δ iterations later. It follows that a δ-redundant store is redundant in all but the final δ loop iterations. Hence, the store can be eliminated from all but the final δ iterations. This is achieved by removing the store from the loop and by unpeeling the final δ loop iterations as shown in Fig. 6 (ii).

4.2.2. Eliminating Redundant Loads

A load of an array element e is redundant, if, on all paths, it is preceded by a statement that either loads or defines the same array element without an intermediate redefinition. The problem of detecting and eliminating redundant loads is a special case of the register allocation problem for subscripted variables described in Section 4.1. Thus, the same framework instance for computing the δ-available values is used. Using this information, redundant loads can be eliminated in a similar way as available subexpressions are eliminated for scalars [1]. Thus, for each redundancy a scalar temporary is created that can be held in a register to avoid memory accesses. Fig. 7 shows an example of a 1-redundant load and its removal.

```
do i=1,1000
        {1-redund. load}
    if cond then ...:=A[i];
    A[i+1]:=...; {store}
enddo
```
(i)

```
t:=A[1];
do I=1,1000
    if cond then ...:=t;
    A[i+1]:=...;
    t:=A[i+1];
enddo
```
(ii)

Fig. 7. Example of a 1-redundant load (i) and the transformed loops with the redundancy eliminated (ii).

† Recall, that, in a backward problem, IN denotes node exit information.

4.3. Controlled Loop Unrolling

When compiling a loop body for execution on a fine-grained parallel architecture, it may be beneficial to unroll the loop [8]. The original loop body may not have enough parallelism to exploit the available architecture. Sufficient parallelism could then be provided by the larger loop body that is obtained by unrolling. However, loop unrolling may also impact the code in a negative way. Loop unrolling leads to an increase in code size and to an increase in register pressure due to longer live ranges in the unrolled loop body. Moreover, loop unrolling should only be performed if more parallelism is actually created. There may be loop-carried dependencies in the original loop that become loop-independent in the unrolled loop, and thus may prevent an increase in parallelism. Ideally, the effects of loop unrolling are predicted in advance, so as to avoid this transformation if no sufficient improvements of the code can be achieved. Clearly, to be of use, such a prediction must be obtainable efficiently. We describe in this section, how the developed array analysis framework can be used to approximate the impact of loop unrolling on the parallelism in the loop body. A similar strategy may be used to predict the effect of loop unrolling on the register pressure in the loop.

To measure the amount of parallelism consider the length l of the critical path (i.e., the longest chain) in the dependence graph for the loop body. The length l_{unroll} of the critical path in the unrolled loop is $l \le l_{unroll} \le 2 \times l$, where $l_{unroll} = l$ if there are no loop-carried dependencies with distance 1, and $l_{unroll} = 2 \times l$ if there is a dependence from the last statement in the chain in one iteration to the first statement in the chain in the next iteration.

A strategy to control loop unrolling based on the advance calculation of the critical path length l_{unroll} was described for region scheduling [13]. In this strategy, unrolling is performed incrementally under the assumption that all relevant dependence information is available. During each step the length l_{unroll} is calculated from information about loop-carried dependencies with distance 1. Only if l_{unroll} is below a certain threshold value τ, where $l \le \tau < 2 \times l$ is unrolling actually performed. This process continues until either sufficient parallelism has been created or until no more usable parallelism is created by further unrolling the loop. For such a controlled loop unrolling strategy to be efficient, it is critical that information about loop-carried dependences with distance 1 can quickly be made available during each step. We briefly show how our framework can be used to compute this dependence information.

There are three types of loop-carried dependencies: flow dependencies, anti-dependencies, and output dependencies [17]. Flow and anti-dependencies can be detected based on reaching definition information. To include the detection of anti-dependencies the uses of subscripted variables are propagated in addition to the definitions. The data flow analysis for dependence detection computes the δ-reaching references.

> A reference r at point p is δ-*reaching* at point p' if there is a path from p to p' along which the array element referenced by r is not redefined.

The framework for δ-reaching references is specified by the set $G[n]$ containing the definitions and uses of subscripted variables, and by the set $K[n]$ that contains only the definitions in node n. Reaching reference analysis is a may-problem, i.e., potential dependencies are discovered. The fixed point solution $IN[n,r]$ describes at each node n and for each reference $r \in G$ the maximal iteration distance up to which reference r reaches the entry of node n.

Dependencies are determined by examining the computed reaching information at each node Let $r_1 = X[f_1(i)]$ be a reference in node n, and let $r_2 = X[f_2(i)]$ be another reference that reaches node n up to distance δ. A dependence from r_1 to r_2 exists if

$$\exists \delta' \ni 0 \le \delta' \le \delta \colon \ \exists i \in I \colon \ f_1(i) = f_2(i - \delta').$$

A conservative test of the above condition proceeds in a similar way as described in Section 3.1.2 for the definition of preserve functions. Let δ_0 be the smallest value for δ' for which the two references may overlap. A dependence from r_1 to r_2 with iteration distance δ_0 is determined. Instances of r_1 and r_2 with an iteration distance less than δ_0 are dependence-free. Finally, the type of dependence between the two references, i.e., flow, anti-, or output dependence, merely depends on whether r_1 and r_2 are definitions or uses of subscripted variables.

5. Related Work

Previous extensions to the scalar data flow framework to devise a flow-sensitive analysis of array references can be categorized into one of two approaches. The target of frameworks in the first approach is to improve the accuracy of conventional data dependence information and the frameworks operate on array regions. The target of the second approach, and the approach taken in this work, is to provide the data flow information needed in fine-grained optimizations of individual array references. Therefore, the framework operates on individual reference instances. Both approaches share the advantage that data flow frameworks can handle conditional control flow inside loops.

Techniques that follow the first approach include the frameworks described by Gross and Steenkiste [12], Granston and Veidenbaum [11], Rosene [24], and Hanxleden et al [14]. In these techniques, the array regions accessed by an individual array reference are locally approximated by some form of area descriptor. The locally determined summary information is globally propagated during an interval-based or iterative data flow computation. The accuracy as well as the costs of these techniques depends on the choice of representation for the accessed array region and the resulting complexity of the meet operation. Although it is in general possible to extract from the access summaries the instance information that is of interest for fine-grained optimizations, our framework shows that this information can be computed directly in a more efficient way. An array analysis framework that follows the second approach and operates on reference instances as opposed to access summaries was proposed by Rau [22]. This analysis propagates textual names of referenced array elements throughout the program. Unlike the analysis presented in this paper, the number of iterations over the program is in general unbounded and is thus, in practice, limited by a chosen upper bound resulting in a limited amount of information. In contrast to our analysis, this technique would fail to recognize recurrent access patterns in all-paths problems if they arise only after some initial start-up iterations of an enclosing loop.

Other work on the analysis of array references is targeted at improving the accuracy of conventional data dependence tests used in parallelization. By incorporating flow information as an additional constraint on the presence of a dependence, false dependence reports are avoided. These techniques provide more accurate data dependence information at the costs of additional complexity, which is acceptable in relation to the performance improvements of loop parallelization. A general framework for obtaining flow-sensitive data dependence information by incorporating sequencing information was presented by Feautrier [10]. Other work to obtain flow-sensitive data dependence information includes the techniques by Maydan et al [19], Pugh and Wonnacott [21], Brandes [2], Ribas [23], and by Kallis and Klappholz [15].

Some research specifically addresses the exploitation of reuse opportunities for individual array elements. Scalar replacement [3] improves register allocation for array elements by introducing new scalar temporaries for each dependence in a similar way as we have described for eliminating redundant loads. This method is based on conventional data dependence information and thus may miss reuse opportunities in the presence of conditional control flow. Register allocation for array elements across loop iterations in the context of software pipelined loops is described in [20].

6. Conclusions

In this paper we addressed the problem of efficiently and effectively analyzing array references to provide the information needed for various optimization problems targeted at sequential or fine-grained parallel architectures. A data flow framework was presented that extends the traditional scalar framework to analysis of subscripted variables by incorporating iteration distances. The framework models the detection of recurrent access patterns among subscripted references that arise during the execution of loops that may contain conditionals. Analyses phrased in this framework are practical in that they require only a small number of passes over a loop body. This efficiency makes the analyses attractive for use in optimizing compilers for sequential or fine-grained parallel architecture. Applications of the framework were demonstrated in various optimizations of individual array references.

The framework, as stated in this paper, models the analysis of a single loop as the primary source for fine-grained optimizations. We are currently investigating how the framework can be extended to the analysis of tight loop nests to provide information about recurrent access patterns that arise simultaneously with respect to multiple enclosing loops.

Acknowledgements

We thank the referees for their comments which helped in improving this paper.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman, in *Compilers, principles, techniques, and tools*, Addison-Wesley Publishing Company, Massachusetts, 1986.
2. T. Brandes, ‘‘The importance of direct dependences for automatic parallelism,’’ *Proc. of the ’88 Int. Conf. on Supercomputing*, pp. 407-424, 1988.
3. D. Callahan, S. Carr, and K. Kennedy, ‘‘Improving register allocation for subscripted variables,’’ *Proc. of the ACM SIGPLAN ’90 Conf. Programming Language Design and Implementation*, pp. 53-65, Jun. 1990.
4. G. J. Chaitin, ‘‘Register allocation and spilling via graph coloring,’’ *Proc. of the ACM SIGPLAN ‘82 Symp. on Compiler Construction*, pp. 201-207, Jun. 1982.
5. F. Chow and J. Hennessy, ‘‘Register allocation by priority-based coloring,’’ *ACM SIGPLAN Notices*, vol. 19, no. 6, pp. 222-232, 1984.
6. P. Cousot and R. Cousot, ‘‘Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,’’ *Conf. Record of the 4th Annual ACM Symp. on Principles of Programming Languages*, pp. 238-252, Jan. 1977.
7. J.C. Dehnert, P.Y.-T. Hsu, and J.P. Bratt, ‘‘Overlapped loop support in the Cydra 5,’’ *Proc. of the 3rd Int. Conf. on Architectural Support for Programming Languages and Operating Systems.*, pp. 26-39, Apr. 1989.
8. J.J. Dongarra and A.R. Hinds, ‘‘Unrolling loops in FORTRAN,’’ *Software-Practice and Experience*, vol. 9, no. 3, Mar. 1979.
9. E. Duesterwald, R. Gupta, and M.L. Soffa, ‘‘Register pipeling: An integrated approach to register allocation for scalar and subscripted variables,’’ *Proc. of the 4th Int. Workshop on Compiler Construction, LNCS 641 Springer Verlag*, pp. 192-206, Oct. 1992.
10. P. Feautrier, ‘‘Data flow analysis of array and scalar references,’’ *Int. Journal of Parallel Programming*, vol. 20, no. 1, 1991.
11. E. Granston and A. Veidenbaum, ‘‘Detecting redundant accesses to array data,’’ *Proc. of Supercomputing ’91*, pp. 854-865, Nov. 1991.
12. T. Gross and P. Steenkiste, ‘‘Structured dataflow analysis for arrays and its use in an optimizing compiler,’’ *Software - Practice and Experience*, vol. 20, no. 2, pp. 133-155, Feb. 1990.
13. R. Gupta and M.L. Soffa, ‘‘Region scheduling: an approach for detecting and redistributing parallelism,’’ *IEEE Trans. on Software Engineering*, vol. 16, no. 4, pp. 421-431, Apr. 1990.
14. R. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz, ‘‘Complier analysis for irregular problems in Fortran D,’’ *5th Workshop on Languages and Compilers for Parallel Computing*, pp. 65-77, Aug. 1992.
15. A. D. Kallis and D. Klappholz, ‘‘Reaching definitions analysis on code containing array references,’’ *Conf. Rec. of the 4th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1991.
16. J.B. Kam and J.D. Ullman, ‘‘Monotone data flow analysis frameworks,’’ *Acta Informatica*, vol. 7, no. 3, pp. 305-317, Jul. 1977.
17. D.J. Kuck, R.H. Kuhn, D. Padua, B.R, Leisure, and M. Wolfe, ‘‘Dependence graphs and compiler optimization,’’ *Proc. of the 8th ACM Symp. on Principles of Programming Languages*, pp. 207-218, Jan. 1981.
18. T.J. Marlowe and B.G. Ryder, ‘‘Properties of data flow frameworks, a unified model,’’ *Acta Informatica*, vol. 28, no. 2, pp. 121-163, Dec. 1990.
19. D.E. Maydan, S.P. Amarasinghe, and M.S. Lam, ‘‘Array data-flow analysis and its use in array privatization,’’ *Proc. of the 20th ACM Symp.on Principles of Programming Languages*, pp. 2-15, Jan. 1993.
20. Q. Ning and G.R. Gao, ‘‘A novel framework of register allocation for software pipelining,’’ *Proc. of the 20th ACM Symp. on Principles of Programming Languages*, pp. 29-42, Jan. 1993.
21. W. Pugh and D. Wonnacott, ‘‘Eliminating false data dependences using the omega test,’’ *Proc. of the SIGPLAN ’92 Conf. on Programming Language Design and Implementation*, pp. 140-151, Jun. 1992.
22. B. R. Rau, ‘‘Data flow and dependence analysis for instruction-level parallelism,’’ *Conf. Rec. of the 4th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1991.
23. H. Ribas, ‘‘Obtaining depedence vectors for nested-loop computations,’’ *Proc. of the 1990 Int. Conf. on Parallel Processing*, pp. 212-219 (II), 1990.
24. C. Rosene, ‘‘Incremental dependence analysis,’’ *Ph.D. thesis*, Rice University, March 1990.
25. M. Wolfe and U. Banerjee, ‘‘Data dependence and its application to parallel processing,’’ *Int. Journal of Parallel Programming*, vol. 16, no. 2, pp. 137-178, 1987.
26. M. Wolfe, ‘‘Optimizing supercompilers for supercomputers,’’ *Pitman Publishing Company, London, MIT Press*, Cambridge, Massachusets, 1989.