Register Pipelining: An Integrated Approach to Register Allocation for Scalar and Subscripted Variables[†]

Evelyn Duesterwald   Rajiv Gupta   Mary Lou Soffa

Department of Computer Science, University of Pittsburgh
Pittsburgh, PA 15260

Abstract. Conventional compilers typically ignore the potential benefits of keeping array elements in registers for reuse, due in part to the fact that standard data flow analysis techniques used in register allocation are not expressive enough to distinguish among individual array elements. This paper introduces the concept of *register pipelining* as an integrated approach to register allocation for both scalar and subscripted variables. A register pipeline is a set of registers that is allocated to the live ranges of array elements inside a loop. By preserving the computed array elements in the pipeline stages, reuse is enabled across loop iterations. We present an efficient data flow algorithm that extends the construction of live ranges to array elements. To enable a fair competition among the live ranges of subscripted and scalar variables for the available registers, we developed an integrated version of the standard graph coloring algorithm for register allocation.

1 Introduction

Variables are allocated to registers to enable fast reuse of computed values and avoid expensive memory accesses. Current compilers employ sophisticated register allocation strategies, such as allocation via graph coloring [6, 7] for holding scalar variables in registers. However, most compilers typically ignore reuse opportunities for references to array elements, partly because standard data flow analysis does not distinguish among individual array elements and thus treats the whole array as one scalar, and partly because of the potential of aliasing. Failing to keep array elements in registers may cause execution deficiencies when dealing with scientific applications. Scientific programs typically spend most of their time executing loops that process large amounts of data involving references to both scalar and subscripted variables. Exploiting reuse opportunities for these variables can significantly reduce the memory traffic and thus improve the overall performance of the program.

Consider, for example, the loop in Fig. 1 (i). Conventional compilers typically generate memory load and store instructions for each reference to an array element inside the loop. Thus, one load and one store instruction is executed per iteration of the loop as shown in the generated code in Fig. 1 (ii). However, statement 1 contains a recurrence and computes a value A[I] in each iteration that is used two iterations later. The computed values can be preserved in registers for reuse, thereby avoiding memory load instructions, by allocating a *register pipeline*. A register pipeline is a set of registers constituting the

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

```
Do I=3,1000                  load rX <- X                   load r1 <- A(2)
 (1) A[I]:=A[I-2]+X;         rI <- 3                        load r2 <- A(1)
 Enddo                   L1: r0 <- rI-2                     load rX <- X
                             load r0 <- A(r0)               rI <- 3
                             r0 <- r0+rX                L2: r0 <- r2+rX
                             store r0 -> A(rI)              store r0 -> A(rI)
                             rI <- rI+1                     rI <- rI+1
                             if rI<=1000 goto L1            r2 <- r1
                                                            r1 <- r0
                                                            if rI<=1000 goto L2

        (i)                        (ii)                           (iii)
```

Fig. 1. 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).

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. 1 (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 (see instruction: r0 <- r2+rX) and at the end of each iteration the values currently in the pipeline progress one stage further (see instructions: r2 <- r1 and r1 <- r0). With proper initialization of the pipeline, the values of the array elements referenced by A[I-2] 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.

There have been several approaches to optimize loop performance by allocating registers to recurrences [3,4,8,15]. The number of registers required to exploit reuse in a recurrence is determined from the recurrence degree (i.e., the number of iterations a computed value is live). Reuse is enabled only if the complete number of required registers can be allocated to the recurrence, i.e., a partial register assignment cannot be utilized. For this reason, previous techniques have processed recurrences as a new compiler optimization performed in a phase separate from standard scalar register allocation. However, in a typical loop both subscripted and scalar variables compete for the available registers. Consequently, separately handling array recurrences from scalar register allocation causes the dilemma of sacrificing the benefit of one allocation for the other. Ideally, register allocation should be performed in an integrated fashion such that both classes of variables can compete for the available registers in a fair and uniform manner.

We present in this paper the concept of *register pipelining* as a novel approach to register allocation that integrates the allocation problem for both subscripted and scalar variables. In order to guarantee a fair competition among scalar and subscripted variables for the available registers, we developed an integrated variation of the standard priority-based coloring strategy for register allocation [6,7].

Register pipelining introduces a number of new challenges. In order to allocate a register pipeline, the number of iterations that a computed values is live, i.e., the pipeline depth, must be determined. Constructing live ranges for array elements (*A-ranges*) is complicated due to the dynamic naming mechanism of subscripted references. Two array references with distinct subscripts inside a loop may access the same array element in different iterations. Thus, new techniques are needed that extend the construction of live ranges to array elements.

Data dependence theory [11,17,18] has intensively addressed the problem of determining whether two array references with distinct subscript expressions may denote the same array element, in which case they are said to be dependent. Determining data dependencies is crucial in parallelizing compilers, and a large body of powerful and expensive data dependence tests has been developed to determine whether two array references are *dependence-free*. In register allocation, however, we are interested in the opposite question; that is we are interested in detecting guaranteed reuses of values. The techniques developed in data dependence theory do not appear suitable for the problem of allocating registers to array elements for primarily two reasons. First, unlike parallelization, control flow information must be taken into account in order to determine whether a computed value is still available at uses that reference the same array element. Secondly, only a restricted class of dependencies among array references offers opportunities for reuse across loop iterations.

We have developed an efficient data flow based algorithm to determine live ranges for array elements inside a loop. The notion of *δ-available values* is introduced to express that a value of a subscripted variable is available for reuse δ iterations after its generation. Using information about the δ-available values inside a loop, we construct complete live ranges for array elements.

2 Overview of Register Pipelining

Our integrated register allocation algorithm is driven by the loop structure of the program starting with innermost loops and subsequently progressing towards outermost loops and the main program. Thus, a single loop is considered at any one time, and each loop is processed for allocation of register pipelines in the following steps.

Step 1 - Live Range Analysis. The first step consists of determining the live ranges of variable values, which are the entities for register assignment. The live range of a variable value starts with an initial definition or use of the variable and ends at the last use of the same value. Live ranges for scalars (*S-ranges*) are determined by solving the standard data flow problem of finding definition-use chains [1]. Unlike scalars, the value of an array element can be accessed inside its live range under different names that are all aliases of one other. Moreover, the name under which an array element is referenced may change in different iterations of a loop. For example, the expression A[I] in iteration instance I = *i* and the expression A[I-1] in iteration instance I = *i*+1 denote the same array element. Thus, an algorithm to construct live ranges of array elements (*A-ranges*) must necessarily account for the potential of aliasing. We introduce a family of *canonical names* to distinguish among the possible ways of naming array elements. The canonical name of an array element changes with each additional iteration in which the respective value is live. By keeping track of the current canonical names we compute the δ-available values of subscripted variables; that is the values that are available δ iterations after their generation. The information about the δ-available values is then used to construct the A-ranges inside a loop.

Step 2 - Constructing the Integrated Register Interference Graph (IRIG). The live ranges for both scalar and subscripted variables are presented in a common structure, the IRIG. To guarantee a fair competition for the available registers we uniformly calculate priorities of the S-ranges and A-ranges in the graph.

Step 3 - Multi-Coloring. Register pipelines are assigned to the live ranges in the IRIG by *multi-coloring* the IRIG using *k* colors, where *k* is the number of available registers. The nodes in the graph are colored based on their priorities by favoring nodes with highest priority. In standard coloring for register allocation, each node is colored with a single color representing the single register that is needed to enable reuse. A register pipeline, however, may require more resources, i.e., more than one register. It follows that multiple colors must be assigned to such a node. We have extended the standard coloring algorithm to a multi-coloring algorithm, in which a single node may be assigned multiple colors.

Step 4 - Code Generation for Register Pipelines. When all loops in the program have been allocated register pipelines, the final code is generated. Code generation for a register pipeline allocated to an A-range includes the code for appropriately initializing the pipeline in the loop header, the replacement of variable references by accesses to pipeline stages and the code to implement the pipeline progression at the end of each iteration.

As an additional optimization we present a simple loop transformation that can improve register pipelining. The improvement results from a reduction of the pipeline depths for an A-range (i.e., a reduction in the number of registers needed to enable reuse), and thus to a reduction of the overall register pressure. The transformation is a source-to-source transformation and may be applied prior to the register assignment.

The remainder of this paper is organized as follows. Section 3 presents our live range analysis. The Integrated Register Interference Graph is defined in Section 4. Section 5 presents the multi-coloring algorithm. The code generation aspects of register pipelining are discussed in Section 6. Section 7 demonstrates the loop transformation used to improve register pipelining. Section 8 presents related work and concluding remarks are given in Section 9.

3 Live Range Analysis

We construct live ranges for subscripted references by computing the available array elements (*available A-values*) inside a loop. An available A-value v is *generated* by a definition or a use of a subscripted variable V. The value v is available up to a point where variable V is redefined, in which case v is said to be *killed*. An A-range starts at a point where an A-value v is generated (i.e., at a definition or use of an array element) and extends up to the last use of the same array element at which v is still available. We refer to the problem of determining the available A-values at a given program point as the *Avail-problem*.

3.1 The Avail-Problem

The Avail-problem is a forward data flow problem and presents a generalization of the problem of determining reaching definitions; a value may be available from a "reaching use" as well as from a reaching definition. Finding a general solution to the Avail-problem for subscripted variables is a difficult task due to the potential of aliases. Kallis and Klappholz [10] describe a technique to compute reaching definitions for subscripted variables using a number of sophisticated data dependence tests to disambiguate potential aliases. Fortunately, for the purposes of register allocation we are not interested in a general solution to the problem. Instead, we are interested in only those available values that can be preserved in a register for reuse. To illustrate the difference of the two problems consider the loop in Fig. 2 (i).

```
Do I=1,1000
(1) A[2*I]=A[I]+X
Enddo
```

(i)

```
Do I=1,1000
(1) A[I+1]=A[I]+X
Enddo
```

(ii)

Fig. 2. Live ranges inside a loop with varying (i) and with constant (ii) iteration distance.

Half of the values defined in statement 1 reach a use in the same statement some iterations later. However, the values that reach a use cannot be preserved in registers due to reuse for two reasons. First, *not all* values used in statement 1 are available, and second, the ones that are available are not available from an earlier computation with a *constant iteration distance*. For example, the value for the use of A[2] in the second iteration was generated one iteration earlier, but the value of A[4] for the use in the fourth iteration was generated two iterations earlier.

It follows, that we are interested in the more restricted problem of determining for a program point p, the A-values that are (1) simultaneously available for all iteration instances of p and (2) that are generated a constant iteration distance earlier. To capture these requirements we refine the notion of available A-values to δ-available A-values.

Definition. Given a program point p inside a loop L, an A-value v is *δ-available* at p if v is generated by a use or definition of a subscripted variable V at a point p' inside L, and no path from p' to p that circulates δ times around L contains a redefinition of V.

A necessary condition for a value to be δ-available is that it is generated by a subscripted reference, whose subscript is a function of a loop induction variable. We consider only subscripts that are functions of the loop's basic induction variable (i.e., the loop control variable) by assuming that prior to the analysis, all non-basic induction variables have been identified and removed [1] and that all loops have been normalized. We furthermore restrict the candidates for δ-available values to references whose subscripts are linear functions of the loop control variable, and we call such a reference a *generating reference*.

Definition. A subscript expression e is called a *generating subscript* if e is a linear function of the loop control variable I, i.e., $e = f(I)$ and $f(I) = a \times I + c$, where a and c are integer constants. Otherwise, e is called a *non-generating subscript.*

δ-available values are determined by forward propagation. The propagation proceeds along the control paths from points that contain generating references until points are encountered where the respective values are killed. Since the propagation of δ-available values may extend across multiple iterations, we need to keep track of the current name under which the value can be referenced. For this purpose we introduce the notion of a canonical name for a δ-available value.

Definition. Given a δ-available value v for an element of an array A, the *canonical name* for v is A[$f(I - \delta)$], where A[$f(I)$] is the generating reference for v.

Consider again the loop in Fig. 2 (ii). The initial canonical name for the value computed in statement 1 is A[I+1]. The canonical names under which the value can be referenced one and two iterations later are A[I] and A[I-1], respectively. The canonical name describes only one way of naming the respective δ-available value. Any other name for the same value is said to be an *alias*. In general, we consider any two subscripted references that have subscripts that differ by anything other than a constant as potential aliases of one another.

3.2 A Data Flow Algorithm for the Avail-Problem

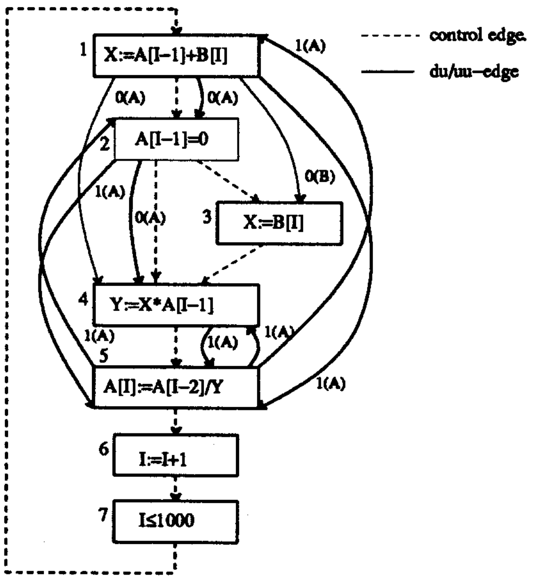
This section presents a data flow algorithm to construct A-ranges by computing the δ-available values inside a loop. Our algorithm operates on the intermediate statement-level control flow graph for a given loop body. Since access patterns of array references are to be analyzed, the intermediate code representation (IR) preserves the original array reference as shown in the example in Fig. 3.

A-ranges are constructed by inserting definition-use (du) edges and use-use (uu) edges for each detected reuse of a δ-available value in the graph. A du/uu-edge is labeled with an *iteration distance* indicating the number of iterations between the two access points that are connected by the edge. Thus, an A-range is represented by a path of du/uu-edges. The control flow graph in Fig. 3 (ii) shows the du/uu-edges that represent the A-ranges inside the loop shown in Fig. 3 (i). The edge $5 \rightarrow 1$ with label 1(A), for example, expresses that the value of array A computed in node 5 is reused in node 1 one iteration later. By ignoring transitive edges we obtain the complete path representing the live ranges for elements of array A as $5 \rightarrow^1 1 \rightarrow^0 2 \rightarrow^0 4 \rightarrow^1 5$. The superscripts refer to the iteration distance of the respective du/uu-edges in the graph.

One approach for the determination of A-ranges inside a loop is as follows. First, the δ-available values are computed at each node up to some maximum value for δ. In a second step, du/uu-edges are created by examining the computed information at each node n to determine which of the δ-available values are reused in n. However, this approach requires saving the avail-sets at each node for all considered δ values. To avoid the unnecessarily high storage requirement we developed a data flow algorithm that combines the two steps and saves only the δ-available values for the currently considered value of δ.

```
Do I=3,1000
(1) X:=A[I-1]+B[I];
(2) If A[I-1]=0 Then
(3)    X:=B[I];
(4) Y:=X*A[I-1];
(5) A[I]:=A[I-2]/Y;
Enddo
```

(i)



(ii)

Fig. 3. A sample loop (i) and its control flow graph annotated with du/uu-edges for A-ranges (ii).

```
Algorithm:  δ-Available Values.
Input:      ( L, $n_{entry}$, $n_{exit}$): control flow graph for a loop with distinguished entry and exit nodes
            $\delta_{MAX}$: maximal iteration distance value
Output:     du/uu edges in L

Begin
(1)  For every n∈L Do                                   /* initialization */
(2)      Gen[n] := { X[e] | e is generating and X[e] is used or defined in n }
(3)      Kill[n] :={ X[e] | X[e] is defined in n }
(4)      IN[l] := ∅; OUT[l] := ∅; Init[n] := Gen[n];
(5)  EndFor
(6)  δ := 0
(7)  Change:= true;
(8)  While $\delta < \delta_{MAX}$ and Change Do                    /* data flow computation */
(9)      Change:= false;
(10)     WS := { $n_{entry}$ };
(11)     While ( WS ≠ ∅ ) Do                            /* one pass through the loop */
(12)         remove a node n from WS;
(13)         IN[n] := $\bigcap_{p \text{ a predeccessor}}$ OUT[p];         /* available values on entry to node n */
(14)         NEWOUT := IN[n];
(15)         For each X[e] ∈ NEWOUT Do
(16)             If X[e] is used in n Then
(17)                 create an edge from the origins of X[e] to s with distance δ;
(18)             EndIf;
(19)             If X[e] ∈ Kill[n] or X[e] is aliased in Kill[n] Then
(20)                 NEWOUT := NEWOUT - {X[e]};
(21)             EndIf;
(22)         EndFor;
(23)         OUT[n] :=  NEWOUT[n] ∪ Init[n];
(24)         Init[n] := ∅;            /* add generating subscripts only in the first pass */
(25)         WS := WS ∪ {p | p a successor of s };
(26)     EndWhile;
(27)     δ := δ + 1;                                    /* prepare for next pass */
(28)     NEWOUT := { X[e - 1] | X[e] ∈ OUT[$n_{exit}$] };   /* update canonical names */
(29)     If NEWOUT ≠ ∅ Then
(30)         Change:= true;
(31)         OUT[$n_{exit}$] := NEWOUT;
(32)     EndIf;
(33) EndWhile;
End.
```

Fig. 4. Data flow algorithm to construct du/uu-edges by computing the δ-available values in a loop.

The detailed algorithm is depicted in Fig. 4. The algorithm performs multiple passes through a loop L and determines during pass i the i-available values in L. For every node n in loop L we define the sets Gen[n] and Kill[n]. Gen[n] contains the canonical names of generating references in n. Thus, Gen[n] describes the candidates for δ-available values. Kill[n] contains the subscripted references of definitions in n. The

sets	first pass: $\delta = 0$	second pass: $\delta = 1$
IN[1]	-	*$A[I-1]^{(5)}$, $A[I-2]^{(1,2,4)}$, $A[I-3]^{(5)}$, $B[I-1]^{(1)}$
OUT[1]	$A[I-1]^{(1)}$, $B[I]^{(1)}$	$A[I-1]^{(5)}$, $A[I-2]^{(1,2,4)}$, $A[I-3]^{(5)}$, $B[I-1]^{(1)}$
IN[2]	*$A[I-1]^{(1)}$, $B[I]^{(1)}$	*$A[I-1]^{(5)}$, $A[I-2]^{(1,2,4)}$, $A[I-3]^{(5)}$, $B[I-1]^{(1)}$
OUT[2]	$A[I-1]^{(1,2)}$, $B[I]^{(1)}$	$A[I-1]^{(5)}$, $A[I-2]^{(1,2,4)}$, $A[I-3]^{(5)}$, $B[I-1]^{(1)}$
IN[3]	$A[I-1]^{(1,2)}$, *$B[I]^{(1)}$	$A[I-1]^{(5)}$, $A[I-2]^{(1,2,4)}$, $A[I-3]^{(5)}$, $B[I-1]^{(1)}$
OUT[3]	$A[I-1]^{(1,2)}$, $B[I]^{(1,3)}$	$A[I-1]^{(5)}$, $A[I-2]^{(1,2,4)}$, $A[I-3]^{(5)}$, $B[I-1]^{(1)}$
IN[4]	*$A[I-1]^{(1,2)}$, $B[I]^{(1)}$	*$A[I-1]^{(5)}$, $A[I-2]^{(1,2,4)}$, $A[I-3]^{(5)}$, $B[I-1]^{(1)}$
OUT[4]	$A[I-1]^{(1,2,4)}$, $B[I]^{(1)}$	$A[I-1]^{(5)}$, $A[I-2]^{(1,2,4)}$, $A[I-3]^{(5)}$, $B[I-1]^{(1)}$
IN[5]	$A[I-1]^{(1,2,4)}$, $B[I]^{(1)}$	$A[I-1]^{(5)}$, *$A[I-2]^{(1,2,4)}$, $A[I-3]^{(5)}$, $B[I-1]^{(1)}$
OUT[5]	$A[I]^{(5)}$, $A[I-1]^{(1,2,4)}$, $A[I-2]^{(5)}$, $B[I]^{(1)}$	$A[I-1]^{(5)}$, $A[I-2]^{(1,2,4)}$, $A[I-3]^{(5)}$, $B[I-1]^{(1)}$
IN[6]	$A[I]^{(5)}$, $A[I-1]^{(1,2,4)}$, $A[I-2]^{(5)}$, $B[I]^{(1)}$	$A[I-1]^{(5)}$, $A[I-2]^{(1,2,4)}$, $A[I-3]^{(5)}$, $B[I-1]^{(1)}$
OUT[6]	$A[I]^{(5)}$, $A[I-1]^{(1,2,4)}$, $A[I-2]^{(5)}$, $B[I]^{(1)}$	$A[I-1]^{(5)}$, $A[I-2]^{(1,2,4)}$, $A[I-3]^{(5)}$, $B[I-1]^{(1)}$
IN[7]	$A[I]^{(5)}$, $A[I-1]^{(1,2,4)}$, $A[I-2]^{(5)}$, $B[I]^{(1)}$	$A[I-1]^{(5)}$, $A[I-2]^{(1,2,4)}$, $A[I-3]^{(5)}$, $B[I-1]^{(1)}$
OUT[7]	$A[I]^{(5)}$, $A[I-1]^{(1,2,4)}$, $A[I-2]^{(5)}$, $B[I]^{(1)}$	$A[I-1]^{(5)}$, $A[I-2]^{(1,2,4)}$, $A[I-3]^{(5)}$, $B[I-1]^{(1)}$

Table 1. Computed data flow sets for the δ-available values in Fig. 3(ii).

algorithm computes at every node n the data flow sets IN[n] and OUT[n]. At the end of the i-th iteration of the main loop (see lines 7-33), IN[n] contains the canonical names of values that are i-available on entry to node n and OUT[n] contains the canonical names of those values from IN[n] that pass through n, i.e., that are i-available on exit from node n. An i-available value with canonical name A[$f(I-i)$] does not pass through node n if the value is killed in n by a definition to A[$f(I-i)$] or to a potential alias thereof (see lines 19-2).

Finally the question arises of how to determine the maximum δ value to be considered, i.e., the number of passes through the loop. For each additional iteration that a value is to be preserved for reuse, an additional register is needed. Thus, a δ value that exceeds the number of available registers does not need to be considered. It follows that the number of available registers can be used to control the number of passes in the algorithm. We illustrate the execution of the algorithm with the example of Fig. 3 (i). Table 1 shows the computed sets at the end of the first and the second iteration of the algorithm in Fig. 4. Superscripts of table entries denote the statement number of the respective reference. Entries marked with * denote the points where a du/uu-edge is created.

After processing a loop L, a *summary node* is created in the enclosing loop. During the analysis, a summary node is treated as any other node. For the determination of the summary sets Gen[L] and Kill[L] for a summary node L, we observe that a subscripted reference that is a generating reference in L may be a non-generating reference in the enclosing loop. Similarly, a reference that is non-generating or killing in L can be a generating reference in the enclosing loop. Thus, the subscripted references in loop L are re-inspected to determine whether they belong to one of the sets Gen[L] and Kill[L]. By incorporating the summary nodes in the analysis of the enclosing loop, it is possible to construct A-ranges that contain references in both L and its enclosing loops.

Although we have described the algorithm for one-dimensional arrays, multi-dimensional arrays are handled in the same manner by propagating collections of canonical names or, alternatively, by linearizing multi-dimensional arrays. To determine the time complexity of the algorithm we observe that each loop is analyzed once and each node inside a loop at most r times, where r is a constant denoting the number of available registers. Thus, the analysis requires $O(r \times N)$ node visits, where N is the number of statements in the program. The time required to process each node is determined by the size of the sets IN and OUT, which is bounded by the maximal number of statements immediately contained inside a loop in the program.

4 The Integrated Register Interference Graph

The problem of register allocation can be formulated as the problem of k-coloring the register interference graph [1], where k is the number of available registers. The nodes in the interference graph represent live ranges of variables. Two nodes are connected, i.e., *interfere*, if the corresponding live ranges overlap and cannot be held in the same register. Live ranges are assigned priorities that express the benefits of keeping the corresponding variable in a register, and registers are assigned to live ranges by coloring the corresponding nodes based on their priorities. We extend the traditional structure of the register interference graph to represent live ranges for subscripted variables as well as scalar live ranges. The resulting graph is called the *Integrated Register Interference Graph (IRIG)*.

The IRIG for a loop L is an undirected graph $G = (N, E)$, where N is a set of nodes representing the S-ranges and A-ranges in L, and $E \subseteq N \times N$ is a set of interference edges. For each live range that is inserted as a node n in the IRIG, the interference edges to other nodes already in the graph are determined. For live ranges that do not cross iteration boundaries, interferences are determined according to standard register allocation methods. To determine the interferences for an A-range that crosses iteration boundaries, we observe that a live range that crosses entire iterations necessarily interferes with any other live range in that loop. Thus, a node in the IRIG representing such an A-range has interferences edges to all other nodes in the IRIG.

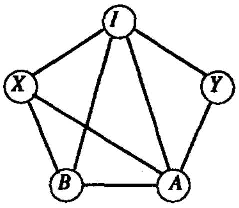
We define two functions on the set of live ranges represented by the nodes in N. Function P is a priority function that assigns to each node $n \in N$ a priority $P(n)$. The second function *depth* assigns to each node $n \in N$ a value $depth(n)$. $Depth(n)$ denotes the depth of the register pipeline that is needed to preserve the generated values in lr. To determine the depth of an A-range lr, we consider the number of iterations that each value in lr is live. Thus, we consider the iteration distance of the latest access points in lr, denoted as $\delta_{max}(lr)$. $Depth(lr)$ is then computed as:

$$depth(lr) = \begin{cases} 1 & \text{if } lr \text{ is scalar} \\ \delta_{max}(lr)+1 & \text{otherwise} \end{cases}$$

Next we determine the priority $P(lr)$ of a live range lr. The priority calculation is based on the savings in memory load instructions. The calculated savings are normalized with respect to the live range length (the number of statements in lr) so as to favor live ranges that require the register resources for only short periods. To determine priorities uniformly over S-ranges and A-ranges, we calculate the utilization of each required register. Let $Cost_{LD}$ be the cost of executing a memory load instruction, $access(lr)$ be the number of access points in lr, and let $|lr|$ denote the length of lr. $P(lr)$ is then computed as:

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

Note, that a memory load instruction is saved for all but the initial access in lr. The IRIG from the example of Fig. 3 is depicted in Fig. 5 (i). The values of the priority and depth functions are shown in the table in Fig. 5 (ii), assuming the cost for a memory load instruction is one, i.e., $Cost_{LD} = 1$. The loop induction variable I requires special treatment in the priority calculation. I is no longer used in the computation of a subscript in an array reference that was found to represent a reuse of an already available value. Thus, I is actually only used in the computation of subscripts that belong to the initial generating references of a live range. It follows that, in this example, there are two uses of I in subscript expressions (for generating values in A and B) plus two additional uses at the beginning and end of the loop (i.e, $access(I) = 4$).



(i)

node	*access*-1	*length*	*depth*	*P*
I	4	7	1	0.57
X	1	4	1	0.25
Y	1	2	1	0.5
A	4	7	2	0.29
B	1	3	1	0.33

(ii)

Fig. 5. The IRIG (i) for the loop from Fig. 3 and the priority and depth values (ii).

5 Multi-Coloring the Register Interference Graph

Register pipelines are assigned to live ranges by multi-coloring the IRIG. In the standard coloring strategy for the (scalar) register interference graph, the coloring of nodes that have fewer interferences (i.e., neighbors) than available registers is postponed knowing that these nodes can always be colored. These nodes are called the *unconstrained* nodes [7]. A node that has more interferences than available registers is called a *constrained* node. The constrained nodes are split, creating two or more new nodes with fewer neighbors. This process terminates when there are no more constrained nodes in the graph and all nodes can be assigned a color.

The situation is slightly different when coloring the IRIG, due to that fact that a node may require a register pipeline that consists of more than one stage and thus requires more than one color. The number of interferences that determines whether a node is an unconstrained node is no longer obtained by counting the neighbors in the graph. The possibly varying register requirements of the neighbor nodes as well as the register requirement of the node itself must be taken into account. Thus, a node n in the IRIG $G = (N, E)$ is an unconstrained node if

$$depth(n) + \sum_{(n,m) \in E} depth(m) \leq k.$$

If the above inequality holds , there will always be *depth(n)* colors to multi-color node n. The remaining portion of the coloring procedure is essentially unchanged with respect to the standard scalar case. If there is a constrained node that cannot be multi-colored, the corresponding live range is split into one or more smaller live ranges that have either fewer interferences or a smaller depth. If there are multiple candidates for splitting, the one with lowest priority is split first. A strategy to decide where to split a live range is described in [7].

6 Code Generation

This section discusses the code generation aspects that are unique to allocating register pipelines to live ranges. The use of a register pipeline for an A-range involves three phases. During the first phase, the pipeline is *initialized* at the end of the loop preheader. The second phase consists of the actual *use of the register pipeline*, during which references to subscripted variables are replaced by accesses to pipeline stages. Finally, the last phase consists of the *pipeline progression* at the end of the loop body. A value located at stage i of the pipeline progresses to stage $i+1$ for use in the next iteration.

We sketch the code generation aspects given an A-range lr with depth $depth(lr)$, the register pipeline $r_0, \ldots, r_{depth(lr)-1}$ allocated to lr, the generating subscript expression $f(I)$ and the initial value I_0 of loop induction variable I.

(1) Initialization. For each stage r_i, where $i = 1,\ldots,depth(lr)-1$ generate:

$$\text{load } r_i \leftarrow A(f(I_0 - i))$$

(2) Usage. Replace every access with canonical name $A[f(I-d)]$ by an access to pipeline stage r_d. Generate a store (load) instruction for the generating definition (use) in lr.

(3) Pipeline progression. For each stage r_i, where $i = 1,\ldots, depth(lr)-1$ generate:

$$r_i \leftarrow r_{i-1}$$

Fig. 6 shows the generated code for the example of Fig. 3 assuming that all live ranges were allocated physical registers. If standard register allocation techniques are used, six memory load instructions and one store instruction are executed per iteration due to the array references. By applying our integrated approach that allocates register pipelines to the A-ranges for A and B, the code can be significantly improved as shown in Fig. 6 (ii). The memory traffic for subscripted references is reduced to a single load and a single store instruction per iteration at the cost of two additional register-to-register move instructions.

Note that the code in Fig. 6 (ii) does not explicitly represent the address calculation for load and store instructions of array elements. We assume in this paper that the target instruction set provides appropriate addressing mechanisms. If, however, no such mechanisms exist, the necessary address calculation may require additional registers. These additional register requirements can be incorporated into our techniques by including the address calculation in the intermediate code representation.

The pipeline progression phase has an overhead that is absent if register pipelines are not used. The number of required register-to-register move instructions for the pipeline progression is $s-1$, where s is the pipeline depth. Thus, if the depth becomes very high, the overhead of the necessary register-to-register move instructions may actually undo the savings that are gained by using a register pipeline. One way to address this problem of overallocation is to incorporate the pipeline progression as a negative factor into the priority calculation (i.e., by subtracting the costs of necessary copy instructions from the calculated savings). Only live ranges with positive priorities are considered for allocation so that non-beneficial use of a register pipeline due to the overhead of the pipeline progression is prevented.

An alternative approach to overcome the additional overhead of the pipeline progression uses hardware support. The Cydra 5 [8] architecture provides hardware mechanisms to improve the execution of loops. There is a special hardware register called the *Iteration Control Pointer* (ICP) that is implicitly addressed by each register reference. The

```
Do I=3,1000
(1) X:=A[I-1]+B[I];
(2) If A[I-1]=0 Then
(3)     X:=B[I];
(4) Y:=X*A[I-2];
(5) A[I]:=A[I-2]/Y;
Enddo
```

(i)

```
    load rA1 <- A(2)        /* initialization */
    load rA2 <- A(1)
    rI <- 3
L:  load rB <- B(rI)        /* statement 1 */
    rX <- rA1+rB
    store rX -> X
    if rA1=0 goto L1        /* statement 2 */
    rX <- rB                /* statement 3 */
    store rX -> X
L1: rX <- rX*rA1            /* statement 4 */
    store rX -> Y
    rA0 <- rA2/rX           /* statement 5 */
    store rA0 -> A(rI)
    rI <- rI+1
    rA2 <- rA1         /* pipeline progression*/
    rA1 <- rA0
    if rI<=1000 goto L
```

(ii)

Fig. 6. Source code from Fig. 3 (i) and the generated pseudo-target code (ii) with the following register pipelines: (rA0, rA1, rA2) for array A, rB for array B, register rX for the scalars X and Y, and rI for scalar I.

register referenced by an instruction is computed as the sum of the register identifier in the instruction and the ICP. By appropriately updating the ICP after each iteration, this architecture can be used to implement the pipeline progression phase in a register windowing scheme.

Another hardware solution provides a concurrent register-to-register move instruction. If subsequent registers $r_0, \ldots, r_n$ are assigned to a register pipeline, then a concurrent register-to-register move instruction *CMOV* $r_n \leftarrow r_0$ performs the individual moves $r_i \leftarrow r_{i-1}$ for $i = 1, \ldots, n$ simultaneously within one cycle.

7 Loop Transformations

We discuss in this section a loop transformation to improve register pipelining. Applying this transformation results in a reduction of the depth of the register pipelines that are needed to preserve values for reuse. Thus, the benefits of this transformation is a reduction of the overall register pressure.

Callahan et al. [4] have shown that *loop interchange* and *unroll-and-jam*, two loop transformations originally developed for parallelizing compilers, can reduce the iteration distance between subscripted references of the same array element. These transformations are applied to nested loops and shorten the dependence distance among references by moving dependencies carried by outer loops to inner loops. We consider in this section a transformation that is applied to a single-level loop. The transformation is a simplified variation of a technique used in software pipelining [2, 12] The idea borrowed from software pipelining is to modify the original iteration boundaries. However, unlike software pipelining, the original execution order of statements remains unchanged. Thus, applying the transformation is always 'safe', in that all data dependencies are guaranteed to be preserved.

Consider the loop in Fig. 7 (i) that contains a single A-range. The value computed in statement 2 is used in statement 1 two iterations later. Thus, three registers are required to preserve the computed values for reuse. To demonstrate how the use in statement 1 can be moved one iteration closer to the definition, we unroll the loop once as shown in Fig. 7

```
Do I=3,1000
(1) B[I]:=A[I-2];
(2) A[I]:=f(I);
Enddo;
```

(i)

```
Do I=3,1000,2
┌(1) B[I]:=A[I-2];
└(2) A[I]:=f(I);      ┐
┌(1) B[I+1]:=A[I-1];  ┘
└(2) A[I+1]:=f(I+1);
Enddo;
```

(ii)

```
(1) B[3]:=A[1];
Do I=3,999
(2) A[I]:=f(I);
(1) B[I+1]:=A[I-1];
Enddo;
(2) A[1000]:=f(1000);
```

(iii)

Fig. 7. A sample loop (i), the loop unrolled once (ii) and the transformed loop (iii).

(ii). The brackets to the left denote the original iteration window and the brackets to the right show the iteration window moved forward by one statement. Fig. 7 (iii) shows the restructured and transformed loop with the new iteration window. The transformation has reduced the iteration distance by one. Thus, only two registers are necessary to preserve the values for reuse in the transformed loop.

In general, we apply the transformation in situations where a statement s_1 uses a value that is available from a use (definition) d iterations earlier in a statement s_2 and s_1 occurs before s_2 in the loop body. To apply the transformation with respect to the iteration distance d from s_1 to s_2, i.e., to apply *Transform*(s_1, s_2), we first unroll the loop once. The original iteration window in the unrolled loop is moved forward to start immediately after the first occurrence of s_1. The transformed loop is obtained using the new iteration window and code for the fragmental first and last iteration is appropriately inserted before and after the transformed loop. The iteration distance between the accesses in s_1 and s_2 is reduced to $d-1$ in the transformed loop.

8 Related Work

Exploiting reuse opportunities for recurrences involving array references has been addressed as an additional compiler optimization by several authors. Benitez and Davidson [3] describe an algorithm to handle recurrences in loops in the context of an access/execute architecture for streaming accesses to structured data. Individual read/write pairs are detected and code that eliminates the memory load is generated. A data flow based approach to optimize subscripted references by removing redundant memory loads is described by Rau [15]. In a dynamic single assignment model, it is first assumed that all computed values inside a loop are preserved in *expanded virtual registers*. A data flow algorithm determines which of the preserved values are reused, i.e, which load instructions are redundant. The algorithm assumes unconstrained register resources and does not address the mapping of virtual to actual registers.

A different approach for exploiting reuse opportunities for array elements has been presented in form of a source-to-source transformation called *scalar replacement* [4]. A similar technique has also been used in the Cydrome compiler for the Cydra 5 architecture [8, 15]. In a preprocessing phase, scalar replacement considers data dependencies among array references and replaces some of the array references by references to new temporary scalar variables, under the assumption that a standard coloring based register allocator will assign registers to these temporaries. Scalar replacement is based on data dependence information that is insensitive to the control flow inside a loop, so that reuse opportunities in the presence of conditional control flow may not be recognized.

Recently, data dependence testing has been formulated in a data flow framework to determine reaching definitions for array references [10]. Data dependence tests are applied to subscripted references to determine whether a reaching definition is *killed* by an array reference. The developed techniques could be adapted to our integrated register allocator as an optional refinement that is applied when an increased compile-time overhead due to the application of data dependence tests is acceptable.

The memory latency problem for array references inside a loop has also been addressed on a different level of the memory hierarchy by improving the cache performance for these references. Several authors describe algorithms to rearrange the loop computation to improve the data locality of a loop nest and thus reduce cache miss latencies [9, 13, 14, 16]. A technique called *Software Prefetching* [5] improves cache performance for array references through a cache prefetch instruction. Compiler generated prefetch instructions are inserted in the loop code to load array elements into the cache that are accessed in later iterations.

9 Conclusion

In this paper we addressed the problem of effectively integrating the allocation of registers to both subscripted and scalar variables to enable a fair competition for the available registers. Register pipelining was introduced to uniformly formulate the allocation problem for both classes of variables. Based on the concept of register pipelining, we developed an integrated variation of the priority-based coloring strategy for register allocation. In order to extend standard data flow techniques for the construction of live ranges to array elements, we introduced the notion of δ-available values. An efficient data flow algorithm is presented to compute δ-available values across loop iterations for the construction of live ranges. Although we have presented our data flow algorithm to compute δ-available values in a way that was tailored to the register allocation problem, the value of the developed techniques is not restricted to register allocation. We are currently investigating how the computed information can be used to extend the application of traditional scalar optimizations, such as constant propagation, to subscripted variables. Handwritten experiments have shown that the use of a register pipeline can significantly reduce the generated memory traffic while requiring only low analysis overhead. We are currently implementing the integrated register allocator to obtain experimental data on the savings that can be achieved by using register pipelines.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman, in *Compilers, principles, techniques, and tools*, Addison-Wesley Publishing Company, Massachusetts, 1986.
2. A. Aiken and A. Nicolau, "Optimal loop parallelization," *Proc. of the ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pp. 308-317, Atlanta, Georgia, June 1988.
3. M. E. Benitez and J. W. Davidson, "Code generation for streaming: an access/execute mechanism," *Proc. of the 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems.*, pp. 132-141, Santa Clara, California, April 1991.
4. 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, White Plains, New York, June 1990.

5. D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching," *Proc. of the 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 40-52, Santa Clara, California, April 1991.

6. G. J. Chaitin, "Register allocation and spilling via graph coloring," *(Proc. of the ACM SIGPLAN 82 Symp. on Compiler Construction), ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 201-207, June 1982.

7. F. Chow and J. Hennessy, "Register allocation by priority-based coloring," *ACM SIGPLAN Notices*, vol. 19, no. 6, pp. 222-232, 1984.

8. 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, Boston, Massachusetts, April 1989.

9. D. Gannon, W. Jalby, and K. Gallivan, "Strategies for cache and local memory management by global program transformation," *Journal of Parallel and Distributed Computing*, no. 5, pp. 587-616, 1988.

10. A. D. Kallis and D. Klappholz, "Reaching definitions analysis on code containing array references," *4th Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, California, August 1991.

11. 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, Williamsburgh, Virginia, January, 1981.

12. M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," *Proc. of the ACM SIGPLAN '88 Conf. Programming Language Design and Implementation*, pp. 318-328, Atlanta, Georgia, June 1988.

13. M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," *Proc. of the 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 63-74, Santa Clara, California, April 1991.

14. A. Porterfield, "Software methods for improvement of cache performance on supercomputer applications," *Ph.D. thesis*, Rice University, May 1989.

15. B. R. Rau, "Data flow and dependence analysis for instruction-level parallelism," *4th Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, California, August 1991.

16. M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," *Proc. of the ACM SIGPLAN '91 Conf. Programming Language Design and Implementation*, pp. 30-44, Toronto, Ontario, Canada, June 1991.

17. 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.

18. M. Wolfe, "Optimizing supercompilers for supercomputers," *Pitman Publishing Company, London, MIT Press*, Cambridge, Massachusets, 1989.