Array Data Flow Analysis for Load-Store Optimizations in Superscalar Architectures*

Rastislav Bodík and Rajiv Gupta

Dept. of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260

Abstract. The performance of scientific programs on superscalar processors can be significantly degraded by memory references that frequently arise due to load and store operations associated with array references. Therefore, register allocation techniques have been developed for allocating registers to array elements whose values are repeatedly referenced over one or more loop iterations. To place load, store, and register-to-register shift operations without introducing fully/partially redundant and dead memory operations, a detailed value flow analysis of array references is required. We present an analysis framework to efficiently solve various data flow problems required by array load-store optimizations. The framework determines the collective behavior of recurrent references spread over multiple loop iterations. We also demonstrate how our algorithms can be adapted for various fine-grain architectures.

1 Introduction

The performance of a superscalar processor is adversely affected by frequent references to memory. Therefore, techniques for effective allocation of registers to scalars have received considerable attention [5]. In scientific programs, a significant fraction of memory accesses arise from references to array elements. In order to reduce the frequency of these memory references, recent research has also considered the assignment of registers to array elements in inner loops [1, 3, 4, 10, 11].

The loop in Fig. 1(a) contains references that in turn access the same array element over five consecutive loop iterations. The value of an array element flows among the references forming a live range. As illustrated in Fig. 2, the value of $A[i+4]$ is defined in the k^{th} iteration and then redefined or used in the subsequent iterations by the shadowed references. A set of registers can be used to form a pipeline that holds the repeatedly referenced array elements throughout the live range. The definition of $A[2\times i]$ is a killing reference that interferes with the value flow shown in Fig. 2 at two points. Since it may redefine the value kept in the register pipeline, additional load operations must be introduced to ensure that values of array elements are available in appropriate registers when needed. Also,

* Partially supported by National Science Foundation Presidential Young Investigator Award CCR-9157371 to the University of Pittsburgh and a grant from Hewlett-Packard Laboratories.

```
i    for i in 1...N
ii       use A[i+2]
iii      use A[i]
iv       if (odd(i)) then
v            use A[i+3]
vi           def A[i+3]
vii      else
viii         def A[2×i]
ix       endif
x        def A[i+4]
xi   endloop
```

(a) Source loop.

```
load R1, A[4]
load R2, A[3]
load R3, A[2]
load R4, A[1]
for i in 1...N                  i
    use R2                      ii
    use R4                      iii
    if (odd(i)) then            iv
        use R1                  v
        shift R3, R4
        shift R2, R3
        def R2                  vi
        store R2, A[i+3]        vi
    else                        vii
        store R1, A[i+3]
        def R2                  viii
        store R2, A[2×i]        viii
        shift R3, R4
        shift R1, R2
        load R3, A[i+2]
    endif                       ix
    def R1                      x
endloop                         xi
store R1, A[N+4]
```

(b) After the optimization.

Fig. 1. The example.

store operations are inserted to save newly computed values of array elements in memory. Lastly, register-to-register copy operations have to be introduced to shift the contents of the pipeline once per iteration.

In this paper we present an analysis framework to efficiently solve various data flow problems required by the desired load-store optimizations. In the pro-

$i = k$	$i = k + 1$	$i = k + 2$	$i = k + 3$	$i = k + 4$

use $A[i+2]$
use $A[i]$
if (odd(i))
  use $A[i+3]$
  def $A[i+3]$
else
  def $A[2 \times i]$
endif
$\boxed{\text{def } A[i+4]}$
	use $A[i+2]$
	use $A[i]$
	if (odd(i))
	  $\boxed{\text{use } A[i+3]}$
	  $\boxed{\text{def } A[i+3]}$
	else
	  def $A[2 \times i]$
	endif
	def $A[i+4]$
		$\boxed{\text{use } A[i+2]}$
		use $A[i]$
		if (odd(i))
		  use $A[i+3]$
		  def $A[i+3]$
		else
		  $\boxed{\text{def } A[2 \times i]}$
		endif
		def $A[i+4]$
			use $A[i+2]$
			use $A[i]$
			if (odd(i))
			  use $A[i+3]$
			  def $A[i+3]$
			else
			  def $A[2 \times i]$
			endif
			def $A[i+4]$
				use $A[i+2]$
				$\boxed{\text{use } A[i]}$
				if (odd(i))
				  use $A[i+3]$
				  def $A[i+3]$
				else
				  $\boxed{\text{def } A[2 \times i]}$
				endif
				def $A[i+4]$

Fig. 2. Flow of the value among array elements during any five consecutive loop iterations.

posed method, all references accessing the same element are considered in concert during a window of consecutive iterations similar to that in Fig. 2. The span of the window covers the entire live range of the array references. In addition, killing references that do not interfere with the value flow through the window are ignored, resulting in more precise analysis. For example, the definition of $A[2 \times i]$ kills the value of $A[i+4]$ from the k^{th} iteration only two and four iterations later. Thus, the definition of $A[i+4]$ and the definition of $A[i+3]$, when considered together, lead us to conclude that the value for use in reference to $A[i+2]$ is always available. Due to these properties, the framework exposes precise collective behavior of the recurrent references to which a register pipeline will be allocated. Our work was inspired by the framework of Duesterwald [11] which is based on reference analysis and thus is not capable of determining that two references may in concert generate the same value. To overcome this deficiency, we base our framework on value analysis.

Arbitrary placement of loads and stores may introduce fully/partially redundant loads and fully/partially dead stores which can be precisely characterized as follows.

Definition 1. A load operation, **load** $R, A[i]$, at a program point p is **partially/fully redundant** if, along *some/all paths* leading to p, the value of $A[i]$ is already available in R.

Definition 2. A store operation, **store** $R, A[i]$, at a program point p is **partially/fully dead** if, along *some/all paths* leaving from p, every potential use of the current value of $A[i]$ is read from register R.

We describe the application domain of our framework through load-store optimization which performs partial redundancy elimination (PRE) and partial dead code elimination (PDE) of the memory operations. Figure 1(b) shows the optimal placement of loads, stores, and shifts for the example loop. A load before the use of $A[i]$ would be partially redundant because, if i is odd in the $k+2^{nd}$ iteration, the value of $A[i+2]$ was not killed by the definition of $A[2 \times i]$ in this iteration (see Fig. 2). A load is appropriately inserted at line 20 to restore the pipeline in case the value was killed in this iteration. A store placed immediately after the definition of $A[i+4]$ in the k^{th} iteration would be partially dead because the value is rewritten one iteration later by the definition of $A[i+3]$. Therefore, the store is delayed to the next iteration and performed at line 15. Shifts are placed in the holes of the live range which results in two operations per iteration, as opposed to four when shifts are placed at the end of the loop body. In summary, the optimized code performs 0/1 load and 1/2 stores per iteration. In contrast, load placement which does not consider PRE and PDE of memory references would perform 2 loads and 2 stores. Existing PRE and PDE techniques have primarily focussed on scalars and thus they do not perform optimization of array references over multiple loop iterations [8, 9, 15, 17].

2 Analyzing Congruent References

This section presents the data flow analysis framework. The analysis considers a group of array references that refer to the same array element during the execution of one or more consecutive loop iterations. Problems such as must availability and must anticipability of a value related to the array element can be solved using the framework. The group of array references being analyzed collectively is termed as a *congruent class* of references. The solution of an analysis of a congruent class of references will be *pictorially* represented using a *stretched loop*. A stretched loop offers a concise view at the window of consecutive iterations shown in Fig. 2 and conceptually illustrates the pattern of the flow of values associated with a congruent class. Figure 3(d) depicts the live range of the example loop; it contains only the references that influence the value, i.e., the references in boxes and the appropriate control statements. The number of loop iterations in the stretched loop for every congruent class is a small constant, typically less than 5, as is suggested by our experiments. Not only is the stretched loop short, but it is also merely an abstract visualization aid. The source loop is never unrolled and the framework computes the solution corresponding to the stretched loop on the source loop itself.

A given loop is analyzed for each array that is referenced in the loop. The references to a given array are divided into disjoint congruent classes based on their index expressions and each congruent class is processed individually, considering killing effects of references from other congruent classes of the same array. Analysis of a congruent class over the length of the stretched loop is sufficient to perform optimizations applicable to the references in the congruent class. Precise definitions of congruence are given below.

Definition 3. References r_1 and r_2 to an array A are **congruent** if they are of the form $r_1 = A[c \times i + k_1]$ and $r_2 = A[c \times i + k_2]$, where i is the normalized loop induction variable and $k_1 \bmod c = k_2 \bmod c$.

Definition 4. A **congruent class** of references to an array A, denoted as $[A, c \times i + k]$, is the largest set of congruent references in the loop of the form $A[c \times i + j]$, such that $j \bmod c = k \bmod c$. For a given loop the references to array A in the loop can be partitioned into disjoint congruent classes.

In Fig. 1(a), the references to array A are divided into two congruent classes. The congruence class $[A, 1 \times i]$ contains references $A[i], A[i+2], A[i+3], A[i+4]$; the congruent class $[A, 2 \times i]$ contains the reference $A[2 \times i]$.

Next, we describe the data flow framework in detail. The framework is able to solve both forward and backward data flow problems. We will present the analysis in the context of forward problems. As illustrated in Fig. 3, the solutions of various analyses span multiple loop iterations. To compute such solutions, we do not explicitly construct a stretched loop. Instead, the solution of an analysis on the conceptual stretched loop is represented in form of a vector associated with each node in the original flow graph. The solutions corresponding to distinct iterations in the stretched loop are computed into distinct positions in the vector.

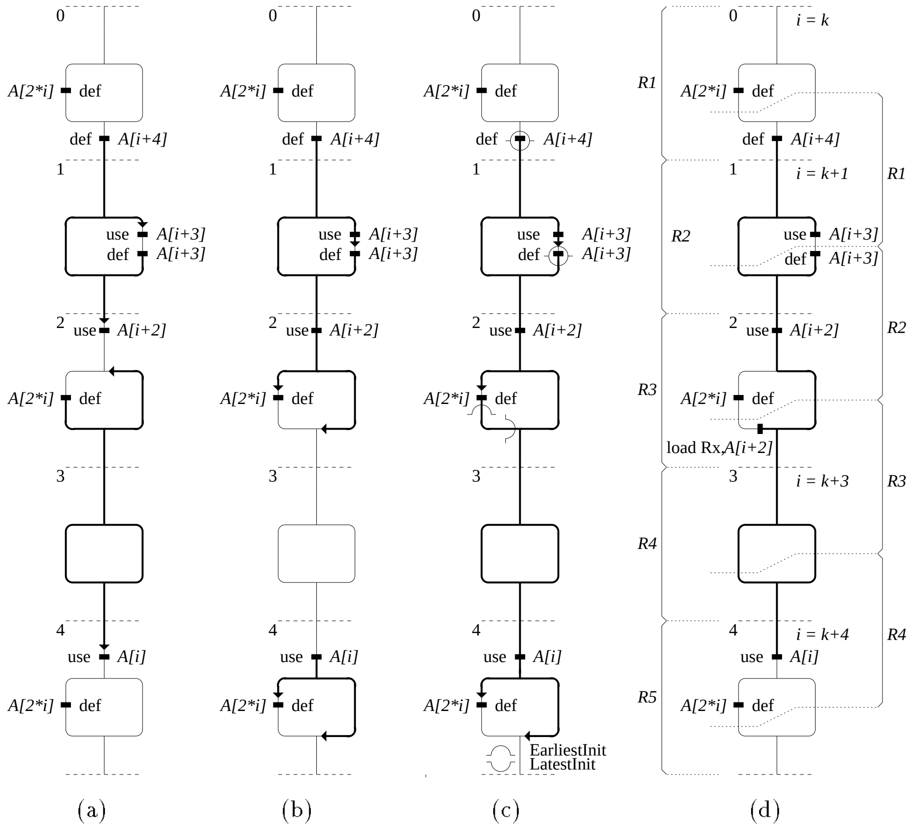


Fig. 3. Data flow solutions on the stretched loop for congruent class $[A, 1 \times i]$: (a) Range of must anticipability of congruent uses; (b) Range of must availability of congruent values; (c) EarliestInit and LatestInit points in the merged range; and (d) Usable congruent live range and shift placement points: naive shift placement on the left, optimal shift placement on the right.

In general, if d is the iteration distance between the lexicographically most distant references in a congruence class, then the stretched loop for the congruent class contains at most $d+1$ consecutive loop iterations which are numbered from 0 to d. Therefore, the size of the solution vector can be at most $d+1$. The size of the solution vector can be reduced by omitting the iterations that contain no array references and kills. For example, in Fig. 3, the iteration numbered 3 can be omitted from the vector. Since a node n in the original loop appears in multiple iterations of the stretched loop, we will use n_i to denote the instance of node n in the i^{th} iteration of the stretched loop.

Since we are mainly interested in problems such as availability of values, the binary lattice $\{\top, \bot\}$, where $\top$ is the greatest element, is used to represent the solutions. The data flow solution value $\top$ indicates that the characteristic of interest holds while the value $\perp$ indicates that characteristic does not hold. The parametric meet operator $\sqcap$ can be either $\wedge$ or $\vee$ where the operator $\wedge$, which is used for must problems, returns the least value and the operator $\vee$, which is used for may problems, returns the highest value.

A straightforward implementation of data flow over the stretched loop requires traversing the loop body $d+1$ times. However, we present a better approach in which all $d+1$ iterations in the stretched loop are processed simultaneously using bit-vector operations. We use the chain lattice $\{\top, Cond, \perp\}$ in which $\top$ and $\perp$ have the same meaning as above. The value $Cond$ at node n_i indicates that the solution for n_i is the same as the solution for $start_i$, where $start_i$ is the start node of the loop body in the i^{th} iteration of the stretched loop. In the first step, the algorithm simultaneously processes all $d+1$ iterations under the assumption that the solution for $start_i$'s is $Cond$. The value $Cond$ is propagated to nodes whose solution depends on the solution of the appropriate $start_i$ node. In the second step, a single pass over the $start_i$'s is used to compute the true solution for $start_i$'s. In the third step again nodes from all $d+1$ iterations are simultaneously processed such that the $Cond$ solution at each node instance n_i is changed to $\top$ if the solution at $start_i$ is $\top$ and to $\perp$ if the solution at $start_i$ is $\perp$. Therefore, the above solution requires only two passes over all the nodes during which vector operations are performed. Next, we describe the computation of GEN and $KILL$ sets and present the above algorithm in detail.

For a given congruence class C, we define set GEN^C to contain all generating references for class C and $KILL^C$ to contain references that kill values of class C. For the must availability problem, GEN^C contains all congruent definitions and uses from C and $KILL^C$ is the set of definitions that kill values generated by references in C. Shown in Fig. 4(a) is how the placement of references from these two sets on the stretched loop is computed. The computation of the positions in function $position^C$ requires two pieces of information, the iteration number $it(r)$ of every reference r in C and the killing distance between a reference s from C and any other reference r. The iteration number of reference R is equal to the number of the stretched loop segment where r resides.

A single killing reference may kill the flow of the congruent value in various distances from the generating references and hence appear in multiple iterations of the stretched loop. For instance, consider the placement of the killing reference $A[2 \times i]$ in the stretched loop for class $[A, 1 \times i]$ of the example loop. For forward/backward *must* data flow problems, it is sufficient to place the killing reference r, for every congruent reference s, in the single killing point closest to s in the appropriate direction of data flow. The value of k_r^s is the shortest killing distance between reference s and a killing reference r, that is, for $k_r^s - 1$ iterations after the value of s is generated it is *not* killed by reference r. The value of k_r^s can be computed using the simple approach suggested in [11], which is equivalent to computing line intersection. If k_r^s is ∞, then there is no dependence between references r and s. On systems with full dependence analysis, accuracy of a finite k_r^s is verified by any data dependence test such as the *gcd* test [18].

procedure *Specialize*
  for every reference $r \in GEN^C$
    for every $i \in positions^C(r)$
      add r to GEN_i^C
  for every reference $r \in KILL^C$
    for every $i \in positions^C(r)$
      add r to $KILL_i^C$

function $positions^C(r)$
  if $r \in C$ then $P := \{it(r)\}$
  else for every $s \in C$ do
      if $k_r^s \neq \infty$ then
        $P := P \cup \{it(s) + k_r^s\}$
  return P

function $f_{n_i}^C(sol)$
  if n contains $r \in GEN_i^C$
    return $\top$
  else if n contains $r \in KILL_i^C$
    return $\perp$
  else return sol

(a)

Step I: Conditional solution
Initialize *start* point in every iteration:
$IN^C[start][0..d] := (Cond, \ldots, Cond)$

for each node n in reverse postorder do
  for $i := 0$ to d do (vectorizable loop)
    $IN^C[n][i] := \sqcap_{m \in pred(n)} OUT^C[m][i]$
    $OUT^C[n][i] := f_{n_i}^C(IN^C[n][i])$

Step II: Solution for *start* points
$IN^C[start][0] := \perp$
for $i := 1$ to d
  if $IN^C[start][i] = Cond$ then
    $IN^C[start][i] := IN^C[start][i-1]$

Step III: Resolving cond. solutions
for every node n do
  for $i := 0$ to d do (vectorizable loop)
    if $IN^C[n][i] = Cond$ then
      $IN^C[n][i] := IN^C[start][i]$
    if $OUT^C[n][i] = Cond$ then
      $OUT^C[n][i] := IN^C[start][i]$

(b)

Fig. 4. Data flow framework for forward problems: (a) $f_{n_i}^C$: flow function for node n_i with respect to congruent class C; and (b) Data flow algorithm.

The *flow function* for a node n corresponding to the i^{th} iteration of the stretched loop is defined in Fig. 4(a). If the node contains a generating (killing) reference, then the flow function returns $\top$ ($\perp$), otherwise the flow function is the identity function. The computation of the data flow solution using the three step process discussed earlier is shown in Fig. 4(b).

In the above analysis framework, the inner loops of Steps I and III are vectorized using bit-vector operations. Since all congruent classes can be processed simultaneously, only two passes over all the nodes in the loop are required by this algorithm. The solution vector at every node n is represented by two parallel bit vectors in which n_i owns the two i^{th} bits. The encoding of the conditional lattice is $\{\perp = 00, Cond = 01, \top = 11\}$. The runtime complexity of the algorithm is expected to be $O(N)$ bit-vector operations or $O(N^2)$ bit-operations where N is the number of array references in the loop. The bound is low due to typically low values of d_{max}, the length of the longest stretched loop, and d_{sum}, the cumulative length of all stretched loops in the loop, as shown in Sect. 4. The bit-vector implementation and the complexity analysis are elaborated in [2].

The framework can also analyze loops with multiple entries and exits. This is possible because we are interested in determining the flow of values in the stabilized middle portion of the iteration space, not the flow in the first or last iterations which may be excuted only partially, due to multiple entries or exits.

3 The Optimization Algorithms

This section presents the applications of the array data flow framework. First, the optimal load-store placement algorithm is described. Then, we show how the framework can improve software pipelining and discuss architectural support provided by existing architectures which can assist in array register allocation.

3.1 The Load-Store Placement Algorithm

To minimize the number of memory references, loop invariant loads and stores can be moved out of the loop. Here we present algorithms for determining the placement of loads, stores, and shifts operations which cannot be moved out of the loop. Our approach uses virtual register names; a subsequent register allocation phase will map the names to physical registers. Under the assumptions that the branching structure of the program and the ordering of code statements is preserved, the algorithm presented achieves the following characteristics:

1. Minimal number of loads and stores are introduced along every path through the loop and therefore the resulting code performs minimal number of memory related operations for array references. This form of optimality is a direct consequence of employing a formulation for partial redundancy elimination developed by Knoop *et al.* [15].
2. The lengths of array live ranges are minimal, that is, the lengths cannot be shortened without introducing additional memory references.
3. Corresponding to a minimal placement of loads and stores, the placement of register-to-register shift operations introduced is such that the number of shifts executed on an average is minimized.
4. Corresponding to a load, store, and shift placement, the number of virtual registers introduced for a given live range is minimal. Furthermore, the scalar ranges created through the use of virtual registers will be precisely identifiable by a scalar register allocator such as the one presented in [13].

The algorithm consists of five steps: (1) Load placement; (2) Store placement; (3) Shift placement and introduction of virtual registers names in loads, stores, and shift operations; (4) Prologue and epilogue creation to ensure that values are appropriately loaded into and stored out of virtual registers; and (5) Register allocation. Each congruent class is processed separately. The algorithms are applied with respect to a given branching structure and code ordering, that is, we do not modify the branching structure or reorder program statements during these algorithms. However, since arbitrary graphs may prevent code placement optimizations, a dummy node is inserted along every *critical edge*[2] [8, 15].

Load Placement. The main objective of this step is to identify points at which the placement of loads does not introduce partial redundancy. An approach for eliminating partially redundant scalar expressions was developed by

[2] A critical edge is an edge leading from a node with more than one successor to a node with more than one predecessor.

Knoop *et al.* [15]. This approach is adapted for application to loads for array references by utilizing the framework introduced in the preceding section.

In order to avoid partial redundancy, we must place loads *as early as necessary* to ensure that later uses can reuse the value and the need for later loads is eliminated. We refer to a point at which a virtual register is initialized as an *initialization point.* To avoid partial redundancy we must consider only those points as initialization points at which the value under consideration is *must anticipable*, that is, there is a use of the value along all paths from the point. Furthermore, if the value under consideration is *must available* at the initialization point, then no load is needed at this point. The initialization points which satisfy the above conditions are called the *earliest initialization* points. Instead of placing loads at earliest points, we compute the *delaylability* of earliest points and insert loads *as late as possible* to shorten the live range during which the values are kept in registers. The resulting points are called *latest initialization* points. This step causes loads to be delayed to the closest use point or to a merge point some of whose incoming paths could not delay their load. The initialization points can be either array uses or array definitions; if the value is being defined at a latest point, then it is directly computed into the appropriate virtual register and no load is required. The above algorithm is summarized in Fig. 5. As we can see, the framework from the preceding section is used to perform *must anticipability*, *must availability*, and *delaylability* analysis. For the example loop of Fig. 1(a), the results of these analyses corresponding to congruence class $[A, 1 \times i]$ are shown in Figs. 3(a), 3(b), and 3(c).

Placement of stores. The values of array references directly computed into virtual registers must be written from the registers to the memory preceding killing uses, congruent loads, and prior to loop termination (assuming the array is used after the loop). The process of placing stores so that partially dead stores are avoided is analogous to the introduction of loads so that partially redundant loads are avoided. A store is partially dead if a later store causes the earlier store to become redundant along a program path. Therefore, the store placement analysis is, with respect to data flow direction, opposite of load placement analysis. While load placement requires backward anticipability, forward availability, and forward delaylability analysis, the placement of stores requires *forward anticipability* and *backward availability* of congruent definitions, and *backward dispatchability* of stores.

Placement of register shifts. Array live ranges longer than one iteration are allocated a set of virtual registers which must shift their values once per iteration using register copy operations so that each reference finds its value in the same register during each loop iteration. Existing techniques [1, 3, 4, 10] place the shifts at the *end* of the loop body, as illustrated in Fig. 3(d). In this example, such a placement requires four copy operations because the iteration boundary crosses the live range four times.

However, when a shift copy is placed in a live range hole, it becomes redundant because the destination register will be reloaded later. Since our data flow framework computes precise shape of live ranges, we can place shifts optimally

Definitions:

Let $\mathcal{R}$ be the set of all references to a specific array in a loop and $C \subseteq \mathcal{R}$ a congruent class. Then

$C_{def} \subseteq C$ is the set of defining references in C,
$C_{use} \subseteq C$ is the set of using references in C,
$(\mathcal{R} \setminus C)$ is the set of references in $\mathcal{R}$ that are not in C,
$(\mathcal{R} \setminus C)_{def}$ is the set of defining references in $(\mathcal{R} \setminus C)$, and
$(\mathcal{R} \setminus C)_{use}$ is the set of using references in $(\mathcal{R} \setminus C)$.

Note: $(\mathcal{R} \setminus C)$ is the set of references that may have data dependence on references in C and thus may appear as killing references in the stretched loop for C.

Load Placement:

AnticUseRange $= \{n_i \mid OUT^C_{AnUse}[n][i] = \top\}$
where $OUT^C_{AnUse}[n][i]$ is solution to *must anticip. of congr. uses* (i.e., potential loads) for n_i
Direction: backward $\quad GEN^C = C_{use} \quad KILL^C = C_{def} \cup (\mathcal{R} \setminus C)_{def} \quad$ Meet $= \wedge$

AvailValRange $= \{n_i \mid IN^C_{AvVal}[n][i] = \top\}$
where $IN^C_{AvVal}[n][i]$ is the solution to *must availability of the congruent values* (i.e., congruent references) for n_i
Direction: forward $\quad GEN^C = C \quad KILL^C = (\mathcal{R} \setminus C)_{def} \quad$ Meet operator $= \wedge$

AALoadRange = **AnticUseRange** $\cup$ **AvailValRange**
EarliestInit $= \{n_i \mid \exists m \in pred(n) : n_i \in$ **AALoadRange** $\wedge$
$(m_i \notin$ **AALoadRange** $\vee\ n_i$ contains r from $C_{def})\}$[a]
set of nodes where register initialization achieves minimal number of loads

DelayRange $= \{n_i \mid IN^C_{Del}[n][i] = \top\}$
where $IN^C_{Del}[n][i]$ is the solution to *delaylability* problem for n_i
Direction: forward $\quad GEN^C =$ **EarliestInit** $\quad KILL^C = C \quad$ Meet operator $= \wedge$

LatestInit $= \{n_i \mid \exists m \in succ(n) : n_i \in$ **DelayRange** $\wedge$
$(m_i \notin$ **DelayRange** $\vee\ n_i$ contains r from $C)\}$
set of nodes where register initialization achieves shortest live ranges
$\textbf{LatestInit}_{def} = \{n_i \in \textbf{LatestInit} \mid n \text{ contains } r \text{ from } C_{def}\}$
$\textbf{LatestInit}_{load} = \textbf{LatestInit} \setminus \textbf{LatestInit}_{def}$

Insert **def** R_x to nodes in $\textbf{LatestInit}_{def}$,
Insert **load** R_x at the entry of nodes in $\textbf{LatestInit}_{load}$ if they contain a reference from C_{use},
Insert **load** R_x at the exit of nodes in $\textbf{LatestInit}_{load}$ if they contain a reference from $(\mathcal{R} \setminus C)_{def}$.

[a] If $n = start$ then end_{i-1} is considered instead of m_i.

Store Placement:

AvailDefRange $= \{n_i \mid IN^C_{AvDef}[n][i] = \top\}$
where $IN^C_{AvDef}[n][i]$ is the solution to *must availability of congruent definitions* (i.e., definitions of registers) for n_i
Direction: forward $\quad GEN^C = C_{def} \quad KILL^C = (\mathcal{R} \setminus C) \quad$ Meet operator $= \wedge$

AnticDefRange $= \{n_i \mid IN^C_{AnDef}[n][i] = \top\}$
where $IN^C_{AnDef}[n][i]$ is the solution to *must anticipability of congruent definitions* (i.e., congruent value redefinitions)
Direction: backward $\quad GEN^C = C_{def} \quad KILL^C = \textbf{LatestInit}_{load} \cup (\mathcal{R} \setminus C)_{use} \quad$ Meet $= \wedge$

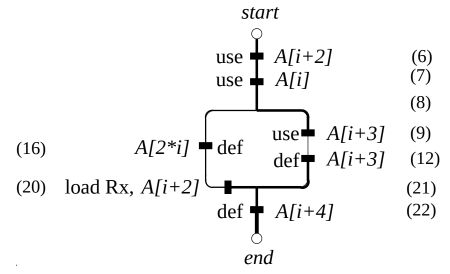
AAStoreRange = **AvailDefRange** $\cup$ **AnticDefRange**
LatestStore $= \{n_i \mid \exists m \in succ(n) : n_i \in$ **AAStoreRange** $\wedge$
$(m_i \notin$ **AAStoreRange** $\vee\ n_i$ contains r from $\textbf{LatestInit}_{load}\}$
set of nodes where register stores achieve minimal number of stores

DispatchRange $= \{n_i \mid OUT^C_{Disp}[n][i] = \top\}$
where $OUT^C_{Disp}[n][i]$ is the solution to *dispatchability* for n_i
Direction: backward $\quad GEN^C =$ **LatestStore** $\quad KILL^C = C_{def} \quad$ Meet $= \wedge$

EarliestStore $= \{n_i \mid \exists m \in pred(n) : n_i \in$ **DispatchRange** $\wedge$
$(m_i \notin$ **DispatchRange** $\vee\ n_i$ contains r from $C_{def})\}$
set of nodes where stores achieve earliest stores

Insert **store** R_x at the exit of nodes in **EarliestStore**.

Fig. 5. Load Placement and Store Placement algorithms.



n	appearance	p_n
$start$	4	1.0
6	4	1.0
7	3	1.0
8	2, 3	1.0
9	2	0.5
12	3	0.5
16	2	0.5
20	3	0.5
21	3	1.0
22	4	1.0
end	4	1.0

S	$Cost(S)$
$\{end\}$	**4.0**
$\{21\}$	3.0
$\{16, 12\}$	2.5
$\mathbf{\{16,\ 9\}}$	**2.0**
$\{7\}$	3.0
$\{6\}$	4.0

Fig. 6. Shift placement positions in the example loop.

by finding a set of nodes that cut the loop body through the live range holes. The optimal placement (Fig. 3(d)) requires only two copy operations per iteration.

Next, we present an algorithm that places shifts so that the average number of shifts expected to be executed per iteration is minimized. First, the precise live range is determined for a given congruent class using one-pass data flow analysis. Next, the number of times every node appears in the live range is weighted by the node execution probability to obtain the *weighted appearance* of the node. This is a trivial step because each node is attached a vector with live range solutions for each of its $d+1$ positions in the stretched loop. It suffices to traverse the vector and count the number of vector elements equal to $\top$.

A set of nodes S is a valid candidate for the placement of shifts if *exactly* one node from S is encountered along any path through the loop body. Such set is a generalized dominator [12] of end. The shifts are placed into nodes of set S which minimizes the sum of weighted appearances of its members.

The optimal dominator set is identified by means of solving the maximum network flow problem [6]. The network flow is defined by setting the flow capacity of every node n to the weighted appearance of the node. In a network with saturated flow, there exists a set of nodes S that cut the loop body such that the maximum flow from $start$ to end is equal to the cumulative flow capacity of the nodes in S [6]. Obviously, there is no cut with a lower flow capacity than that of S. Since S collectively dominates end and its cost is the minimum among all cuts, S represents the optimal shift placement. An $O(N^3)$ algorithm for the network flow problem exists [6] and, since the number of basic blocks in a loop is typically very small, the algorithm is practical for our purposes.

After the positions of shifts are fixed, the algorithm assigns the names to the registers in the shifts inserted in the range as well as to the anonymous virtual registers inserted in the previous stages. Figure 6 illustrates the computation of costs for several cuts of the loop body in Fig. 1(b). The optimal cut $\{13, 8\}$ requires two shifts. On the other hand, the approach used by Carr [4], which places the shifts at the end of the loop (i.e, at end), requires four shifts.

Prologue and epilogue. A *prologue* may be required to initialize the register pipeline prior to entering the loop and an *epilogue* may be required to flush the values from the pipeline after exiting the loop. The prologue contains a sequence of loads for loading values expected in the registers in the loop body during the first iteration. Since the store-elimination algorithm sometimes delays stores of array elements to later iterations, when the loop is exited, the registers may contain values that must be written by generating a series of stores after the loop. For multiple-exit loops, a different epilogue is generated for each exit.

If the newly computed values are not live when the loop is exited, the epilogue is formed by issuing the stores immediately after all definitions in the last k iterations, where k depends on how far the stores have been delayed. This solution handles the case of multiple-exit loops as well.

Register Allocation. Our algorithm determines precise shapes of array element live ranges and splits them into scalar ranges by inserting the shifts. The precision allows a scalar register allocator to use the same register for more array live ranges that do not overlap. For instance, in Fig. 1(b) lines 16 and 17 use register **R2** allocated to the congruent live range. Since the resulting scalar live ranges cross iterations, the allocator developed by Hendren *et al.* [13] is very suitable for subsequent register allocation. To eliminate the register pressure caused by allocating registers to all live ranges, the cost/benefit ratio can be determined for every candidate range, as is done in [4]. When only a part of a live range can be allocated registers, the most beneficial segment of the range can be found with a data flow analysis using our framework, as is done for scalars in the load/store range analysis developed by Kolte and Harrold [16]. Register placement of nested loops is performed in the manner similar to that of [13], starting from the innermost loop and working outward. A nested loop is represented in its enclosing loop by a summary node that references all virtual registers of the inner loop.

3.2 Elimination of Load Latency

The optimization from the previous section determines the shortest live range for every congruent value. Consequently, some loads will immediately precede their corresponding uses and the memory latency penalty will be incurred. In this subsection we present a simple data flow approach for moving loads at earlier points to hide the latency.

The latency elimination is realized as an additional phase performed before the load-placement step of the algorithm in Sect. 3.1. It is assumed that all except the load statements have already been scheduled. The proposed method finds for every array use the point where the load must be issued to be completed in time. These *latency* points are identified for every congruent class by means of backward data flow analysis on the stretched loop and are treated during the load-placement algorithm as references that use the value but for which no code is generated.

The data flow solution is computed on the integer chain lattice $L = \{0, 1, \ldots, T\}$, where T is the largest load latency $l(u)$, for any reference u. The solution value s on the input of a node signifies that the load must be issued s cycles before the node. The flow functions propagate the data flow solution as follows. Congruent uses raise the value of the solution to the value of their estimated latency. This value is subsequently decreased on every instruction by the execution time t of the instruction. Latency points are nodes with a non-zero solution at the output and a zero solution on the input. Since loads cannot be moved across definitions, killing nodes reset the solution to zero. When loads are moved above a merge point from both true and false branches, the load that must be issued earlier is considered by causing the $\vee$ operator returns the maximum value. The flow functions are defined as follows.

$$f_{n_i}^C(s) = \begin{cases} \max(l(u), s) & \text{if } n_i \text{ contains a congruent use } u, \\ 0 & \text{if } n_i \text{ contains any definition,} \\ \max(s-t, 0) & \text{otherwise.} \end{cases}$$

It should be noted that, since the data flow analysis is performed on the stretched loop, the loads can be moved across one or more iterations without special treatment. Furthermore, the integration of this step with the previous algorithm provides the implicit change of the index expression when such a move occurs.

It remains to be shown how the algorithm in Fig. 4(b) is modified to compute integer chain lattice solutions. The conditional solution determined in Step I for every node of the stretched loop now consists of two values, IN_0 and IN_T, that are computed separately. These two solutions correspond to zero and full contribution from the $start_i$ nodes and are obtained by the initialization with vectors $(0, \ldots, 0)$ and $(T, \ldots, T)$, respectively.

The final solution for the node n_i is computed from IN_0 and IN_T based on the following observations. The final solution of the node is not lower than IN_0, the solution with no contribution from the start node. Because the $\vee$ merge operator composes the solutions, when the contribution from the start node is T, the final solution is not lower than IN_T. Similarly, due to the linearity of the flow functions, when the solution at the start node is $T-d$, the solution at the node is not lower than $IN_T - d$. Assume that we know $IN[start][i]$, the final solution for the $start_i$ node. Then, the final solution for node n_i is given by

$$IN[n][i] = \max(IN_0[n][i], IN_T[n][i] - (T - IN[start][i])).$$

This formula is used to modify Steps II and III of Fig. 4(b).

3.3 Exploiting Fine-Grain Architectures.

Data Speculative Load in PlayDoh [14]. The algorithm inserts a load after every killing reference that appears in the live range of the congruent value. This load updates the register pipeline in case the killing reference wrote to a memory location that is being stored in a pipeline register. Since upward movement of such a load is not possible, the load latency may complicate scheduling. In this subsection we discuss the utilization of an existing run-time memory disambiguation mechanism [14] for eliminating the load latency and propose a modification to the mechanism.

```
1  do i = 1, n
2      store R1, A[i]
3      store R2, A[f(i)]
4      load R3, A[i]
5  end do
```

(a) Load after a kill.

```
1  do i = 1, n
2      store R1, A[i]
3      LDS R3, A[i]
4      store R2, A[f(i)]
5      LDV R3, A[i]
6  end do
```

(b) LDS/LDV.

```
1  do i = 1, n
2      store R1, A[i]
3      R3 = R1
4      LOG R3, A[i]
5      store R2, A[f(i)]
6      LDV R3, A[i]
7  end do
```

(c) LOG/LDV.

Fig. 7. The use of data speculative load.

In Fig. 7(a) the live range of the value $A[i]$ is split by the definition of $A[f(i)]$ and thus a load must be placed at line 4. The PlayDoh architecture [14] offers data speculative load (LDS) and load verification (LDV) instructions that can eliminate latency of the load as is shown in Fig. 7(b). The speculative load is started before the killing store is issued and an entry $\langle address(A[i]), \mathbf{R3}\rangle$ is placed in the processor data speculative log. If $f(i) = i$, then the subsequent store invalidates the entry. If the entry remains valid at line 5, the LDV instruction does not reissue the load.

The LDS/LDV pair is ideal for register allocation with congruent references because a killing reference is likely to invalidate the pipeline only a few times per loop execution. However, if the value to be loaded is available when the LDS is issued, the LDS load is redundant. We propose a new instruction LOG which inserts the entry to the log but does not issue the load. The use of LOG (Fig. 7(c)) eliminates the memory traffic related to reloading the killed values. Whether an LDS operation can be replaced by a LOG is decided based on the solution of the must-availability problem solved in the load-placement algorithm.

The LDS operations can be moved upwards so that the LDS load is completed when the LDV operation is issued. The LDS positions are determined in our framework by solving a data flow problem under a numeric chain lattice. The details are beyond the scope of the paper.

Streaming in WM [1]. The WM architecture provides *streaming* load/store instructions that enable vector-like access to array elements with a fixed displacement through a FIFO memory queue. Streaming loads can be used in our register allocation if no two loads in the live range lie on the same path, that is, every congruent value is loaded exactly once. This can be verified from the positions of the loads by a data flow analysis within our framework.

Cyclic Register Files in Cydra 5 [7]. The design of Cydra 5 [7] took into consideration array register allocation and provided a cyclic register file in which it is possible to shift the contents of a subset of registers in one cycle. When cyclic registers are used in load-store elimination, the single shift copy operation can be placed at the end of the loop body and the shift placement algorithm need not be consulted.

Software Pipelining. The minimal length of a software pipeline schedule is governed by resource constraints and loop carried dependencies. To achieve short schedules, it is necessary to consider only the value-based dependencies and eliminate the spurious memory-based dependencies.

Our framework can inexpensively eliminate spurious dependencies as follows. We consider all congruent classes present in the loop. A value-based def-use dependence between any two loop statements a and b exists if there is a def-use path from an instance of a to an instance of b in any of the congruent classes. The positions of the statements in the stretched loops are determined based on the array references in the statements.

4 Experiments and Related Work in Optimization

We are currently implementing presented techniques in the PDGCC compiler at the University of Pittsburgh. To verify the usefulness of our optimizations, we have determined by hand the register placement for the Livermore Loops benchmark suite and compared the results to the register placement of the Gnu C compiler (version 2.6.0). The number of loads due to array references was determined by profiling the suite, examining the assembly output of the compiler, and by considering the loads required by the manually optimized kernels. `Gcc` was unable to eliminate congruent loads across iterations even when loop unrolling, the CSE optimization, and all expensive optimizations were enabled. This implies that PRE of congruent memory references in the presence of conditional control flow is not performed in `gcc`.

Figure 8 gives the dynamic count of loads before and after the optimization. As we can see, a large fraction of loads are avoided. Since we were only able to optimize simple loops, we did not encounter any opportunities to eliminate stores, or find holes in array live ranges that would enable shift optimization. Figure 8 also gives the values of d_{max}, the largest stretched loop, and d_{sum}, the cumulative length of stretched loops for all congruent classes, for the examined programs. As we can see, the length of the stretched loop is typically small.

Techniques described in [1, 3, 10, 11] do not handle full conditional control flow. The only available method for array PRE is Scalar Replacement of Carr *et al.* [4]. However, we can obtain a superset of their optimizations. For instance, when a congruent definition is killed only outside the stretched loop window, their algorithm considers it to be an unsafe generator for the live range, even though for the register allocation purposes the kill has no effect. Moreover, the

Livermore loop no.	1	5	7	11	12	15	20	23
Loads placed by gcc -O2 -funroll-all-loops	1200	3000	1080	2000	2000	52500	8000	50000
Loads placed by our register placement	800	2000	360	1000	1000	35000	6000	40000
Loads eliminated	33%	33%	66%	50%	50%	33%	25%	20%
d_{max}	1	1	6	1	1	1	1	2
d_{sum}	4	4	10	3	3	13	8	10

Fig. 8. Dynamic count of array-related register loads before and after optimization. The maximum and cumulative lengths of stretched loops.

algorithm is very complex; eleven data flow problems must be solved to place the loads (we solve only three problems). While we can prove the optimality of our load-placement algorithm, their approach uses heuristic decisions. Also, the prologue created by their algorithm unrolls the loop, whereas ours only places a sequence of loads. As has been shown in Sect. 3.3, the compiler may need to solve various data flow problems to customize the load-store placement for a particular architecture. Our framework allows an easy formulation and provides precise calculation of these problems. Consequently, Carr's algorithm cannot place the shift operations optimally.

References

1. M.E. Benitez and J.W. Davidson, "Code Generation for Streaming: an Access/Execute Mechanism," *Proceedings of Arch. Support for Programming Languages and Operating Systems-IV*, pages 132-141, 1991.
2. R. Bodik and R. Gupta, "Optimal Placement of Load-Store Operations for Array Accesses in Loops," *Technical report 95–03*, DCS, Univ. of Pittsburgh, 1995.
3. D. Callahan, S. Carr, and K. Kennedy, "Improving Register Allocation for Subscripted Variables," *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, White Plains, New York, pages 53-65, June 1990.
4. S. Carr and K. Kennedy, "Scalar Replacement in the Presence of Conditional Control Flow," *Software-Practice and Experience*, Vol. 24, No. 1, pages 51-77, Jan. 1994.
5. G.J. Chaitin, "Register Allocation and Spilling via Graph Coloring," *Proceedings of the SIGPLAN Symposium on Compiler Construction*, SIGPLAN Notices, vol. 17, no. 6, pages 98-105, June 1982.
6. T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, 1990.
7. J.C. Dehnert, P.Y.-T. Hsu, and J.P. Bratt, "Overlapped Loop Support in the Cydra 5," *Proceedings of ASPLOS-III*, pages 26-39, 1989.
8. D.M. Dhamdhere, "Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise," *ACM Transactions on Programming Languages and Systems*, Volume 13, No. 2, pages 291-294, April 1991.
9. D.M. Dhamdhere, B.K. Rosen and F.K. Zadeck, "How to Analyze Large Programs Efficiently and informatively," *Proc. of the SIGPLAN PLDI*, San Francisco, California, pages 212-223, June 1992.
10. E. Duesterwald, R. Gupta, and M.L. Soffa, "Register Pipelining: An Integrated Approach to Register Allocation for Scalar and Subscripted Variables," *Proc. of International Workshop on Compiler Construction, LNCS 641* Springer Verlag, pages 192-206, Paderborn, Germany, October 1992.
11. E. Duesterwald, R. Gupta, and M.L. Soffa, "A Practical Data Flow Framework for Array Reference Analysis and its Application in Optimizations," *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.68-77, Albuquerque, New Mexico, June 1993.
12. R. Gupta, "Generalized Dominators and Post-Dominators," *The Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 246-257, Albuquerque, New Mexico, January 1992.
13. L. Hendren, G.R. Gao, E.R. Altman, and C. Mukerji, "A Register Allocation Framework based upon Hierarchical Cyclic Interval Graphs," *Intnl. Workshop on*

Compiler Construction, *LNCS 641* Springer Verlag, pages 176-191, Germany, 1992.
14. V. Kathail, M. Schlansker, and B. Rau, *HPL PlayDoh Architecture Specification: Version 1.0*, HPL-93-80, February, 1994.
15. J. Knoop, O. Ruthing, and B. Steffen, "Optimal Code Motion: Theory and Practice," *ACM TOPLAS*, vol. 16, num. 4, 1117-1155.
16. P. Kolte and M.J. Harrold, "Load/Store Range Analysis for Global Register Allocation," *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, pages 268-277, June 1994.
17. E. Morel and C. Renvoise, "Global Optimization by Suppression of Partial Redundancies," *Communications of the ACM*, Volume 22, No. 2, pages 96-103, 1979.
18. M. Wolfe and U. Banerjee, "Data Dependence and Its Application to Parallel Processing," *Intl. Journal of Parallel Programming*, Vol. 16, No. 2, April 1987.

This article was processed using the LaTeX macro package with LLNCS style