

Load/Store Range Analysis for Global Register Allocation*

Priyadarshan Kolte[†] and Mary Jean Harrold

Department of Computer Science

Clemson University

Clemson, SC 29634-1906

harrold@cs.clemson.edu

Abstract

Live range splitting techniques improve global register allocation by splitting the live ranges of variables into segments that are individually allocated registers. Load/store range analysis is a new technique for live range splitting that is based on reaching definition and live variable analyses. Our analysis localizes the profits and the register requirements of every access to every variable to provide a fine granularity of candidates for register allocation. Experiments on a suite of C and FORTRAN benchmark programs show that a graph coloring register allocator operating on load/store ranges often provides better allocations than the same allocator operating on live ranges. Experimental results also show that the computational cost of using load/store ranges for register allocation is moderately more than the cost of using live ranges.

1 Introduction

Register allocation maps variables in an intermediate language program to either registers or memory locations in order to minimize the number of accesses to memory during program execution. Due to the significant difference between access times of registers and memory, a good register allocation scheme

can produce appreciable speedup in program execution time[14]. Various compiler optimizations such as procedure inlining and code scheduling further increase the demand for good register allocation.

Register allocation is divided into two subproblems: *register spilling* and *register assignment*[1]. First, register spilling determines which live ranges will be assigned to registers (*allocated live ranges*) and which live ranges will be assigned to memory (*spilled live ranges*). Then, register assignment maps allocated live ranges to registers so that no register contains more than one live range at each program statement.

Graph coloring on *interference graphs* of live ranges is an established paradigm for register allocation[6, 7]. The nodes of an interference graph are live ranges, and there are edges between nodes that are simultaneously live at any statement in the program. Colors represent physical registers, and the nodes of the graph are assigned colors such that neighboring nodes do not share a color. Nodes that cannot be colored have to be spilled. Thus, under the graph coloring paradigm, the register assignment problem becomes the *coloring* problem, which determines whether all nodes of a given interference graph can be colored with a given number of colors. Similarly, the *spilling* problem becomes the node deletion problem which finds the minimum number of nodes to spill such that the resulting interference graph can be colored. Since both the coloring and the spilling problems are NP-complete[12], heuristic algorithms are employed to obtain a good register allocation.

Register allocation requires a weaker constraint than that imposed by interference graphs of live ranges, namely, two variables cannot share a register at *any particular* statement in the program. Thus, *segments* or *partitions* of a live range of a variable may be assigned different registers or even spilled to memory in different regions of the program. These

*This work was partially supported by NSF under grant CCR-9109531 to Clemson University.

[†]Current address: Department of CS&E, Oregon Graduate Institute, Beaverton, OR 97006. Email: pkolte@cse.ogi.edu

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

ACM-SIGPLAN-PLDI-6/93/Albuquerque, N.M.

© 1993 ACM 0-89791-598-4/93/0006/0268...\$1.50

partitions are connected to one another by register-register or register-memory transfer instructions, and the resulting allocation is valid even though it may not correspond to a valid coloring of the interference graph. To produce better register allocations, *live range splitting* techniques that partition live ranges have been implemented to solve both the spilling problem[5, 8, 18] and the assignment problem[8, 9]. Although live range splitting is promising, there is no theoretical or empirical evidence that these techniques provide better allocations than the graph coloring algorithm using live ranges. Further, there is no guarantee that the improvement in register allocation outweighs the connection costs, where *connection costs* represent the execution cost of the instructions that are inserted to connect the split live range segments of a variable that are assigned different registers.

In this paper, we present a new technique, *load/store range analysis*, that addresses the register spilling problem by splitting live ranges into partitions that can be independently and profitably allocated (or spilled). Load/store ranges are computed using a standard iterative data flow analysis algorithm that is similar to live variable analysis[1]. We construct the interference graph of a program using load/store ranges instead of live ranges, and use existing heuristics to spill and color the nodes of this graph. We incorporated our load/store range analysis into a register allocator that uses Chaitin's graph coloring scheme[7] with delayed spilling[3] and the Haifa suite of spill heuristics[2], and compared it to a similar register allocator that used live range interference graphs. Experiments on a small suite of well known C and FORTRAN benchmark programs show that the allocator operating on load/store ranges often produces better spilling than the one operating on live ranges.

The main benefit of our approach is that, unlike live ranges that are coarse and do not adequately account for clustered accesses of variables[8, 15], our technique provides a fine granularity of candidates for register allocation that are based on the access patterns of variables. Additionally, our load/store ranges can be used instead of live ranges with any existing graph coloring algorithms, and our technique can even complement other live range splitting schemes.

In the next section, we give background information. Section 3 describes the load/store range analysis technique. In Section 4, we discuss the use of load/store ranges with an existing register allocation scheme. Section 5 presents our experimental results. Related work is briefly discussed in Section 6. Finally, section 7 presents conclusions and future work.

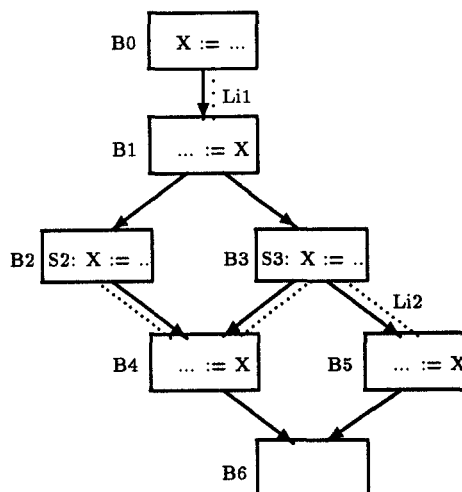


Figure 1: For variable X , Name1 contains $\{B0, B1\}$, and Name2 contains $\{B2, B3, B4, B5\}$.

2 Background

A *control flow graph* is a directed graph in which nodes represent program statements and edges represent flow of control between statements. A *subpath* in a control flow graph is a finite sequence of nodes (n_1, n_2, \dots, n_k) such that for $i = 1, 2, \dots, k-1$, there is an edge from n_i to n_{i+1} . A subpath (i, n_1, \dots, n_m, j) is *definition-free* with respect to a variable v from nodes i to j if there is no definition of v in any of the n_i on the path. A use U of variable v is *reachable* from statement S if there is at least one definition-free subpath with respect to v from S to U . A variable v is *live* at program statement S if there is a use of v that is reachable from S . The *live range* of variable v is the set of all statements over which v is live[1]. The live range of a variable may contain a number of non-contiguous regions, called *names*[6], where each name is a set of connected statements.

For example, Figure 1 shows a program with variable X defined in $B0, B2$ and $B3$, and used in $B1, B4$ and $B5$. Live range analysis reveals that X is live in all basic blocks except $B6$, and hence the live range of X is $Li(X) = \{B0, B1, B2, B3, B4, B5\}$. However, since X is not live on the edges $(B1, B2)$ and $(B1, B3)$, the live range Li contains two non-contiguous regions (names). These two names are Name1= $\{B0, B1\}$ and Name2= $\{B2, B3, B4, B5\}$, and they are shown in Figure 1. Although names are not identical to live ranges, we traditionally use the two terms synonymously.

Two live ranges *interfere* with each other if they are simultaneously live at any statement. The *degree* of live range Li is the number of live ranges that interfere with Li . If we assume that a memory access (load or store) takes one CPU clock cycle, the *cost* or *profit* associated with a live range is the weighted sum of the number of definitions and uses of the variable, where the weights are the the execution frequencies of the definitions and uses.

Chaitin's heuristic for deciding which live ranges to spill[7] chooses the candidate with the lowest *profit/degree* ratio. Briggs, et al.[3] introduced the idea of *delayed spilling*, which always produces allocations that are at least as good as those produced by Chaitin. Bernstein, et al.[2] developed the Haifa register allocator and refined Chaitin's spilling heuristic to produce a set of three complementary heuristic functions: $cost/degree^2$, $cost/(area \times degree)$, and $cost/(area \times degree^2)$. Their experiments showed that although none of these three heuristic functions consistently dominated one another, the "best-of-three" always outperformed Chaitin's heuristic for spilling.

3 Load/Store Range Analysis

The goal of load/store range analysis is to achieve better spilling by localizing profits and register requirements of the accesses to variables. *Register requirements* are the statements over which a register is required. *Accesses* are the definitions and uses of the variable. The *store range* of a definition is the set of statements over which the variable must be allocated a register to avoid a store at the definition. The *load range* of a use is the set of statements over which the variable must be allocated a register to avoid a load at the use. A store range may contain a number of load ranges, and there is at least one load range for every use in the store range. Store ranges are based on the observation that, in order to avoid a store at a definition, it is necessary and sufficient to allocate a register only to those statements in the live range of a variable where the definition reaches. The main observation used to construct load ranges is that allocating a register over all statements between the use of a variable and the "most recent" accesses of the variable is necessary and sufficient to avoid a load at the use.

Load ranges and store ranges are partitions of live ranges that can be independently and profitably allocated or spilled. Independent allocation means that the allocation, including the computation of the heuristic function for spilling costs, of a partition of

a live range is independent of all other partitions of that live range. Profitable allocation means that every live range segment has a positive savings in program execution time if it is allocated. An additional desirable property for a live range splitting scheme is that it produce minimum size partitions. Minimum size partitions are the smallest sets of statements that are profitable and independent, and hence every live range results in the maximum number of such partitions. Although load and store ranges usually satisfy the requirement of minimum size partitions, it may be possible to obtain finer partitions based on the sub-paths within the ranges.

3.1 Examples of load and store ranges

Example of load ranges: Figure 2 illustrates the usefulness of load ranges of a variable. There is a definition of X at statement $S1$, and uses of X at statements $S2$, $S3$, and $S4$. X is live between $S1$ and $S4$, and hence, the live range of X is $Li1:\{S1, \dots, S4\}$. Assuming that the basic block is executed once, the profit of allocating $Li1$ is 4 because it has 1 definition and 3 uses. Store range analysis produces the store range $St1:\{S1, \dots, S4\}$ that also has a profit of 4. Load range analysis produces the load ranges $Lo1:\{S1, \dots, S2\}$, $Lo2:\{S2, \dots, S3\}$ and $Lo3:\{S3, \dots, S4\}$. The profit of allocating $Lo1$ to a register is 1 because a load is saved at $S2$. The profit of allocating $Lo2$ is 1 because we assume that a load is used to access X at $S2$ and a register is used to access X at $S3$ and thus, we save a load at $S3$. Similarly the profit of $Lo3$ is 1.

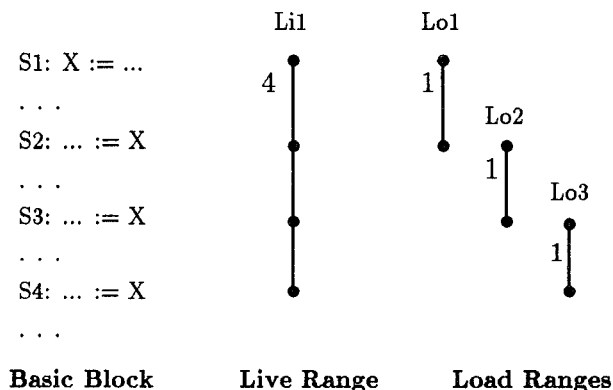


Figure 2: Load ranges $Lo1$, $Lo2$ and $Lo3$ are profitable and independent of one another. Store range $St1$ is not shown.

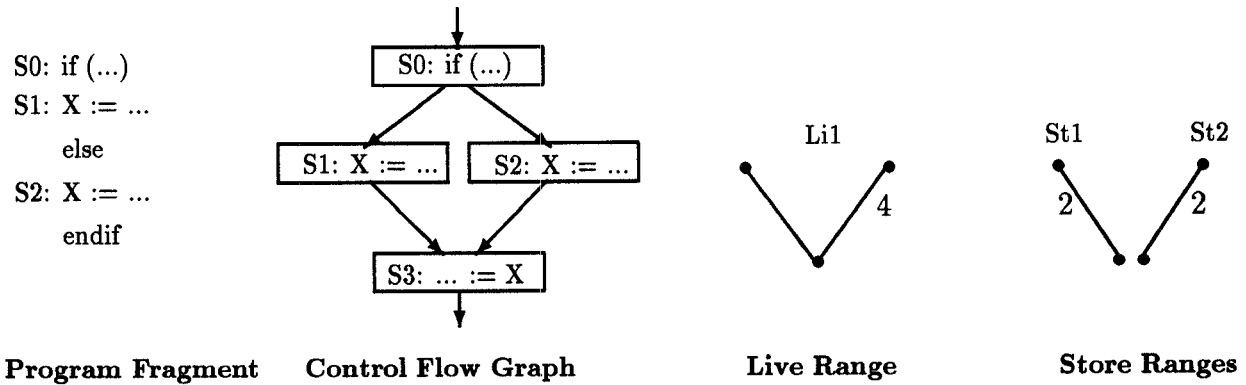


Figure 3: Store ranges $St1$ and $St2$ are profitable and independent of each other.

If there is a sufficient number of registers available in the basic block, allocating store range $St1$ is the same as allocating live range $Li1$ and yields a profit of 4. On the other hand, suppose there are too many live variables in the region between statements $S2$ and $S3$, and variable X must be spilled. If we spill the live range of X , $Li1$, there is no profit, but if we spill load ranges, we may be able to allocate the load ranges $Lo1$ and $Lo3$ to obtain a profit of 2. Load ranges $Lo1$, $Lo2$ and $Lo3$ are independent partitions because the allocation as well as the computation of the profit for any one of them does not depend on that of the others. When the entire live range lies within a single basic block, load ranges are minimum size partitions because it is not possible to find smaller partitions that yield any profit.

Example of store ranges: The program fragment shown in Figure 3 illustrates the usefulness of store ranges. Variable X is defined at statements $S1$ and $S2$, and is used at statement $S3$. The live range of X is $Li1:\{S1,\dots,S2,\dots,S3\}$. Assuming that $S1$ and $S2$ are each executed once and $S3$ is executed twice, the profit of $Li1$ is 4. Store range analysis produces two store ranges: $St1:\{S1,\dots,S3\}$ and $St2:\{S2,\dots,S3\}$ each with a profit of 2. Suppose a register is not available in the region $S1,\dots,S3$, but a register is available in the region $S2,\dots,S3$. Then, considering store ranges as candidates for spilling is beneficial because only $St1$ is spilled while $St2$ is allocated a register. The load X for $St1$ can be placed on the edge between blocks $B1$ and $B2$. It is clear that $St1$ and $St2$ are the minimum size partitions that are independent and profitable.

3.2 Computation of load and store ranges

The store range (St) of definition D of variable v is the set of all statements S that D reaches such that there exists a reachable use U of v at S .

$$St(D) = \{S \mid D \text{ reaches } S \wedge \exists U \text{ reachable from } S\}$$

We introduce additional terminology to define a load range. An *access* of a variable v is either a definition or a use of v . An *access-free subpath* with respect to variable v and two statements S_i and S_j is a subpath from statement S_i to the statement S_j that contains no accesses of the variable v . An access of variable v at statement S_i *arrives* at statement S_j if there is at least one access-free subpath with respect to v from S_i to S_j . Use U of variable v is *exposed* at statement S if there is at least one access-free subpath with respect to v from S to U . A definition of a variable is never exposed. The load range (Lo) of two accesses a_i and a_j in a store range is the set of statements between a_i and a_j .

$$Lo(a_i, a_j) = \{S \mid a_i \text{ arrives at } S \wedge a_j \text{ is exposed at } S\}$$

Computing store ranges requires reaching definition and reachable use analyses, which are performed using standard iterative data flow analysis algorithms[1]. After store ranges are computed, we assign unique subscripts to the definitions and uses of the variables in order to identify the accesses within store ranges. Then we use algorithm `ComputeLoadRanges`, given in Figure 4, to compute load ranges.

```

algorithm ComputeLoadRanges
input:      store range G with set of statements S and edge set E
output:    load ranges Lo of store range G
declare:   A_IN, A_OUT, E_IN, E_OUT, GEN, USE and KILL are sets of accesses
              such that there is one such set for each statement  $s \in S$ ;

begin

  /* step 1 : initialization of sets */
  for  $s \in S$  do
    GEN [s] := { a | a is an access in s };
    USE [s] := { a | a is a use in s };
    KILL [s] := { a | a is an access of a variable that occurs in s };
    A_IN [s] :=  $\phi$ ; /* A_IN is set of arriving accesses at a point just before s */
    A_OUT [s] := GEN [s]; /* A_OUT is set of arriving accesses at a point just after s */
    E_IN [s] := USE [s]; /* E_IN is set of exposed accesses at a point just before s */
    E_OUT [s] :=  $\phi$ ; /* E_OUT is set of exposed accesses at a point just after s */
  endfor

  /* step 2 : compute arriving accesses by iterating until a steady state */
  A_IN [s] :=  $\bigcup_{\substack{p \text{ is a prede-} \\ \text{cessor of } s}} \text{A\_OUT [p]}$ ;
  A_OUT [s] := GEN [s]  $\cup$  (A_IN [s] - KILL [s]);

  /* step 3 : compute exposed uses by iterating until a steady state */
  E_OUT [s] :=  $\bigcup_{\substack{t \text{ is a suc-} \\ \text{cessor of } s}} \text{E\_IN [t]}$ ;
  E_IN [s] := USE [s]  $\cup$  (E_OUT [s] - KILL [s]);

  /* step 4 : compute load ranges for all pairs of accesses in S */
  Lo ( $a_i, a_j$ ) := { s |  $a_i \in \text{A\_IN [s]} \wedge a_j \in \text{E\_IN [s]}$  }

end ComputeLoadRanges

```

Figure 4: Algorithm to compute load ranges.

In step 1, `ComputeLoadRanges` initializes the sets for the data flow analysis. For program statement s , $\text{GEN}[s]$ is the set of accesses in s , and $\text{USE}[s]$ is the set of uses in s . $\text{KILL}[s]$ is the set of accesses that, if exposed at the point after s will not be exposed at the point before s . The $\text{GEN}[s]$, $\text{USE}[s]$, and $\text{KILL}[s]$ are computed locally for each statement s . $\text{A_IN}[s]$ is the set of arriving accesses that reach the point just before s , and $\text{A_OUT}[s]$ is the set of arriving accesses that reach point just after s . $\text{A_IN}[s]$ is initially empty and $\text{A_OUT}[s]$ is initially $\text{GEN}[s]$. Similarly, $\text{E_IN}[s]$ is the set of exposed accesses at the point just before s and $\text{E_OUT}[s]$ is the set of exposed accesses at the point just after s . $\text{E_IN}[s]$ is initially $\text{USE}[s]$ and $\text{E_OUT}[s]$ is initially empty.

In step 2 of `ComputeLoadRanges`, the arriving accesses are computed using forward data flow analysis and solving the data flow equations by iterating over the statements until a steady state is reached.

Similarly, in step 3 of `ComputeLoadRanges`, the exposed uses are computed by iterating over the statements and solving the equations until a steady state is reached, except that flow is in a backward direction.

Finally, in step 4, the arriving accesses and the exposed uses are considered for each pair of accesses to determine the load ranges.

4 Register Allocation with Load/Store Ranges

Register allocation using graph coloring involves constructing the interference graph of the program, applying the spilling and coloring heuristics to the interference graph, and assigning registers to the allocated ranges.

Constructing the interference graph of a program using load/store ranges requires the usual three steps:

Step 1: compute the nodes

Step 2: compute the weights of nodes

Step 3: compute the edges between the nodes

The first step of constructing an interference graph computes the nodes representing sets of statements in the load/store ranges using the techniques described in section 3.2. The second step attaches weights to the nodes that represents the profits of allocating the nodes. To compute the profits of the ranges, we estimate that each access is executed 10^d times where d is the loop-nesting depth at which the access occurs. The profit associated with a store range is the sum of the number of definitions and uses in the store range that have been weighted by the execution frequencies of the definitions and uses. The profit associated with load range $Lo(a_i, a_j)$ is the number of times that an access-free subpath from a_i to a_j is executed. We statically estimate the profit of $Lo(a_i, a_j)$ as the quotient of the execution frequency of a_j and the number of accesses that statically reach a_j .

The third step of constructing interference graphs determines the edges between nodes by computing the interferences among the load and store ranges as follows. Load and store ranges of the same variable do not interfere with one another. The load and store ranges of distinct variables interfere with one another if they intersect at some statement. Thus, the degree of a range of a variable is the number of other variables with which the range interferes.

For a graph coloring register allocator, the computational expense of using load/store ranges instead of live ranges is the additional time required to compute the load/store ranges and the increase in allocation time due to the increased number of nodes handled by the allocator. If we assume that each statement of the program defines one result and uses at most two operands, then the number of accesses in the program is at most three times the number of definitions in the program (and at most one and a half times the number of uses in the program). Since the time required by the iterative data flow analysis algorithm is linear in the size of the set of values being propagated, the times required for computing reaching definitions, reachable uses, arriving accesses and exposed accesses are roughly similar. Thus, the time required for computing store ranges is close to that for live ranges. Although computing arriving accesses and exposed uses with the method given in steps 2 and 3 of Figure 4 is efficient, the execution time of algorithm **ComputeLoadRanges** is dominated by step 4 where all

possible pairs of accesses are examined. An enhancement to **ComputeLoadRanges** is to perform step 4 in a lazy manner in which we compute the load ranges of only those store ranges that might be spilled by the coloring/spilling heuristics of the register allocator.

The number of store ranges in a program is roughly equal to the number of live ranges in the program, but the number of load ranges in a store range can be as large as the square of the number of accesses of the variable in the store range. Though the total number of load ranges in the program is potentially large, in our experiments, we found that the number of load ranges is between two and three times the number of live ranges. Thus, the increase in register allocation time due to the use of load or store ranges is not prohibitive.

5 Experimental Results

We implemented two register allocators that are based on Chaitin's graph coloring scheme[7] with the delayed spilling enhancement[3] and the Haifa spill heuristics[2]. One allocator uses live ranges as the nodes of its interference graph and the other uses load/store ranges. We experimented with a few well known C and FORTRAN programs in order to compare the two allocators, and give our results in Tables 1 and 2. The C programs that we considered include **linpack**, which is a linear algebra kernel, **lloops**, which is a kernel containing the 14 Lawrence Livermore loops, **nsieve**, which is the sieve of Eratosthenes. We also examined the **dhrystone**, **whetstone** and **dhampstone** synthetic benchmarks, but do not report the results because both allocators produced nearly identical allocations. The FORTRAN programs that we considered are the **tomcatv** mesh generation program and the **matrix300** matrix multiplication program from the SPEC benchmarks suite. Further details of our experiments can be found elsewhere[16].

The live ranges and load/store ranges of the C programs are computed from the definition/use information obtained from a modified version[13] of the GNU C compiler **gcc**[19]. The definition/use information of the FORTRAN programs is obtained by using **f2c** to translate the programs into C[11], and then compiling the resulting C programs with the modified **gcc** compiler.

Table 1 compares the two allocators for both C and FORTRAN programs. The first and second columns contain the names of the program and its procedure respectively. The third column lists the weighted number of loads and stores in the procedure

Program	Procedure	Total Loads/ Stores	Number of Registers	Using Live Ranges	Using Lo./St. Ranges	Gain (%)
C Programs						
nsieve	main	328	1	156	136	20 12.8%
			2	91	86	5 5.5%
			4	40	36	4 10.0%
			8	0	0	0 0%
	sieve	12042	1	4696	4384	312 6.6%
			2	3491	3447	44 1.3%
			4	1087	809	278 25.6%
			8	26	35	-9 -34.6%
loops	main	190057	1	108547	105814	2733 2.5%
			2	68866	64010	4856 7.1%
			4	34138	34822	-684 -2.0%
			8	8247	6841	1406 17.0%
	init	98290	1	55988	54380	1608 2.9%
			2	35847	33445	2402 6.7%
			4	10262	8986	1276 12.4%
			8	0	0	0 0%
linpack	main	2191	1	744	722	22 3.0%
			2	305	278	27 8.9%
			4	26	29	-3 -11.5%
			8	0	0	0 0%
	matgen	10290	1	5249	4879	370 7.0%
			2	3719	3158	561 15.1%
			4	2178	1934	244 11.2%
			8	642	305	337 52.5%
	dgefa	9567	1	4665	4310	355 7.6%
			2	3728	3296	432 11.6%
			4	2156	1923	233 10.8%
			8	464	224	240 51.7%
	daxpy	2090	1	1169	1070	99 8.5%
			2	786	684	102 13.0%
			4	442	377	65 14.7%
			8	75	78	-3 -4.0%
	ddot	1940	1	1070	1003	67 6.3%
			2	699	615	84 12.0%
			4	389	335	54 13.9%
			8	54	27	27 50.0%
	dscal	1579	1	797	714	83 10.4%
			2	515	464	51 9.9%
			4	203	155	48 23.6%
			8	0	0	0 0%
FORTRAN Programs						
tomcatv	main	162852	1	63876	62390	1486 2.3%
			2	38620	36646	1974 5.1%
			4	19726	15745	3981 20.2%
			8	6629	4766	1863 28.1%
matrix300	main	6828	1	2632	2503	129 4.9%
			2	1798	1621	177 9.8%
			4	517	505	12 2.3%
			8	4	2	2 50.0%
	prnt	1688	1	982	761	221 22.5%
			2	550	465	85 15.5%
			4	257	266	9 -3.5%
			8	11	11	0 0%
	sgemv	1252	1	618	555	63 10.2%
			2	470	409	61 9.9%
			4	293	274	19 6.5%
			8	156	157	-1 -0.1%

Table 1. Allocations for C and FORTRAN programs.

Program	Procedure	Store Ranges	Load Ranges	Single Ranges	Loads/Stores	Live Ranges
nsieve	main	24	30	19	35	21
	sieve	34	80	18	96	28
lloops	main	1261	1964	1116	2109	1219
	init	163	335	130	368	147
linpack	main	376	430	353	453	368
	matgen	51	136	33	154	43
	dgefa	91	250	79	262	89
	daxpy	101	210	81	230	93
	ddot	96	189	74	211	86
	dscal	67	142	55	154	64
tomcatv	main	1037	1645	944	1738	1013
matrix300	main	203	229	192	240	204
	prnt	25	56	14	67	23
	sgemv	93	174	68	199	85

Table 2. Number of nodes in the interference graphs.

and the fourth column gives the number of general purpose registers available for allocation. We experimented with 1, 2, 4 and 8 registers to get our results. The fifth column contains the weighted number of loads and stores obtained after allocating R registers to the live ranges of the procedure; the sixth column shows the corresponding number for the allocation of load/store ranges. The last two columns show the absolute difference and the percentage difference between the two allocations respectively. In cases where the resulting number of loads/stores is small, such as `nsieve` with 8 registers, `linpack`, `matgen` with 8 registers, `matrix300`, `main` with 8 registers and `matrix300`, `saxpy` with 8 registers, the percentage gains appear extreme. However, even for other cases, the allocations with the load/store ranges usually results in a reduction in the number of loads and stores in the program.

To determine the growth in the size of the interference graph using load/store ranges, we recorded the number of nodes produced in each allocator. Table 2 compares the number of nodes in the load/store interference graphs of our test programs with the number of nodes in the live range interference graphs of the same programs. The third column in Table 2 lists the number of store ranges and the fourth lists the number of load ranges. The fifth column gives the number of store ranges that contained only one load range. We optimized the number of load/store ranges by eliminating the load ranges that were the only members of their store ranges from the interference graph. The

resulting number of load/store ranges is shown in the sixth column. The last column in Table 2 shows the number of live ranges of the procedure. Comparing the last two columns of this table reveals that the number of load/store ranges is generally less than three times the number of live ranges. Although there are a few programs where the ratio of the number of load/store ranges to the number of live ranges is higher, these are small programs with few live ranges. These results lead us to believe that, in practice, the cost of allocating load/store ranges is much less than the theoretical worst case estimates.

6 Related Work

Bernstein et al.[2] proposed that spilling of live ranges be performed only in the “busy regions” instead of over the entire program as in Chaitin’s implementation. Callahan and Koblenz[5] proposed a scheme that computes the spill costs of variables based on variable usage patterns between spills and reloads rather than the usage over the entire program. Their hierarchical graph coloring scheme first covers the program with a set of “tiles” that reflect the control flow of the program, then computes the interferences and allocates registers in the tiles in a depth first manner, and finally assigns registers in a top-down manner in the tile tree. Although this scheme is intuitively appealing, it is much more complex than Chaitin’s original scheme, and there are no theoretical or empirical results to demonstrate its effectiveness.

Chow and Hennessy presented an alternate graph coloring scheme called *priority-based coloring*[8, 17]. Nodes of the interference graph are assigned *priorities* using a *cost/size* function that is similar to Chaitin's cost/degree heuristic. Nodes are colored in order of decreasing priority, and a node that cannot be colored is split into partitions that can be colored. Although the live range splitting can produce colorings where Chaitin fails, the greedy coloring approach may split live ranges that might have been assigned a single color by Chaitin's coloring heuristic. Thus, it may introduce unnecessary register-register transfer instructions. Further, it is not clear whether the priority-based coloring produces any improvement in spilling over Chaitin's heuristic. Bernstein's team reports[2] that their modification to Chaitin's spilling heuristic universally outperforms the priority-based coloring approach, but they do not provide any statistics.

Cytron and Ferrante[9] presented an algorithm for live range splitting to improve the colorability of interference graphs. They described an algorithm that guarantees a coloring using MAXLIVE colors, where MAXLIVE is the maximum number of variables that are simultaneously live at any program statement. If the number of available registers, is less than MAXLIVE, the assignment pass is preceded by an allocation pass which uses weights on the nodes such as Chow and Hennessy's priorities. However, the technique for determining such an allocation is not described. Like the priority-based coloring, this algorithm can produce colorings where Chaitin fails, but it may also split live ranges where Chaitin's coloring heuristic would have worked.

Proebsting and Fischer[18] presented a global register allocation technique called *probabilistic register allocation* in which the spilling heuristic uses probabilities that estimate the chances of live ranges being spilled. These probabilities serve the same purpose as the $1/degree^2$, $1/(area \times degree)$ and $1/(area \times degree^2)$ components of the Haifa spilling heuristics. They concentrated on reducing the number of loads by splitting live ranges in a manner that is equivalent to our computation of load ranges. Unlike our scheme, their heuristic further splits load ranges at loop boundaries. After load ranges are allocated registers, they allocate registers to store ranges which have no spilled load ranges. It would be interesting to compare the allocations produced by their probability based heuristic for spilling with those produced by the Haifa spilling heuristics operating on load/store ranges.

Briggs et al.[4] suggested various schemes for splitting live ranges such as splitting live ranges

around loops and splitting at forward and reverse dominance frontiers[10]. They report that in spite of comprehensive experimentation, they were unable to find a live range splitting scheme that consistently produced better allocations than using live ranges.

7 Conclusions and Future Work

We have presented a new technique to split live ranges into load/store ranges that provide a finer granularity of candidates for register allocation than using live ranges. Our algorithms to compute load/store ranges are standard iterative data flow algorithms. Our experiments show that the computational cost of our technique is only moderately more than the cost of using live ranges but we get a better allocation in most cases.

In this paper, we concentrate on using live range splitting to improve the spilling of variables in regions of high register pressure. The problem of assigning registers to the live range segments and inserting the minimum possible connection statements still remains. Two possible approaches to solving this problem are to use *conservative coalescing* in which the connected segments of a live range that can be coalesced without spilling are coalesced[4], and *biased coloring* in which, when possible, a live range segment is assigned the same color as its connected segments[4].

Load subpaths are sets of statements within load ranges such that each statement (except the last one) contains exactly one control flow successor in the load subpath. Load subpaths contain statements between a use and a "most recent" access, instead of *all* most recent accesses as in the case of load ranges. Thus, load subpaths represent an even finer granularity than load ranges, and we plan to experiment with register allocators that use these.

It is well known[4, 5, 18] that moving loads and stores out of loops is beneficial. At present, load/store range analysis implicitly assumes that the loads are avoided only at uses and stores are avoided only at definitions, and hence it does not take loops into account when constructing the ranges. We plan to extend our analysis to initially assume that all live variables are used (with execution frequency 0) at all loop boundaries, and study the effects of this extension on register allocation.

Acknowledgements

We wish to thank Preston Briggs, Jeff Offutt, Todd Proebsting and Mark Smotherman for their valuable comments on various drafts of this paper.

References

- [1] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.
- [2] D. Bernstein, D. Q. Goldin, M. C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon and R. Y. Pinter, "Spill code minimization techniques for optimizing compilers", *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, Sigplan Notices*, vol. 24, no. 6, pp. 258-263, June, 1989.
- [3] P. Briggs, K. D. Cooper, K. Kennedy and L. Torczon, "Coloring heuristics for register allocation", *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, Sigplan Notices*, vol. 24, no. 6, pp. 275-284, June, 1989.
- [4] P. Briggs, K. D. Cooper and L. Torczon, "Rematerialization", *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, Sigplan Notices*, vol. 27, no. 7, pp. 311-321, June, 1992.
- [5] D. Callahan and B. Koblenz, "Register allocation via hierarchical graph coloring", *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Sigplan Notices*, vol. 26, no. 6, pp. 192-203, June, 1991.
- [6] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke and P. W. Markstein, "Register allocation via coloring", *Computer Languages*, vol. 6, pp. 47-57, January, 1981.
- [7] G. J. Chaitin, "Register allocation and spilling via coloring", *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, Sigplan Notices*, vol. 17, no. 6, pp. 98-105, June, 1982.
- [8] F. Chow and J. Hennessy, "The priority-based coloring approach to register allocation", *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 4, pp. 501-536, October, 1990.
- [9] R. Cytron and J. Ferrante, "What's in a name? The value of renaming for parallelism detection and storage allocation", *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 19-27, August, 1987.
- [10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph", *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451-490, October, 1991.
- [11] S. I. Feldman, D. M. Gay, M. W. Maimone and N. L. Schryer, *A Fortran-to-C converter*, Computing Science Technical Report No. 149, AT&T Bell Laboratories, Murray Hill NJ, November, 1990.
- [12] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1989.
- [13] M. J. Harrold and P. Kolte, "Combat: A compiler based data flow testing system", *Proceedings of the 10th Pacific Northwest Software Quality Conference*, October, 1992.
- [14] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990.
- [15] W. -C. Hsu, C. N. Fischer and J. R. Goodman, "On the minimization of loads/stores in local register allocation", *IEEE Transactions on Software Engineering*, vol. 15, no. 10, pp. 1252-1260, October, 1989.
- [16] P. Kolte, "Load/store range analysis for global register allocation," Masters Paper, Clemson University, August, 1992.
- [17] J. R. Larus and P. N. Hilfinger, "Register allocation in the SPUR Lisp compiler", *Proceedings of the ACM Symposium on Compiler Construction, Sigplan Notices*, vol. 21, no. 6, pp. 255-263, June, 1986.
- [18] T. A. Proebsting and C. N. Fischer, "Probabilistic register allocation", *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, Sigplan Notices*, vol. 27, no. 7, pp. 300-310, June, 1992.
- [19] R. M. Stallman, "Using and porting GNU CC (version 1.37.1)," Free Software Foundation, Inc., Cambridge MA, February, 1990.