IEEE Region 10 Conference, Tencon 92
11th - 13th November, 1992
Melbourne, Australia

Estimating Register Cost Using Spots

Feipei Lai, Chia-Cheng Yeh,
Dept. of Electrical Engineering
& Dept. of Computer Science&
Information Engineering
National Taiwan University
Taipei, Taiwan, R.O.C.
Email : flai@cad.ee.ntu.edu.tw

Hung-Chang Lee
Dept. of Information Management
Tamkang University, Tamsui
Taipei, Taiwan, R.O.C.

Abstract

Register allocation is a necessary component of most compilers, especially those for RISC machines. Former graph-coloring techniques [Chai 81-82] have been recognized as an effective method. However, techniques based on graph-coloring suffer from the long live range problem and the trouble of manipulating the large interference graph [Laru 86, Brig 89]. In this paper, a modified register allocation algorithm whose register cost using distinct spots is introduced. With this method, live range of a variable can be viewed as a collection of spots, which are the coordinate distances where the variables used. Together with the usage counts of a variable and the increased weight on variables in loop structures, we estimate the cost of each variable already in a register. In case a spill is not avoidable, a variable in a register with minimum cost is chosen. Primary result show that this method increase speed about 21%, in term of the number of load/store instructions, when compared with Chow's [Chow 84] graph-coloring method.

1. Introduction

Register allocation is one of the most important parts of an optimizer. The functionality of register allocation can be viewed as the process of mapping an unlimited number of variables into the finite sets of registers provided by a computer. An elegant formalization of this problem is the graph-coloring approach first used by Chaitin [Chai 81, Chai 82]. In this method, it constructs an interference graph to represent the interference between the live ranges of variables. There is an edge between two nodes if the values of the respective nodes are live at a statement in the program. The coloring problem is to assign a fixed set of colors to the nodes, subject to the constraint that two nodes connected by an edge cannot be given the same color. Each different color in the graph-coloring method represents a different register in the register allocation.

The problem of coloring graph is an NP-complete problem. In addition, many graphs cannot be colored because the minimum number of registers required is greater than the number of available registers. This introduces the spill situation, i.e., spilling code for loading and storing some temproraries. In Chaitin's method, he removes nodes that have fewer than **R** neighbors in the interference graph and the edges adjacent to these nodes since these nodes can be trivially colored. This process continues until no nodes with fewer than **R** neighbors are left.

Another method developed by Chow [Chow 83, 84, 90] is called *priority-based coloring*. In the method, each temporary is assigned a priority that is the estimated additional cost if the temporary resided in memory rather than in a register. Temporaries with more than **R** neighbors are assigned to registers in decreasing order of priority. A temporary that cannot be assigned to a register because **R** or more of its neighbors have been colored must be split and spill code introduced. A temporary is spilt by dividing the set of blocks in which the temporary is live into two sets and allocating separately in each.

However, experience [Laru86] shows that there may be some live ranges of variables that cannot be allocated a register because they have too many neighbors and cannot be spilt further. Under this case, as the paper suggested, there must be a load instruction inserted before each use of the variables and a store instruction inserted after each definition.

Other interesting refinements to the graph-coloring methods, have been proposed by [Brig 89], [Bern 89], and [Gupt 89]. [Brig 89] made a sharp observation to the internal execution flow of graph-coloring method and proposed a more efficient execution flow. [Bern 89] developed a *cleaning* technique which avoids some of the insertion of spill code in non-busy regions. [Gupt 89] presented a clique separator method to reduce the size of interference graph and thus reduce both in execution time and space.

The live time of a variable, called *live range* in tradition, is connected by many distinct *spots*. Each *spot* of a variable is the coordinate distance in which the variable is used. The coordinate distance is dertermined by the program flow and

0-7803-0849-2/92 $3.00

the separation between the statement that use the variable and the statement that *considers* register allocation. For example, see Figure 1, the live ranges of variables **a** and **b** have the same length 10, and their total usage counts are both 5. It seems that **a** and **b** are the same from these points of view. However, from the viewpoint of statement 8, it will more advantage to keep variable **b** in register rather than variable **a**, since **b** is used more times and earlier than **a** is.

Based on the observation, we present a modified algorithm that uses distinct spots. In summary, the proposed algorithm has the following features:

(1) It can present the usage count and the distribution of distinct spots where a variable appears,

(2) It exclude the construction of interference graph.

2. Definitions

This section defines some terms which our algorithm needed. Without losing generality, we assume that all the statements are three-address statements. A program can be divided into basic blocks and these basic blocks form a flow graph.

Definition 1 (Next statement) We use I_0 to denote the first statement of a program. Of course, I_0 is also the first statement to be executed. Statement X is called a ***next statement*** of statement Y if statement X is one of the statements that may be executed immediately after statement Y executed.

Definition 2 (Path & Length) A ***path*** from statement i to j is a sequence of statements $(i, S_1, S_2, ..., S_{n-1}, j)$, such that statement S_1 is a next statement of statement i, S_2 is a next statement of statement S_1, ..., and j is a next statement of statement S_{n-1}. There may be more than one path between two statements.

The ***length*** of a ***path*** is defined as the number of the statements in a sequence of the path minus one. We use $Length(i,j)$ to denote the maximum length of all the paths from statement i *to* statement j. The length of path $(S_1, S_2, ..., S_n)$ is equal to $n - 1$.

Definition 3 (Coordinate number) Each statement of a program is given a natural number, called ***coordinate number***. It can be defined by a function k, k is a mapping from S to positive integers, where S is the set of all the statements of a program. The function k is defined as

$$k(c) = Length(I_0,c),$$

where c is an element of S. Especially $k(I_0) = 1$.

The *coordinate number* of statement **x** is equal to the maximum number of statements passed by all paths from the original point to statement **x** without cycles. Thus, for any two statements **i** and **j**, if statement **i** is done before statement **j** (not due to loop), then the coordinate number of **i** is smaller than the coordinate number of statement **j**, i.e. **k(i) < k(j)**. The source and target of a branch has the same coordinate numbers at the beginning. This makes the method more fair and the estimation via the coordinate numbers information better.

Definition 4 (Weight) The weight of statement **i** is denoted as $w(i)$, and $w(i) = W^d$, where d is the loop-depth of statement i and W is a positive constant selected for weighting the executing number of a loop.

In our algorithm, we assume that every loop is executed 10 times for simplification as in [Chai 82]. That is, $w(i) = 10^d$, where **d** is the loop-depth of statement **i**.

Definition 5 (Distance & Weighted distance) The ***distance*** from statement j to i, denoted as $d(i,j)$, is defined as the absolute value of i minus j, i.e.

$$d(i,j) = abs(k(i) - k(j)).$$

The ***weighted distance*** between statement i and j is denoted as $wd(i,j)$, and it is defined as

$$wd(i,j) = w(j) \cdot d(i,j).$$

In order to present the real conditions of executing, we need some functions to describe the executing situation. Assume the program is executing statement **c**. We use a binary function $\mathbf{use_x(i,c)}$, where **i** is a statement and **x** is a variable.

$\mathbf{use_x(i,c)} = 1$ if **x** is a source variable in statement **i** and **k(i) > k(c) = 0** otherwise.

In other words, $\mathbf{use_x(i,c)} = 1$ if statement **i** will be done after statement **c**.

Definition 6 (Position Sum) The ***position sum*** of variable x is denoted as $D_x(c)$, and

$$D_x(c) = \Sigma_i\, w(i) \cdot k(i) \cdot use_x(i,c).$$

Definition 7 (Usage Count) The ***usage count*** of variable x is denoted as $N_x(c)$, and

$$N_x(c) = \Sigma_i\ w(i) \cdot use_x(i,c).$$

The definition of *usage count* is not the same with the definition in the graph-coloring method. We consider only the *future* usage at any point that considers register allocation. On the other hand, the definition of usage count in the graph-coloring method is *static*, that is, its value will not change through the algorithm.

Definition 8 (Average Distance) The

average distance of variable x is denoted as $AD_x(c)$, and

$$AD_x(c) = \Sigma_i\, wd(i,c) \;/\; N_x(c).$$

After making a reduction, we obtain

$$AD_x(c) = D_x(c) \;/\; N_x(c) - k(c).$$

Definition 9 (Estimation function) An ***estimation function*** of a variable x, denoted as $EF_x(c)$, as current executing statement being c is defined as

$$EF_x(c) = AD_x(c) \;/\; N_x(c)$$
$$= (D_x(c) \;/\; N_x(c) - k(c)) \;/\; N_x(c).$$

3. Algorithm and complexity

This section describes the proposed algorithm, and its complexity. An example to compare the graph-coloring methods with the proposed one.

The input of our method is a three-addressed-statement program. The algorithm for register allocation is divided into two passes.

PASS 1:

(A) Give each statement a *coordinate number* as defined.

(B) For each variable **v**, compute the usage count $N_v(I_0)$ and position sum $D_v(I_0)$, they are the initial conditions of the variable **v** before program execution. We use two variables N_v and D_v to store the information, that is, $N_v = N_v(I_0)$ and $D_v = D_v(I_0)$. And another variable EF_v to store the value of estimation function is assigned to the value $(D_v/N_v - k(I_0))/N_v$ as the initial value.

(C) Repeat steps **A** and **B** until the whole program has been scanned.

PASS 2:

(A) Allocating registers: We scan the program again to allocate registers to variables. Assume that the current statement is statement **c** and variable **v** is to be put in register in statement **c**, our register allocator will do the following steps sequentially:

(1) If **v** is in a register already, then the allocator leaves it alone.

(2) If there exists an empty register, then the allocator assigns the empty one to **v**.

(3) Calculate the value of estimation function for each variable which resides in register now. Choose the variable, say **y**, with the maximum of estimate function value and store it to memory. At last, the allocator assigns the register which **y** just resided in to **v**.

(B) Updating values: After scanning statement **c**, we would update the values of D_v and N_v as

$D_v = D_v - w(c) \cdot k(c)$, and
$N_v = N_v - w(c)$.

(C) Repeat steps (A) and (B) until all the statements have been scanned.

Assume that the number of program size is **N** and the total available registers is **R**. The time cost by step PASS1 in the algorithm is **O(N)** because the program is scanned two times and the amount of calculation for every statement is equal. This algorithm spends much more time in PASS2 is more complicated. It needs **O(R)** time when a spill code inserted and **O(1)** in other cases. So it costs **O(N•R)** in worst case. But the spilling is not much in most cases, the complexity is just **O(N)**. The algorithm complexity is near **O(N)** in average.

An example to illustrate the allocation procedure between graph-coloring and the proposed one is shown in Figures 2. In Figure 2.c, the interference graphs in the graph-coloring method are shown, including the graph before and after spilling the live ranges of variables. Assume that there are 4 registers available. There would be 3 times spill-out (variable **c** once and **a** twice) and variable **m** cannot reside in registers if we use the graph-coloring method of Chow. Every spill code needs a store instruction to write back to memory and a load instruction for next use. And variable **m** which cannot reside in any register has one use and one definition, two memory access statements are needed. Thus, It needs 8 (i.e. 2•3+2 = 8) memory access statements totally. But when using the proposed method, only 4 memory access statements are needed involving 2 spill-out as shown in Figure 2.b.

4. Performance measurement

The performance of the proposed algorithm was measured over the eleven programs. In the following, we will present how we measure, the results, and the comparison with Chow's method.

We choose a machine-independent intermediate language U-Code [Perk 79] as a input basis to test our algorithm. We count the number of memory-access statements (load or store statements) of MIPS-X assembly codes generated by the two distinct methods and record the results.

The results are shown in Tables 1 and 2. In Table 1, we record the number of memory access statements of benchmarks varying in the number of available registers without register allocation. The percentage of the numbers of memory access statements reduced are more than 50% in average and more than 36% for all cases.

In Table 2, we make a comparison between the results of graph-coloring used in the original U-

code compiler system and the proposed method. For these benchmarks, proposed method improves more than 21% with 6 registers available and 28% with 9 registers available on average.

5. Conclusion

During the benchmarks evaluation, we found there are several situations which illustrate why our proposed method do better than graph-coloring method.

1. The average distance introduced in proposed alogorithm can help the allocator understand the distribution of statements. It enhances the sensitivity of the register allocation.

2. The graph-coloring method allocates registers using basic blocks as fundamental units in majority. It implies that the assignment of registers cannot be changed within a basic block even though the contents in some registers become useless. The proposed one has no such problems.

3. The primary assumption of the proposed algorithm is to put all variables into registers while variables computing. It is more suitable for those machine with load/store architectures than graph-coloring method since graph-coloring assumes all variables reside in memory, and when the benefit of loading data from memory to register is no more than keeping in memory, it would keep data in memory rather than load to registers.

References

[Bern 89]D. Bernstein, D. Q. Goldin, M. C. Golumbic, H. Kraqczyk, Y. Mansour, I. Nahshon, and R. Y. Pinter, "Spill Code Minimization Techniques for Optimizing *Conference on Programming Language Design and Implementation, SIGPLAN Notices,* Vol. 24, No. 7, July 1989, pp. 258-263.

[Brig 89]P. Briggs, K. D. Cooper, K. K. Kenndy, and L. Torczon, "Coloring Heuristics for Register Allocation," *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, SIGPLAN Notices,* Vol. 24, No. 7, July 1989, pp. 275-284.

[Chai 81]G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register Allocation Via Coloring," *Computer Languages*, Vol 6, 1981, pp. 47-57.

[Chai 82]G. J. Chaintin, "Register Allocation & Spilling Via Graph Coloring," *Proceedings of the SIGPLAN '82 Symposium on Computer Construction, SIGPLAN Notices*, Vol 17, No. 6, June 1982, pp. 98-105.

[Chow 83]Fredrick C. Chow, "A Portable Machine-Independent Global Optimizer-Design and Measurements," *Technical Note* no. 83-254, Computer Systems Laboratory, Stanford University, December 1983.

[Chow 84]Frederick Chow and John Hennessy, "Register Allocation by Priority-based Coloring," *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, Vol. 19, No. 6, June 1984, pp. 222-232.

[Chow 90]Frederick Chow and John L. Hennessy, "The Priority-Based Coloring Approach to Register Allocation," *ACM Trans. Program. Lang. and Sys.*, Vol. 12, No. 3, July 1990.

[Gupt 89]R. Gupta, M. L Soffa, and T. Steele, "Register Allocation Via Clique Separators," *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, SIGPLAN Notices,* Vol. 24, No. 7, July 1989, 264-274.

[Laru 86] J. R. Larus and P. N. Hilfinger, "Register Allocation in the SPUR Lisp compiler," *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices*, Vol. 21, No. 7, July 1986, pp. 255-263.

[Perk 79]D. Perkins and R. Sites, "Machine-independent Pascal Code Optimization," *Proceedings of the SIGPLAN '79 Symposium on Compiler Construction, ACM SIGPLAN Notices*, Vol. 14, No. 8, August 1979, pp. 201-207.

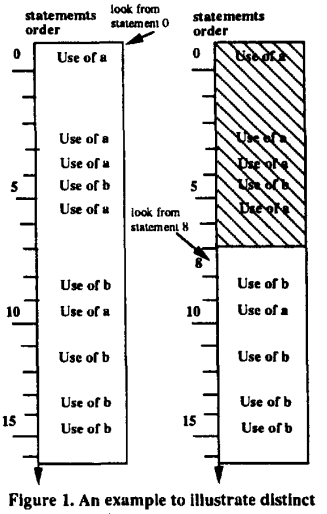

Figure 1. An example to illustrate distinct spots.

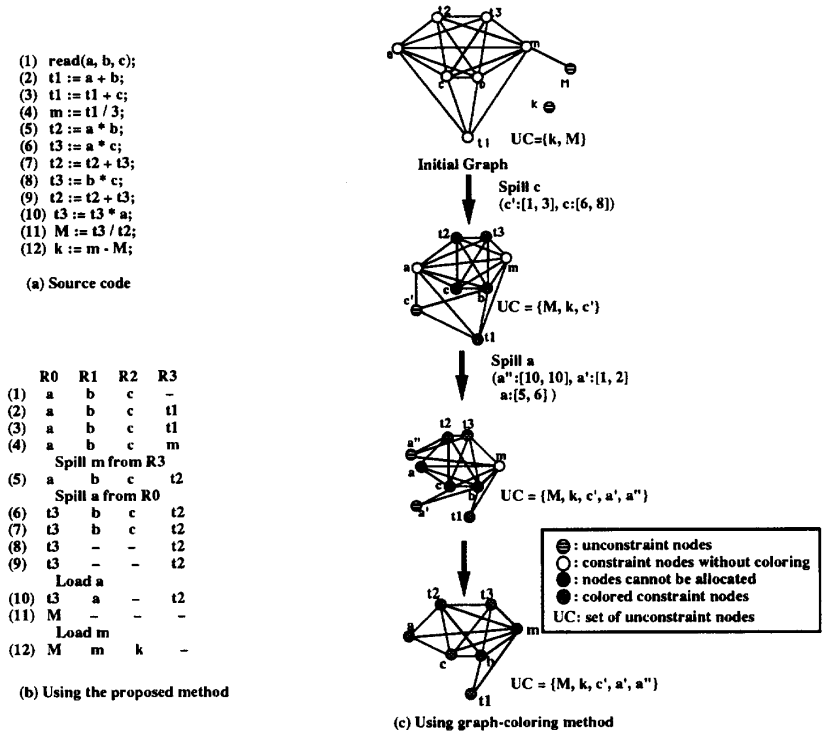

Figure 2. An example to compare the proposed algorithm and the graph-coloring one.

Table 1. Comparison between no register allocation and the proposed one.

#regisers \ benchmarks		Perm	Queen	Intmm	Dhrys	Puzzle	Shell	Bubble	Tree	Sieve	Quick	Quick2	Average
0	#Inst	82	108	114	328	523	101	110	170	47	106	151	---
6	#Inst	44	68	66	209	142	51	60	103	16	55	62	---
	Impro.	46.3%	37.0%	42.1%	36.3%	72.8%	49.5%	45.5%	65.0%	66.0%	48.1%	58.9%	51.6%
9	#Inst	44	52	66	194	125	38	56	103	14	47	49	---
	Impro.	46.3%	51.9%	42.1%	40.9%	76.1%	62.4%	49.1%	65.0%	70.20%	55.7%	67.8%	57.0%

Table 2. Comparison between graph-coloring method and the proposed one.

#regisers \ benchmarks		Perm	Queen	Intmm	Dhrys	Puzzle	Shell	Bubble	Tree	Sieve	Quick	Quick2	Average
6	gr-c	59	78	88	252	180	64	68	115	16	70	85	---
	prop	44	68	66	209	142	51	60	103	16	55	62	---
	Ratio	34.1%	14.7%	33.3%	17.1%	21.1%	25.5%	13.3%	11.7%	0.0%	27.3%	37.1%	21.4%
9	gr-c	59	72	81	244	180	46	63	115	16	60	76	---
	prop	44	52	66	194	125	38	56	103	14	47	49	---
	Ratio	34.1%	38.5%	22.7%	25.8%	44.0%	21.1%	12.5%	11.7%	14.3%	27.7%	55.1%	28.0%