MULTICS TECHNICAL BULLETIN MTB-020

DATE: December 4, 1973

TO: Distribution

FROM: R. A. Freiburghouse

SUBJECT: Register Allocation Via Usage Counts

Attached is a paper on "Register Allocation Via Usage Counts" which has been sent to communications of ACM for publication.

Register Allocation Via Usage Counts

R. A. Freiburghouse

Honeywell Information Systems Incorporated
575 Technology Square
Cambridge, Massachusetts 02139

Abstract: This paper introduces the notion of usage counts, shows how usage counts can be developed by algorithms that eliminate redundant computations, and describes how usage counts can provide the basis for register allocation. The paper compares register allocation based on usage counts to other commonly used register allocation techniques, and presents evidence which shows that the usage count technique is significantly better than these other techniques.

Keywords and Phrases: optimization, redundant computations, common subexpressions, register allocation, compilers, programming languages, virtual memory, demand paging.

CR Categories: 4.12, 4.2, 4.39

References:

1. Allen, F. Program Optimization. Annual Review of Automatic Programming, Pergamon Press, 5(1968), 239-307.

2. Busam, V., and Englund, D. Optimization of Expressions in Fortran. Comm. ACM Vol 12, 12(Dec 1969), 666-674.

3. Cocke, J. Global Common Subexpression Elimination. Proceedings of a Symposium on Compiler Optimization. SIGPLAN Notices, Vol 5, 7(July 1970), 20-24.

4. Freiburghouse, R. The Multics PL/I Compiler. AFIPS Conf. Proc. FJCC (1969), 187-199.

5. Green, P. An Implementation of SEAL on Multics. Batchelor's Thesis. Dept. of Electrical Engineering. MIT, (May 1973).

6. Gries, D. Compiler Construction for Digital Computers. John Wiley and sons, Inc., New York, (1971).

7. The Multics PL/I Language. AG94, Honeywell Information Systems Inc., (1972).

8. Horowitz, L., Karp, R., Miller, R., and Winograd, S. Index Register Allocation. Jour. ACM Vol 13, 1(Jan 1966), 43-61.

9. Lowry, E., and Medlock, C. Object Code Optimization. Comm. ACM Vol 12 1(Jan 1969), 13-22.

10. Luccio, F. A Comment on Index Register Allocation. Comm. ACM Vol 10, 9(Sept 1967), 572-574.

1. Introduction

Algorithms for eliminating redundant computations are well known and widely implemented(1,2,3,6,9). Similarly, several techniques for register allocation have appeared in the literature(2,8,9,10). This paper intoduces a very simple mechanism, the usage count, which ties these two subjects together and which provides the basis for an easily implemented technique for register allocation that is remarkably efficient.

2. Terminology

A computation is an operation f(a1,a2,...,an) which yields a result value that is determined solely by the input operands (a1,a2,...,an), and which produces no side-effects; i.e. it has no effect except to yield a result value.

A computation is redundant if there exists a previously evaluated computation which yields the same result. Since a compiler must determine the redundancy of a computation without evaluating it, a compiler generally considers a computation f(a1,a2,...,an) to be redundant if there exists a previous computation g(a1,a2,...,an) such that g(a1,a2,...,an) is always evaluated before f(a1,a2,...,an), no input operand ak has been assigned a value since g(a1,a2,...,an) was evaluated, and f and g yield the same result for a given set of input operands.

A true usage count is the number of times that a given value is used during the execution of a program. A true usage count cannot be determined without knowing exactly how many times each reference to a given value will be evaluated. Since, in general, a compiler does not have this information, it cannot develop a true usage count. Therefore, we define a usage count to be the number of distinct references to a given value in the text of a program. Usage counts are easily derived from most algorithms that eliminate redundant computations.

A linear region is a region of program which has one entry and one exit; i.e. the flow of control through the region is a straight line.

3. A Simple Algorithm

The algorithm given here is derived from an algorithm described by Gries(6). It eliminates redundant computations and develops usage counts over regions of a program that are bounded by labels.

Consider a linear representation of a program in which each operation is represented by a prefix operator and one or more operands. Each operation is represented on a single numbered line, and has an associated usage count. If an operation is not a computation as defined in section 2, its usage count is zero. If an operation is a computation, its usage count is the number of times its value is referenced in the program.

Each variable declared in the program is represented by a symbol table entry that contains an integer, S, which identifies the line that last stored a value into the variable. Initially S is zero.

"S of x" refers to the integer S stored in the symbol table entry for the variable x.

The function limit(y) returns a line number. The input operand y is either a line number or the name of a variable. If y is the name of a variable, limit(y) returns the value of S of y; otherwise, it returns:

```
max(limit(a1),limit(a2),...,limit(an))
```

where a1,a2,...,an are the operands on line y.

When y is a line number, the effect of limit(y) is to return the highest line number stored in the symbol table entry of any variable that is an operand of line y or of any line which yields a value that is input to line y.

Note that an actual implementation of this algorithm could avoid recomputing the limit for a given line by associating it with the line when that line is processed.

The program:

```
L1:  a:= b+c
```

is represented as:

```
line count operator operands
   1      0 label    L1     defines L1 as a label
   2      1 value    b      yields the value of b
   3      1 value    c      yields the value of c
   4      1 add      2,3    adds the values of lines 2 and 3
   5      1 address  a      yields the address of a
   6      0 store    5,4    stores the value of line 4 into
                            the storage addressed by line 5
```

To eliminate redundant computations and develop usage counts for programs in this representation, let the last labeled line number, L, be zero; and perform the following for each line beginning with line one.

Select the applicable case:

1. Case(the operator on this line is a store)

   1.1 Let j be the 1st operand of this line. Let x be the variable identified by the 1st operand on line j. Set S of x to the current line number.

2. Case(the operator on this line defines a label)

   2.1 Let L be the current line number. Note that an operator that defines a label does not compute a value and does not store into a variable, it only defines the occurrence of a label.

3. Case(the operator on this line does not define a label or store into a variable)

   3.1 Let k be the current line number.

   3.2 Let r be max(L,limit(a1),limit(a2),...,limit(an)) where a1,a2,...,an are the operands of the current line.

   3.3 Replace each operand that identifies a line whose operator is "use" by the operand of that line, and mark this operand as having been replaced.

   3.4 Examine each line between line r and the current line looking for a line that is equivalent to the current line. Note that the number of lines searched can be arbitrarily limited to avoid excessive search time.

   3.5 If no equivalent line was found, add one to the usage count of each line identified by an operand marked in step 3.3.

   3.6 If an equivalent line was found, replace the current line by a line whose usage count is zero, and which has the form:

       ```
       use n
       ```

       where n is the line number of the equivalent line.

Given the program:

```
a:= b+c*d
x:= b+y+c*d
```

The initial and optimized representations are given below:

Initial Representation				Optimized Representation		
line	count	operator	operands	count	operator	operands
1	1	value	c	1	value	c
2	1	value	d	1	value	d
3	1	multiply	1,2	2	multiply	1,2
4	1	value	b	2	value	b
5	1	add	3,4	1	add	3,4
6	1	address	a	1	address	a
7	0	store	6,5	0	store	6,5
8	1	value	b	0	use	4
9	1	value	y	1	value	y
10	1	add	8,9	1	add	4,9
11	1	value	c	0	use	1
12	1	value	d	0	use	2
13	1	multiply	11,I2	0	use	3
14	1	add	10,13	1	add	10,3
15	1	address	x	1	address	x
16	0	store	15,14	0	store	15,14

4. Register Allocation

To avoid recomputing a value unnecessarily, a code generator must hold the result of a computation in a register or temporary until it is no longer needed. However, to minimize the number of times that the contents of registers are stored or reloaded, a code generator must insure that it does not hold a value in a register longer than it is needed. This aspect of register allocation we call the _value retention problem_.

When there is excess demand for registers, a code generator must chose which register to load so as to minimize the number of stores and loads. This aspect of register allocation we call the _register demand problem_.

To solve these problems by means of usage counts, a code generator must maintain a model of the object program's machine state. The model is a record of which value is currently held in each register and temporary, and includes the usage count of each value so held.

As a code generator scans the program and generates instructions, it decrements the usage count of a value each time it encounters a reference to the value. When the usage count drops to zero, the register or temporary holding the value can be released for reuse. When there is an excess demand for registers, the register containing the value with the lowest usage count can be selected for loading. Its previous value need be stored only if a copy of it does not already exist in storage.

5. The Value Retention Problem

Usage counts developed for linear regions provide an optimal solution to the value retention problem. Usage counts developed for nonlinear regions, do not provide an optimal solution to the value retention problem, as is shown by the following program.

```
line count operator operands
  1     3 value     x
  2        ...                  region 1
  3     0 if-goto   y,L1
  4     0 use       1
  5        ...                  region 2
  6     0 goto      L2
  7     0 label     L1
  8        ...                  region 3
  9     0 use       1
 10     0 label     L2
```

In this program, the value of line 1 is used three times. However, in any given execution of the program it will be used only twice. A code generator following the register allocation scheme described here would retain the value of x in a register or temporary throughout line 5, even though the value will not be used. Usage counts developed for each region of the program could avoid this problem.

6. The Register Demand Problem

For linear regions in the previously described representation, the optimal solution to the register demand problem is given by:

1. Let R1,R2,...,Rn be the values contained in the registers at the time that line j is to be compiled.

2. For k=1,2,...,n search forward from line j to find the next line that references Rk. If no line references Rk, the register containing Rk is the register to load; otherwise, let Lk be the number of the line that references Rk, and continue.

3. Let m be max(L1,L2,...,Ln).

4. The register to load is the register containing the value referenced on line m.

This solution is optimal because it minimizes the number of loads by retaining in registers those values which will be referenced on the lines following any given line.

Unfortunately, the optimal solution is often not practical because it requires look-ahead and only works for linear regions. Therefore, most compilers use either the "least-recently-used" criterion or the "least-recently-loaded" criterion when selecting a register to load.

To test the relative performance of the optimal, usage count, least-recently-used, and least-recently-loaded criteria, 2500 linear regions each consisting of 20 lines were input to a program which allocated registers using each criterion. The program counted the number of loads produced using each criterion for each region.

The number of registers was fixed at two, but the number of distinct values referenced in a given region was varied between 4 and 8, thereby varying the ratio of values to registers.

To insure that the regions contained realistic patterns of references, each region was constructed by selecting a random point within the region and building a cluster of references to a given value around that point. This procedure was followed for each distinct value to be referenced in the region. The density of these clusters was varied so as to create five types of regions: regions containing very dense clusters of references, regions containing dense clusters of references, regions containing moderately dense clusters of references, regions containing loose clusters of references, and regions containing a nearly random distribution of references.

Table 1 shows that the usage count criterion generated fewer loads than either the least-recently-used or the least-recently-loaded criterion, and was remarkably close to the optimal method. Table 1 also shows that the least-recently-used criterion generated fewer loads than the least-recently-loaded criterion. The relative performance of these critera held for all types of regions.

Table 2 compares the relative performance of these criteria by giving the total number of cases in which each criterion produced the fewest loads.

The number of loads generated by the usage count criterion would have been fewer if the program had decided tie cases by means of the least-recently-used criterion, rather than the least-recently-loaded criterion.

Based on its performance in linear regions, we would expect the usage count criterion to work well in nonlinear regions, but no experimental evidence has been produced to verify this hypothesis.

Since the register demand problem is essentially the same as the demand paging problem, we would expect that a paging algorithm based on usage counts would perform better than algorithms based on program history, such as algorithms based on the least-recently-used criterion.

7. Implementation Experience

A compiler for an algol68-like language was constructed which used the register allocation algorithm given here(5). During the implementation of the compiler, usage counts were found to be a valuable check on the correct operation of the code generator. If a usage count dropped below zero or remained greater than zero, it was an indication of a compiler error.

The Multics PL/I compiler(4,7) uses a directed graph representation of programs which includes usage counts. The compiler eliminates redundant computations across an entire procedure or begin-block, and it develops a single usage count for each value. The code generator uses usage counts to determine how long to retain a value in a register or temporary, and to determine which register to store in cases of excess demand. Experience with this implementation suggests that usage counts are an effective, simple, and practical basis upon which to allocate registers and temporaries.

Table 1

Loads Generated Using Each Criterion

Region Type	Optimal	Usage Count	Least Recently Used	Least Recently Loaded
4-1	444	450	485	505
4-2	467	479	524	556
4-3	530	553	630	666
4-4	552	579	661	698
4-5	584	622	721	727
5-1	524	527	555	581
5-2	588	607	664	698
5-3	648	666	758	771
5-4	683	707	820	856
5-5	687	718	826	844
6-1	639	643	679	692
6-2	677	685	750	778
6-3	725	747	827	878
6-4	785	814	947	968
6-5	828	851	1004	1026
7-1	737	739	778	794
7-2	748	759	812	844
7-3	798	808	907	939
7-4	876	895	1030	1060
7-5	915	935	1095	1125
8-1	838	838	884	900
8-2	851	855	922	951
8-3	909	921	1034	1067
8-4	970	991	1133	1159
8-5	980	1008	1166	1191

The high-order digit of the region type is the number of distinct values referenced in the region. The low-order digit indicates how the references to the values were clustered: 1 very dense, 2 dense, 3 moderately dense, 4 loose, 5 nearly random. Each entry in the table gives the number of loads generated for 100 regions of the indicated type.

Table 2

Usage Count	Least Recently Used	Least Recently Loaded
1549	55	
1827		54
	846	277

Each row of table 2 compares two criteria by giving the number of cases in which each of the two criteria produced fewer loads than the other criterion listed in that row. The optimal method produced fewer loads than any other criterion in 284 cases. A total of 2500 cases were compared.