THE UNIVERSITY OF MICHIGAN

COMPUTING RESEARCH LABORATORY[1]

A SURVEY OF ALGORITHMS FOR REGISTER ALLOCATION IN STRAIGHT-LINE PROGRAMS

Vaclav Rajlich and M. Drew Moshier

CRL-TR-14-84

FEBRUARY 1984

Room 1079, East Engineering Building
Ann Arbor, Michigan 48109
USA
Tel: (313) 763-8000

[1]Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

A Survey of Algorithms for Register Allocation in Straight-Line Programs*

Vaclav Rajlich
M. Drew Moshier

Department of Computer and Communication Sciences
University of Michigan
Ann Arbor, MI 48109

*This survey was partially supported by a grant from IBM Corporation.

Abstract

Two models of register allocation for straight-line programs are investigated in the literature: the permutation model, and the temporary-spilling model. The problem of register allocation in the permutation model is *NP*-complete, while in the temporary-spilling model it is polynomial. However, in practice, the polynomial solution is intractable because the degree of the polynomial is too large for practical consideration. The paper surveys the results related to the temporary-spilling model. Heuristic algorithms are also surveyed.

1. Introduction

Register oriented architectures are the most commonly used computer architectures. Registers are special memory locations, which play a different role than the rest of the memory: access time to the registers is much shorter than to the other memory locations, hence registers improve the efficiency of programs; registers may also perform specific tasks which cannot be performed by the ordinary memory locations, like to hold the index value for index addressing, etc.

To improve the efficiency of a program, it is advantageous to assign certain information to the registers. If an operation produces a temporary result which will be reused, then both the time of storing and retrieving can be shortened by storing the temporary result in a register. However registers are a scarce resource; typically there are many more temporary results than available registers. Register allocation algorithms then select those candidates for which the over-all contribution to the efficiency of the program is the greatest.

In the literature, several algorithms are given which provide allocation of registers. The algorithms are usually presented in the context of an idealized computer architecture and a set of constraints, under which the algorithms work. These constraints may involve whether or not we allow permutation of the instructions, whether or not we allow storing the contents of a register even if that value will be used sometime in the future, whether or not we allow branch instructions, etc. The idealized architecture and the constraints taken together are usually referred to as a *model.* The properties and algorithms differ substantially for different models.

The models are usually grouped into three different categories, based on the programs allowed:

Local models deal with programs which are in fact a single expression. The results related to these models can be found in [AHO76a], [BUS69], [NAK67] and [SET70]. Local models are not reviewed in this survey.

Straight-line models deal with straight-line programs, i.e., programs without the control instruction.

Global models deal with general programs. Global models are not reviewed in this survey. The relevant results can be found in [AGR79], [BEA74], [CHA82], [DAY70], [HAR75], [KEN72], [LEV81] and [WAR78].

This survey deals with properties and algorithms of the straight-line models. In the literature, two different models of this category are investigated. We call them the *program permutation model* and the *temporary-spilling model.*

The program permutation (or pebbling) model is investigated in [AHO76b], [BRU76], [GLT80] and [SET75] and can be characterized in the following way: We are looking for both optimal utilization of registers, and optimal sequence of instructions in a straight-line code. It has been shown that the register allocation for this model is NP-complete [AHO76b], [SET75], and P-space complete if we allow repetition of some computations [GLT80]. This means that no effective algorithm for this model is known; all known algorithms guaranteed to find the optimal solution require an exponential number of steps. However several heuristic algorithms were published in [AHO76b], and [BRU76], and they give approximate solutions to the problem. The permutation model is not covered in the survey.

Temporary-spilling model is investigated in [FRE74], [HOR66], [KEN72] and [LUC67], and can be characterized in the following way: We shall assume that the instruction sequence cannot be changed, and are trying only to find the optimal utilization of the registers. We allow temporary storing of intermediate values in the memory. It was shown that an algorithm for this problem is polynomial, if we assume that the number of registers is constant (which is true for a given computer). However this polynomial algorithm is practically intractable, because the degree of the polynomial is too large in most instances. (It is equal to the number of registers.) Several heuristics are published in the literature [BEL66], [FRE74]. However we shall make distinction between the general results, and the specialized results which apply to the index registers only. The general case is treated in Section 2 and first part of Section 3, while the specialized index register allocation problem is treated in the second part of Section 3. Section 4 contains several heuristic algorithms.

2. Temporary spilling model

In this section, we shall deal with the *temporary spilling* model of the register allocation. The important feature of the model is the fact that the code sequence is fixed -- the algorithm is not allowed to change the sequence.

Without loss of generality, we will assume that all instructions are one-operand instructions. The rationale behind this assumption is based on the fact that it is easy to convert any other instruction set into one-operand instructions. For example, instruction $a := b \times c$ will be converted into the sequence bca' , where b stands for instruction "load b ", c stands for "load c (and execute operation $\times$)", and a' stands for "store a ". Note that store instructions will be denoted by the name of the variable followed by apostrophe $'$, while load instructions will be denoted by the name of the variable.

We will disregard the difference between the individual operations, considering only the difference between loading and storing of a variable. This simplified model is easy to investigate and it preserves all important features of the programs, and hence it is useful for practical purposes.

Intuitively, a variable either can be located in memory only (in the following definition denoted by M), or it can be located both in memory and in a register, in which case the value in the register may be the same as the value in memory (denoted R), or the values may differ (denoted D). Formally, this is defined in the following way:

2.1. Definition

Let A be a set of *variables*, then a *state* is a function $S : A \to \{R,M,D\}$. For states we will use the following alternative notation:

$S = \{a \mid S(a) = R\} \cup \{b' \mid S(b) = D\}$. Also, set A will be called the set of *load instructions,* set $A' = \{a' \mid a \text{ in } A\}$ the set of *store instructions,* and set $B = A \cup A'$ the set of *instructions.* A (straight-line) *program* is a (nonempty) string of instructions. If p is a program, then $|p|$ will denote the length of the program, and $p(i)$ will denote the i^{th} instruction.

Note that in the definition above, the "state" is defined in two different ways (as a function and as a set), and the sets A , A' , have two different meanings (as variables and instructions, respectively). This choice of the notation corresponds to [HOR66], and makes usage very flexible. From the context, it will be always obvious which particular meaning is being dealt with. These definitions are illustrated by the following example:

Example 2.1.

As an example, consider set of variables $A = \{a,b,c\}$ and program $p = aba'c'ac$, which corresponds to the sequence of instructions

```
load   a
load   b
store  a
store  c
load   a
load   c.
```

Here, $|p| = 6$, and $p(1) = a$, $p(2) = b$, $p(3) = a'$, etc. As previously remarked, we shall disregard actual operations which may be executed in the program, such as multiplication, addition, etc.

If we have a state $S = \{a,b'\}$, then $S(a) = R$ i.e., variable a is located both in a register and memory, and the value in register is the same as the value in memory; $S(b) = D$, i.e., the variable b is located both in a register and memory, and the value in register is different from the value in the memory. For variable c, we have that c, $c' \notin S$, hence $S(c) = M$, i.e., the variable c is located in the memory only.

In the next definition, we shall define the allocated program.

Definition 2.2.

Let r denote the *number of registers.* For the states we will consider, the number of variables (either with apostrophe or without) in S always will be less than or equal to r. An *allocated program* is a couple $q = <p,s>$ where p is a program, $p = p(1)\, p(2) \cdots p(M)$, and s is a sequence of states $s = s(0)\, s(1) \cdots s(M)$. Triple $q(i) = <p(i),\, s(i\text{-}1),\, s(i)>$ will be called a *step.*

The previous notions are illustrated by the following example:

Example 2.2.

Suppose we have 3 variables a,b,c and $r = 2$ registers. Then possible states are $\{a,b\}$, $\{a' ,b\}$, $\{a,c\}$, etc.

An example of an allocated program is a program $q = <p,s>$ where $p = aba'\ c'\ ac$, and $s = \{a,b\}\ \{a,b\}\ \{a,b\}\ \{a' ,b\}\ \{a' ,c' \}\ \{a' ,c' \}\ \{a' ,c' \}$. Then $s(1) = \{a,b\}$, $s(4) = \{a' ,c' \}$, etc. Steps $q(1) = <a,\ \{a,b\},\ \{a,b\}>$, $q(2) = <b,\ \{a,b\},\ \{a,b\}>$, etc.

In the next definition, we shall define the cost of an allocated program. The cost will be based on the cost of a step:

Definition 2.3.

Let a be an instruction, and S, T be the states. Then *cost of step* is a function $\text{cost}(a,S,T) > 0$. Let $q = <p,s>$ be an allocated program, for which $|p| = M$, then *cost of program* $\text{cost}(q) = \text{cost}(q(1)) + \text{cost}(q(2)) + \cdots + \text{cost}(q(M))$.

Now we can formulate the allocation problem in the following way:

Definition 2.4.

Let p be a program, then the ***temporary-spilling problem*** is to find s such that $<p,s>$ is an allocated program and cost($<p,s>$) is minimal. Sequence s is then called the ***minimal allocation.***

In [HOR66] an algorithm for the temporary-spilling problem for index registers is introduced. The algorithm is explained in the form of "rules", some of which apply to general-purpose registers, while others apply to index registers only. We explain a somewhat generalized version of the algorithm for general-purpose registers.

Algorithm 2.1.

```
Create a list of all possible states, and for each state $s$ define
        Cost $(0,s) = 0$ ;
For $i := 1$ TO $M$ DO {i.e., for every step of the program do}
        BEGIN
        Create a list of all possible states, and for each state define
                Cost $(i,s) = \min_t \{\text{Cost}\ (i-1,t) + \text{cost}\ (p(i-1),t,s) | t \text{ is a state}\}$.
        Make a link from state $s$ back to state $t$ .
        END
```

Select the state T such that Cost(M,T) is the minimum. Reconstruct the sequence $s(0), s(1), \ldots, s(M) = T$ and denote it s . Then $<p,s>$ is the allocated program with minimum cost.

Proof of the correctness of this algorithm is by induction on M .

The complexity of the algorithm is given by the following formula: Let b be the number of the variables, r the number of registers, and M the number of steps. Then the asymptotic estimate of the number of distinct states is $0(b^r)$, and the number of different steps is $0(b^{2r})$, hence the complexity of the whole algorithm is $0(M.b^{2r})$.

For a fixed computer architecture, the number of registers r is constant, and hence this is a polynomial algorithm. However the size of the exponent is such that the use of this algorithm is impractical. As an illustration, consider a situation with 16 registers, 20 variables and program of length 100; then the number of the steps of the algorithm will be approximately 4.10^{27}, which is clearly way beyond any reasonable usefulness. In [HOR66], several techniques are presented which lower the number of states the algorithm has to generate. Some of them are applicable to general-purpose registers, while others are applicable to index registers only. They are based on a more detailed investigation of the structure of the cost function, and they will be dealt with in Section 3. Another approach is based on heuristic methods, where we try to evaluate how likely a given state is to be a part of the minimal allocation, and consider only the most promising states. This approach is dealt with in Section 4.

3. Improvements in the basic algorithm

The algorithm presented in the previous section is clearly unacceptable because of its complexity. One of the ways to achieve a more acceptable algorithm is to trim the set of states which are generated by the algorithm for every step. [HOR66] provides several techniques which are repeated here. These techniques are based on a more detailed definition of the cost function, which allows improvements in the algorithm. As remarked earlier, some of them apply to the general-purpose registers, while others apply to the index registers only.

First we consider the generally applicable techniques.

Definition 3.1.

Let I be a set of integers, let c: $\{M,R,D\} \rightarrow I$ be a function called ***cost of instruction*** and let C: $\{M,R,D\} \times \{M,R,D\} \rightarrow I$ be a function called ***cost of change***. Define cost of a step as a sum

$$\text{cost}\ (a,s,t) = c(s(a)) + \sum\{C(s(x),\ t(x))|\ s(x) \neq t(x),\ x \neq a\}\ .$$

Intuitively speaking, the cost of change from one state to another consists of the cost of the instruction, plus the cost of changes of states of all individual variables. The cost of these changes is in fact the cost of extra load and store instructions, which have to be included into the program, if we want to change a variable's state.

Then we can trim the set of the states using the following "rule of minimal change":

Algorithm 3.1.

Let p be a program. Let $p(i)$ be an operation, and $s(i\text{-}1)$ be a state, then define $s(i)$ in the following way:

(a) If $p(i) = a$ and either $a \in s(i-1)$ or $a' \in s(i-1)$, then $s(i) = s(i-1)$.

(b) If $p(i) = a'$ and $a' \in s(i-1)$, then $s(i) = s(i-1)$. If $a \in s(i-1)$, then $s(i) = (s(i-1) - \{a\}) \cup \{a'\}$.

(c) If $p(i) \notin s(i-1)$, then either $s(i) = s(i-1)$ or generate all possible $s(i)$ in which one variable is replaced by $p(i)$.

Proof of the correctness of this algorithm is by induction on the size of the program.

This improvement cuts down the complexity of the algorithm. For each step, we now have $O(b^r.r)$ different possibilities, and hence the complexity of the algorithm is $O(M.b^r.r)$. This is a somewhat better, but still impractical complexity.

Another improvement is the following algorithm:

Algorithm 3.2. Let s, t be two states. Then define the cost of change from s into t to be $C'(s,t) = \sum\{C(s(x),t(x)): s(x) \neq t(x)\}$. Then if in a step i, $C(i,t) \geq C(i,s) + C'(s,t)$, then do not generate t.

While this rule again trims the number of states to be generated in a step, it still preserves the asymptotic complexity introduced above.

In [HOR66, LUC67], part of the results were oriented towards index registers. There are several important distinctions between index registers and general registers. Index registers are both read and updated in operations of the type a', while the general registers are written only. Also index values cannot be read from the memory and immediately used in an operation; they first must be loaded into a register. If we accept these restrictions, we can apply the following rule which trims the number of the states generated even further (in the case of general registers, the rule does not apply):

Algorithm 3.3.

Let i be the current step of the program and let $j > i$ be the first step after i in which either x or x' occurs, $k > i$ be the first step in which x' occurs. Denote $j-i = d(i,x)$, and $k - i = d(i,x')$. If x' never occurs after step i, set $d(i,x') = \infty$, and if x also never occurs after i, set $d(i,x) = \infty$. Then we can trim the number of the states generated by the following additional rules:

Let s, t be two states of step i, which differ exactly in one element. Let this element be a or $a' \in s$, b or $b' \in t$. Eliminate state t in any of the following cases:

(a) $a \in s$, b or $b' \in t$, $d(i,a) \leq d(i,b)$, and $C(i,s) \leq C(i,t)$.

(b) $a' \in s$, $b \in t$, $d(i,a') \leq d(i,b)$, and $C(i,s) \leq C(i,t)$.

(c) $a' \in s$, $b' \in t$, $d(i,a') \leq d(i,b)$, and $C(i,s) \leq C(i,t)$.

As a corollary, suppose $p(i) \notin s(i-1)$, a or $a' \in s(i-1)$, and b or $b' \in s(i-1)$. Then do not generate $s(i)$ by replacing a or a' if:

(a) $a,b \in s(i-1)$ and $d(i,a) < d(i,b)$

(b) $a',b \in s(i-1)$ or $a',b' \in s(i-1)$ and $d(i,a') < d(i,b)$.

In [KEN72], practical experiments with index register allocation are reported, where the number of states was trimmed by all the rules above. It is reported that the average number of states generated in a step was about 1.5, which indicates that the trimming rules are very effective for index registers.

4. Heuristic algorithms

In the previous two sections, we listed several algorithms for register allocation of both general-purpose and index registers. It is obvious that in the case of general-purpose registers, the algorithms are not very promising from the point of view of immediate usability. However there are several heuristic algorithms published in [FRE74], which are much more efficient and hence more suitable for practical use. The common characteristic of all these algorithms is that they generate just one state for each step, and they only approximate the optimal solution to a larger or lesser degree.

The first algorithm of this section is published in [BEL66] and reprinted in [FRE74]. The heuristic is based on the idea that the general-purpose registers are treated as index registers.

Algorithm 4.1.

Let p be a program. Then for each step i, include the variable $p(i)$ into the state. If the number of variables in the state becomes bigger than the number of registers, then store variable x for which $d(i,x)$ is the maximum.

The algorithm is of complexity $O(M)$ and it requires either two passes or backward scanning. It gives an optimal or near optimal solution in the situations in which $c(M) > C(M,R) + C(R,M) + c(R)$. This is certainly true for paging mechanisms and index registers, for which the algorithm was originally published, where direct reading of memory is not available and hence its cost is ∞. However for general-purpose register allocation, this is usually false, and the algorithm gives the optimal solution only sometimes. The following example contains a situation where the algorithm fails:

Example 4.1.

Let us consider a computer with two registers. In the following columns, we have the program (the left column), the states produced by Algorithm 4.1 (the central column), and the optimal solution (the right column).

	c,a	c,b
c	c,a	c,b
a	c,a	c,b
b	b,a	c,b
a	b,a	c,b
b	b,a	c,b
c	c,a	c,b
c	c,a	c,b
c	c,a	c,b

Suppose we have the following cost functions: $c(R) = 1$, $c(M) = 2$, $C(M,R) = C(R,M) = 3$. Then the cost of the allocation produced by the Algorithm 4.1 is 14, while the optimal solution costs 10.

Another heuristic algorithm published in [FRE74] is the so-called *usage count algorithm*. It is defined in the following way:

Algorithm 4.2.

Let p be a program, and let $M(i,a)$ be the number of consequent references of the current value of variable a (called usage count). Then for each step i, include the variable $p(i)$ into the state, and decrement its usage count. If the number of variables in the state becomes bigger than the number of registers, then store variable x for which $M(i,x)$ is the minimum.

The complexity of the algorithm is $O(M)$ and again it requires either two passes or backward scanning. This algorithm works well when references to variables are uniformly distributed in a program, or when large clusters of references are concentrated at the beginning of the program. If there is a large cluster at the end of the program, the algorithm gives poor results as demonstrated by the following example:

Example 4.2.

Let us again consider a computer with two registers. In the following columns, we have a program (left column), state as allocated by the Algorithm 4.2.(central column), and optimal solution (right column):

	b,c	a,b
b	b,c	a,b
a	b,c	a,b
a	b,c	a,b
b	b,c	a,b
a	b,c	a,b
b	b,c	a,b
c	b,c	b,c
c	b,c	b,c
c	b,c	b,c

Suppose we have the following costs: $C(M,R) = C(R,M) = 1$, $c(R) = 1$, and $c(M) = 2$, then the cost of the allocation by Algorithm 4.2. is 12, while for the optimal program in the right-most column it is 10.

Both previous heuristic algorithms require backward scanning or two passes through the program. However in certain applications, neither one is acceptable; hence, other criteria for register allocation must be used, as in the following algorithm:

Algorithm 4.3. (Least recently used heuristic.)

Let p be a program. For each step i, include variable $p(i)$ into the state. If number of variables in the state becomes bigger than the number of registers, then store the variable which was least recently used.

Effectiveness of this heuristic obviously strongly depends on the distribution of variable references within the program. The advantage is that very little computation is needed; hence this algorithm is very effective.

A similar general idea is the basis for the following algorithm:

Algorithm 4.4. (Least recently loaded heuristic.)

Let p be a program. For each step i, include the variable $p(i)$ into the state. If the number of variables in the state becomes bigger than the number of registers, then store the variable which was least recently loaded into a register.

Again in this case, the effectiveness of this heuristics strongly depends on the distribution of the variable references in the program. The effectiveness of this and the previous heuristics are experimentally studied in [FRE66]. The data indicate that the usage count algorithm gives results which are very close to the optimum (within 5% range of the optimum), while least recently used and least recently loaded algorithms give somewhat worse results (by approximately 10%). Of the two, the least recently used algorithm gives somewhat better results.

5. BIBLIOGRAPHY

[AGR79] Agresti, W.W.
Register Assignment in Tree-Structured Programs. "Information Sciences Journal", 18:83-94, 1979.

[AHO76a] Aho, A.V. and Johnson, S.C.
Optimal Code Generation for Expression Trees. "Journal of the ACM", 23(3):488-501, "August", 1976.

[AHO73] Aho, A.V. and Ullman, J.D.
The Theory of Parsing, Translation and Compiling, Vol. II: Compiling, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[AHO76b] Aho, A.V., Johnson, S.C. and Ullman, J.D.
Code Generation for Expressions with Common Sub-expressions. In *Third Annual Symposium on Principles of Programming* Languages, pages 19-31. SIGPLAN-SIGACT, "January", 1976.

[BEA74] Beatty, J.C.
A Register Assignment Algorithm for Generation of Highly Optimized Object Code. "IBM Journal of Research and Development", 18(1):20-39, "January", 1974.

[BEL66] Belady, L.A.
A Study of Replacement Algorithms for Virtual Storage Computers "IBM Systems Journal", 5(2):78-101,1966.

[BRE69] Breuer, M.A.
Generation of Optimal Code for Expressions via Factorization. "Communications of the ACM", 12(6):333-340, "June", 1969.

[BRU76] Bruno, J. and Sethi, R.
Code Generation for a One-Register Machine. "Journal of the ACM", 23(3):502-510, "July", 1976.

[BUS69] Busam, V.A. and Englund, D.E.
Optimization of Expressions in Fortran. "Communications of the ACM", 12(12):666-674, "December", 1969.

[CHA82] Chaitin, G.J.
Register Allocation and Spilling via Graph Coloring. "SIGPLAN Notices", 17(6):98-105, "June", 1982

[COC70] Cocke, J. and Schwartz, J.T.
Programming Languages and their Compilers. Courant Institute, New

York University, New York, 2nd ed. 1970.

[DAY70] Day, W.H.E.
Compiler Assignment of Data to Registers. "IBM Systems Journal", 9(4):281-317, 1970.

[FRE74] Freiburghouse, R.A.
Register Allocation via Usage Counts. "Communications of the ACM", 17(11):638-642, "November", 1974.

[HAR75] Harrison, W.
A Class of Register Allocation Algorithms. Research Report RC5342, "IBM Thomas J. Watson Research Center", "March", 1975.

[HOR66] Horwitz, L.P., Karp, R.M., Miller, R.E. and Winograd, S.
Index Register Allocation. "Journal of the ACM", 13(1):43-61, "January", 1966.

[KAR72] Karp, R.M.
Reducibility among Combinitorial Problems. *Complexity of Computer Computations*. Plenum Press, N.Y., pages 85-104.

[KEN72] Kennedy, K.
Index Register Allocation in Straight-line code and simple loops. *Design and Optimization of Compilers,* Rustin, R. (ed), Prentice-Hall, Englewood Cliffs, N.J. 1972, pages 51-63.

[LUC67] Luccio, F.
A comment on Index Register Allocation. "Communications of the ACM", 10(9):572-574, "September",

[NAK67] Nakata, I.
On Compiling Algorithms for Arithmetic Expressions. "Communications of the ACM", 10(8):492-494, "August", 1967.

[SET75] Sethi, R.
Complete Register Allocation Problems. "SIAM Journal of Computing", 4(3):226-248, "September", 1975.

[SET70] Sethi, R. and Ullman, J.D.
The Generation of Optimal Code for Arithmetic Expressions. "Journal of the ACM", 17(4):715-728, "October", 1970.

[WAR78] Warren, H.S.
Static Main Storage Packing Problems. *Acta Informatica,* 9:355-376,1978.

[LEV81] Leverett, B.W.
Register allocation in optimizing compilers, Ph.D. Thesis, Carnegie-Mellon University, Dept. of Computer Science, 1971.

[GLT80] Gilbert, J.R. Lengauer, T., Tarjan, R.E.,
The pebbling problem is complete in polynomial space, *(SIAM J. Comput. 9(3)* (1980):513-524.