# A compiler optimization technique

*By* Mark Finkelstein*

**A technique is presented for effecting more optimal machine code, as the result of compilation of algorithmic language programs. The concept of a 'deferred store' is introduced, and it is indicated how this concept may be used in compilation schemes. The technique is particularly effective on computers with multiple accumulators.**

(First received February 1967, and in revised form November 1967)

In this paper I should like to discuss a particular type of optimization, one which is normally performed when a program is 'hand coded', but for which there seems to be no simple way of providing optimal 'algebraic code', and which is not normally performed by the optimization phase of a compiler.

Consider the following operation: $\sum_{i=1}^{n} a_i$. If we wish to perform this computation, in ALGOL we write:

*sum*:= 0; **for** *i*:= 1 **step** 1 **until** *n* **do** *sum*:= *sum* + *a*[*i*]; (1)

Note that in most cases this computation will be performed in a rather inefficient manner; there will be an unnecessary fetch and store of *sum* for each iteration of the loop. The hand coder would avoid this by making use of a loop which formed a partial sum in the accumulator and added to it on each iteration. In a problem of this type, the inner loop compiled from ALGOL will be about 67% slower than the hand coding, requiring 5 instructions as opposed to 3 when hand coded (these numbers, of course, depend upon the instruction repertoire of the particular machine).

This paper discusses a method of compilation which will allow the above ALGOL program to be compiled into roughly the same machine code as would be written by hand. The type of optimization presented will effect considerable saving when compiling for machines which have a large number of high-speed registers.

The optimization feature which will enable us to produce compilations as indicated above is the concept of a 'deferred store'. Basically, the idea is this: suppose that the compilation has progressed to the point where the source program has been translated into a collection of nested macros, where we are now ready to generate machine code instructions. The basic rule will be to defer 'store' operations until the accumulator(s) which contains the quantity to be stored is needed for another computation. We hope that by holding the quantity in the accumulator, it will be needed for a subsequent calculation before the accumulator is needed again. Obviously, this approach will fail when compiling for a very simple computer, i.e. one which has only one accumulator and no index registers. On such a machine, the accumulator is needed for almost every instruction, and we would not be able to save a quantity in the accumulator for very long. However, on a machine such as the CDC 6600, where we have multiple accumulators and index registers, it is quite possible to hold a quantity in one of the accumulators over a period of perhaps 20 or 30 machine instructions without hampering the operation of the program over those next several instructions. That is, we suppose (and in fact it is generally the case) that most of the time, although we have 7 accumulators, we can really do with 6 (at least in the short run). There is no reason why the idea cannot be extended, to hold simultaneously two or three quantities in accumulators (or even index registers), if this will reduce the number of accesses to main memory significantly. In the example above, we can compile the inner loop in four instructions, a fetch from memory, an add, a test, and increment of index register.*

Effectively, in compiling the instruction

*sum*:= *sum* + *a*[*i*];

the store instruction which would normally be generated at the end of the compilation of the statement (the store into SUM)† will be deferred, with the quantity SUM held in the accumulator, and it will be found there when it is needed again (which will be on the next iteration of the loop). The remainder of this paper is concerned with outlining how this concept of *deferred store* could be implemented in a compiler.

In order to detail the description of the 'deferred store', we shall suppose that we are using a system which is approximately the Compiler Generator System of Computer Associates, Inc. (Cheatham and Sattley, 1964; Warshall and Shapiro, 1964). A detailed description of the CGS system can be found in (Computer Associates, 1963). Basically, the system operates as follows: The input string is 'parsed' by an analysis phase of the

* On the CDC 6600, counting instructions is not really the proper way to count, as the instructions on this machine are very heavily overlapped.

† At the suggestion of the referee, to minimize confusion, we shall adopt the convention that lower case italic letters indicate source identifiers, and upper case letters represent the quantity or location of the identifier once it has been (partially) translated into the language of the machine. Thus, our references to *sum* will be as an identifier in an ALGOL program, while our references to SUM will be to the corresponding quantity or location in memory.

* *University of California, Irvine, California.*



Downloaded from https://academic.oup.com/comjnl/article/11/1/22/424265 by guest on 19 April 2024

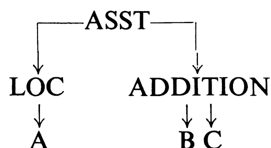
compiler, which produces sets of nested macros representing an initial translation of the various instructions of the input program. For example, the ALGOL statement

$$a := b + c;$$

might be parsed into the set of macros

(ASSIGNMENT, (LOC, A), (ADDITION, B, C)).

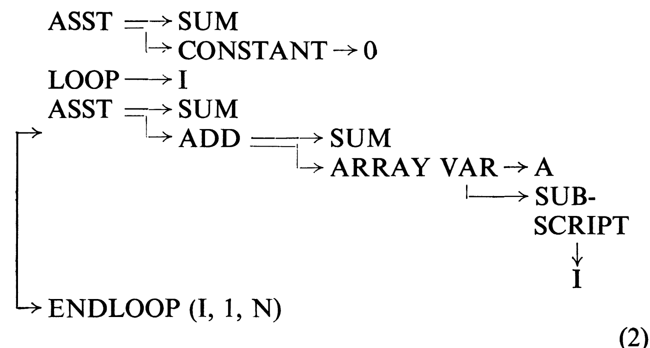
Diagrammatically,




The set of macros is then processed by an optimization phase, the result of which is again a set of macros in which various information has been added or changed. The final phase of the compiler is the code generator, which processes the macros into machine code. The code generator constructs a table as it proceeds, the table representing a simulation of the states of the special registers of the machine. Now the code generators can effect considerable saving of space and time in the compiled program if when a macro requests that a certain quantity be loaded into an index register, it checks its table and finds that the quantity need not be fetched from storage, because the quantity is already residing in another high speed register. The idea of using code generators in this syntax-directed compiler represents an improvement over the original syntax-directed compiler (Irons, 1961), because here we are no longer bound to produce stereotyped translations of macros, but may use information about the context (provided in this simulation of the registers) to produce more efficient code.

The implementation of the 'deferred store' occurs in the code generation phase in conjunction with the optimization phase. Rather than set down an algorithm, which would necessarily be quite complicated, I shall work through an example in detail, from which hopefully the reader will be able to construct an implementation of this process.

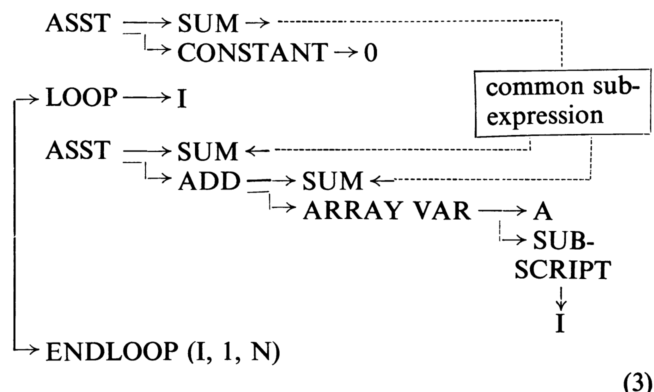
Consider the following parse of (1):




(2)

This set of macros contains essentially all the information necessary to compile (1) into machine language. In order to implement the deferred store algorithm, we shall have to recognise the common sub-expressions in this set of macros. Two expressions (in the algebraic language) are common sub-expressions if they are identical strings of symbols, and if they represent the (computation of the) same quantity. That is, whether two identical expressions are common sub-expressions (CSE's) or not depends very highly on the context in which they occur.

We suppose the optimization phase has recognised CSE's, so that (2) becomes




(3)

When the statement '$sum := 0;$' is processed, an instruction of the form 'load accumulator with constant 0' is generated, but the 'store accumulator in location SUM' (hereafter STO SUM) which would normally be generated at this point is deferred, since SUM is a CSE, and hence we may have use for it again without needing a memory reference. The fact that SUM is now residing in the accumulator and not in memory is noted. The loop control is now processed, and we assume that no instructions involving the accumulator are generated, so that we are still deferring the store instruction. When the statement '$sum := sum + a[i];$' is processed, the left-hand side is noted to be the same as the destination of the currently deferred store. It is noted, then that the deferred store may be ignored, as this new assignment statement will supercede it. The quantity SUM is noted to live only in the accumulator. In the compilation of the right-hand side, the quantity SUM is known to be in the accumulator, and consequently no fetch from memory need be made. (Indeed, no fetch from memory can be made!) The instructions for 'ADD A[I]' are generated, completing the processing of the right-hand side of the statement. To complete the processing of the left-hand side, an instruction STO SUM would normally be generated, but by our deferred store algorithm, we note again that SUM is again in the accumulator, and defer generation of the STO SUM instruction.

The end-of-loop processing is now performed. Let us suppose for the sake of completeness that the accumulator is needed to perform this processing. In this case, immediately preceding the new use of the accumulator, the instruction STO SUM would be generated, and immediately preceding the transfer to the 'top' of



Downloaded from https://academic.oup.com/comjnl/article/11/1/22/424265 by guest on 19 April 2024

the loop, where SUM is expected to be in the accumulator, the instruction 'load accumulator from SUM' would be generated. In this case we would have gained nothing, as we could not carry the deferred store to completion.

Suppose on the other hand that the end-of-loop processing does not require the accumulator. Then as we complete the loop, SUM is still in the accumulator, and at the time the transfer to the top of the loop is generated a check is made to see whether that transfer is within the region where the accumulator is expected to have the quantity SUM in it. Since it is, this transfer can be executed without losing the quantity SUM.

At the completion of the loop, code is generated for the subsequent statements. Still, the quantity SUM is in the accumulator. When the accumulator is first needed, a line of code STO SUM will be generated, to preserve the quantity SUM. Note that if the next statement following the loop should require the quantity SUM, this will be referenced in the accumulator rather than from storage, providing that this occurrence of SUM has been recognised as a CSE with the previous occurrence.

We have chosen a particularly simple example for the sake of illustration, and the reader will no doubt observe that minor perturbation of the problem will render the device of the deferred store inoperative. However, on machines with multiple accumulators the device seems to be potentially rather successful. Consider the following matrix multiplication problem:

**for** $i := 1$ **step** 1 **until** $n$ **do**
**for** $j := 1$ **step** 1 **until** $n$ **do**
**begin** $c[i, j] := 0$
**for** $k := 1$ **step** 1 **until** $n$ **do**
$c[i, j] := c[i, j] + a[i, k] \times b[k, j]$; **end**

If we compile this program for a machine such as CDC 6600, using our deferred store technique, the innermost loop would involve an accumulation in the accumulator of the products $a[i, k] \times b[k, j]$, which would then be stored in $c[i, j]$ only on completing one iteration of the '$j$-loop', at which time the address $c[i, j]$ would be changing.

The parsed program would look like **Fig. 1** (we give only the pertinent parts of the parse).

As has been observed by the referee, the machinery necessary to perform this optimization is quite great. For example to recognise common sub-expressions in all instances requires a complete flow analysis, coupled with a source language-level analyser which can determine the regions in which quantities do or do not change their values. The following two examples illustrate this point:

(I)
$$A := (B + C) \times D;$$
**if** $B = 0$ **then go to** $LABEL$;
$$D := 0$$
$$LABEL\colon E := B + C;$$

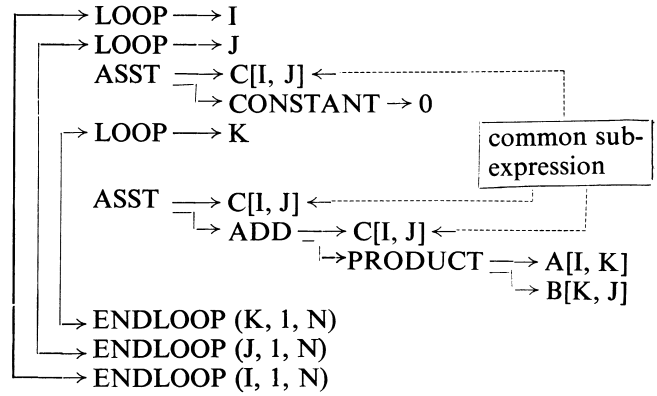



**Fig. 1**

(II)
$$A := (B + C) \times D;$$
**if** $B = 0$ **then go to** $LABEL$;
$$C := 0;$$
$$LABEL\colon E := B + C;$$

Note that in (II), '$B + C$' is not a CSE, for '$B + C$' has a different meaning in its second occurrence—$C$ having been set to 0 in the interim. In (I) '$B + C$' will be a CSE, provided that there are no other jumps to $LABEL$, or (more sophisticated) provided that all other jumps to $LABEL$ already have '$B + C$' marked for CSE, and no change to either variable in the CSE takes place.

This type of flow analysis is a difficult piece of work, and the author is not suggesting that it be undertaken solely for the purpose of implementing the deferred store. However, as the demand for more and more sophisticated compilers increases, the builders of these compilers are led more and more in this direction. The tools necessary to implement the deferred store may be close at hand within this 'software generation'. These tools and some of the ideas set forth in this paper are now being implemented in a restricted form, in the LRLTRAN compiler of the Lawrence Radiation Laboratory, Livermore, California. There, the deferred store concept is partially implemented, requiring, however, that the source language programmer specify the names of the variables which are to be kept in registers. In our example (1), it would require that we name *sum* to be kept in an accumulator. In the 'ideal implementation', this request would, of course, be generated by the optimization phase of the compiler itself.


## Acknowledgements

This work was performed while the author was employed by Control Data Corp., Palo Alto, California. I would like to express my thanks to them for their support, and particularly to Messrs. Warren Cash and Richard Bielsker for many fruitful discussions regarding this work.




Downloaded from https://academic.oup.com/comjnl/article/11/1/22/424265 by guest on 19 April 2024

**References**

CHEATHAM, T. E., Jr., and SATTLEY, K. (1964). Syntax-Directed Compiling, presented at the Spring Joint Computer Conference April 21–23, 1964. (Computer Associates document # CA-64-1-R.)

Computer Associates (1963). Compiler Generator Systems Program Descriptions. (Computer Associates document # CA-63-4-SD, 1 July, 1963.)

IRONS, E. T. (1961). A Syntax-Directed Compiler for ALGOL 60, *Communications of the ACM*, Vol. 4, pp. 51–55.

WARSHALL, S., and SHAPIRO, R. M. (1964). A General-Purpose Table-Driven Compiler, presented at the Spring Joint Computer Conference, April 21–23, 1964. (Computer Associates document # CA-63-4-R.)

---

# Errata

'Seasonal adjustment and forecasting in the presence of a trend', by R. W. Hiorns.

The ALGOL procedure presented at the end of the above paper, which appeared in Vol. 10, p. 143, is incorrect. The correct procedure is as follows.

```
procedure season (y, m, f, n, N, a, e, w, P1, P2, pred, pe);
value y, m, f, n, N, P1, P2; array y, f, a, e, pred, pe;
integer array n; integer m, N, P1, P2; real w;
comment This procedure calculates, simultaneously,
estimates of seasonal and trend constants by multiple linear
regression and provides forecasts and errors for a specified
period. The model used has the demand variable repre-
sented by a trend component, seasonal component and
random component combined additively. The trend term
is assumed to consist of a single function whose values are
supplied to the procedure in an array f[1 : N] where N is
the number of observed values of the demand variable, also
supplied, in an array y[1 : N]. These values correspond
to consecutive time periods, there being m seasons in a
year, represented by m seasonal constants, but N need not
be a multiple of m. N must satisfy N ⩾ m + 2. The
number of observed values for each season must be
supplied in the array n[1 : m].
  Estimates are left by the procedure as follows: the trend
constant in a[0] and the m seasonal constants in the
remainder of the array a[0 : m]. Standard errors for the
constants are in the array e[0 : m]. The residual variance
estimate is in w.
  Forecasts (or predictions) are made for consecutive time
periods from P1 to P2. These are left in pred[P1 : P2]
and their standard errors in pe[P1 : P2].
  It should be noted that if the values of P1 and P2 do not
both lie within the range 1 to N, then the array f will
require new bounds, the lesser of 1 and P1 and the greater
of N and P2, respectively. In any case, values must be
supplied to the whole of the array f before activation;

begin integer i, j, k; real F, YF, p, q; array T, Y[1 : m];
  YF:=F:=p:=q:=0;
    for i:=1 step 1 until m do
      begin
      k:=i; T[i]:=Y[i]:=0;
      for j:=1 step 1 until n[i] do
        begin
        F:=F+f[k]↑2; T[i]:=T[i]+f[k];
        Y[i]:=Y[i]+y[k]; YF:=YF+y[k]×f[k];
        k:=k+m
        end;
      p:=p−T[i]↑2/n[i]; q:=q−T[i]×Y[i]/n[i]
      end;
    p:=p+F; q:=q+YF;
  estimates:
    a[0]:=q/p;
    for i:=1 step 1 until m do
    a[i]:=(Y[i]−a[0]×T[i])/n[i];
  sumsquares:
    w:=0;
    for i:=1 step 1 until m do
      begin
      k:=i;
      for j:=1 step 1 until n[i] do
        begin
        w:=w+(y[k]—a[i]−a[0]×f[k])↑2; k:=k+m
        end
      end;
    w:=w/(N—m—1); e[0]:=sqrt(w/p);
    for i:=1 step 1 until m do
    e[i]:=sqrt((1+T[i]↑2/(p×n[i]))/n[i]×w);
  predictions:
    for k:=P1 step 1 until P2 do
      begin
      i:=k−m×((k−1)÷m);
      if i⩽0 then i:=i+m;
      pred[k]:=a[i]+a[0]×f[k];
      pe[k]:=w×(1/n[i]+(T[i]/n[i]−f[k])↑2/p);
      pe[k]:=sqrt (pe[k])
      end
    end of season
```



Downloaded from https://academic.oup.com/comjnl/article/11/1/22/424265 by guest on 19 April 2024