The Compilation of Loop Induction Expressions

RICHARD L. SITES
University of California, San Diego

In an optimizing compiler, it is often desirable to compile subscript expressions such as Abase−2 + I*2 so that the value of the expression is available in a register and is simply incremented whenever I is incremented, thus avoiding the multiplication inside the loop. This change is effected by a standard optimization called strength reduction. Program loops often contain several such expressions, stemming perhaps from references to operands A(I), A(I+1), B(I), and C(K, J, I, L). Under what circumstances can we do better than keeping four addresses in four separate registers or temporaries? A general technique is presented which minimizes the number of registers needed to hold such values, while simultaneously minimizing the amount of computation inside the loop. In the above collection, it is possible to use as few as two registers for the four I-dependent values.

Key Words and Phrases: register allocation, common subexpressions, optimizing compilers, code generation, computer architecture
CR Categories: 4.12, 6.2, 6.32

1. PROBLEM STATEMENT

Our discussion is motivated by a desire to have an optimizing compiler that compiles a nontrivial subscript expression inside a simple loop as a temporary containing the exact address of the array element reference on a given iteration and a simple increment of this value at the end of the loop. This transformation of expensive subscript calculations (typically involving one or more multiplications) into simple current values and increments is a standard optimization technique used in many compilers and almost universally practiced by assembly language programmers [2–4, 6]. Our concern here is not the basic strength reduction transformation, but the more subtle issues of minimizing the number of temporaries needed for a related collection of subscript expressions and minimizing the number of values which must be incremented inside the loop. The techniques described below are applicable to a wide variety of hardware architectures and are also applicable to a larger class of expressions than just subscripts.

Our discussion is phrased in terms of a multiple-register machine, such as the IBM 370, CDC 6600, Cray-1, or PDP-11, but the ideas can be used on single-register or stack machines, because any transformation which decreases the

This work was supported in part by the Los Alamos Scientific Laboratory under Contract NP8-2322E-1.

Author's address: Department of Electrical Engineering and Computer Sciences, C-014, University of California at San Diego, La Jolla, CA 92093.

2. LOOP INDUCTION EXPRESSIONS

amount of computation inside a loop will be of benefit on almost any architecture. The discussion is phrased in terms of subscript calculations, because the desire to optimize is easy to motivate and the examples are easy to follow. The same techniques apply to nonsubscript expressions.

A loop induction expression (LIE) is any expression which is a linear function of the number of iterations through portions of a loop. Assuming that I is an iteration control variable and J is a loop constant, this definition includes expressions such as 7*I + J and I*J. Also included are expressions containing variables that are conditionally incremented or incremented in more than one place. This definition is essentially the same as the definition of recursive integer polynomials in the CDC 6600 FTN Fortran compiler [5, 7] or the definition of induction variable in [1, sec. 13.5]. If A is the loop induction expression $C_1*I + C_2$, where C_1 and C_2 are loop constants, then C_1 is called the *stride* of A.

Different LIEs within a loop which have the same stride can often be combined to minimize the number of registers and increments used. For example, in the PL/I DO loop

```
DO I = 1 TO N;
  A(I) = B(I);
  END;
```

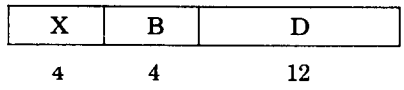
the two LIEs might be 4*I + Abase−4 and 4*I + Bbase−4, where Abase and Bbase are the origins of the respective arrays. If the first expression is kept in register R1 and the second in R2, then the machine address of B(I) is 0(R2) and that of A(I) is 0(R1), where 0 is a constant displacement or address field in some machine instruction, and R*x* specifies an index register. This style of addressing requires two registers and two updates at the end of the loop. Since the strides are the same, we could keep only the first expression in a register and address B(I) via Bbase − Abase(R1), where again Bbase − Abase is a (loop-) constant displacement. This style uses only one register and one update at the end of the loop. Presented subsequently are related strategies for using the minimal number of index registers for a group of LIEs with the same stride.

Extending the treatment in [1, sec. 13.5], we define that two expressions (involving perhaps different variables) have the same stride if the expressions are incremented by the *same* loop-constant amount (including zero) in each basic block of a loop. Thus, the expressions 4*I − 2 and 2*J + 5 have the same stride in a loop that only modifies I and J via the adjacent pair of increments I := I + 1 and J := J + 2. Detection of the common expression increment involves compile-time arithmetic on the coefficients of the linear expressions. Some existing optimizing compilers detect common strides in this way (e.g. see [6] and the examples in Table II).

3. SPECIFIC MACHINES

In this section, we examine specific rules for compiling loop induction expressions on four different architectures, the IBM 370, Cray-1, PDP-11, and CDC 6600. These four cover almost all variations on addressing architectures: double indexing, single indexing, progressive indexing, no address field, small address field, and full-size address field.

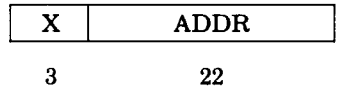
The IBM 370 addressing architecture (RX instructions) allows an index register, a base register, and a constant displacement:



Thus the 370 is a machine which allows double indexing at little or no cost. The displacement field cannot contain an entire machine address, but only a small integer in the range 0–4095. Given a group of LIEs with common stride, the minimization algorithm takes five steps:

1. For each LIE in turn, assume that it is to be kept in a register, and the other LIEs in the group are not.
2. Form the differences between the other LIEs and the chosen one. These differences are all loop constants, but may not be compile-time constants (for example, base addresses of dynamically allocated arrays are not known until execution time).
3. If all the differences are constants between 0 and 4095, then our assumption in step 1 is correct, and the group is compiled with the chosen LIE in a register, no other register used, and one increment at the end of the loop.
4. If the differences are constant, but not in range, try another LIE in an attempt to find the "front" of a group. This step is not necessary on machines which have full-size address fields.
5. Choose the LIE for which the most differences are in the range 0–4095 as the one to keep in a register. For each other LIE whose difference is either out of range or not known at compile time, put that difference in another register and use double indexing with a zero displacement. This method never uses more registers than the number of LIEs in the group and guarantees only one increment sequence for the entire group.

The Cray-1 addressing architecture allows a single-index register and a full-size address field:



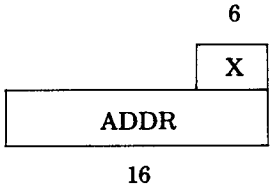
The Cray-1 is representative of a wide variety of single-index machines. Because the address field can always hold a constant difference, positive or negative (indexing uses two's complement arithmetic with no overflow detection), the algorithm is a bit simpler than for the 370:

1. Put any LIE in the group in a register. For all other LIEs in the group which have a (compile-time) constant difference from the chosen LIE, put this difference in the address field and use the same register as an index. Increment the single register at the end of the loop.
2. If any LIEs remain (differences not known at compile time), put one in another register and repeat step 1. Since the machine does not have double indexing, it is best to use a new register and to increment it at the end of the loop as well. The alternative is to assign the unknown difference to a register and when needed, add the two registers, putting the result in a third. This method uses more registers, and adding the two registers is at least as expensive as a separate increment at the end of the loop.

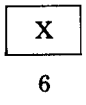
For both the 370 and Cray-1, a second-order tuning can be done. Note that our rules above always guarantee that some displacement/address field will be zero (the one associated with the chosen LIE(s)). To run loops this way, typically there is an addition of Abase outside of the loop. It is sometimes possible to save this initialization addition by shifting the range of index register contents from

(Abase + 1 through Abase + N) to (1 through N). The Abase gets buried back into the displacement/address or possibly (on the 370) into the base register.

The PDP-11 addressing architecture allows four forms which concern us here. There can be an index register and a full-size address field, as in the Cray-1:



Or there can just be an index register, which must contain exactly the address desired:



The third and fourth forms are just like the second, except the address can be incremented/decremented by 1 when used (progressive indexing, called autoincrement/autodecrement by DEC). Because of this short-form progressive indexing, there is no clear-cut strategy: if two address values A(I) and B(I) differ by exactly 100, it is possible to keep the address of A(I) in a single register and make references to 0(Rx), a short instruction, and to 100(Rx), a long instruction, or to keep two addresses in two registers and use short instructions for each reference. The autoincrement/autodecrement provides "free" updating, so the use of two registers can actually be faster. Whatever code we choose to generate, note that there is a bias toward exact addresses in registers. The PDP-11 caters to this bias by having short-form instructions which do not waste an entire address/displacement field just to say "zero offset."

The CDC 6600 addressing architecture is quite different from that of the other machines. On the 6600, addresses are formed by adding/subtracting two registers

Table I. Source Code of Two Example Fortran Subroutines

```
      SUBROUTINE T1(A,B,C)
      DIMENSION A(500), B(500), C(500)
      DO 10 I = 1, 500
10      A(I) = B(I) * C(I)
      RETURN
      END

      SUBROUTINE T2
      DIMENSION A(500), B(500), C(500), D(500)
      J = 1
      DO 20 I = 1, 500
        IF (A(I) .GT. 0) J = J + 1
20      A(I) = B(I) + C(J) * D(J)
      RETURN
      END
```

Table II. Comparison of Proposed Strategy With Existing Compiler
(The existing compilers were run at their highest optimization level. The proposed strategy usually saves registers, memory accesses, and/or increment instructions.)

Program	Machine	Compiler	#I	#R	#M	#+	LIE	Inst.	Loop Const. Register	Changing Register	Memory	None
T1	IBM/370	Fortran H	4	4	0	1	A(I)	0	Abase (1)	I*4 (2)		
							B(I)	0	Bbase (3)	I*4		
							C(I)	0	Cbase (4)	I*4		
		Proposed	4	3	0	1	A(I)	0		Abase+I*4 (1)		
							B(I)	0	Bbase-Abase(2)	Abase+I*4		
							C(I)	0	Cbase-Abase(3)	Abase+I*4		
T1	Univac1108	FTN	4	3	0	3	A(I)	0		Abase+I (1)		
							B(I)	0		Bbase+I (2)		
							C(I)	0		Cbase+I (3)		
		Proposed	(the same)									
T1	PDP-10	Fortran V	10	1	3	4	A(I)	-1		I(1)	Abase+1(1)	
							B(I)	-1		I	Bbase+1(2)	
							C(I)	-1		I	Cbase+1(3)	
		Proposed	6	3	0	3	A(I)	0		Abase+I(1)		
							B(I)	0		Bbase+I(2)		
							C(I)	0		Cbase+I(3)		
T1	CDC6600	FTN	3½	3	0	3	A(I)	0		Abase+I(1)		
							B(I)	0		Bbase+I(2)		
							C(I)	0		Cbase+I(3)		
		Proposed	(the same)									
T2	IBM/370	Fortran H	8½	5	0	3	A(I)	112	Abase-112(1)	I*4 (2)		
							B(I)	2112	Abase-112	I*4		
							C(J)	0	Cbase (3)	J*4 (4)		
							D(J)	2000	Cbase	J*4		
							J			J (5)		
		Proposed	8	2	0	2	A(I)	0		Abase+I*4(1)		
							B(I)	2000		Abase+I*4		
							C(J)	0		Cbase+J*4(2)		
							D(J)	2000		Cbase+J*4		
							J					J
T2	Univac 1108	FTN	11	2	1	3	A(I)	Abase		I (1)		
							B(I)	Bbase		I		
							C(J)	Cbase		J (2)		
							D(J)	Dbase		J		
							J				J(1)	
		Proposed	8	2	0	2	A(I)	0		Abase+I (1)		
							B(I)	500		Abase+I		
							C(J)	0		Cbase+J (2)		
							D(J)	500		Cbase+J		
							J					J
T2	PDP-10	Fortran V	8	2	0	2	A(I)	Abase		I (1)		
							B(I)	Bbase		I		
							C(J)	Cbase		J (2)		
							D(J)	Dbase		J		
							J			J		
		Proposed	(the same)									
T2	CDC6600	FTN	11	3	0	2	A(I)		Abase (1)	I (2)		
							B(I)	Bbase		I		
							C(J)	Cbase		J (3)		
							D(J)	Dbase		J		
							J			J		
		Proposed	10	2	0	2	A(I)			Abase+I (1)		
							B(I)	500		Abase+I		
							C(J)			Cbase+J (2)		
							D(J)	500		Cbase+J		
							J					J

Notes. #I: Number of "words" of instructions in the inner loop (one "word" is 30, 32, or 36 bits).
#R: Number of index registers used for LIEs in the inner loop. Other register usage is excluded. The contents of these registers are identified by numbers of parenthesis in the rightmost columns.
#M: Number of memory locations used for LIEs in the inner loop.

Notes continued.

#+: Number of LIE increments done in the inner loop, either with progressive indexing or with explicit instructions.

LIE: Loop induction expression in the inner loop. This expression value is kept in some combination of register(s), main memory, and instruction address field. Alternatively, the expression is not materialized in the loop at all, but is only reconstructed on exit from the loop. The remaining columns of the table show exactly how the expression value is maintained.

Inst: Compile-time constant part of an LIE kept in an instruction address field (also called displacement or constant field). A name in this column represents the full memory address of the corresponding variable. A zero in this column represents an address field containing all zero bits. A blank in this column represents a short-format instruction that has no address field. A zero therefore implies an architecture that wastes bits by having address fields full of zeros.

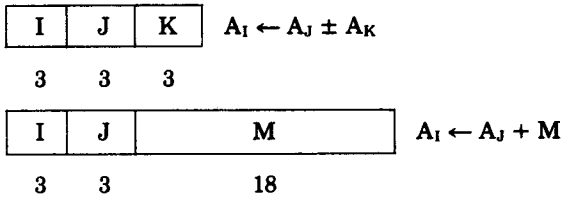
Const. Register: Part of an LIE kept in an index register that is not incremented within the loop. A nonblank entry implies an architecture with double indexing.

Changing Register: Part of an LIE kept in an index register that is incremented within the loop. The number of distinct registers in this column plus the number of LIEs incremented in memory equals the number of LIE increments done in the loop.

Memory: Part of an LIE kept in main memory.

None: Indicates an LIE that is not used inside the loop and is therefore reconstructed from some other LIE on exit.

or a register and a constant field and putting the result in a third (possibly different) register. This architecture allows double indexing,



progressive indexing, single indexing, and subtractive indexing. If an exact address is kept in a register, it can be incremented at the end of the loop either by a constant buried in an instruction, or by a constant in another register. It is better that this increment be done at a reference inside the loop, because the *only* way an address can be generated on the 6600 is via an add/subtract: using an address without changing it must be done by adding zero. So progressive indexing is "free." In addition, double indexing can be used to generate related addresses in a group, just as on the 370. One other feature of the 6600 is subtractive indexing. On a 370, the best way to generate addresses for A(I), A(I + 1), and A(I + 2) is to keep the address of A(I) in a register and refer to 0(Rx), 1(Rx), and 2(Rx) (assuming 1-byte array elements). On a 6600, the best strategy is to keep the address of A(I + 1) in a register, say A4, the constant 1 in a second register, say B1, and refer to A4 − B1, A4 + B0 (B0 is always zero), and A4 + B1. In other words, the 1 in +1 and −1 is picked up as a common subexpression.

4. INTERACTIONS BETWEEN LOOP INDUCTION EXPRESSIONS AND COMMON SUBEXPRESSIONS

The formation of loop induction expressions via strength reduction of subscript expressions needs to be done in a compiler *before* common subexpression elimination is done. Note that for A(I) = B(I)*C(I) in a loop on I, the address

expressions are transformed to three distinct values, and the I is not directly found to be common (indirectly, it is the reason that all three expressions have the same stride). What would happen if the common subexpression I were found first? The result would be a common value I kept and incremented in a register and perhaps three distinct base addresses kept invariant in three more registers (if all three arrays were parameters, or otherwise had unknown addresses) for a total of *four* registers, one increment, and a requirement for either double indexing or three more additions. This code is potentially much worse than the result of not noticing the common subexpression at all, in which case we can guarantee that no more than *three* registers be used.

Table I shows the source code for two example Fortran subroutines T1 and T2. For these, Table II shows a comparison of code generated by our proposed strategy with that generated by some existing optimizing Fortran compilers.

The IBM 370 is described above. The Univac 1108 is a single-index machine, like the Cray-1 above. In addition, the Univac 1108 has progressive indexing, so few increments require explicit instructions. The PDP-10 is a single-index machine without progressive indexing. The CDC 6600 is described above.

Subroutine T1 has three arrays as parameters, so no relationship between their addresses can be assumed at compile time. The proposed strategy saves one register compared to Fortran H, saves an increment and a register compared with the PDP-10 compiler (in addition to moving three addresses from memory to registers), and does not improve upon the CDC or Univac compilers.

Subroutine T2 has four arrays declared as local variables, so compilers can calculate the difference between two array base addresses at compile time. T2 also illustrates a conditionally-incremented variable J. Because the increments of I and J are in different basic blocks, they are assumed to have different strides (in another example, flow analysis could perhaps detect increments in different basic blocks that in fact represent common strides). In T2, J is a loop induction expression that has a common stride with C(J) on the word-addressed machines, but not on the byte-addressed IBM 370. However, the Univac 1108 compiler fails to take advantage of this, so both J and Cbase + J are stored and incremented separately in the loop. In fact, J is never used in the loop and need not be maintained at all. Instead, the value of J can be reconstructed on exit from the loop by subtracting Cbase from the register containing Cbase + J (for the IBM 370, a division by 4 or right shift of 2 bits must also be done). The reconstruction of an unused LIE value from a related one used inside a loop saves a register (or memory accesses) inside the loop, plus any time spent manipulating the LIE. The proposed strategy again usually saves registers, memory accesses, and/or increment instructions when compared with the existing compilers.

5. SUMMARY

The basic idea developed is conceptually to dedicate a separate register to each loop induction expression and to increment each expression when one of its loop induction variables is changed. Starting with this conceptual allocation, expressions with the same stride can often be combined profitably as a single "base" expression and loop-constant differences from this base. If the differences economically match the hardware architecture, then only one register and one

increment are needed inside the loop. In other cases, the scheme degrades gracefully to never needing more than the conceptual number of registers or increments. If a loop induction expression is a linear function of another and is not used inside the loop, then it can be reconstructed on exit from the loop.

ACKNOWLEDGMENTS

The ideas in this paper were developed during extended discussions with Phil Shaw and Nick Tindall. The referees' comments on an earlier version of the paper prompted the examples given in Table II. Dan Perkins, John Baird, Jeff Finger, and Douglas Albert helped to obtain the data in Table II.

REFERENCES

1. AHO, A.V., AND ULLMAN, J.D. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1977.
2. ALLEN, F.E. Program optimization. In *Annual Review in Automatic Programming, Vol. 5.* Pergamon Press, New York, 1969, pp. 239–307.
3. ALLEN, F.E., AND COCKE, J. A catalogue of optimizing transformations. In *Design and Optimization of Compilers,* R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 1–30.
4. COCKE, J., AND SCHWARTZ, J.T. Programming languages and their compilers. Courant Inst., New York, 1971.
5. FTN 4.6 internal maintenance specifications, section 64—global register assignment. Control Data Corp., Minneapolis, Minn.
6. LOWRY, E.S., AND MEDLOCK, C.W. Object code optimization. *Comm. ACM 12,* 1 (Jan. 1969), 13–22.
7. JASIK, S. Private communication.

Received October 1977; revised March 1979