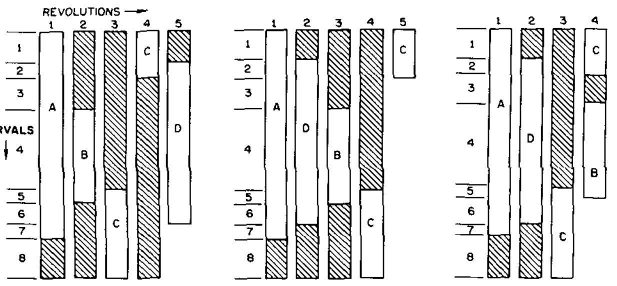
Fig. 3. Possible schedules for accessing the records described in Figure 2: from left to right, (a) a feasible schedule; (b) an SLTF schedule; and (c) the minimum latency schedule.



bottom before moving on to the next. The shaded areas in the figure show drum latency. The record label indicates the period during which a record is being accessed.

Figure 3(b) shows an SLTF schedule for the same collection. It is not difficult to show that this is the unique SLTF schedule for this example since no two records share a starting point. In general, however, when two or more distinct SLTF schedules exist, they may not all be of the same length. Note that the SLTF schedule in Figure 3(b) has less latency time than the schedule in Figure 3(a).

A Bound on the Near-Optimality of SLTF Schedules

We show that an SLTF schedule for a collection of records is never as much as one drum revolution longer than an optimal schedule.

Definition. A *critical interval* for a schedule of a collection of records is any interval in which the schedule is never latent.

Figure 3(a) has no critical intervals, whereas interval 5 is a critical interval in Figure 3(b).

LEMMA. *Every SLTF schedule has at least one critical interval.*

PROOF. Let the initial interval of the last record in the schedule be called interval *S*. We prove that this interval is a critical interval. For if not, then during some revolution prior to the last revolution, the drum is latent while it passes interval *S*. But at this point the last record is the next record to come under the read head, so it would have been scheduled here under the SLTF discipline, which proves the lemma.

Let a drum for which all information is transferred in records of equal size and whose records are located in nonoverlapping intervals on its surface be called a *paging drum* [1]. It is easy to prove that an SLTF discipline always minimizes total latency time for a paging drum. This follows as a trivial consequence of the above discussion since in every case the SLTF schedule terminates immediately after the critical interval.

In the general case, existence of a critical interval establishes near-optimality of the SLTF discipline.

THEOREM. *An SLTF schedule for a collection of records is never as much as one drum revolution longer than an optimal schedule.*

PROOF. Since an SLTF schedule has a critical interval, no schedule can pass the critical interval fewer times than the SLTF schedule. In the worst possible case, an SLTF schedule can terminate $M-1$ intervals past the critical interval, which is just one interval less than a full drum revolution. An optimal schedule might terminate just after the critical interval, but no sooner. This proves the theorem.

Figure 3(c) shows an optimal schedule for the collection of records shown in Figure 2. Inspection shows that it is not an SLTF schedule and that it is five intervals shorter than the SLTF schedule in Figure 3(b).

Received December 1971; revised November 1972

References
1. Denning, P.J. Effects of scheduling on file memory operations. Proc. 1967 AFIPS SJCC, Vol. 30, AFIPS Press, Montvale, N.J. pp. 9–21.
2. Fuller, S.H. An optimal drum scheduling algorithm. *IEEE Trans. Comput. C-21*, 11 (Nov. 1972), 1153–1165.

Short Communications
Computer Systems

Adapting Optimal Code Generation for Arithmetic Expressions to the Instruction Sets Available on Present-Day Computers

Peter F. Stockhausen
Bell Laboratories

Key Words and Phrases: arithmetic expressions, code generation, compilers, object-code optimization, register assignment, trees
CR Categories: 4.12, 5.24, 5.32

The October 1970 issue of the *Journal of the ACM* contains an article by Sethi and Ullman on the generation of optimal code. The authors took pains to make their design relevant to present-day hardware. I quote:

It is important to note that commands of the type

OP [*C*(storage), *C*(register)] → *C*(register)

are not permitted. This restriction is in keeping with the instruction sets of many present-day machines. For example, in a divide operation, the dividend is constrained to be in a register. Permitting commands of this type would have the same effect as making all operators commutative. Commands of type 4 apply an operator

The work reported here was performed at Bell Laboratories. Author's present address: Wisconsin Telephone Company, Milwaukee, WI 53200.

to the contents of two registers, leaving the result in a third (not necessarily distinct) register.

However, allowing the result of a register-register operation to be placed into any register makes their results irrelevant, since such generality is not available in most present-day machines. Instead, most machines place the result in the register containing the first operand. Thus, a division of A by B can have the result placed only in A. The Sethi-Ullman algorithm as it stands needs the more flexible instructions to handle noncommutative operators. For example, an operation of the form "divide A by B, placing the result in B" is required by their algorithm for the following expression: $W/(X - Y * Z)$.

The algorithm produces $X - Y * Z$ in the first register, W in the second, and divides the second by the first placing the result in the first register. The algorithm requires any subexpression to place its result in the first register available to it. This requirement is useful because it allows expressions such as $(\cdots) - W/(X - Y * Z)$ to be evaluated in a mechanical procedure.

I propose modifying the algorithm so that any register-register operation is defined as placing its result in the register of the first operand. Hence, the quotient of A/B will be placed into the register containing A. The Sethi-Ullman definition of register-register operations then becomes the following.

OP [C(register 1), C(register 2)] $\rightarrow$ C(register 1)

The new algorithm works with a vector b of N cells, each of which contains a register number. If $b(m)$, $b(m + 1), \ldots, b(N)$ are available for evaluating a subexpression, the result will be placed in the register specified by $b(m)$.

Algorithm for Optimal Code Generation

(1) For each node $\underset{\sim}{n}$ of the tree assign a label, $L(\underset{\sim}{n})$, from the bottom up.
(A) If $\underset{\sim}{n}$ is a leaf, set $L(\underset{\sim}{n})$ to 0 if $\underset{\sim}{n}$ is the right descendant of its parent, and to 1 if it is the left descendant or the only descendant.
(B) If $\underset{\sim}{n}$ has descendants with labels L_1 and L_2, set $L(\underset{\sim}{n})$ to the larger of L_1 and L_2, provided $L_1 \neq L_2$; if $L_1 = L_2$, set $L(\underset{\sim}{n})$ to $L_1 + 1$.

(2) Apply (3) to the root with registers $b(1)$, $b(2)$, $\ldots$, $b(N)$ available. Routine (3) will evaluate the expression represented by the subtree extending from the node to which it is applied. The result will appear in the register identified by $b(1)$.

(3) Evaluate node $\underset{\sim}{n}$ with label $L > 0$ using registers $b(m)$, $b(m+1), \ldots, b(N)$, where $1 \leq m \leq N$.

First, suppose that $L = 1$.
(A) If $\underset{\sim}{n}$ is a leaf it must be a left descendant. Generate [$\underset{\sim}{n} \rightarrow b(m)$], so that the initial storage value of $\underset{\sim}{n}$ is loaded into the register identified by $b(m)$.
(B) If $\underset{\sim}{n}$ is not a leaf, its right descendant must be a leaf, else $\underset{\sim}{n}$'s label would be at least 2. Apply (3) to its left descendant with registers $b(m)$, $b(m+1)$, $\ldots$, $b(N)$ available. The value of the left descendant will be returned in the register identified by $b(m)$; the value of the right descendant, S, must be a leaf, in storage. Generate [$b(m)$ OP $S \rightarrow b(m)$].

For $L > 1$, let the descendants of $\underset{\sim}{n}$ have labels L_1, L_2.
(C) If $L_1, L_2 \geq N$ apply (3) to the right descendant with registers $b(m)$, $b(m+1), \ldots, b(N)$ available. Store the value of the right descendant in a temporary location, $\underset{\sim}{t}$. Apply (3) to the left descendant with registers $b(m)$, $b(m+1), \ldots, b(N)$ available, obtaining the result in $b(m)$. Generate [$b(m)$ OP $\underset{\sim}{t} \rightarrow b(m)$].
(D) If $L_1 \geq L_2$ and $L_2 < N$, apply (3) to the left descendant with registers $b(m)$, $b(m+1), \ldots$, $b(N)$ available. If $L_2 > 0$, apply (3) to the right descendant with registers $b(m+1)$, $b(m+2), \ldots$, $b(N)$ available, and generate [$b(m)$ OP $b(m+1) \rightarrow b(m)$]. If $L_2 = 0$, generate [$b(m)$ OP $S \rightarrow b(m)$], where S is the storage value of $\underset{\sim}{n}$'s right descendant, which must be a leaf, in storage.
(E) If $L_1 < L_2$ and $L_1 < N$, apply(3) to the right descendant of $\underset{\sim}{n}$ with registers $b(m+1)$, $b(m)$, $b(m+2)$, $b(m+3), \ldots, b(N)$ available, obtaining the result in $b(m+1)$. Apply (3) to the left descendant with registers $b(m)$, $b(m+2)$, $b(m+3)$, $\ldots$, $b(N)$ available, obtaining the result in $b(m)$. Generate [$b(m)$ OP $b(m+1) \rightarrow b(m)$].

As an example, observe the code generated by the original and by the new algorithm for the expression $A - (B/(C - D/E))$.

	Original	Modified		Original	Modified
1.	D $\rightarrow$ 1	D $\rightarrow$ 2	5.	B $\rightarrow$ 2	B $\rightarrow$ 2
2.	1/E $\rightarrow$ 1	2/E $\rightarrow$ 2	6.	2/1 $\rightarrow$ 1	2/1 $\rightarrow$ 2
3.	C $\rightarrow$ 2	C $\rightarrow$ 1	7.	A $\rightarrow$ 2	A $\rightarrow$ 1
4.	2-1 $\rightarrow$ 1	1-2 $\rightarrow$ 1	8.	2-1 $\rightarrow$ 1	1-2 $\rightarrow$ 1

Several points should be noted. The new algorithm produces code in the same manner as the old one, with the same number of registers available at each node of the tree. The novelty of this algorithm is that it uses indirect register specification. As a result, the registers can be ordered to fit the instruction set of present day machines. To use the code produced by the old algorithm for the example above, instructions 4, 6, and 8 would have to be extended to several instructions on most current machines, making it no longer an optimal strategy.

The old algorithm was accompanied by proofs of its correctness and optimality. These proofs are still valid for the improved algorithm since the new algorithm does not violate any of the hypotheses of those proofs.

Received May 1972; revised September 1972