ISE-TR-84-43

TIME-OPTIMAL SHORT-CIRCUIT EVALUATION OF BOOLEAN EXPRESSIONS

by

Masataka Sassa

April 26, 1984

Time-Optimal Short-Circuit Evaluation of Boolean Expressions

Masataka Sassa

Institute of Information
Sciences and Electronics
University of Tsukuba
Sakura-mura, Niihari-gun
Ibaraki-ken, 305, Japan

Abstract

A method of realizing time-optimal short-circuit evaluation of boolean expressions is described. A boolean expression is assumed to be made up of boolean operators "and", "or" and "not".

Formulae are derived to estimate the expected execution time of short-circuit evaluation, given a boolean expression and an evaluation time and probability for each boolean primitive.

Using these formulae, we present a theorem to minimize the expected execution time of short-circuit evaluation by reordering the evaluation sequence of boolean subexpressions, based on laws of commutativity and associativity of "and" and "or" operations. The theorem utilizes the concept of dynamic programming.

Methods and conditions are given in an application of the theorem to code generation for programming languages.

A comparison based on an experimental implementation and some statistics from real programs are also given.

Key words

Short-circuit evaluation, Boolean expressions,
Expected execution time, Optimization, Code generation

(An early version of this report appeared in [Sassa])

1. Introduction

Boolean expressions are an important constituent of programming languages and query languages for data base systems.

Evaluation of boolean expressions by short-circuiting is a familiar method which skips over the evaluation of boolean primitives no longer relevant to the value of the expression as a whole. It often results in an attractive optimization of execution time, both in programming languages [Nakata] [Aho] [Logothetis] and in data base query languages [Gudes] [Breitbart]. However, commutativity and associativity of "and" and "or" operations, which can allow further optimization, is not taken into account in previous works.

In this paper, we focus on short-circuit evaluation of boolean expressions and present a method of realizing time-optimal short-circuit evaluation by reordering the evaluation sequence.

The boolean expressions we deal with are the following.

Definition D A boolean expression (in this paper) is composed of boolean primitives (relational expressions, boolean constants, variables or functions etc.) and of boolean operators "and", "or" and "not" (and their derivative, e.g., "**imply**").

In general, the execution time of short-circuit evaluation varies between the cases when it yields "true" and "false". We first derive formulae to estimate the expected execution time of short-circuit evaluation in each case, given a boolean expression and an evaluation time and probability for each boolean primitive of the expression.

Using these formulae, we present a theorem to minimize the expected execution time of short-circuit evaluation by reordering the evaluation sequence of boolean subexpressions, based on laws of commutativity and associativity of "and" and "or" operations. The theorem utilizes the concept of dynamic programming.

Next, we present a practical application of the theorem to code generation of programming languages. To apply the theorem, a kind of invariance under reordering must be satisfied for subexpressions. We show that this invariance is satisfied in usual forms of intermediate code and in representative actual machine architectures.

A comparison based on an experimental implementation and some statistics from real programs are also given.

Although the main purpose of this paper is a practical application to programming languages etc., we want to clarify conditions and assumptions for the proposed method so that the method can be utilized in other fields of computer applications. From this standpoint, we clarify the assumption for the formulae of the expected evaluation time and the theorem to be presented.

Assumption I For a given boolean expression, it is assumed that the evaluation time of a boolean subexpression and the probability that a boolean subexpression yields a "true"/ "false" value are independent of previously evaluated boolean subexpressions.

This assumption is related to the treatment of common sub-expressions. Optimization on common subexpressions may sometimes violate the above mentioned independency. Dealing with optimization in such a case is reserved for future studies.

In this paper, we discuss only the short-circuit evaluation method. However, by simply estimating the expected evaluation time, we can easily choose the best evaluation method between the traditional method (by "logical and" and "logical or" operations) and the short-circuit method with or without reordering the evaluation sequence.

2. Formulae for the expected time of short-circuit evaluation

Assume that a given boolean expression is analyzed and a syntax tree is constructed. If the same binary operators come in succession in the boolean expression, it should be represented as a single <u>n-ary</u> operator node with multiple sons owing to the law of associativity. The leaf nodes are boolean primitives. An example is shown in Fig. 1.

When a syntax tree of a boolean expression is given, the short-circuit evaluation sequence (from left to right) is determined uniquely. We show that the expected execution time of a short-circuit evaluation can be computed in a bottom-up manner by applying the formulae to be presented shortly to each node of the syntax tree. (As for execution time, we can either use the strict instruction time or approximate it by the number of code steps.)

In the following, for notational convenience, we sometimes identify a node or a (sub)tree of the syntax tree with its corresponding (sub)expression.

For a node b_j of a syntax tree, let

p_j be the probability that the evaluation of (the subexpression corresponding to) node b_j yields "true", and

T_j^t and T_j^f be the expected evaluation time of node b_j in cases it yields "true" and "false", respectively.

We also use $\bar{p}_j = 1-p_j$ for notational convenience.

Thus, T_j, the expected evaluation time of node b_j (regardless of its "true"/"false" value), becomes

$$T_j = p_j T_j^t + \bar{p}_j T_j^f$$

Formulae for "and" nodes (Fig. 2(a))

For an "and" node b with n sons b_i's (i=1,...,n), assume that p_i, T_i^t, T_i^f and T_i (i=1,...,n) are given as the probability and expected evaluation times of b_i. Then the formulae for p, T^t, T^f and T of the "and" node b become as follows. (The derivation of the formulae is given in Appendix A1.)

$$p = p_1 p_2 \cdots p_n = \Pi_{i=1}^{n} p_i \tag{1}$$

$$T^t = T_1^t + T_2^t + \ldots + T_n^t = \Sigma_{i=1}^{n} T_i^t \tag{2}$$

$$T^f = [\ \Sigma_{i=1}^{n} [(\Pi_{j=1}^{i-1} p_j)\bar{p}_i \{(\Sigma_{j=1}^{i-1} T_j^t) + T_i^f\}]\ ]/\bar{p} \tag{3}$$

$$\text{or } T^f = [\ \bar{p}_1 T_1^f + p_1\bar{p}_2 T_2^f + p_1 p_2 \bar{p}_3 T_3^f + \ldots + p_1 \cdots p_{n-1}\bar{p}_n T_n^f + (p_1 - p) T_1^t + (p_1 p_2 - p) T_2^t + \ldots + (p_1 \cdots p_{n-1} - p) T_{n-1}^t\ ]/\bar{p}$$

$$= [\ \Sigma_{i=1}^{n} (\Pi_{j=1}^{i-1} p_j)\bar{p}_i T_i^f + \Sigma_{i=1}^{n} (\Pi_{j=1}^{i} p_j - p) T_i^t\ ]/\bar{p} \tag{3'}$$

(n-th term is 0)

$$(= (T - pT^t)/\bar{p}\ )$$

$$T = pT^t + \bar{p}T^f$$

$$\text{or } T = T_1+p_1T_2+p_1p_2T_3+\dots+p_1p_2\dots p_{n-1}T_n = \Sigma_{i=1}^{n}(\Pi_{j=1}^{i-1}p_j)T_i \quad (4)$$

T^f can be computed more easily by $T^f = (T - pT^t)/\bar{p}$ using p, T^t and T given in (1), (2) and (4), respectively.

Note that T for an "and" node depends only on the p_i's and T_i's of its sons, and not explicitly on T_i^t's and T_i^f's.

Recall also Assumption I which states that probability p_i and expected evaluation times T_i^t, T_i^f and T_i are independent of previously evaluated subexpressions. The regularity of the formulae is due to this assumption.

Formulae for "or" nodes (Fig. 2(b))

Since an "or" node has duality with an "and" node, the formulae can be easily derived by interchanging the "true" and "false" cases, i.e., p_i<->$\bar{p}_i$, p<->$\bar{p}$, and T_i^t<->T_i^f.

$$\bar{p} = \bar{p}_1\bar{p}_2\dots\bar{p}_n, \quad \text{i.e., } p = 1-\bar{p}_1\bar{p}_2\dots\bar{p}_n = 1-\Pi_{i=1}^{n}\bar{p}_i$$

$$\begin{aligned} T^t &= [\ (\bar{p}_1-\bar{p})T_1^f+(\bar{p}_1\bar{p}_2-\bar{p})T_2^f+\dots+(\bar{p}_1\bar{p}_2\dots\bar{p}_{n-1}-\bar{p})T_{n-1}^f \\ &\quad +p_1T_1^t+\bar{p}_1p_2T_2^t+\bar{p}_1\bar{p}_2p_3T_3^t+\dots+\bar{p}_1\bar{p}_2\dots\bar{p}_{n-1}p_nT_n^t\ ]/p \\ &= [\ \Sigma_{i=1}^{n}(\Pi_{j=1}^{i}\bar{p}_j-\bar{p})T_i^f + \Sigma_{i=1}^{n}(\Pi_{j=1}^{i-1}\bar{p}_j)p_iT_i^t\ ]/p \end{aligned}$$

(n-th term is 0)

$$(\ = (T - \bar{p}T^f)/p\ )$$

$$T^f = T_1^f+T_2^f+\dots+T_n^f = \Sigma_{i=1}^{n}T_i^f$$

$$T = T_1+\bar{p}_1T_2+\bar{p}_1\bar{p}_2T_3+\dots+\bar{p}_1\bar{p}_2\dots\bar{p}_{n-1}T_n = \Sigma_{i=1}^{n}(\Pi_{j=1}^{i-1}\bar{p}_j)T_i$$

Formulae for "not" nodes (Fig. 2(c))

Considering that no evaluation corresponding to a "not" operator is necessary in the short-circuit method and that a "not" operator complements "true" and "false", formulae for a "not" node are as follows.

$$p = \bar{p}_1$$

$$T^t = T_1^f$$

$$T^f = T_1^t$$

$$T = pT^t + \bar{p}T^f = \bar{p}_1T_1^f + p_1T_1^t = T_1$$

Example 1 As an example for a boolean expression, we take (eoln **or** ch=tab **or** ch=' ') **and** flag

Its syntax tree with the values of p's and T's attached is shown in Fig. 3(a) with the following values assumed for boolean primitives or leaf nodes.

node no. i	bool. prim.	p_i	T_i^t	T_i^f	T_i
1	eoln	0.05	5	5	5
2	ch=tab	0.1	3	3	3
3	ch=' '	0.3	3	3	3
4	flag	0.5	3	3	3

(<u>Notes</u> Estimation of probabilities for boolean primitives will be discussed in section 6. The evaluation time is approximated here by the number of steps of code. The number of steps is for the IBM/370 series machine. T_i^t and T_i^f of a boolean primitive often become equal for a reason to be shown in section 4.)

Calculation of p's and T's is made bottom-up. First, calculation for the "or" node (node no. 5) is as follows.

$$p_5=1-\bar{p}_1\bar{p}_2\bar{p}_3=1-0.95\text{x}0.9\text{x}0.7=0.4015$$

$$T_5^t=[(\bar{p}_1-\bar{p}_5)T_1^f+(\bar{p}_1\bar{p}_2-\bar{p}_5)T_2^f+p_1T_1^t+\bar{p}_1p_2T_2^t+\bar{p}_1\bar{p}_2p_3T_3^t]/p_5$$
$$=[(0.95-0.5985)\text{x}5+(0.95\text{x}0.9-0.5985)\text{x}3$$
$$+0.05\text{x}5+0.95\text{x}0.1\text{x}3+0.95\text{x}0.9\text{x}0.3\text{x}3]/0.4015=9.543$$

(can be calculated more easily by $(T_5-\bar{p}_5T_5^f)/p_5$)

$$T_5^f=T_1^f+T_2^f+T_3^f=5+3+3=11$$

$$T_5=T_1+\bar{p}_1T_2+\bar{p}_1\bar{p}_2T_3=5+0.95\text{x}3+0.95\text{x}0.9\text{x}3=10.415$$

$(=p_5T_5^t+\bar{p}_5T_5^f)$

Next, calculation for the "and" node (node no. 6) is as follows.

$$p_6=p_5p_4=0.20075$$

$$T_6^t=T_5^t+T_4^t=9.543+3=12.543$$

$$T_6^f=[\bar{p}_5T_5^f+p_5\bar{p}_4T_4^f+(p_5-p_6)T_5^t]/\bar{p}_6$$
$$=[0.5985\text{x}11+0.4015\text{x}0.5\text{x}3+(0.4015-0.20075)\text{x}9.543]$$
$$/(1-0.20075)=11.388 \qquad (= (T-p_6T_6^t)/\bar{p}_6)$$

$$T_6=T_5+p_5T_4=10.415+0.4015\text{x}3=11.620 \quad (= p_6T_6^t+\bar{p}_6T_6^f)$$

3. Minimizing expected execution time by evaluation reordering

Utilizing the commutativity and associativity of "and" and "or" operations, we can minimize the expected execution time of short-circuit evaluation by reordering the evaluation sequence. The reordering is made in a bottom-up manner using the syntax tree. For example, we can minimize the expected evaluation time of the boolean expression of Fig. 3 (a) by reordering the sub-expressions as in Fig. 3 (c). (The details will be given in Example 2 later).

In order to assure that this reordering is possible, the boolean expression must satisfy the following.

Assumption A The program semantics of the boolean expression is not altered by reordering the short-circuit evaluation sequence.

This assumption is related to side-effects. Since the validity of this assumption depends on the specific program and also on the programming language used, we leave its discussion to a later section.

Under this assumption, a theorem can be proved if the following condition holds.

Condition C For each son i of an "and" or "or" node in the syntax tree of the given boolean expression, p_i and T_i are invariant under reordering.

This condition states that the probability and the expected evaluation time of a son do not depend on its position among sons of an "and"/"or" node. For example, in the syntax tree of Fig. 3(b) which is made by reordering one in Fig. 3(a), Condition C is satisfied if $p_1=0.05$, $T_1^t=5$, $T_1^f=5$, etc. still hold in Fig. 3(b).

If Assumption A and Condition C are satisfied, the following theorem holds.

Minimization Theorem If Assumption A and Condition C are satisfied, then the following statements hold in the syntax tree of a boolean expression as defined in Definition D.

(i) For an "and" node with n sons, the expected evaluation time T of (the subexpression corresponding to) that node by the short-circuit method is minimized by reordering its sons in the order

$$i_1, i_2, \ldots, i_n$$

such that

$$\frac{T_{i_1}}{1-p_{i_1}} \le \frac{T_{i_2}}{1-p_{i_2}} \le \ldots \le \frac{T_{i_n}}{1-p_{i_n}} \qquad (5)$$

(ii) Similarly for an "or" node, the expected evaluation time T is minimized by reordering its sons in the order

$$i_1, i_2, \ldots, i_n$$

such that

$$\frac{T_{i_1}}{p_{i_1}} \le \frac{T_{i_2}}{p_{i_2}} \le \ldots \le \frac{T_{i_n}}{p_{i_n}}$$

(iii) For a "not" node, which has only one son, reordering is meaningless.

(Proof) The proof is given in Appendix A2. It can be considered as an application of dynamic programming.

Informally speaking, the strategy of the above reordering is the following. For an "and" node, first evaluate sons with a high probability of yielding a "false" value and with a low evaluation time so that the rest of the sons may be quickly skipped over.

By applying the reordering in a bottom-up manner in a syntax tree, we can minimize the expected time for the short-circuit evaluation of the whole syntax tree. Note that only p_i's and T_i's (not T_i^t's and T_i^f's) are necessary for applying reordering.

Example 2 Consider the boolean expression of Example 1 (Fig. 3(a)). For the subexpression corresponding to the "or" node (node 5) which has 3 sons, six different evaluation orders can be considered. The T_5's corresponding to these orders constitute a lattice as shown in Fig. 4 for each evaluation order. $T_5(1,2,3)$ corresponds to Fig. 3(a), and $T_5(3,2,1)$ corresponds to Fig. 3(b) and (c). The best evaluation order of sons of the "or" node is (node 3, node 2, node 1) (Fig. 3(b)) since

$$\frac{T_3}{p_3} = \frac{3}{0.3} < \frac{T_2}{p_2} = \frac{3}{0.1} < \frac{T_1}{p_1} = \frac{5}{0.05}$$

Next, the best evaluation order of sons of the "and" node (node 6) is (node 4, node 5) (Fig. 3(c)) since

$$\frac{T_4}{1-p_4} = \frac{3}{0.5} < \frac{T_5}{1-p_5} = \frac{8.25}{0.5985}$$

As a whole, the expected evaluation time of the expression is improved from T_6=11.620 (Fig. 3(a)) to T_6=7.125 (Fig. 3(c)).

Canonical forms of a syntax tree

In the above reordering, use of an n-ary "and"/"or" operator node is essential. For example, if only binary operator nodes are used, the syntax tree of Fig. 5(a) which corresponds to

b_2 **and** (b_1 **and** b_3)

can not be reordered into the syntax tree of Fig. 5(b), i.e.,

b_1 **and** b_2 **and** b_3

Moreover, a sequence of apparently different operators may be transformed into a single n-ary operator. For example, the syntax tree of Fig. 5(c), i.e.,

b_1 **or not** (b_2 **and** b_3)

can be transformed into that of Fig. 5(d), i.e.,

b_1 **or not** b_2 **or not** b_3

Therefore, in order that the bottom-up application of the reordering can truly realize minimum evaluation time of the whole syntax tree, the syntax tree must be represented in some kind of canonical form.

An example of a canonical form is to represent the syntax tree using only "and" and "not" as boolean operators. This is achieved by utilizing de Morgan's law. We could use only "or" and "not" as boolean operators as well.

Another canonical form is to use a syntax tree such that in the path from the root node to any leaf node (i.e., boolean primitive) the "and" and "or" nodes strictly alternate, except for the node prior to the leaf node which may be a "not" node.

Example 3 For the following boolean expression,

(eoln **or not** (ch<>tab **and** ch<>' ')) **and** flag the first canonical form is Fig. 6(a) which corresponds to

```
          not (not eoln and ch<>tab and ch<>' ') and flag
```

and the second canonical form is Fig. 6(b) which corresponds to

```
          (eoln or not ch<>tab or not ch<>' ') and flag
```

This may be reduced to another second canonical form of Fig. 3(a) which corresponds to

```
          (eoln or ch=tab or ch=' ') and flag
```

The minimization theorem can be applied to both canonical forms, and yields the same result.

4. Application of the Minimization Theorem to code generation for programming languages

In the previous section, we have shown that the Minimization Theorem holds under Condition C. Here, we concentrate on compilation of programming languages and present a practical method of optimal code generation. We will show that under some natural premises, our method can satisfy Condition C.

4.1 Intermediate code

First, we define the class of intermediate code to be dealt with as satisfying the following.

Premise P1 We generate short-circuit code corresponding to an intermediate code (input to the code generator) of the form

```
          if b then goto l        , or
          if not b then goto l
```

where b is a boolean expression. (b is not restricted to a boolean primitive.)

This means that in the object code of a boolean expression one of the "true"/"false" exits falls to the position immediately following the relevant code (this position is often called the "fall-thru" position [Logothetis]).

A large class of intermediate code falls into this category. For example, the intermediate code for conditional and loop statements will be

```
        if b then $s_1$ else $s_2$
        -> if not b then goto $l_1$ ; $s_1$ ; goto $l_2$ ; $l_1$: $s_2$ ; $l_2$:
        while b do s
        -> $l_1$: if not b then goto $l_2$ ; s ; goto $l_1$ ; $l_2$:
```

and that of assignment statements to a boolean variable will be

```
        v := b
        -> if not b then goto $l_1$ ; v:=true ; goto $l_2$ ;
           $l_1$: v:=false ; $l_2$:
    or, -> v:=false ; if not b then goto l ; v:=true ; l:
```

4.2 Short-circuit code for n-ary boolean operators

Under Premise P1, we can generalize the usual generation method of short-circuit object code for binary boolean operators [Nakata] and get a method for n-ary boolean operators as follows.

Let O(b,c,l) be the short-circuit object code for

if b = c **then goto** l

where b is a boolean expression (not restricted to a boolean primitive), c is either "true" or "false", and l is a label. (We sometimes identify a boolean expression b with the corresponding subtree in the syntax tree.) O(b,c,l) can be defined recursively, i.e., top-down in a syntax tree, as in Table 1, and finally as in Table 2 for boolean primitives.

If no reordering is allowed, this method produces optimal short-circuit code in the sense that no redundant evaluations of boolean primitives and no redundant branches are made within the code of b [Nakata].

(Notes Optimization on common subexpressions is outside the scope of this paper. Non-redundancy of evaluation holds for the code in the level above boolean primitives. Register level optimizations can be further applied. Somewhat inefficient code may be generated for the code outside b. For example, redundant branches may arise if code is always generated according to

if b **then** s -> O(b,f,l); s; l:

even when s is a simple transfer of control (**goto, return,** etc.) [Logothetis].)

Example 4 When b is the syntax tree of Fig. 3 (c), its object code O(b,t,l_1) is derived as in Table 3(a). Fig. 7 shows the code together with its expected evaluation time. The same object code will be produced from

$O($flag **and** (**not** (ch<>' ') **or** ch=tab **or** eoln), $t, l_1)$, or

$O($flag **and not** (ch<>' ' **and** ch<>tab **and not** eoln), $t, l_1)$

etc. Similarly, the code $O(b,f,l_3)$ for the same b of Fig.3(c) becomes as shown in Table 3(b).

4.3 Target machine architecture

In order that the Minimization Theorem can be utilized for generating optimal code, we also assume the following for the architecture of the target machine.

Premise P2 Let b_p be a boolean primitive. Codes for both $O(b_p,t,l)$ and $O(b_p,f,l')$ are supported, and their execution times are the same, i.e.,

$$T^t \text{ of } O(b_p,t,l) = T^t \text{ of } O(b_p,f,l'), \text{ and}$$
$$T^f \text{ of } O(b_p,t,l) = T^f \text{ of } O(b_p,f,l').$$

This premise is usually satisfied if $O(b_p,t,l)$ and $O(b_p,f,l)$ are realized as in Table 2 by the target machine architecture. For example, if b_p is a boolean variable, the object codes are

$O(b_p,t,l)$ = load reg,b_p ; br_true l and

$O(b_p,f,l')$ = load reg,b_p ; br_false l'

and their T^t's and T^f's are the same (2 steps). We can see that codes essentially equal to Table 2 will be used in many representative actual machine architectures such as the VAX 11 and the IBM 370 (see Appendix B for a concrete form), and thus Premise P2 is usually satisfied.

4.4 The Method and rationale

Now, we show a lemma stating that if Premise P1 and P2 are satisfied and if code is generated according to Tables 1 and 2, Condition C holds. From this lemma, we can utilize the Minimization Theorem and generate a time-optimal short-circuit code for a boolean expression by reordering the evaluation sequence. Note that the optimality is in the sense given in section 4.2.

Lemma If Premise P1 and P2 are satisfied and if code is generated according to Tables 1 and 2, then Condition C holds for

a boolean expression as defined in Definition D and satisfying Assumption I.
(Proof) The proof is given in Appendix A3.

A practical method for time-optimal short-circuit code generation is as follows.

Step 1. Make a syntax tree in a canonical form.
Step 2. Trace the tree in a bottom-up manner, estimating the probability and evaluation time of each subtree and performing reordering at each node.
Step 3. Trace the tree in a top-down manner and generate code according to Tables 1 and 2.

In some analyzers, steps 1 and 2 may be further merged into a single pass.

Estimation of probabilities for boolean primitives is discussed in section 6.

5. Some results

5.1 Experimental implementation and comparison

We have made an experimental code generator based on the method shown in section 4 [Yamaguchi].

A comparison of several code generation methods is made. For an "if" statement containing the boolean expression of Example 1,

```
if (eoln or ch = tab or ch = ' ') and flag
then ... else ...
```

object codes and their expected execution times are given in Fig. 8. The target machine is Hitachi's M-170, an IBM/370 compatible machine.

Fig. 8 (a) is the code produced by Pascal 8000 (AAEC version) [Cox] using the traditional (non short-circuit) method. Fig. 8 (b) and (c) are the codes produced by our implementation using the short-circuit method without and with reordering, respectively. The expected number of execution steps is improved from 17 in (a) to 11.620 in (b) and further to 7.125 in (c).

Note that optimization in the register and condition code level is out of the scope of this paper.

5.2 Statistics of boolean expressions in some real programs

To get statistics for real programs, we investigated the source program of Ammann et al.'s Pascal P4 Compiler [Pemberton] and that of a Petri Net analyzer [Kuse]. We used Lex [Lesk], a lexical analyzer generator in Unix*, to collect this statistical data.
(* Unix is a Trademark of Bell Laboratories)

Some results are shown in Table 4. In this table, we assumed (rather restrictively) that boolean expressions satisfying (e) are those for which the short-circuit method is applicable. In estimating the execution time of short-circuit codes, we assumed that probability p of each boolean primitive is 1/2 and that the expected execution time T of each boolean primitive satisfies the following:

time of boolean var. access < time of relational operation
  < time of set membership op. < time of system function call
time of single var. access < time of pointer var. access

Since we do not take all time differences, such as in accessing parameters, global variables, etc., into account, these assumptions should be considered as merely a first approximation.

It was shown that some 12% to 20% of boolean expressions to which the short-circuit method is applicable, can be further improved by reordering the evaluation sequence.

6. Discussions

6.1 About Assumption A

Assumption A requires invariance of program semantics under reordering of boolean subexpressions. To satisfy this, the non-existence of side effects is a sufficient condition. Usually in programming languages, we can regard an expression without function calls, division, pointer operations, access to an array element etc. as having no side effects. If there is a possibility for side effects, we may abandon reordering, for example as in boolean expressions like

```
random(i) < 0.5 and i <> 0
x = 0 or y/x < 0.01
p <> nil and p^.f = 0
i >= 0 and i <= 10 and a[i] = 0
```

(While such expressions might have violated the strict definition of **and**, **or** operations in the programming language used, we don't go too far into it.)

Also, we may omit reordering for boolean expressions involving short-circuit boolean operators defined in the programming language used, such as "&&" and "||" in C, or "**and then**" and "**or else**" in Ada.

A more elaborate approach would be to apply our optimization technique even in the case where side effects and short-circuit boolean operators occur. It can be done by entrusting the compiler with determination of semantic equivalence of a boolean expression under evaluation reordering. However, this determination may require extensive analysis, for example in the case where exceptions are to be detected strictly as in Ada.

An interesting discussion can be found in Appendix C of [Logothetis].

6.2 Other boolean operators

Short-circuit evaluation of boolean expressions including boolean operators other than "and", "or" and "not" is straightforward if they can be represented with "and", "or" and "not" and if the operands are evaluated no more than once:

$$
\begin{aligned}
b_1 \ \textbf{imply}\ b_2 &= (\textbf{not}\ b_1)\ \textbf{or}\ b_2 \\
b_1 <= b_2 &= (\textbf{not}\ b_1)\ \textbf{or}\ b_2 \\
b_1 >= b_2 &= b_1\ \textbf{or}\ (\textbf{not}\ b_2) \\
b_1 < b_2 &= (\textbf{not}\ b_1)\ \textbf{and}\ b_2 \\
b_1 > b_2 &= b_1\ \textbf{and}\ (\textbf{not}\ b_2)
\end{aligned}
$$

Here, b_1 and b_2 are boolean expressions. The last four are from Pascal.

Operators like **equiv** (≡, =), **xor** (≢, <>) are not suited to short-circuit evaluation, since they always require evaluation of both operands, or alternatively, both operands appear twice if the operations are represented with "and", "or" and "not". Still, there remains the possibility of evaluating their operands by the short-circuit method.

6.3 Estimation of probabilities for boolean primitives

Our method requires the estimation of probabilities of boolean primitives or leaf nodes in a syntax tree.

Without any knowledge on the behavior of the program or data base, we can only assume that the probabilities for a boolean primitive to yield "true" and "false" values are the same, i.e., $p = 1/2$. This is a usual assumption in papers dealing with optimizing problems of decision trees [Breitbart]. However, for system functions, such as "eof(f)" and "eoln(f)" of Pascal, we may assume plausible predefined probabilities as in Example 1.

A more elaborate approach would be to analyze the dynamic behavior of the program or query and to estimate probabilities based on it. This approach would be useful in system programs.

6.4 Related works and effectiveness of the optimization method

One of the motivations of this work has been to find the formulae for the expected time of short-circuit evaluation and their theoretical lower bound after reordering. This was achieved and these values will serve as a criterion for assessing the quality of code optimization of boolean expressions.

In application to short-circuit code generators of programming languages, none of the previous works [Nakata] [Logothetis] utilized commutativity and associativity of "and"/"or" operations, on which our optimization using evaluation reordering is based. However, due to some degree of reordering costs, our code optimization method could be restricted to middle or higher level optimization.

We can also conceive of a range of optimization levels, including the traditional method which uses "logical and" and "logical or" instructions and the short-circuit methods with or without reordering. Selecting a suitable method would be done by comparing their expected execution times. However, we are always faced with the general trade-off between compilation time and execution time.

In data base queries, our approach seems to be more promising since the gain in evaluation time in accessing the secondary storage overcomes the overhead of CPU work for making an evaluation strategy. Theoretically, our work aims at a similar purpose to that of Breitbart and Reiter [Breitbart] who studied a nearly optimal evaluation ordering for data base queries. However, our result is stricter than Breitbart et al.'s since their result was an approximation to the optimal evaluation order. Our method takes both evaluation time and probabilities of boolean primitives into account and deals with "not" operators, while theirs does not.

Gudes et al.'s work [Gudes] had a different vector. They presented a practical run-time algorithm to determine whether the evaluation of a boolean primitive can be skipped when its evaluation does not affect the value of the result. The evaluation sequence is a given one, and no reordering of evaluation is made. Evaluation time and probabilities of boolean primitives are not considered.

A weakness in our method is that we do not take common subexpressions into account, while Breitbart et al.'s and Gudes et al.'s consider a restricted case of common subexpressions, namely, the case where the same boolean primitive appears more than once. However, it is probable that no strict methods exist in the presence of common subexpressions as in the case of arithmetic expressions [Bruno].

7. Conclusion

We have described a method of realizing time-optimal short-circuit evaluation of boolean expressions. A boolean expression is assumed to be made of boolean operators "and", "or" and "not".

Formulae are derived to estimate the expected execution time of short-circuit evaluation, given a boolean expression and the evaluation time and probability for each boolean primitive.

Using these formulae, we have presented a theorem to minimize the expected execution time of short-circuit evaluation by reordering the evaluation sequence of boolean subexpressions. It is based on the laws of commutativity and associativity of "and" and "or" operations.

Assumptions and conditions for the formulae and the theorem have been clarified so that the method can be utilized in other fields of computer applications.

We have also presented a practical application of the theorem to time-optimal short-circuit code generation for programming languages. It was shown that the condition of the theorem is satisfied when applied to the usual form of intermediate code and in representative actual machine architectures. The statistics from sample real programs show that some 12% to 20% of the short-circuit code can be further improved by reordering the evaluation sequence. Hence, the method can be applied as an optimization technique for compilers of programming languages.

Application to data base query seems to be promising and further studies will be welcomed.

Acknowledgements The author wishes to thank Ikuo Nakata, Yoshitsugu Yamamoto and Hisashi Takayama for helpful discussions, and Yoshikazu Yamaguchi for developing an experimental implementation.

Appendix A1

Derivation of the formulae for "and" nodes

Formula (1) This formula is straightforward, since (the whole subexpression corresponding to) the given "and" node yields true only if all sons yield true.

Formula (2) This formula comes also from the fact that for (the whole subexpression corresponding to) the "and" node to yield true all sons must be evaluated and must yield true.

Formula (3) This formula is somewhat complicated. The given "and" node yields false in one of the following cases:

(i) the 1st son yields false, or

(ii) the 1st son yields true, but the 2nd yields false, or

(iii) the 1st and 2nd sons yield true, but the 3rd son yields false, or

:

:

(n-1) the 1st and ... (n-1)-th son yield true, but the n-th son yields false.

The formula is given by summing up the evaluation times of all cases weighted by probability. Each case has the following execution time and probability.

	exp. exec. time	probability	whole probability
(i)	T_1^f	$\bar{p}_1$	
(ii)	$T_1^t+T_2^f$	$p_1\bar{p}_2$	
(iii)	$T_1^t+T_2^t+T_3^f$	$p_1p_2\bar{p}_3$	$\bar{p}$
:			
:			
(n-1)	$T_1^t+T_2^t+\ldots+T_{n-1}^t+T_n^f$	$p_1p_2\cdots p_{n-1}\bar{p}_n$	

Therefore,

$$T^f = [\ \bar{p}_1T_1^f+p_1\bar{p}_2(T_1^t+T_2^f)+p_1p_2\bar{p}_3(T_1^t+T_2^t+T_3^f) \\ \ldots+p_1p_2\cdots p_{n-1}\bar{p}_n(T_1^t+T_2^t+\ldots+T_{n-1}^t+T_n^f)\ ]/\bar{p} \qquad \text{(a)}$$

$$= [\ \Sigma_{i=1}^{n}[(\Pi_{j=1}^{i-1}p_j)\bar{p}_i\{(\Sigma_{j=1}^{i-1}T_j^t)+T_i^f\}]\ ]/\bar{p}$$
$$= (3)$$

Furthermore,

$$\begin{aligned}(a) = [\ & \bar{p}_1T_1^f+p_1\bar{p}_2T_2^f+p_1p_2\bar{p}_3T_3^f+\ldots+p_1p_2\cdots p_{n-1}\bar{p}_nT_n^f \\ & +(p_1\bar{p}_2+p_1p_2\bar{p}_3+\ldots+p_1p_2\cdots p_{n-1}\bar{p}_n)T_1^t \\ & +(p_1p_2\bar{p}_3+\cdots+p_1p_2\cdots p_{n-1}\bar{p}_n)T_2^t \\ & \ldots \\ & +(p_1p_2\cdots p_{n-1}\bar{p}_n)T_{n-1}^t\ ]/\bar{p}\end{aligned}$$

$$= [\ \Sigma_{i=1}^{n}(\Pi_{j=1}^{i-1}p_j)\bar{p}_iT_i^f + \Sigma_{i=1}^{n-1}\{\Sigma_{j=i+1}^{n}(\Pi_{k=1}^{j-1}p_k)\bar{p}_j\}T_i^t\ ]/\bar{p} \qquad (b)$$

To show (3'), we use the following equality relation

$$\begin{aligned}&\Sigma_{j=i+1}^{n}(\Pi_{k=1}^{j-1}p_k)\bar{p}_j \\ &= p_1\cdots p_i\bar{p}_{i+1} + p_1\cdots p_{i+1}\bar{p}_{i+2} + p_1\cdots p_{i+2}\bar{p}_{i+3} \\ &\quad + \ldots + p_1\cdots p_{n-1}\bar{p}_n \\ &= p_1\cdots p_i(1-\cancel{p_{i+1}}) \\ &\quad +p_1\cdots p_ip_{i+1}(\cancel{1}-\cancel{p_{i+2}}) \\ &\quad +p_1\cdots p_ip_{i+1}p_{i+2}(\cancel{1}-p_{i+3}) \\ &\quad \ldots \\ &\quad +p_1\cdots p_ip_{i+1}\cdots p_{n-1}(\cancel{1}-p_n) \\ &= p_1\cdots p_i - p_1\cdots p_n \\ &= \Pi_{j=1}^{i}p_j - p \qquad \text{by (1).}\end{aligned} \qquad (c)$$

Noting also that if $i = n$,

$$(\Pi_{j=1}^{i}p_j - p) = p_1\cdots p_n - p = 0 \quad (\text{for } i = n) \qquad (d)$$

holds, formula (3') is derived by (b), (c) and (d).

Formula (4) This formula is derived as follows.

$$\begin{aligned}&T=pT^t+\bar{p}T^f \\ &=p\Sigma_{i=1}^{n}T_i^t+[\Sigma_{i=1}^{n}(\Pi_{j=1}^{i-1}p_j)\bar{p}_iT_i^f+\Sigma_{i=1}^{n}(\Pi_{j=1}^{i}p_j-p)T_i^t] \text{ by (2) and (3')} \\ &=\Sigma_{i=1}^{n}(\Pi_{j=1}^{i}p_j)T_i^t+\Sigma_{i=1}^{n}(\Pi_{j=1}^{i-1}p_j)\bar{p}_iT_i^f \\ &=\Sigma_{i=1}^{n}(\Pi_{j=1}^{i-1}p_j)(p_iT_i^t+\bar{p}_iT_i^f) \\ &=\Sigma_{i=1}^{n}(\Pi_{j=1}^{i-1}p_j)T_i\end{aligned}$$

Alternatively, this formula (4) can be derived directly from the fact that T is the sum of the evaluation times of the following cases weighted by probability.

(i) the 1st son is always evaluated.

(ii) the 2nd son is evaluated when the 1st son yields true.

(iii) the 3rd son is evaluated when the 1st and 2nd sons yield true.

:

:

(n) the n-th son is evaluated when the 1st and ... (n-1)-th son yield true.

Each case has the following evaluation time and probability.

	exp. exec. time	probability
(i)	T_1	1
(ii)	T_2	p_1
(iii)	T_3	p_1p_2
:		
:		
(n)	T_n	$p_1p_2\cdots p_{n-1}$

Therefore,

$$T = T_1 + p_1T_2 + p_1p_2T_3 + \dots + p_1p_2\cdots p_{n-1}T_n = \Sigma_{i=1}^{n}(\Pi_{j=1}^{i-1}p_j)T_i$$

Appendix A2

Proof of the Minimization Theorem

We only show the proof for an "and" node since an "or" node is dual (p_i <-> $1-p_i$) to an "and" node.

Notice that the formula for T^t (2) is invariant under reordering. So, the minimization of the expected time applies only to the formula for T^f (3'), or, since p is invariant and $T=pT^t+\bar{p}T^f$, to the formula for T (4). Here we are applying minimization to the formula for T (4).

Let $T(i_1,i_2,\dots,i_n)$

$$=T_{i_1}+p_{i_1}T_{i_2}+p_{i_1}p_{i_2}T_{i_3}+\dots+p_{i_1}\dots p_{i_{n-1}}T_{i_n}$$

$$=\Sigma_{k=1}^{n}(\Pi_{j=1}^{k-1}\ p_{i_j})T_{i_k}$$

where $(i_1,i_2,\dots,i_n)$ is a permutation of $(1,2,\dots,n)$.

First, we show that if

$$T_{i_k}/(1-p_{i_k}) \leq T_{i_{k+1}}/(1-p_{i_{k+1}}) \qquad (a)$$

is satisfied, then

$$T(i_1,\ldots,i_{k-1},i_k,i_{k+1},i_{k+2},\ldots,i_n)$$
$$\leq T(i_1,\ldots,i_{k-1},i_{k+1},i_k,i_{k+2},\ldots,i_n) \qquad (b)$$

holds. This is because

left hand side - right hand side

$$= (p_{i_1}\cdots p_{i_{k-1}}T_{i_k}+p_{i_1}\cdots p_{i_{k-1}}p_{i_k}T_{i_{k+1}})$$
$$- (p_{i_1}\cdots p_{i_{k-1}}T_{i_{k+1}}+p_{i_1}\cdots p_{i_{k-1}}p_{i_{k+1}}T_{i_k})$$
$$= p_{i_1}\cdots p_{i_{k-1}}(T_{i_k}+p_{i_k}T_{i_{k+1}}-T_{i_{k+1}}-p_{i_{k+1}}T_{i_k})$$

≤ 0, from (a) and $0 < p_{i_h} < 1$ (for all i_h).

Now assume, without loss of generality, that

$$\frac{T_1}{1-p_1} \leq \frac{T_2}{1-p_2} \leq \cdots \leq \frac{T_n}{1-p_n} \qquad (c)$$

If in $T(j_1,\ldots,j_k,j_{k+1},j_{k+2},\ldots,j_n)$, there is a reverse ordering pair (j_k,j_{k+1}) such that $j_k > j_{k+1}$, it follows from (a), (b) and (c) that

$$T(j_1,\ldots,j_{k+1},j_k,j_{k+2},\ldots,j_n) \leq T(j_1,\ldots,j_k,j_{k+1},j_{k+2},\ldots,j_n)$$

Repeating this step as long as there are reverse ordering pairs, it is shown that T(...)'s constitute a lattice and T(1,2,...,n) is the minimum element. (Q.E.D.)

Appendix A3

Proof of the Lemma (section 4.4)

Owing to Premise P1, we can generate short-circuit code according to Tables 1 and 2. We want to show that T_i^t and T_i^f of a boolean subexpression b_i are invariant before and after reordering. For example, in the case when b_i is a son (subexpression) of an "and" node b and code for O(b,t,1) is to be produced (see Table 1), the code for b_i before and after reordering may remain unchanged or may vary between O(b_i, f, newlabel) and $O(b_i,t,l)$. Other cases are similar. If the code remains unchanged, Condition C trivially holds. If not, we see in general that the code for b_i may vary between either of $O(b_i,t,l)$ and $O(b_i,f,l')$ before and after reordering. By applying Table 1 (recursively) and Table 2 for this b_i, we observe that the codes $O(b_i,t,l)$ and $O(b_i,f,l')$ are very similar.

In fact, when we rename labels, they differ only in the position of labels and in the code of the last boolean primitive. (For example, see (a) and (b) of Table 3, renaming labels l_3 to l_2 and l_4 to l_1. They differ only in the codes $O(eoln,t,l_1)$ and $O(eoln,f,l_3)$ for the last boolean primitive "eoln".) Note that the difference in label positions does not affect the execution time. As to the code of the last boolean primitive (let us call it b_p), its code is either $O(b_p,t,l_p)$ or $O(b_p,f,l_p')$ of Table 2, and its execution time is equal by Premise P2. By these investigations, we conclude that T^t of $O(b_i,t,l)$ = T^t of $O(b_i,f,l')$ and T^f of $O(b_i,t,l)$ = T^f of $O(b_i,f,l')$, hence T of $O(b_i,t,l)$ = T of $O(b_i,f,l')$.

The invariance of p_i under reordering is trivial from Assumption I. (Q.E.D.)

Appendix B

A concrete form of Table 2 for VAX 11 and IBM 370 series computers

	VAX 11		IBM 370	
(1):	tstl	b	L	reg,b ; LTR reg,reg
(2):	beql	l	BNZ	l
(3):	bneq	l	BZ	l
(4):	cmpl	b_1,b_2	L	reg,b_1 ; C reg,b_2 (when b_1 is not in a register)
(5):	beql	l	BE	l
(6):	bneq	l	BNE	l
(7):	calls	...	BALR	reg_1,reg_2
(8):	movl	r0,r10	L	reg,result ; LTR reg,reg

References

[Sassa] Sassa, M., Time-Optimal Short-Circuit Evaluation of Boolean Expressions, Preprint of WGSF (Software Foundation) 8-1, IPSJ (Mar. 1984).

[Nakata] Nakata, I., Compiling Algorithms for Logical Expressions, Johoshori, Vol. 16, No. 3, pp. 186-194 (Mar. 1975) (in Japanese).

[Aho] Aho, A.V. and Ullman, J.D., Principles of Compiler Design, Addison-Wesley (1977).

[Logothetis] Logothetis,G. and Mishra,P., Compiling Short-circuit Boolean Expressions in One Pass, Software- Practice and Experience, Vol. 11, pp. 1197-1214 (1981).

[Gudes] Gudes, E. and Reiter, A., On evaluating boolean expressions, Software- Practice and Experience, Vol. 3, pp. 345-350 (1973).

[Breitbart] Breitbart, Y. and Reiter, A., Algorithms for fast evaluation of boolean expressions, Acta Informatica, Vol. 4, pp. 107-116 (1975).

[Bruno] Bruno, J. and Sethi, R., Code Generation for a One-Register Machine, J. ACM, Vol. 23, No. 3, pp. 502-510 (July 1976).

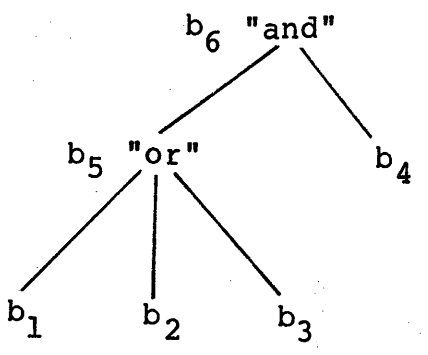
[Yamaguchi] Yamaguchi, Y., Generation of optimal code for boolean expressions considering expected execution time, Bach. Thesis, Univ. of Tsukuba (Feb. 1984) (in Japanese).

[Cox] Cox, G.W. and Tobias, J.H., Pascal 8000 IBM 360/370 version for OS and VS environments Reference manual version 1.2 (1978).

[Pemberton] Pemberton, S. and Daniels, M.C., Pascal Implementation: The P4 Compiler, and ibid: Compiler and Assembler / Interpreter, Ellis Horwood (1982).

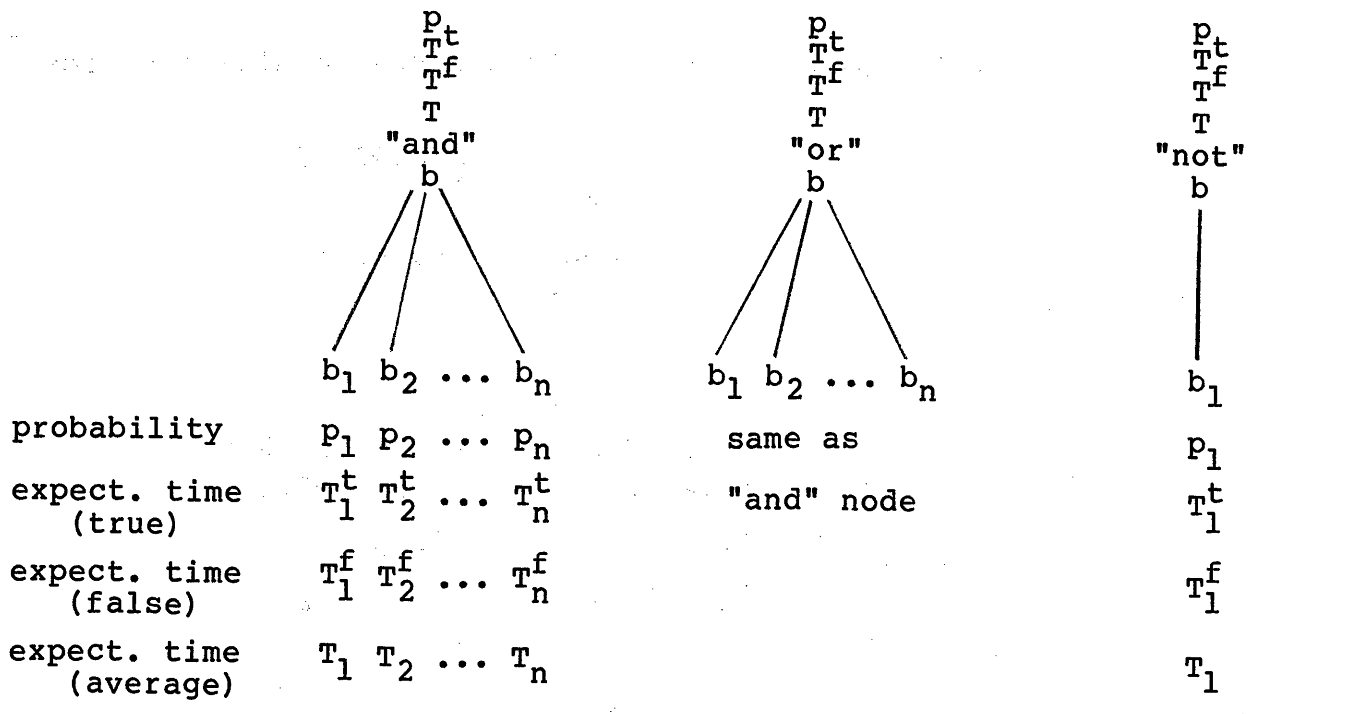
[Kuse] Kuse, K., Sassa, M. and Nakata, I., Analysis and transformation of concurrent processes connected by streams, Symposium on Mathematical Methods in Software Science and Engineering 1983, RIMS Report 511, Univ. of Kyoto (Feb. 1984).

[Lesk] Lesk, M.E., Lex - A Lexical Analyzer Generator, Comp. Sci. Tech. Rep. 39, Bell Laboratories.



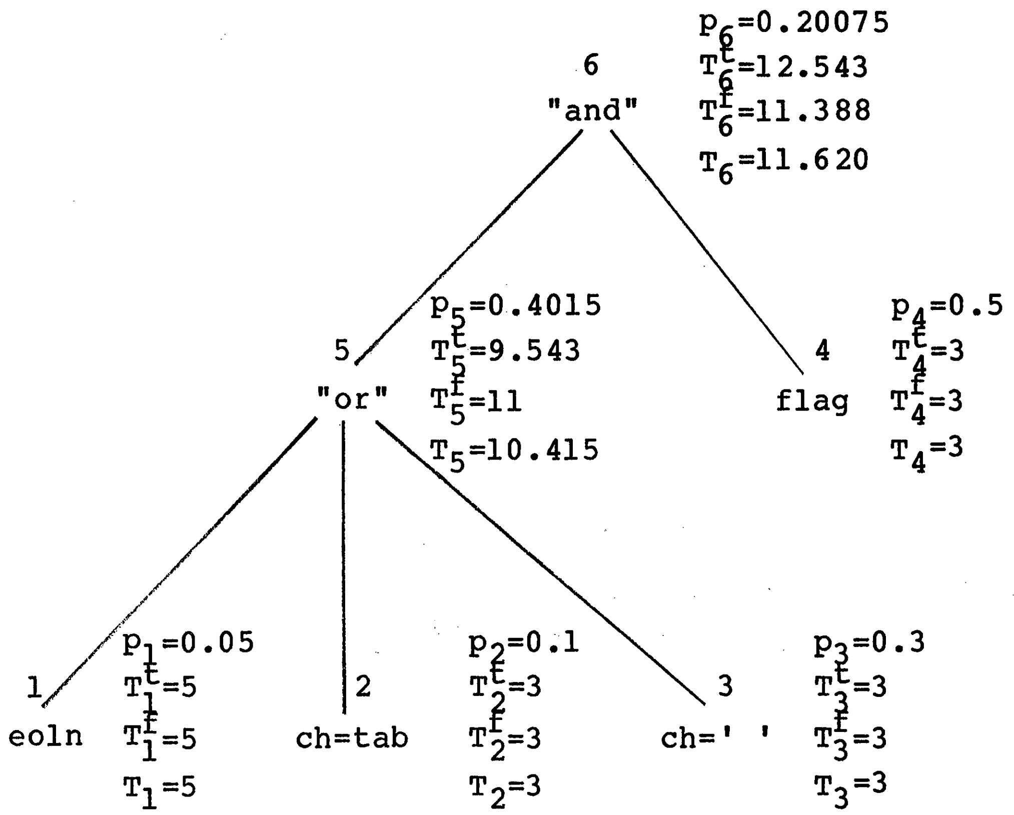
(b_1, b_2, b_3 and b_4 are boolean primitives)

Fig. 1 Syntax tree of the boolean expression "(b_1 **or** b_2 **or** b_3) **and** b_4"

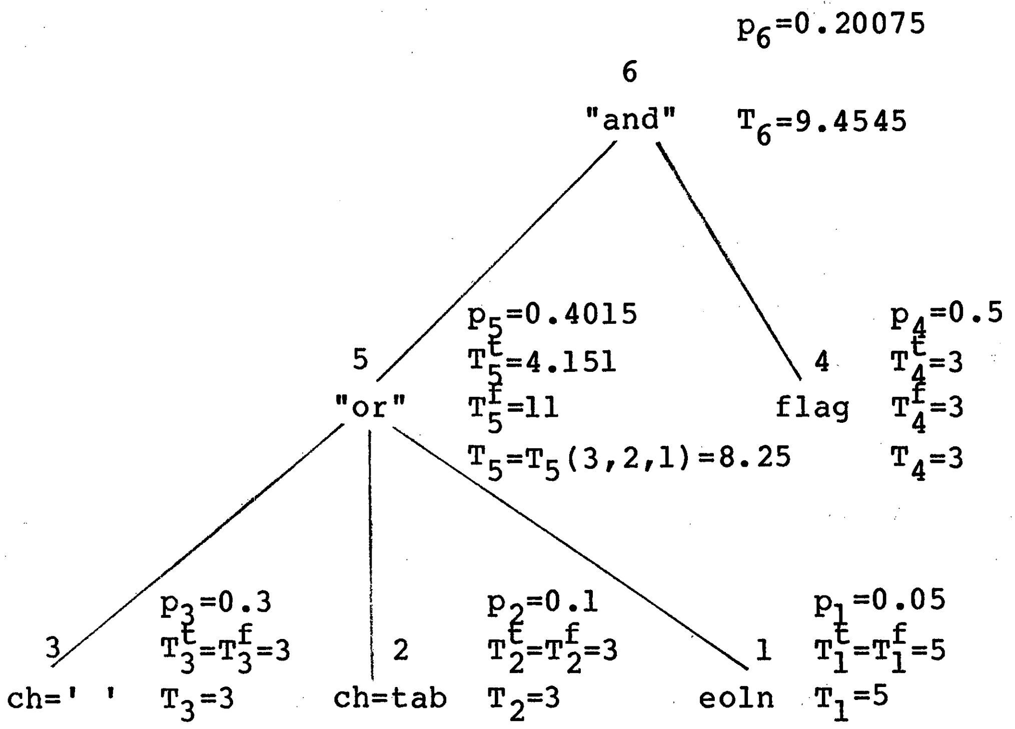


(a) "and" node (b) "or" node (c) "not" node

Fig. 2 Probabilities and expected evaluation times for "and", "or" and "not" nodes

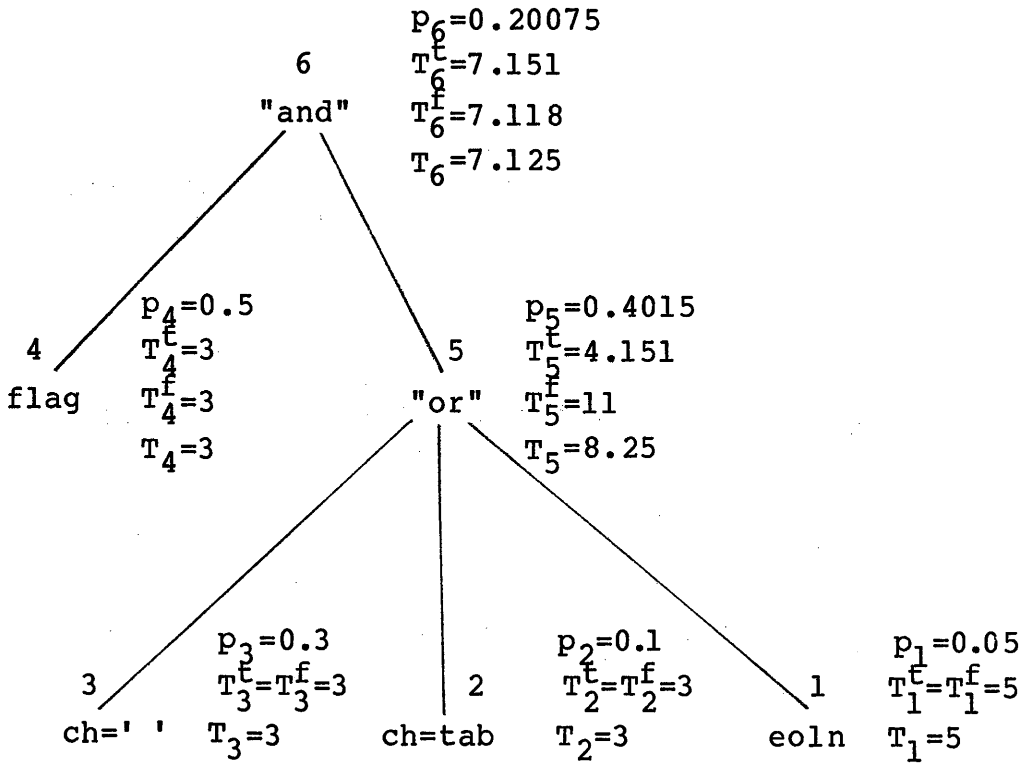


(a) original syntax tree and its expected evaluation time



(b) syntax tree and expected evaluation time after reordering the subtree of node 5 in (a)

Fig. 3 An example reordering of the syntax tree for "(eoln **or** ch=tab **or** ch=' ') **and** flag" (continued on next page)



The syntax tree corresponds to
"flag **and** (ch=' ' **or** ch=tab **or** eoln)"

The subtree below node 5 is the same as (b)

(c) optimal syntax tree after reordering and its expected evaluation time

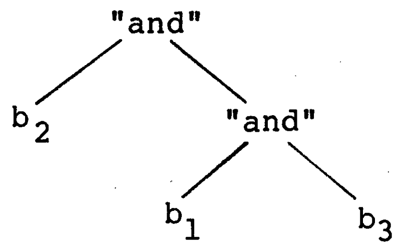
Leaves in the syntax trees are boolean primitives

Fig. 3 An example reordering of the syntax tree for "(eoln **or** ch=tab **or** ch=' ') **and** flag"

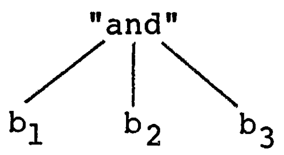
$$\begin{tikzcd}[column sep=small]
 & \begin{array}{l}\text{maximum}\\ T_5(1,2,3) = 5+0.95\times3+0.95\times0.9\times3\\ \qquad\qquad\; = 10.415\end{array} \arrow[dl, dash] \arrow[dr, dash] & \\
\begin{array}{l}T_5(2,1,3) = 3+0.9\times5+0.9\times0.95\times3\\ \qquad\qquad\; = 10.065\end{array} \arrow[d, dash] & & \begin{array}{l}T_5(1,3,2) = 5+0.95\times3+0.95\times0.7\times3\\ \qquad\qquad\; = 9.845\end{array} \arrow[d, dash] \\
\begin{array}{l}T_5(2,3,1) = 3+0.9\times3+0.9\times0.7\times5\\ \qquad\qquad\; = 8.85\end{array} \arrow[dr, dash] & & \begin{array}{l}T_5(3,1,2) = 3+0.7\times5+0.7\times0.95\times3\\ \qquad\qquad\; = 8.495\end{array} \arrow[dl, dash] \\
 & \begin{array}{l}\text{minimum}\\ T_5(3,2,1) = 3+0.7\times3+0.7\times0.9\times5\\ \qquad\qquad\; = 8.25\end{array} &
\end{tikzcd}$$

$T_5(i_1,i_2,i_3)$ corresponds to the evaluation order (node i_1, node i_2, node i_3).

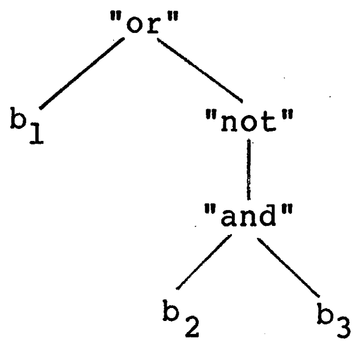
Fig. 4 Expected evaluation times for each reordering of sons of node 5 in Fig. 3(a)
(They form a lattice)



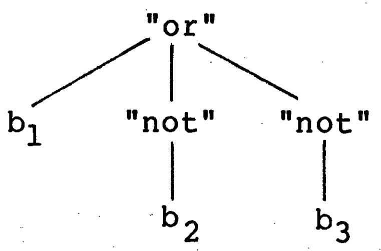
(a) b_2 **and** (b_1 **and** b_3)



(b) b_1 **and** b_2 **and** b_3

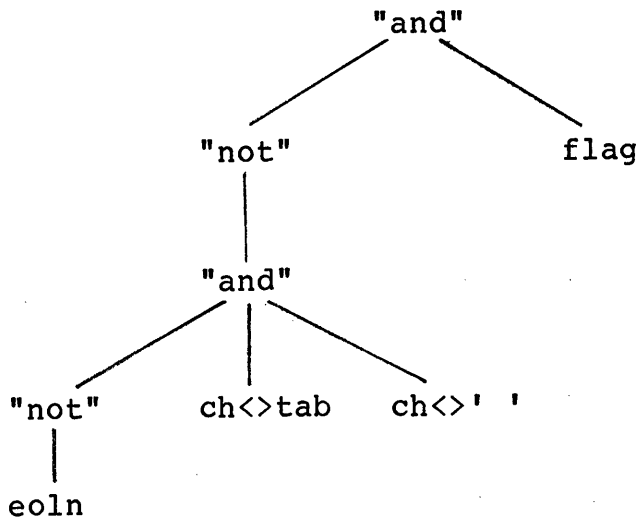


(c) b_1 **or not** (b_2 **and** b_3)

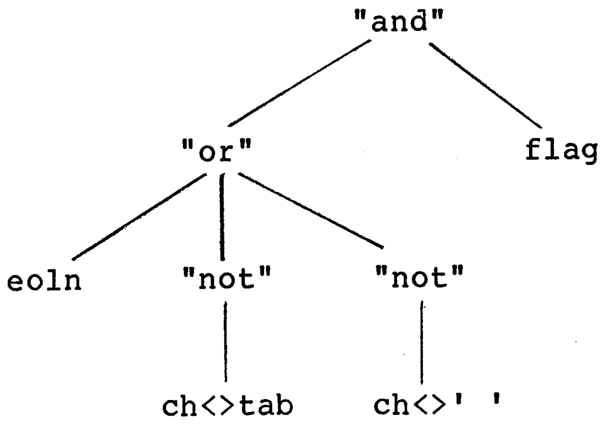


(d) b_1 **or not** b_2 **or not** b_3

Fig. 5 Syntax trees related to canonical forms



(a) first canonical form



(b) second canonical form

Fig. 6 Syntax trees in two canonical forms for "eoln **or not** (ch<>tab **and** ch<>' ')) **and** flag"

	$O(b,t,l_1)$		exec. time (*1)	probability of exec.	expected eval. time
	load	reg,flag	3	1	3
	br_false	l_2	(T_4)		(T_4)
	compare	ch,' '	3	0.5	1.5
	br_equal	l_1	(T_3)	(p_4)	(p_4T_3)
	compare	ch,tab	3	0.35	1.05
	br_equal	l_1	(T_2)	$(p_4\bar{p}_3)$	$(p_4\bar{p}_3T_2)$
	call	eoln			
	load	reg,result	5 (*2)	0.315	1.575
	br_true	l_1	(T_1)	$(p_4\bar{p}_3\bar{p}_2)$	$(p_4\bar{p}_3\bar{p}_2T_1)$
l_2:	(false)				-----------
				total	7.125

$$(T_6=T_4+p_4(T_3+\bar{p}_3T_2+\bar{p}_3\bar{p}_2T_1))$$

*1: for IBM/370 series machines
*2: for Pascal 8000 AAEC version compiler (cf. Fig. 8(c))

Fig. 7 Object code and expected execution time for Fig. 3(c)

```
     L     10,$EOLN   }
     SRL   10,1       } 3
     N     10,=X'1'   }
     L     11,CH      }
     C     11,TAB     }
     LA    11,1       } 4.5
     BE    L1         }
     XR    11,11      }
L1   OR    11,10      } 1
     L     10,CH      }
     C     10,=X'40'  }
     LA    10,1       } 4.5
     BE    L2         }
     XR    10,10      }
L2   OR    10,11      } 1
     N     10,FLAG    }
     LTR   10,10      } 3
     BZ    LF         }
     (then part)

LF   (else part)
--------------------------
     exp. exec. step = 17
```

(a) Pascal 8000 (AAEC)
Traditional code
(non short-circuit)

```
     L     10,$EOLN
     SRL   10,1
     N     10,=X'1'
     LTR   10,10
     BNZ   L1
     L     10,CH
     C     10,TAB
     BE    L1
     L     10,CH
     C     10,=X'40'
     BNE   LF
L1   L     10,FLAG
     LTR   10,10
     BZ    LF
     (then part)

LF   (else part)
------------------------
exp. exec. step = 11.620
(cf. Example 1)
```

(b) Our implementation.
Short-circuit code
without reordering

```
     L     10,FLAG
     LTR   10,10
     BZ    LF
     L     10,CH
     C     10,=X'40'
     BE    LT
     L     10,CH
     C     10,TAB
     BE    LT
     L     10,$EOLN
     SRL   10,1
     N     10,=X'1'
     LTR   10,10
     BZ    LF
LT   (then part)

LF   (else part)
-----------------------
exp. exec. step = 7.125
(cf. Example 2)
```

(c) Our implementation.
Short-circuit code
with reordering

The target machine is Hitachi's M-170, an IBM/370 compatible machine

Fig. 8 Comparison of codes and their expected execution times for "**if** (eoln **or** ch = tab **or** ch = ' ') **and** flag **then** (then part) **else** (else part)"

Table 1. O(b,c,l): Short-circuit object code of "**if** b=c **then goto** l"

*1 means:
O(b_1,f,newlabel) followed by O(b_2,f,newlabel) ...
... followed by newlabel:

b \ c	t (true)	f (false)

b= b_1 **and** b_2 ... **and** b_n	O(b_1,f,newlabel)
	O(b_2,f,newlabel)
	:
	O(b_{n-1},f,newlabel)
	O(b_n,t,l)
	newlabel: *1
		O(b_1,f,l)
		O(b_2,f,l)
		:
		:
		O(b_n,f,l)

b= b_1 **or** b_2 ... **or** b_n	O(b_1,t,l)
	O(b_2,t,l)
	:
	:
	O(b_n,t,l)
		O(b_1,t,newlabel)
		O(b_2,t,newlabel)
		:
		O(b_{n-1},t,newlabel)
		O(b_n,f,l)
		newlabel:

b= **not** b_1	O(b_1,f,l)	O(b_1,t,l)
b is a boolean primitive	(see on Table 2)	

Table 2. Object code O(b,c,l) for boolean primitive b

N.B. Although this table only includes limited types of boolean primitives, other primitives could be conveniently incorporated, such as relational operations with character strings or set membership operations.

b \ c	t (true)	f (false)
b is a variable	load reg,b (1) br_true l (2)	load reg,b br_false l (3)
b is a relational expression	compare b_1,b_2 (4) br_equal l (5) etc.	compare b_1,b_2 br_not_equal l (6) etc.
b is a function call	call function (7) load reg,result (8) br_true l	call function load reg,result br_false l

Table 3.   Object codes of
           "flag **and** (ch=' ' **or** ch=tab **or** eoln)"

(a) Object code of O(flag **and** (ch=' ' **or** ch=tab **or** eoln), t, l_1)

```
O(flag and (ch=' ' or ch=tab or eoln), t, $l_1$)
= O(flag,f,$l_2$)                                ...  load      reg,flag
                                                      br_false  $l_2$
  O(ch=' ' or ch=tab or eoln, t, $l_1$)
  = O(ch=' ',t,$l_1$)                            ...  compare   ch,' '
                                                      br_equal  $l_1$
    O(ch=tab,t,$l_1$)                            ...  compare   ch,tab
                                                      br_equal  $l_1$
    O(eoln,t,$l_1$)                              ...  call      eoln
                                                      load      reg,result
                                                      br_true   $l_1$
  $l_2$:                                         ... $l_2$:
```

(b) Object code of O(flag **and** (ch=' ' **or** ch=tab **or** eoln), f, l_3)

```
O(flag and (ch=' ' or ch=tab or eoln), f, $l_3$)
= O(flag,f,$l_3$)                                ...  load      reg,flag
                                                      br_false  $l_3$
  O(ch=' ' or ch=tab or eoln, f, $l_3$)
  = O(ch=' ',t,$l_4$)                            ...  compare   ch,' '
                                                      br_equal  $l_4$
    O(ch=tab,t,$l_4$)                            ...  compare   ch,tab
                                                      br_equal  $l_4$
    O(eoln,f,$l_3$)                              ...  call      eoln
                                                      load      reg,result
                                                      br_false  $l_3$
    $l_4$:                                       ... $l_4$:
```

Table 4. Statistics of boolean expressions in two real programs

	Pascal P4		Petri Net Analyzer	
(a) no. of source lines	4000		2528	
(b) no. of boolean expressions (except the RHS of boolean assignment statements)	791		318	
(c) classification by the place of appearance				
in an if statement	702		272	
while statement	41		45	
repeat statement	48		1	
	------		------	
	791		318	
(d) classification by the number of boolean primitives				
with 1 boolean primitive	732		242	
2	57		56	
3	1	59	6	76
4	1		5	
5			6	
6			2	
7			0	
8			1	
	------		------	
	791		318	
(e) no. of boolean expressions for which the short-circuit method is applicable i.e., boolean expressions with 2 or more boolean primitives and without user functions. (side-effect free system functions are allowed.)	49		71	
(f) no. of boolean expressions in (e) for which the short-circuit code with reordering is better (takes less time) than the short-circuit code without reordering	6 (6/49=12%)		14 (14/71=20%)	

(g) boolean expressions of (f) for Pascal P4 Compiler

(eol) **or** (ch = '''')
(llp <> nil) **and not** redef
(typtr^ . form = scalar) **and** (typtr <> realptr)
(pflev + 1 <> level) **or** (fprocp <> fcp)
(llp <> nil) **and not** found
(lsp^ . form <> scalar) **or** (lsp = realptr)

INSTITUTE OF INFORMATION SCIENCES AND ELECTRONICS
UNIVERSITY OF TSUKUBA
SAKURA-MURA, NIIHARI-GUN, IBARAKI 305 JAPAN

REPORT DOCUMENTATION PAGE	REPORT NUMBER ISE-TR-84-43

TITLE

Time-Optimal Short-Circuit Evaluation of Boolean Expressions

AUTHOR(S)

Masataka Sassa

Institute of Information Sciences and Electronics
University of Tsukuba

REPORT DATE April 26th, 1984	NUMBER OF PAGES 35
MAIN CATEGORY Programming Languages: Processors	CR CATEGORIES D.3.4, H.3.3, F.2.2, G.3

KEY WORDS

Short-circuit evaluation, Boolean expression, Expected execution time, Optimization, Code generation

ABSTRACT

A method of realizing time-optimal short-circuit evaluation of boolean expressions is described. A boolean expression is assumed to be made up of boolean operators "and", "or" and "not".

Formulae are derived to estimate the expected execution time of short-circuit evaluation, given a boolean expression and an evaluation time and probability for each boolean primitive.

Using these formulae, we present a theorem to minimize the expected execution time of short-circuit evaluation by reordering the evaluation sequence of boolean subexpressions, based on laws of commutativity and associativity of "and" and "or" operations. The theorem utilizes the concept of dynamic programming.

Methods and conditions are given in an application of the theorem to code generation for programming languages.

A comparison based on an experimental implementation and some statistics from real programs are also given.

SUPPLEMENTARY NOTES