John Cocke
Peter W. Markstein

Strength Reduction for Division and Modulo with Application to Accessing a Multilevel Store

A method for replacing certain division and modulo operations by additions and subtractions is presented. This optimization allows efficient and easy use of partitioned arrays to access a multilevel store.

0. Introduction

Reduction in strength is an optimization which moves "expensive" calculations from a high-frequency execution region to a lower-frequency region, and replaces the original expensive calculations with "cheaper" ones within the region [1]. The most common examples are "code motion," in which the cheaper operation is no operation at all, and the replacement of multiplications (usually associated with indexing through arrays) by additions. In this paper, we describe a strength reduction procedure for integer division and modulo operations, and demonstrate its use in accessing arrays in a multilevel store.

In this paper, x ÷ c is used to denote integer division, as defined by the Euclidean algorithm: $x \div c = Q$, where Q and R are integers satisfying

$$x = Q * c + R, \qquad 0 \leq R < |c|.$$

R is taken to be the value of mod (x, c). Observe that $x \div c$ is linear in the numerator in the following sense:

$(x + k * c) \div c = x \div c + k$ for all integers k.

1. Strength reduction of division and modulo

Suppose either or both of the following computations appear in a strongly connected region [2] of a program:

x ÷ c, or
mod (x, c),

where c is constant throughout the region and where x is modified only by computations of the form

$x = x + k$, or
$x = k$,

where k is constant throughout the strongly connected region.

Introduce two new variables Q and R by inserting the following computations on entry to (but outside of) the strongly connected region:

$Q = x \div c$,
$R =$ mod (x, c).

In the strongly connected region, replace every computation of $x \div c$ with a reference to Q, and replace every computation of mod (x, c) with a reference to R. Whenever a computation

$x = x + k$

appears in the strongly connected region, insert immediately after that computation the following:

$R = R +$ mod (k, c);
if $R \geq |c|$ then do;
    $R = R - |c|$;
    $Q = Q +$ sgn (c);
    end;
$Q = Q + k \div c$;

and whenever

$x = k$;

appears in the strongly connected region, insert

$R = \text{mod}\ (k, c);$
$Q = k \div c;$

since k and c are region constants, mod (k, c) and $k \div c$ can be computed on entry to the region or at compilation time.

To show that Q and R hold the updated values of $x \div c$ and mod (x, c), suppose that

$Q = x \div c$, and
$R = \text{mod}\ (x, c)$.

If $x' = x + k$, then

$$\begin{aligned}\text{mod}\ (x', c) &= \text{mod}\ [\text{mod}\ (x, c) + \text{mod}\ (k, c), c] \\ &= R + \text{mod}\ (k, c) \\ &\quad \text{if } R + \text{mod}\ (k, c) < |c|, \\ &= R + \text{mod}\ (k, c) - |c| \\ &\quad \text{if } R + \text{mod}\ (k, c) \geq |c|,\end{aligned}$$

and

$$\begin{aligned}x' \div c &= (x + k) \div c \\ &= [(x \div c) * c + \text{mod}\ (x, c) \\ &\quad + (k \div c) * c + \text{mod}\ (k\ c)] \div c \\ &\quad = Q + k \div c \\ &\quad\quad \text{if } R + \text{mod}\ (k, c) < |c|, \\ &\quad = Q + k \div c + \text{sgn}\ (c) \\ &\quad\quad \text{if } R + \text{mod}\ (k, c) \geq |c|.\end{aligned}$$

In most higher-level languages integer division $x \div c$ is defined as

$$x \div c = [|x| \ / \ |c|] * \text{sgn}\ (x * c),$$

so that $[x/c]$ corresponds to the Euclidean algorithm definition of integer division only when $x \geq 0$ and $c > 0$. In particular, the definition of integer division just cited is nonlinear in the numerator, since for $0 < x < |c|$, $|c| > 1$, $x \div c = 0$, and $(x - c) \div c = 0$. For such languages, strength reduction can be applied to modulo without qualifications, but can only be used for division when it can be shown that the numerator is always nonnegative.

2. Arrays in multilevel storage hierarchies

Let us assume that a two-dimensional array **A** having m rows and n columns is stored "row-wise," so that $\mathbf{A}(i, j)$ is located $n * i + j$ elements beyond $\mathbf{A}(0, 0)$. If only a small part of the array can be kept in fastest memory, as is the case with cache memory and with paged memory, then operations which use elements of **A** by row will be very efficient, utilizing all elements in a "page" of fast memory, whereas columnwise operations use only w/n elements, where w is the number of elements in a page of fast memory. If an array will be used principally in a columnwise manner, it is good practice to work with its transpose instead. However, it may be the case that the array is accessed both by rows and by columns, as for example in matrix transposition, matrix inversion, iterative solutions to partial differential equations, etc.

Let $k = [\text{sqrt}\ (w)]$. If we partition arrays into $k \times k$ subarrays, then for use of the array by columns or by rows, k elements per page will be accessed. Rather than burden the programmer with the task of managing partitioned arrays, a compiler can generate code to compute the location of $\mathbf{A}(i, j)$ as being $i' + i'' + j' + j''$ elements from $\mathbf{A}(0, 0)$, where

$i' = (i \div k) * n * k;$
$i'' = [\text{mod}\ (i, k)] * k;$
$j' = (j \div k) * k * k;$
$j'' = \text{mod}\ (j, k);$

(assuming that m and n are multiples of k). In commonly encountered loops, these computations can produce surprisingly little overhead by using the strength reduction given in Section 1. Suppose for example that a program fragment has the structure

```
do i = 0 to m − 1;
        do j = 0 to n − 1;
                .
                .
                .
                access A(i, j)
                .
                .
                .
        end;
end;
```

Straightforward code generation, using matrix partitioning, would produce

```
loop: · · ·
        i' = (i ÷ k) * n * k;
        i'' = [mod (i, k)] * k;
        j' = (j ÷ k) * k * k;
        j'' = mod (j, k);
        · · ·
        access A(i' + i'' + j' + j'')
        · · ·
        j = j + 1;
        if j < n then go to loop;
        j = 0;
        i = i + 1;
        if i < m then go to loop;
        · · ·
```

Applying reduction in strength to division and modulo yields

```
        R' = 0;
        Q' = 0;
        Q'' = 0;
        R'' = 0;
loop:   i' = Q' * n * k
        i'' = R' *k;
        j' = Q''* k * k;
        j'' = R'';
        . . .
        access A(i' + i'' + j' + j'');
        . . .
        j = j + 1;
        R'' = R'' + 1;
        if R'' ≥ k then do;
                R'' = R'' - k;
                Q'' = Q'' + 1;
                end;
        if j < n then go to loop;
        j = 0;
        R'' = 0;
        Q'' = 0;
        i = i + 1;
        R' = R' + 1;
        if R' ≥ k then do;
                R' = R' - k;
                Q' = Q' + 1;
                end;
        if i < m then go to loop;
        . . .
```

If we now apply strength reduction to the multiplications of i', i'', and j' by the loop constants $n * k$, k, and $k * k$, respectively, apply linear test replacement and dead code elimination to computations involving i and j (assuming that they are only used for subscript computation and loop control), and subsume variables, then the above code becomes

```
        k2 = k * k;
        nk = n * k;
        mk = m * k;
        i' = 0;
        i'' = 0;
loop2:  j' = 0;
        j'' = 0;
loop:   . . .
        . . .
        access A(i' + i'' + j' + j'');
        . . .
        j'' = j'' + 1;
        if j'' ≥ k then do;
                j'' = j'' - k;
                j' = j' + k2;
                end;
        if j' < nk then go to loop;
        i'' = i'' + k;
        if i'' ≥ k2 then do;
                i'' = i'' - k2;
                i' = i' + nk;
                end;
        if i' < mn then go to loop2;
        . . .
```

For $k - 1$ out of k times through the loop, the cost of loop closing is thus only one addition, two comparisons, and two branches, which presumably is cheaper than $k - 1$ page faults, if the array is large. There are an additional two or three instructions in the inner loop to combine i', i'', j', and j'' to compute the actual address of $A(i, j)$. The final code shown above can be produced by straightforward code generation and optimization techniques, augmented by strength reduction for division and modulo.

Reference and note

1. John Cocke and J. T. Schwartz, "Programming Languages and Their Compilers," Courant Institute of Mathematical Sciences, New York University, New York, 1970.
2. A strongly connected region of a program is a subset of program statements such that control can flow between any two statements in the subset without leaving the subset. A PL/I do-loop is an example of a strongly connected region.

Received February 29, 1980; revised May 20, 1980

The authors are located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.