LITTLE Newsletter No. 18 November 10, 1972

A new array optimization for basic blocks, and some suggestions concerning the installation of optimizations into LITTLE. J. Schwartz

1. A new array optimization. Frequently in LITTLE code-sequences that occur in the SETL run-time library, the following phenomenon, typical for certain types of recursive programming, are encountered. An array A is used as a stack. Code which in some original recursive source might have appeared as

```
I = I + 1
J = J + 1
K = I * J
```

becomes transformed by the association of I, J, K with stack locations relative to a stack pointer P into code with the following appearance:

```
A(1+P) = A(1+P) + 1
A(2+P) = A(2+P) + 1
A(3+P) = A(1+P) * A(2+P) .
```

The standard basic block optimization technique is incapable of coping with this situation, since it assumes that the A(2+P) = ... changes the *whole* array A, and hence that A(1+P) must be reloaded before its use in the third line.

We shall now describe an improvement of the standard technique which removes this deficiency.

We assume that the following information will be exposed prior to the basic block optimization which we aim to describe. Indexed loads and stores have their indices reassociated as necessary, and are exposed in the form

```
... A(n+e) (a use)    or    A(n+e) = (a redefinition) .
```

Here n represents an integer constant whose value is known at compile time; e a variable, or rather really its value number in the sense of the normal basic block optimization procedure. Other operations are represented in any conventional and convenient way, cf. Cocke and Schwartz, Programming Languages etc., p. ff. Our treatment of indexed stores and loads ias

as follows:

a) As many separate 'auxiliary names' A_n are generated as are needed for all the ...A(n+e) and A(n+e) = which occur in the block.

b) The basic inherent spoil rules which apply are these:

bi) an assignment $A_n(e) = x$ sets the value number of the left-hand side equal to the value number of the right-hand side.

bii) an assignment $A_n(e) =$ does not spoil the value of $A_{n'}(e)$ if e and e' are the same value number and $n \neq n'$.

biii) any assignment $A_m(e') =$ with $e' \neq e$ spoils the value of every $A_n(e)$, $A_{n'}(e)$, etc.

c) These rules can be enforced as follows:
For each array name A introduce two quantities lastindex(A) and indexcount(A). Here
lastindex(A) -- is the value number of the last index e occurring in a store $A_n(e) =$;
indexcount(A) -- is the number of separate indices which have occurred in stores $A_n(e) =$. We count immediately successive stores $A_n(e) =$, $A_{n'}(e) =$, ... only once, but for example consider the pattern

$$A_n(e) = \ , \quad A_m(e') = \ , \quad A_k(e) =$$

to involve three separate indices since the two uses of the index value (number) e are separated by a use of the different index value (number) e'.

The rule for updating lastindex and indexcount is clearly this: on encountering a store $A_n(e)$, execute

```
if lastindex(A) ne e then lastindex(A)=e; indexcount(A)=
        indexcount(A)+1; end if;
```

d) When a value number is recorded for $A_n(e)$ (on encountering an assignment to or a use of $A_n(e)$) record with this value number the current value k of lastindex(A). At a later point of use in the same basic block, $A_n(e)$ has the same value if and only if no entervening stores $A_m(e')$ have occurred. But this is clearly the case if and only if k is still equal to lastindex(A). Hence we may use this scheme: Value numbers (for indexed quantities) are associated with pairs A_m ,e; A_m being a 'qualified array name' (qualified by a, compile-time constant integer) and e being a value number. With each such pair we also associate a quantity 'creationcount', which records the value which indexcount(A) had at the moment at which the current value number was associated with the pair A_m,e. When $A_m(e)$ is used, we check its creationcount. If this is equal to the current value of indexcount(A), the value number associated with $A_m(e)$ is unchanged, and whatever redundant operation elimination is appropriate is performed. In the contrary case, the pair A_m , e is issued a new value number, and the creationcount associated with this pair set equal to the current value of indexcount(A).

It is certainly desirable to extend this technique to an interval-based method which can be used in the presence of flow.

2. Concerning the installation of optimizations into LITTLE.

The easiest optimizations to install, and those which probably have the biggest immediate payoff, are associated rather closely with basic blocks. They can be experimented with as basic block optimizations, supplemented by additional information concerning end-conditions on block exit, which would of course have to be collected by a global analyzer. The most valuable optimizations are probably

a) basic block redundancy and constant propagation.

b) some special casing:

quantities used only for conditional transfers should not be reduced to final boolean form;

divide by 2^n, multiply by 2^n;

mask and other 'easy' constants generated by mask operations rather than loads;

quantities to be stored generated directly into X6, X7, saving moves;

and possibly a few others.

c) suppress store of dead quantities, using global live-dead information.

Going beyond this to an optimization which is more truly global, it is desirable to use live-dead tracing to find interferences within first and second level intervals (inner loops), and to do register packing in these code sections, probably using the fast Cocke packing algorithm described in the SETL notes.