15 High level optimisation 4 - Global data flow analysis

What is data flow analysis?

Many optimisations that we have seen depend on either re-ordering or eliminating statements from the intermediate code representation of the program. It is vital to be certain that such changes are safe, in the sens that they leave the behaviour of the program unchanged. To decide this it is essential to know over what ranges of the program the values used are unchanged and over what ranges they are unused. Since flow of control forms several intertwining paths through the statements of the program, it is not siply a question of identifying a unique section of code where a variable was set, but of identifying those chains of instructions through which it might have been set.

Use-definition (u-d) chaining

The use of a value is any point where that variable or constant is used in the right hand side of an assignment or is evaluating an expression.

The definition of a value occurs implicitly at the beginning of the whole program for a constant and at the point where it is the target of an assignment or read statement for a variable.

A point is defined either prior to or immediately after a statement.

Difficulties occur, as we noted in DAG analysis, when the value is an array element or a dereferenced pointer, since different names may be used in accessing the same location.

Reaching definitions

Following Aho and Ullman, we first consider which definitions in the program can reach a given point. This requires the construction of the following:

(a) A list of all points where each simple variable is defined. A *definition* is a point where a value is set for that variable.

(b) A set called $GEN[B]$, which is the set of generated definitions, i.e. for block B, those definitions which reach its end. We saw how this could be achieved when examining DAG construction, i.e. whether the node for statements which assign to that identifier which still have it as an attached identifier at the completion of the DAG.

(c) A set called $KILL[B]$, which is the set of definitions outside the block B that also have definitions inside B. This involves the sets computed in 0a and 0b).

(d) For all blocks B, the set $IN[B]$, which is all definitions reaching the point just before Bs first statement.

Once this is known, the definitions reaching any use of A within B are found by:
Let u be the statement being examined, which uses A.

1. If there are definitions of A within B before u, the last is the only one reaching u.
2. If there is no definition of A within B prior to u, those reaching u are in $IN[B]$.

To help compute IN we also compute $OUT[B]$ for each block. $OUT[B]$ includes $GEN[B]$ and definitions in $IN[B]$ which persist after B has finished.

Data flow equations

The equations which must be solves are the sets that relate INs and OUTs. For all blocks B:

1. $$OUT[B] = (IN[B] - KILL[B]) \cup GEN[B]$$

2. $$IN[B] = \bigcup_{\forall P \text{ preceding } B} OUT[P]$$

From rule 1: A definition, d, reaches the end of B iff

1. $d \in IN[B]$ and is not killed by B

   or

2. d is generated in B and not subsequently redefined there.

From rule 2: A definition reaches the beginning of B iff
it reaches the end of one of its predecessors

Non-unique solutions

Not all sets of dataflow equations have unique solutions. For instance, a block which is a loop returns to its own start. Thus B is a predecessor to itself.

Let there be a solution where $IN[B]$ and $OUT[B]$ have values IN_0 and OUT_0. Take d, IN_0, OUT_0 or $KILL[B]$. Then

$$IN[B] = IN_0 \cup \{d\}$$

$$OUT[B] = OUT_0 \cup \{d\}$$

and every $IN[B']$ and $OUT[B']$ having d added ($B' \neq B$). This is still a solution. Thus we must take the *smallest* solution for such a system.

Computing u-d chains

We now want the *u-d* chains based on these.

If a use of variable `a` is preceded in its block by a definition of `a`, this is the only one reaching it.

If no such definition precedes its use, all defiunitions of `a` in IN[B] are on its chain.

Uses of u-d chains

1. If the only definition of `a` reaching this statement involved a constant, we can substitute that constant for `a`.

2. If no definition of `a` reaches this point, a warning can be given.

3. If a definition reaches nowhere, it can be eliminated. This is part of *dead code elimination.*

Interprocedural analysis

Where procedure calling is involved, parameters must be tracked and the *transitive closure of the passing of values* determined. This greatly reduces the optimisations possible in some cases. Input of a parameter is equivalent to a definition in the calling procedure.

Optimisations depnding on data flow analysis

With the help of this sort of data flow analysis across blocks, it is possible to perform safely several of the optimisations we have considered only within the scope of a single block. These include:

Invariant code motion may be extended to move code in front of preceding blocks, if the values moved do not get killed in the intervening blocks.

Common sub-expression elimination may be extended to multiple blocks.

Dead code elimination may be extended where no ultimate use is made of a variable live at the end of the block in which its value is generated.

Loop unrolling requires that no value used in iteration k is generated in iteration $k-1$.

Parallelisation and vectorisation

The information generated in an optimising compiler may also be used in automatic parallelisation and vectorisation of code.

Parallelisation is carried out to allow code to execute across several processors in parallel. For two pieces of a program to execute safely in parallel, they must not depend on results from each other. Points where data must be exchanged between sections of code force them to synchronise. Global data flow analysis can help find parallelism in code written for sequential machines in a similar manner to loop unrolling.

SIMD parallel machines, such as the ICL DAP and the Connection Machine, allow concurrent execution of hundreds of iterations of a loop which, for instance, multiplies every element of an array by the corresponding element of another array.

Vectorisation is a technique to exploit pipelining in architectures. Two sections of code, such as those generated from unrolled loops, may not be fully independent, but it may be possible to execute part of one before its predecessor has finished. With a multi-stage pipeline in the architecture of a machine this can be exploited to speed computation.

In the late 1970s and early 1980s many vector processing co-processors were introduced, where array operations were pipelined over the iterations of loops in this way. Although

special versions of Fortran were devised to allow programmers to use whole array operations to match these capabilities, automatic vectorisation of existing code was very important in making this approach popular.

Superscalar and other novel machines have led to ever increasing demands on compilers. In a typical super-scalar processor two integer and one floating point instruction may be issued each cycle. This *micro-parallelism* requires very careful use of the information found in control flow and data flow analysis.

Even newer architectures are being designed along with their compilers to allow speculative execution of code in branches of conditional statements, for example. Estimates of the frequency of execution of alternatives, perhaps based on profiling of earlier executions, may be used here.

Inline substitution of procedures

Non-recursive procedures may be inserted inline at points where they are called. This has been very popular in Fortran compilers, where recursion is not allowed and most data is typically shared through common blocks, rather than parameter passing. Languages like C++ force inline substitution, since their compiler technology is often crude.

There is usually some cost formula used in terms of the size of a procedure and the number of times it is called, when deciding whether to perform inline substitution.