

Optimization of MIRV Programs

Application of structural dataflow

Krisztián Flautner **manowar@engin.umich.edu**
David Greene **greened@eecs.umich.edu**



1.0 Introduction

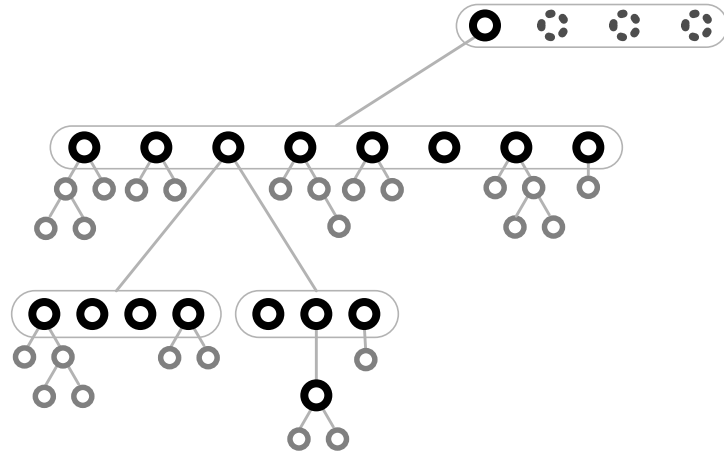
One of the driving ideas behind MIRV is its ability to preserve high level information about the source program. Control-flow constructs are not resolved to a set of labels and goto instructions but are preserved instead. The lack of arbitrary goto instructions in the language means that control-flow graph is always reducible and that the high level MIRV constructs accurately describe the structure of the MIRV program.

The analysis and transformation steps during the optimization phase exploit the availability of the high level information. This means that instead of iterative data-flow analysis on basic blocks, one can perform structural dataflow analysis on the MIRV program. Unlike in many other compilers, the high level structure does not have to be synthesized from the basic-block level since it is inherent in the representation.

The following sections describe the key ideas behind structural dataflow analysis and provide examples of analysis steps that we have implemented.

2.0 Structural dataflow analysis

The smallest unit that is used for structural dataflow analysis is a statement. Unlike in iterative dataflow analysis where a unit of analysis is a basic block, statements do not necessarily imply a sequential path of execution. The control-flow characteristics of a statement are derived from its semantic meaning. Figure 1 on page 2 shows an abstract structured program graph.

FIGURE 1.**A portion of a structured program graph**

The dark circles in the graph represent statements and the lighter nodes represent expressions. The statement nodes correspond to a high level construct such as an if statement or while statement. Statements may contain statement lists or expressions. The structure of the graph does not correspond to the control flow of the program but rather to the operators of the language and their arguments.

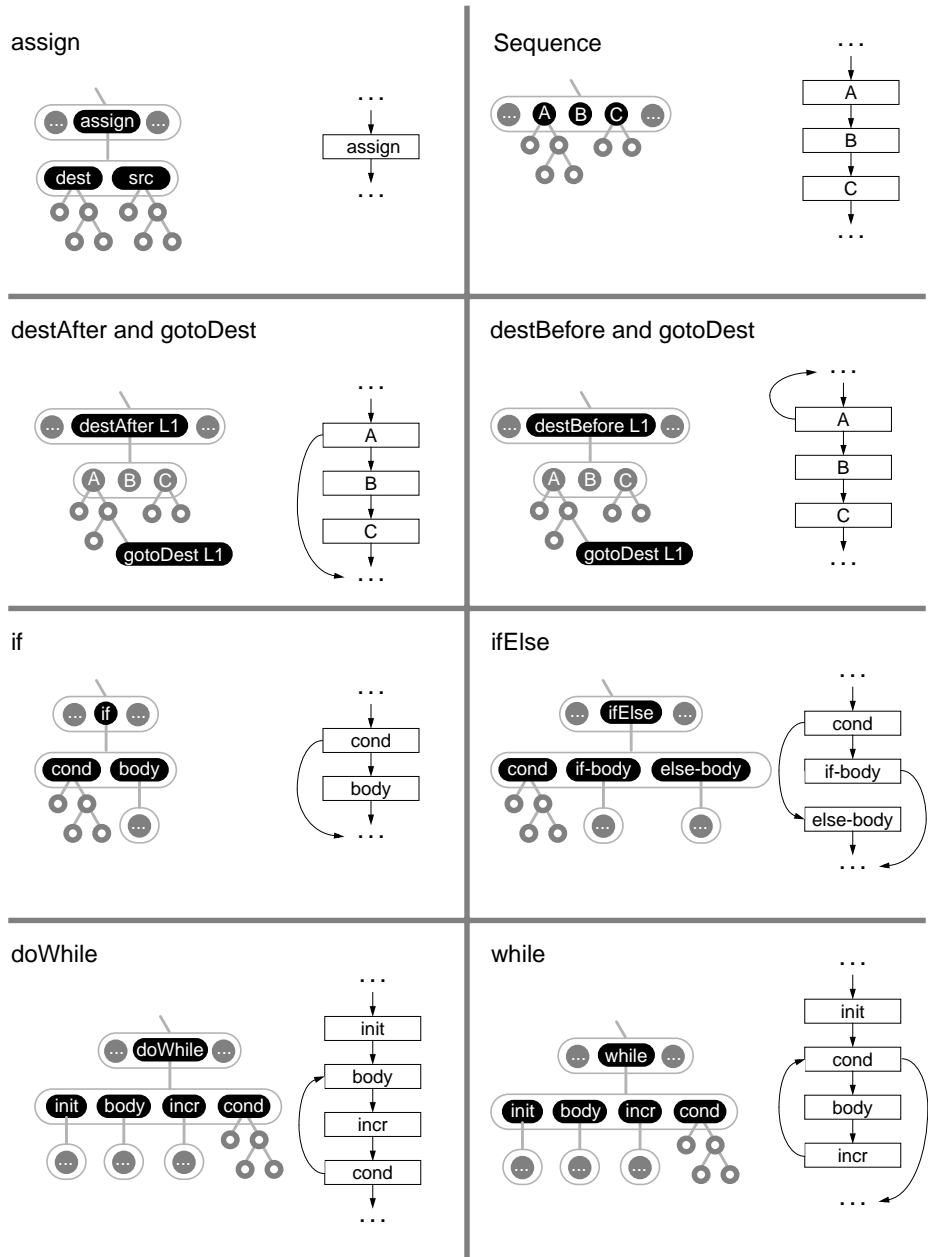
While statements in the program graph define the control-flow characteristics of the program, expressions specify the data values that are accessed. All leaf expression nodes in the tree represent a variable access. Depending on the expression's context the variable is read or written. The expression's context is defined by the set of statements that are found above it in the static program graph (in essence the expression's context is an inherited attribute whose value is computed during a traversal of the parse tree).

A crucial difference between structural dataflow and iterative dataflow analysis is that in structural dataflow temporary variables need no be analyzed. In the structured graph temporary values appear implicitly between expression nodes. However, temporaries have a unique characteristic that differentiates them from other variables; they are written exactly once and are read exactly once. This behaviour completely defines the dataflow characteristics of the node (namely that it is written once and read once by its neighbors in the graph) so that these values need not be included in analysis steps. This means that there is less work to perform using structural dataflow analysis than in the more traditional iterative analysis on basic blocks of triples.

During dataflow analysis, the variable use or definition information is propagated up to the statement to which the expression belongs and the semantics of the particular statement determines how the data is handled and how it is combined with data from other portions of the graph. Figure 2 on page 3 illustrates the statements that can be used in a MIRV program. We are still working on an appropriate jump table (switch statement) implementation, so that is not yet included in the diagram.

FIGURE 2.

Structural units (statements)



The above diagrams illustrate the layout of the structural units in MIRV. The pictures on the left in each quadrant show the structural layout while the pictures on the right show the control flow behaviour of the specified construct. All entry and exit points to/from the statements are marked explicitly on the control-flow graphs.

The destAfter and destBefore constructs are peculiar to MIRV. These constructs represents a branch target that can be the destination of a gotoDest instruction from within

the body of the “dest” statement. This allows one to break out from code at an arbitrary level of nesting in a similar way to a labeled “next” or “continue” instruction in Perl. The structure of this instruction was chosen to be easy to use for structured dataflow analysis.

In order to be able to perform a particular data-flow analysis problem, one must specify how the statements handle the dataflow information passed to them. This operation can be described in terms of inherited and synthesized attributes at a given statement node. The following descriptions of dataflow problems all specify what computations have to be performed by a given statement.

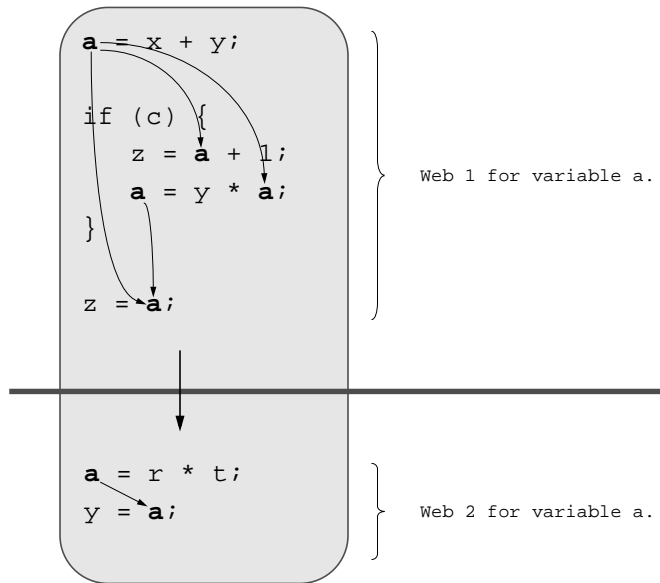
3.0 Web analysis

In order to be able to do a good job at register allocation one must be able to collect information about the liveness ranges of individual variables. Live variable analysis seeks to determine the range of instructions in the program during which a given variable is alive. The number of concurrent variables that are alive at the same time at a given point in the program determine the number of registers that are needed to hold these values.

Web analysis takes this analysis a step further and attempts to distinguish between disjoint liveness ranges (called webs) of a given variable. A web is defined as the intersection of def-use chains of a variable that have a use in common. Figure 3 on page 5 shows two webs associated with a single variable. In essence a web defines a new name for a variable that is referenced in a set of uses and definitions, thereby giving the compiler the ability to make allocation decisions at a finer granularity.

The iterative version of web analysis relies on live variable analysis as well as du-chain information. However, webs can be determined in a simpler way by using a simple modified live variable analysis algorithm if structural dataflow is used; the def-use information is implied by the structure of the graph.

FIGURE 3. Liveness ranges of a variable



This figure illustrates two webs associated with the variable “a”. The arrows in the two halves of the picture connect the variable definitions to their uses. The two webs (by definition) are independent of each other; allocation decisions for variable a in the two webs (a_1 and a_2) can be made independently of each other. In essence a_1 and a_2 are completely different variables. The fact that the programmer used the same variable name for them is incidental.

In order to explain how structured web analysis works, live variable analysis is first outlined.

3.1 Live variable analysis

This analysis seeks to determine which variables *may* be live at any given point in the program. A variable is live at a particular point if there is a path in the control-flow graph of the program between a definition and the exit along which the variable’s value may be used before redefinition. Both the iterative and the structural dataflow versions of the algorithm rely on backward dataflow analysis. Live variable analysis is a backward-may problem.

3.1.1 The iterative algorithm

The program is represented as a control flow graph, where the nodes are basic-blocks of instructions.

The first pass of the algorithm calculates the DEF and USE sets for every basic block in the graph. This is strictly a local analysis, where the DEF set corresponding to a basic block contains all variables that are defined (assigned to) in that block. The USE set

contains all variables that are locally exposed uses of variables (i.e. uses of variables, whose definition comes from the outside of the basic block).

Calculation of the IN and OUT sets is performed as a succession of iterations over the control flow graph until the sets reach a steady state (The IN sets stops changing). The algorithm starts from the last basic block of the control flow graph and setting the OUT set of the last node to be the empty set. The following two equations are applied to every instructions of the basic blocks in the graph as all the nodes are traversed from back to front:

$$\begin{aligned}
 IN(i) &= (OUT(i) - DEF(i)) \cup USE(i) \\
 OUT(i) &= \bigcup_{j \in Succ} IN(j)
 \end{aligned}
 \tag{EQ 1}$$

3.1.2 The structural algorithm

Unlike in the iterative algorithm, a USE of a variable is simply any reference to the variable, since all use of a variable on a structured graph is a locally exposed use. Also, just

TABLE 1.

Live variable analysis algorithms

Statement	IN
assign	$IN = (OUT - DEF) \cup USE$
Sequence	<p>Visit all statements in the sequence in reverse order. Set the OUT_n set for statement n to be IN_{n+1} (if $n + 1 >$ number of elements in the sequence, then $OUT_n = OUT$).</p> $IN = IN_1$
destAfter	$IN = IN_{body}$ $SAVE_{DestID} = OUT$
destBefore	$SAVE_{DestID} = IN_{body}$ <p>Iterate over the body once more.</p> $IN = IN_{body}$
gotoDest	$IN = OUT \cup SAVE_{DestID}$
if	$OUT_{body} = OUT$ $OUT_{cond} = IN_{body} \cup OUT$ $IN = IN_{cond}$

TABLE 1. Live variable analysis algorithms

Statement	IN
ifElse	$OUT_{ifBody} = OUT$ $OUT_{elseBody} = OUT$ $OUT_{cond} = IN_{ifBody} \cup IN_{elseBody}$ $IN = IN_{cond}$
doWhile	$OUT_{cond} = OUT \cup IN_{body}$ $OUT_{incr} = IN_{cond}$ $OUT_{body} = IN_{incr}$ <p>Repeat the previous steps once.</p> $OUT_{init} = IN_{body}$ $IN = IN_{init}$
while	$OUT_{cond} = OUT \cup IN_{body}$ $OUT_{incr} = IN_{cond}$ $OUT_{body} = IN_{incr}$ <p>Repeat the previous steps once.</p> $OUT_{init} = IN_{incr}$ $IN = IN_{init}$

as in the iterative version of this algorithm, a definition of a variable is always in the DEF set.

Table 1 on page 6 shows the process of performing structural live variable analysis. The table contains an algorithm corresponding to each statement in the language that calculates the IN set. The algorithm is best written in terms of synthesized and inherited attributes. The current notation may be a little hard to read, please follow the following rules:

- The OUT set refers to the current statement's OUT set. This set is an inherited attribute that gets passed down to the statement during the tree traversal.
- Assignment to $OUT_{subscript}$ means that a set of data is being passed down to the part of the statement identified as the subscript. This operation refers to the passing of an inherited attribute to a statement at a lower level.
- Referring to $IN_{subscript}$ on the right hand side implies the evaluation if the statement identified by the subscript. The value of $IN_{subscript}$ is the set synthesized by evaluated statement.

Note, that structural analysis does not entirely eliminate iteration from the algorithm. In some cases one must traverse a subgraph multiple times.

Once the IN and OUT sets are computed for a given statement, the two sets are associated with the appropriate statement node. This step is not shown in the previous table since it is always done the same way, at the end of each computation step.

3.2 The structural web creation algorithm

Given the structural version of live variable analysis, it is easy to modify it to get the web information. The only modification that needs to happen is an addition of a preserve set into the IN equation.

$$\begin{aligned}
 IN(i) &= (OUT(i) - (DEF(i) - PRES(i))) \cup USE(i) \\
 OUT(i) &= \bigcup_{j \in Succ} IN(j)
 \end{aligned}
 \tag{EQ 2}$$

The PRES set (preserve set) of a given statement contains variables that should not be taken out of the OUT set, even if they were defined in the statement. Since all the variables in the PRES set are also contained in the OUT set of the given block, the equation can be rewritten as:

$$IN(i) = (OUT(i) - DEF(i)) \cup PRES(i) \cup USE(i)
 \tag{EQ 3}$$

The preserve set causes the meaning of the IN set to be redefined. It will contain a name of a variable even if it is redefined by the current statement, if the current statement is conditionally executed.

TABLE 2.
Web analysis algorithms

Statement	IN
assign	$IN = (OUT - DEF) \cup PRES \cup USE$
Sequence	Visit all statements in the sequence in reverse order. Set the OUT_n set for statement n to be IN_{n+1} (if $n + 1 >$ number of elements in the sequence, then $OUT_n = OUT$). $IN = IN_1$
destAfter	$IN = IN_{body}$ $SAVE_{DestID} = OUT$
destBefore	$SAVE_{DestID} = IN_{body}$ Iterate over the body once more. $IN = IN_{body}$
gotoDest	$IN = OUT \cup SAVE_{DestID}$

TABLE 2.

Web analysis algorithms

Statement	IN
if	$PRES_{body} = OUT$ $OUT_{body} = OUT$ $OUT_{cond} = IN_{body} \cup OUT$ $IN = IN_{cond}$
ifElse	$PRES_{ifBody} = OUT$ $OUT_{ifBody} = OUT$ $PRES_{elseBody} = OUT$ $OUT_{elseBody} = OUT$ $OUT_{cond} = IN_{ifBody} \cup IN_{elseBody}$ $IN = IN_{cond}$
doWhile	$OUT_{cond} = OUT \cup IN_{body}$ $OUT_{incr} = IN_{cond}$ $OUT_{body} = IN_{incr}$ Repeat the previous steps once. $OUT_{init} = IN_{body}$ $IN = IN_{init}$
while	$PRES_{body} = OUT_{body}$ $OUT_{cond} = OUT \cup IN_{body}$ $OUT_{incr} = IN_{cond}$ $PRES_{incr} = OUT_{incr}$ $OUT_{body} = IN_{incr}$ Repeat the previous steps once. $OUT_{init} = IN_{incr}$ $IN = IN_{init}$

When no addition is specified to the preserve set, then the existing contents of the preserve set are passed down to a node below.

In order to find the webs in the program, all one has to do after marking up the graph with the IN/OUT information is to traverse the graph once more and compare the OUT set of the current statement with the IN set of the following statements. If the next IN set contains a variable that is not in the current OUT set, then a new web is created. On the other hand if the current OUT set contains a variable that is not in the next IN set, then a web is destroyed.

4.0 Structured Reaching Definition Analysis

The reaching definition dataflow problem is a *forward-may* problem. The propagation of the dataflow information proceeds in a forward direction because the OUT sets of each structure are calculated from a function of the IN, GEN and KILL sets. It is a “may” problem because a definition appearing in IN is only killed if it can be proven that the definition cannot reach the end of the structure. For example, in an if-else construct, the definition must be killed in both the then and else clauses for it be killed within the if-else structure.

MIRV is a structured intermediate form. Thus structural dataflow analysis maps naturally onto the internal representation used by the compiler. This structural analysis occurs in two phases. First, the GEN and KILL sets for each structure are built from the GEN and KILL sets from child structures. Then (in the case of forward problems), the IN set presented to a structure is propagated down to and through the child structures to produce an OUT set for the high-level structure and its child structures.

The first uses a static set of equations for each structure type to produce the GEN and KILL sets for each structure as it is encountered by the analyzer. That is, given a set of GEN and KILL sets for its child structures, the GEN and KILL sets for the parent structure can be computed. For example, analysis of a MIRV if-else structure involves three child nodes: the condition, then clause and else clause. Reaching definition analysis of such a structure uses the following equations:

$$GEN_{if} = (GEN_{cond} - (KILL_{then} \cap KILL_{else})) \cup GEN_{then} \cup GEN_{else} \quad \text{(EQ 4)}$$

Structured Reaching Definition Analysis

$$KILL_{if} = (KILL_{cond} - (GEN_{then} \cup GEN_{else})) \cup (KILL_{then} \cap KILL_{else}) \quad (EQ 5)$$

Table 3 lists the GEN and KILL equations for each type of MIRV structure. Due to the changing nature of the MIRV language, analysis was not implemented for switch and case statements.

TABLE 3.

Pass 1 Reaching Definition GEN and KILL Equations for MIRV Structures

Type	GEN KILL
destBefore	$GEN_{DB} = GEN_{block}$ $KILL_{DB} = KILL_{block}$
destAfter	$GEN_{DA} = GEN_{block}$ $KILL_{DA} = KILL_{block}$
block	$GEN_{block} = GEN_1 \cup \bigcup_{i=2}^{n-1} (GEN_i - KILL_{i+1}) \cup GEN_n$ $KILL_{block} = \bigcup_{i=1}^{n-1} (KILL_i - GEN_{i+1}) \cup KILL_n$
if	$GEN_{if} = GEN_{cond} \cup GEN_{then}$ $KILL_{if} = KILL_{cond} - GEN_{then}$
ifElse	$GEN_{if} = (GEN_{cond} - (KILL_{then} \cap KILL_{else})) \cup GEN_{then} \cup GEN_{else}$ $KILL_{if} = (KILL_{cond} - (GEN_{then} \cup GEN_{else})) \cup (KILL_{then} \cap KILL_{else})$
while	$GEN_{while} = (GEN_{init} - KILL_{cond}) \cup$ $(GEN_{body} - (KILL_{incr} \cup KILL_{cond})) \cup (GEN_{incr} - KILL_{cond}) \cup GEN_{cond}$ $KILL_{while} = (KILL_{init} - GEN_{cond}) \cup$ $(KILL_{body} - (GEN_{incr} \cup GEN_{cond})) \cup (KILL_{incr} - GEN_{cond}) \cup KILL_{cond}$
doWhile	$GEN_{do} = (GEN_{init} - (KILL_{body} \cup KILL_{incr} \cup KILL_{cond})) \cup$ $(GEN_{body} - (KILL_{incr} \cup KILL_{cond})) \cup$ $(GEN_{incr} - KILL_{cond}) \cup GEN_{cond}$ $KILL_{do} = (KILL_{init} - (GEN_{body} \cup GEN_{incr} \cup GEN_{cond})) \cup$ $(KILL_{body} - (GEN_{incr} \cup GEN_{cond})) \cup$ $(KILL_{incr} - GEN_{cond}) \cup KILL_{cond}$
assign	GEN_{assign} $KILL_{assign}$

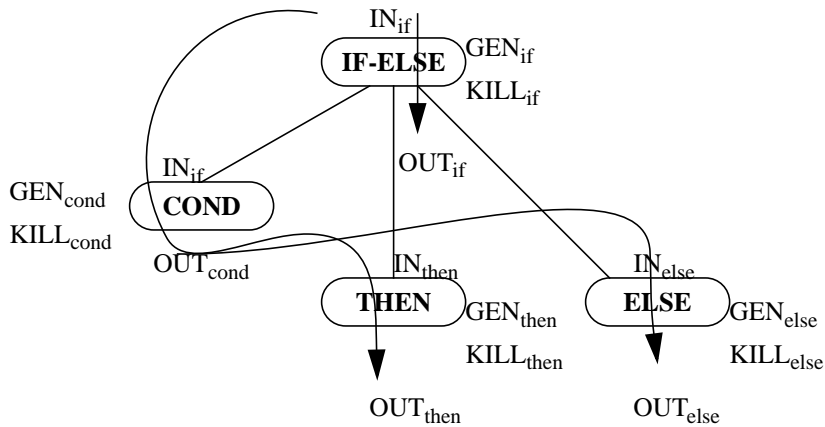
Block statements use an iteration of the classic *sequence* structural equation ([1]). The variable n denotes the number of statements in the block structure. Assignment statements produce the base GEN and KILL sets.

After computing the GEN and KILL sets for all structures in the code, the second pass propagates this information through the structure to the child nodes. It is important to keep the order of execution in mind. In the if-else structure, the IN set to the if structure must be propagated through the condition, whose OUT set becomes the IN sets of the then and else clauses.

This operation is shown in Figure 4 for a MIRV if-else statement. The high-level if-else statement is represented by the top oval. The bottom three ovals, from left to right, represent the condition checked, the then statement and the else statement. Information into the if-else is propagated to the condition node, where an OUT set is calculated. This OUT set becomes the IN set for the then and else clauses, each of which computes an OUT set. The OUT set for the entire structure is computed in terms of the IN set to the structure and the GEN and KILL sets computed for the structure in pass 1.

FIGURE 4.

Pass 2 Structural Analysis of an IF-ELSE Construct



Dataflow information is propagated through the condition node and down the IF-ELSE tree to the then and else child nodes, where the OUT set is computed for each child. The OUT set for the entire IF-ELSE structure is computed from IN_{if} , GEN_{if} and $KILL_{if}$.

Pass 2 is slightly more complex for loop structures. In the case of a loop, iteration must be performed to get the final OUT set for the loop. For example, in a while statement, any OUT from the increment statement must be fed back into the IN of the condition expression and the calculations re-run until a fixed point is reached. A similar operation must occur for a gotoDest within a destBefore statement, as the goto defines a loop edge in the graph.

TABLE 4.

Type	OUT Computation
gotoDest	$OUT_{GD} = (IN_{GD} - KILL_{GD}) \cup GEN_{GD}$ $SAVE_{destID} = OUT_{GD}$
destBefore	<p style="text-align: center;">Iteration Start</p> $IN_{DB} = IN_{DB} \cup SAVE_{destID}$ $IN_{body} = IN_{DB}$ <p style="text-align: center;">Iterate until a fixed point is reached</p> $OUT_{DB} = (IN_{DB} - KILL_{DB}) \cup GEN_{DB}$
destAfter	$IN_{body} = IN_{DA}$ $OUT_{DA} = (IN_{DA} - KILL_{DA}) \cup GEN_{DA}$
block	$IN_i = IN_{block}$ $\forall (i \in \{stmt_{block}\}); IN_i = IN_{i-1}, OUT_i = (IN_i - KILL_i) \cup GEN_i$ $OUT_{block} = (IN_{block} - KILL_{block}) \cup GEN_{block}$
if	$IN_{cond} = IN_{if}$ $IN_{then} = OUT_{cond}$ $OUT_{if} = (IN_{if} - KILL_{if}) \cup GEN_{if}$
ifElse	$IN_{cond} = IN_{if}$ $IN_{then} = OUT_{cond}$ $IN_{else} = OUT_{cond}$ $OUT_{if} = (IN_{if} - KILL_{if}) \cup GEN_{if}$
while	$IN_{init} = IN_{while}$ <p style="text-align: center;">Iteration Start</p> $IN_{cond} = OUT_{init} \cup OUT_{incr}$ $IN_{body} = OUT_{cond}$ $IN_{incr} = OUT_{body}$ <p style="text-align: center;">Iterate until a fixed point is reached</p> $OUT_{while} = (IN_{while} - KILL_{while}) \cup GEN_{while}$

TABLE 4.

Type	OUT Computation
doWhile	$IN_{init} = IN_{while}$ <p style="text-align: center;">Iteration Start</p> $IN_{body} = OUT_{init} \cup OUT_{cond}$ $IN_{incr} = OUT_{body}$ $IN_{cond} = OUT_{incr}$ <p style="text-align: center;">Iterate until a fixed point is reached</p> $OUT_{while} = (IN_{while} - KILL_{while}) \cup GEN_{while}$
assign	$OUT_{assign} = (IN_{assign} - KILL_{assign}) \cup GEN_{assign}$

The equations presented here use similar notations to those in Table 2. The SAVE set for the gotoDest statement allows the back edge from a gotoDest to the head of a destBefore block to be iterated over.

Table 4 describes the operation of reaching definition pass two. Propagation down the tree is implied by assigning the IN set of a child node to be the IN set of the parent node, or the OUT set of a previous node in a sequence. This sort of assignment implies a recursive descent through the MIRV tree.

Once reaching definitions have been calculated, it is useful to construct DU and UD chains. Having these chains greatly simplifies many transformations, including loop induction variable strength reduction. These chains are surprisingly easy to implement. For DU chains, one simply examines the definition IN set at each statement. For each definition in the set, all uses of variables in the statement are added to the chain for that definition. In fact, it is simpler than that. There is no reason to walk through a block structure and keep track of definitions as they are generated and killed. Every statement within the block has a reaching definition IN set associated with it. Only the leaf statements and expressions (in the case of branch conditions) in the MIRV tree need to be considered. Building UD chains requires a similar approach.

For both DU- and UD-chains, having a two-pass dataflow system works quite well. While walking through the MIRV tree, each statement can be annotated with a USE attribute listing the unique uses corresponding to its source variables. Since there is no concept of killing USE attributes, the “pseudo-dataflow” algorithm simply takes the union of child node USE attributes to be the USE attribute GEN set for the parent statement. This occurs in the first pass. Since no USE attributes are ever killed and DU- and UD-chain construction is only concerned with USE attributes at a particular statement (the GEN set), it is not necessary to run the second propagation pass. This can speed up DU- and UD-chain computation, because we can identify large blocks of statements that contain no uses of interest.

5.0 Loop Induction Variables and Strength Reduction

A *loop induction variable* is a variable whose values follow an arithmetic sequence within the loop ([1]). For example, in the code

```
for(i = 0; i < 10; i++) {  
    j = 2*i + 1;  
    a = j;  
}
```

both *i* and *j* are loop induction variables. The sequence followed by *i* is {0, 1, 2, 3, ...}. Variable *j* follows a sequence {1, 3, 5, 7, ...} that depends on *i* and is thus called a *dependent induction variable*. Variable *i* is a *base induction variable* because its values do not depend on the values of any other variables.

Strength reduction can be applied to transform the multiplication of *i* into an addition to *j*. The idea is to discover the relationship between *i* and *j* and exploit that relationship to increment *j* whenever *i* is incremented. The above loop can be transformed to the following:

```
i = 0;  
tj = 2*i + 1;  
tk =  
for(; i < 10;) {  
    j = tj;  
    a = tj;  
    i++;  
    tj += 2;  
}
```

The multiplication has been copied outside the loop, and has been changed to addition within the loop. As will be demonstrated shortly, the replacement of *j* by *tj* in the assignment to *a* is critical in the MIRV compiler. The strength reduction transformation is only valid if *j* maintains the same relationship to *i* along all paths through the loop. This means that if *j* is defined in one branch of an if-else statement, it must be defined by the same relationship to *i* in the other branch. There is also the possibility that *i* is redefined in the loop, in which case *i* may be split into two variables and the correct relationships to dependent induction variables can be determined.

When identifying induction variables, there is a trade-off between speed and simplicity. Take the following example:

```
for(i = 0; i < 10; i++) {  
    j = 2*i;  
    k = 2*j + 3;  
}
```

Variable *j* is dependent on *i*, and *k* is dependent on *j*. Muchnick presents an algorithm to discover the relationship between *k* and *i* in the same pass as the relationship between *j*

and i is discovered. However, determining that k is indeed an induction variable requires some extra analysis. Two additional rules must be enforced: there can be no assignments to j from outside the loop that reach its use in the definition of k . This can happen if k is defined in an if statement. Furthermore, there must not be any assignments to i between the assignment to j and the assignment to k . If there were, it might destroy the j - k relationship.

In the MIRV compiler, a simpler, iterative approach is taken. In the first pass through the loop, base induction variables and induction variables immediately dependent on the base are identified. The strength reduction transformation is performed, generating temporaries for the dependent variables. These temporaries then become base induction variables in the second pass through the loop, at which point we can find induction variables based on them, which are exactly the variables that were dependent on the first-pass dependent induction variables. This eliminates the two extra checks above, because there is no “second-order” relationship to disrupt.

As an example, consider the code above. After the first iteration, j is discovered to be dependent on i and strength reduction is performed:

```
i = 0;
tj = 2*i;
for(; i < 10;) {
    j = tj;
    k = 2*tj + 3;
    i++;
    tj += 2;
}
```

During the second pass, tj is discovered to be a base induction variable, with k dependent on it. A second run of the transformation results in the following code:

```
i = 0;
tj = 2*i;
tk = 2*tj + 3;
for(; i < 10;) {
    j = tj;
    k = tk;
    i++;
    tj += 2;
    tk += 2;
}
```

To handle the requirement that induction variables appearing in one half of an if-else construct must also appear in the other half with the same equation, “pseudo-dataflow” can be used. Recall that pseudo-dataflow (running only the first pass of the general dataflow algorithm) was used to construct USE sets used by the DU- and UD-chain constructors. Here, the first pass verifies the if-else induction variable requirement. The GEN set of an if-else contains only those variables in both branches that have the same

induction equation. This is done by intersecting the GEN sets of the then and else clauses. This approach also solves the nested if problem. A while statement simply takes the union of the GEN sets of its child nodes. There is still a bug in the implementation. When a dependent induction variable appears (with identical induction equations) in both branches of an if-else construct, two different temporaries are used when replacing the assignment to the dependent variable. This will cause problems for any later uses of the dependent variable, because one of the temporaries will be substituted for the use.

The variable splitting problem described earlier could also be handled by the one-pass dataflow. A redefinition of an induction variable could be defined to kill the original definition of that same induction variable in the loop. Members of the loop KILL set are exactly those induction variables that need to be split. This has not yet been implemented in the MIRV compiler.

Strength reduction also includes simple operations such as converting multiplies to shifts and adds. These were not implemented due to time constraints.

6.0 Future Work

In addition to the obvious need to make improvements to handle more cases, especially the variable-splitting case, the following observations about the MIRV compiler framework were observed:

- The attribute system needs a complete overhaul. Most of the ugliness in the current code (extra loops and so forth) are needed to keep the induction variable attributes consistent with each other. For example, when the definition of a dependent induction variable is replaced by an assignment from a temporary, that temporary needs to be used in every other replacement for definitions of the induction variable in the loop. This happens when the two branches of an if-else contain the same definition of a dependent induction variable. The code must loop through all the statements it has recorded as defining this variable, note the temporary to be used, and store the updated attribute back into the attribute system. This is extremely time consuming. Previously, attributes had been unique to each construct, so storing values was fine. Now, communication must occur, and that is better handled through pointers.
- The dataflow engines can be generalized as was done in earlier projects. It should be possible to write four major generalized algorithms: forward-may, forward-must, backward-may and backward-must. If a designer requires a different type of engine, it will have to be written from scratch, but the four major engines should cover most of the need.
- The loop constructs are too complex. We originally thought that the incr and init fields of loops would be useful but we have not yet found a good use for them.

7.0 Appendix

The following pages present sample output from the MIRV reaching definition and strength reduction passes. Because the textual representation of MIRV is quite verbose,

it was necessary to keep the datasets small. Reaching definition data set one simply verifies that the algorithm is correct for a simple for loop. The output includes the DU and UD chains computed. Strength reduction data set one is a simple test of basic functionality. Data set two illustrates the if-else bug described above. The final data set illustrates multilevel induction variable dependence (k depending on j which depends on i).

References

- [1] Muchnick, Steven S., *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco (1997)