Reprinted from the

Proceedings of the GCC Developers' Summit

June 28th–30th, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon Incorporated*
C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
Ian Lance Taylor, *Google*
C. Craig Ross, *Linux Symposium*
Andrew J. Hutton, *Steamballoon Incorporated*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Changes to RTL Dataflow Analysis

Danny Berlin
Google
dannyb@google.com

Kenneth Zadeck
Natural Bridge, Inc.
zadeck@naturalbridge.com

Abstract

Significant revisions have been made to DF, the alternative RTL dataflow analysis module for the dataflow branch, to make it suitable for everyday use as a provider of both backend liveness information, as well as various other dataflow facts (use-def and def-use chains, death notes and register information). These changes include:

- Enhancement of the RTL scanning so that it now encapsulates all of the hard register special cases that were scattered throughout the backend.
- Revision of the interfaces to better support the solution of abstract dataflow problems.
- Replacement of the use-def and def-use chain algorithms that speed up their computation by up to three orders of magnitude.
- Removal of the non working incremental dataflow interface.

Additionally several phases of the compiler, such as the global register allocator, DCE, and DSE have been modified to use the new DF rather than FLOW. This change has provided for better generated code as well as faster compilation.

All of these will be discussed in the full paper.

1 Motivation

The back end of GCC uses `flow.c` to perform the dataflow analysis.

- **The flow analysis engine is archaic.** The generally accepted method for dataflow analysis is to scan a basic block once building a summary of the instructions that occur within that block. Global analysis then uses that summary as its input. `Flow.c` rescans each block at each step of the iteration. This is quite expensive.
- **The iteration technology is primitive.** Significant improvements to iterative dataflow were first developed by Hecht [5] in 1975. The technology in flow is inferior to this.
- **Unrelated problems have been added to the iteration.** Many of the problems, like finding auto inc instructions derive no benefit from being done inside the main iterative loop aside from being able to reuse some intermediate structure.
- **The current algorithm may not terminate.** This has been *solved* by only allowing a certain number of iterations. However, when this limit is reached, flow is currently left with incorrect answers.
- **Flow does not get the *best* solution when used incrementally.** There are many

possible correct solutions to the dataflow equations. However, only one solution is *minimal*. This is the desired solution.

The back end of GCC uses FLOW to perform the dataflow analysis.

2 Underlying Technology

Dataflow analysis is defined over a graph of basic blocks, the *control flow graph*, (cfg). Dataflow problems can be characterized in several ways:

direction Dataflow problems are either *forward*, information flows in the direction of the cfg edges, *backward* information flows against the edges in the cfg or *bidirectional* information can flow in both directions. The *logical predecessors* of a block basic block *b* are the cfg predecessors if the problem is forwards or bidirectional and the cfg successors if the problem is backwards. The *logical successors* are defined in a corresponding way.

In each basic block, two sets are defined, an *in* and and *out* set. For forwards and bidirectional problems the *in* set is at the top of the block and the *out* set is at the bottom of the block. This is reversed for backwards problems.

domain The set of items to be analyzed: commonly used domains are the set of registers, the set uses, or the set of definitions.

confluence The operation to be executed at the merge point in the cfg. Confluence operators are either simple or complex. The simple operations are generally *set union* where the *or* operator is used for bit vectors, or *set intersection* where the operator used at merge points is set intersection.

However complex operations may also be used. In constant propagation, the confluence operator is equality over values. While DF is capable of solving dataflow problems with complex merge operations, the technology used in DF is generally not the appropriate for this kind of problem.

A dataflow problem is a set of simultaneous equations.

> The *in set* of each basic block *b* is defined to be the confluence operator applied to the *out set* of each logical predecessor of *b*.
>
> The *out set* of each basic block *b* is defined to be the transfer function applied to the *in set* of *b*.

There are many possible correct solutions to these equations. The goal of the dataflow is to find the *best* solution to the above system of simultaneous equations. For a set union (intersection) problem we want the *smallest solution (largest solution)*, i.e. the solution with the fewest (most) bits that is *correct*.

2.1 Solving Dataflow Equations

The dataflow equations can be solved in a variety of ways. Most of the techniques fall into two categories:

elimination algorithms The earliest elimination techniques tried was Gaussian elimination. However it was realized very quickly that the structure of the control flow graph could be taken advantage of yield faster solutions. There were many techniques developed [1, 4, 7].

Elimination algorithms are built on the idea of divide and conquer. It is easy to compute the solution to the data flow equations for control flow graphs of certain forms. The idea is to parse the control flow graph into a recursive tree that contains only these forms. Then the dataflow solution can be obtained by walking the tree in a particular order.

While elimination algorithms are generally fast, they do not work if the control flow graph cannot be parsed into these simple forms. In particular, any graph that contains a multiple entry loop must use other techniques.

iterative algorithms The earliest iterative algorithms were simple worklist iteration, described below and implemented in FLOW. This has the advantage over the elimination techniques that it works for all control flow graphs. The disadvantage is that it is slow.

Matthew Hecht [5] observed that if you impose a certain structure onto the worklist, the iteration will tend to converge very quickly. Forward problems process the blocks in reverse postorder and backwards problems process the blocks in postorder. Bidirectional problems can be solved by alternating passes of postorder and reverse postorder. For programs that have only single entry loops, the number of passes is never greater than one plus the maximum loop nesting.

Atkinson and Griswold [2] made a small modification where an additional depth first search is added in certain conditions. They showed that this modification improved Hechts algorithm for many common cases. This is the technique implemented in DF.

There are three steps to solving a dataflow problem:

- The first step in solving a dataflow problem is typically building the *transfer functions* for the basic blocks. The transfer function for block *b* describes how the instruction within *b* can be used to compute the *out set* of *b* from the *in set* of *b*.

  While it is certainly correct rescan *b* each time we wish to compute *b*'s *out set*, this is too expensive since the *out set* may have to be computed many times before the equations converge.

  For most simple dataflow problems it is possible to summarize the action of a basic block into something that is much simpler than rescanning the block. This is almost always true for problem that represent their results as bit vectors. For these problems, the summary for a block is typically represented as two bit vectors, *kill* and *gen* where the kill bitvector knocks bits out of the vector and the gen bitvector adds other bits back.

- The second step is to initialize the *in* and *out* sets for each block. For set union problems the sets can be initialized to the empty set and for set intersection problems the sets can be initialized to the universal set. However it has been observed that convergence is faster if the *out* set is initialized to the *gen* for set union problems or *kill* for set intersection problems.

- The third step is to actually solve the equations. There are a wide variety of techniques that have been proposed over the years. Almost all compilers use some form of fixed point iteration using a worklist.

  The technique implemented in FLOW is based on the simple worklist iteration

```
worklist <- all blocks
until (worklist is empty) {
  take b off the worklist
  b->in = empty set
  foreach logical preds p of b
    b->in |= p->out
  temp = trans_function(b, b->in)
  if (temp != b->out) {
    b->out = temp
    foreach logical succ p of b
      add p to worklist
  }
}
```

above. However, no transfer functions are ever computed. Instead the instructions in a block are rescanned each time the block is processed by the iteration.

3 Underlying Technology

3.1 Predefined Dataflow Problems

Unlike the analysis in FLOW which only solves live variables, DF is capable of solving any forwards or backwards dataflow problem[1] Several of these problems have been defined in `df-problems.c` and are usable in any pass in the backend that maintains a control flow graph. There are nine predefined dataflow problems that are packaged in a way that is easy to use:

Scan (SCAN) DF works on an abstraction of the RTL. The scanning phase builds that abstraction. There are several options control the scanning. The most common option controls if hard registers are to be considered in addition to pseudo registers.

For each instruction, the sets of registers defined and of registers used are created as well as the the set of multiword references. For each basic block *b* there is also the set of artificial uses and defs that occur at the bottom and top of *b*. Artificial registers are implicit uses or definitions of registers that cannot be attached to explicit instructions. For instance, before instruction selection, the stack and frame pointers are considered live everywhere. Exception handling also give rise to artificial uses and definitions.

Technically scanning is not a dataflow problem in that there are not equations to solve. What this problem does is compute datastructure that make the computation of the transfer function for the other problems efficient.

Live Registers (LR) The live registers problem answers the question "what registers still contain values that are used later in the program?"

At the end of computation, there is a bitvector at the bottom of each basic block, *b*, that contains the set of registers whose value may be used on some path reachable from *b* to the exit of the program. Live variables is a backwards, set union, problem where each slot in the bit vector represents one register. *Gen* is the set of registers that are used in the block. *Kill* is the set of assignments to whole registers. The bitvector at the top of *b* is the union of the bitvectors at the bottom of the preds of *b*.

Uninitialized Registers (UR) The uninitialized registers problem answers the question "what registers are used before they are defined?"

At the end of computation, there is a bitvector at the top of each basic block, *b*, that contains the set of registers which

[1] With a small amount of work, bidirectional dataflow problems could be accommodated.

may provide values on some path from the beginning of the function to *b*. Uninitialized registers is a forwards, set union, problem where each slot in the bit vector represents one register. *Gen* is the set of registers that are set in the block. *Kill* is the set of clobbers to whole registers. The bitvector at the bottom of *b* is simple the union of the bitvectors at the bottom of the preds of *b*.

UR with Early Clobber (UREC) This problem is a specialization of the uninitialized registers problem that takes into account "early clobber" instructions. This processing is over conservative and should be incorporated directly into the interference graph building. When this happens, this problem will go away.

Reaching Uses (RU) The reaching-uses problem answers the uses analogue of the Reaching definitions problem, "which uses of a register may reach this definition site?" If a use site reaches some point in the program, it means not every path in the program redefines that register.

This is a very expensive problem to compute because there are a large number of uses in a large function and each use requires one slot in all of the bitvectors. Thus, the use of this problem should be discouraged. Currently it is only used in modulo scheduling. Getting rid of the use of this problem would most likely speed up that phase[2].

Reaching Definitions (RD) The reaching definitions problem answers the question "which definitions of a register may reach this point in the program?" If a definition site reaches some point in the program, it means not every path to that point in the program kills the definition.

Chain Building (CHAIN) Def-Use and Use-Def chains provide explicit chains formed from either the reaching uses, or reaching definitions problems. In particular, given a use of a register, the use-def chains will provide links to all definitions of that register that may reach that use. Given a def of a register, the def-use chains will provide links to all uses of the register the definition reaches.

Building chains is also not technically a dataflow problem because the construction of either type of chain is based on the solution of the reaching definitions problem.

Register Information (RI) Register information is a collection of information about registers used by passes such as the register allocator. This includes the number of references made to the register, how many calls the register lives across, and other miscellaneous information used in register allocation heuristics. This is the same information about how it builds this information so the information that was computed in FLOW with the `PROP_REG_INFO` parameter however the information produced here is more precise that that computed in FLOW.

`REG_DEAD` **and** `REG_UNUSED` **Notes** `REG_DEAD` and `REG_UNUSED` notes are simple information that was previously provided by FLOW. `REG_DEAD` notes represent kills of registers. Anytime a register dies in an instruction, a `REG_DEAD` note is generated. `REG_UNUSED` notes represent the last use of a register. If no further uses of the register occur in the program, a `REG_UNUSED` note is generated. This is

[2] In the original version of DF, use-def chains were built using this problem. Since modulo scheduling used use-def chains, this problem was available for free. The current implementation of DF builds both use-def and def-use chains from reaching definitions so using this problem is expensive.

the same information about how it builds this information so the information that was computed in FLOW with the `PROP_DEATH_NOTES` parameter however the information produced here is more precise that that computed in FLOW.[3]

3.2 Other Features of FLOW

FLOW has also become a catch basin for a wide variety of transformation that have nothing to do with dataflow analysis except that they utilize some datastructure that was private to FLOW. These include:

- Cleanup of conditional assignment statements with a basic block.
- Combining memory references and increment/decrement instructions into pre and post increment instructions.
- Discovery of functions that change the stack pointer.
- Computation of register use statistics.

There is little synergy between these problems and the rest of dataflow analysis. Thus, we have decided to either make these separate passes or make it a separate dataflow problem.

3.3 Dead Code Elimination

Dead code elimination (DCE) and dead store elimination (DSE) are handled in `dce.c`. There are two dead code elimination algorithms and one dead store elimination algorithm.

[3] `LOG_LINKS`, which are now only used by combine have been integrated into `combine`. The use of this datastructure is discouraged but `combine` is difficult to rewrite.

3.3.1 The First Dead Code Elimination Algorithm

The first DCE was developed by Richard Sandiford of CodeSourcery. This algorithm is based on the optimistic dead code elimination in [3] but differs in two important ways: it uses use-def chains rather than SSA form and it currently does not utilize the control dependence graph to remove dead branches. The latter difference will be fixed when time permits.

The overall form of the algorithm is to

1. build use-def chains.
2. mark instructions that can never be dead as live. Everything else is assumed dead.
3. iteratively mark any instructions as live if it is used by something live.
4. delete everything marked live.

3.3.2 Dead Store Elimination

The dead store elimination was also developed by David Sandiford. It deals with two forms of dead stores: stores in the exit block that store to into the stack frame and stores, whose value is stored over before the value can be read.

To find the latter, a dataflow problem is solved where each symbolic address is modeled with position in the bit vector. The flow equations track which stores may reach other instructions.

3.3.3 The Second Dead Code Elimination Algorithm

The second dead code elimination algorithm was implemented by Kenneth Zadeck and is

similar in principle to the existing dead code elimination in FLOW; i.e. it is based on live variable analysis and processes the instructions on a block by block basis.

This algorithm is inferior to the first dead code elimination in that it cannot remove code that only depends on itself (dead induction variables) and cannot be modified to remove control dependent dead code.[4] However, this algorithm is much faster than the first because live variables is much less expensive to compute than use-def chains. Also, this algorithm is generally called as an almost free, side effect of building live variables, which are used for many other passes of the compiler.

The main difference between the algorithm in DF and the one in FLOW is that the basic blocks are processed in postorder. The initial value for liveness that is used at the bottom of the block is either value computed by DF if none of the successors has changed or the union of liveness at the top of the successors if any of them has changed.

As each block is processed (from last to first instruction) the liveness is kept up to date. When the top of the block is reached, this computed liveness is compared with the value at the top of the block computed by DF. If the values differ, the locally computed value replaces the value computed by DF and this block is marked as changed.

The only time that it is necessary to actually go back and re-solve the dataflow equations is when the live variable bitvector changes at the top of a block that is the destination of a cfg back edge. This is quite rare, particularly since DCE is called at the beginning of many passes of the backend. In the DCE in FLOW the equations are resolved if any instruction is deleted.

[4] This difference will only be important when the first dead code algorithm is enhanced with control dependence information.

3.3.4 Status

Currently the first DCE algorithm not called directly but is used as a part of the dead store elimination. When it is enhanced with the control dependence graph, it may be useful to call this as a separate pass at higher optimization levels.

It may also be possible to enhance the first DCE so that it can be called as a side effect of building use-def and def-use chains. The performance issues are not with the dead code part of the algorithm but with the building of the chains. However, in passes such as the modulo scheduler which builds both use-def and def-use chains, it may be possible to integrate the first DCE algorithm and fix up the chains.

4 Abstractions and API

The abstractions used by the dataflow analysis are meant to be both usable, and efficient, at the same time.

There are several important structures provided by the dataflow engine, the main ones being the dataflow reference structure and the insn info structure.

The dataflow reference structure is the heart of dataflow information. It is generated by the dataflow scanner, and represents a def or a use of a register (IE a reference to a register). The information it provides consists of:

- `REG`, the register this reference is referencing.
- `BB`, the basic block in which the instruction occurs.
- `INSN`, a pointer to the instruction containing the reference.

- LOC, a pointer to the place in the instruction containing the reference.
- CHAIN, a pointer to the chain of uses of this reference if it is at def, or a chain of defs of this reference if it is a use.
- ID, a unique id for this reference.
- TYPE, whether the reference is a use or a def, and if it is a use, whether it is a regular use, or part of a memory addressing operation.
- FLAGS, various informational flags about the reference, such as whether it is a clobber, whether the reference is artificial, whether it occurs in a note, and other useful pieces of information.

The dataflow insn info structure is the second most used dataflow structure. It provides information about each instruction in the program consisting of:

- USES, the list of register uses in this instruction.
- DEFS, the list of register definitions in this instruction.
- MWREGS, the list of multiword register uses and defs in this instruction (this is separated out for use by REG_DEAD and REG_UNUSED note generation).

In addition to these structures, there are some small and easily understood structures used by various simpler problems, as well as various tables that contain pointers to the structures defined above . An example of one of these table is the the register-use and register-def table, which is a table of all the def/use structures for a given register.

4.1 Using predefined problems

Using the predefined problems is very simple. An example:

```
struct df *df
  = df_init (DF_HARD_REGS);
df_lr_add_problem (df, 0);
df_analyze (df);

bbinset
  = DF_LR_BB_INFO(df, bb)->in;
bboutset
  = DF_LR_BB_INFO(df, bb)->out;
df_finish (df);
```

The call to df_init initializes the dataflow instance, and is passed flags that tell the DF instance about the details of the info you need. The current flags include DF_SUBREGS, which includes information about subregs, DF_HARD_REGS, which includes information about machine registers, and DF_EQUIV_NOTES, which provides information about references that occur in REG_EQUIV notes.

Once initialized, adding problems to the dataflow instance only requires calling the right df_add_problem function. There is one for each predefined problem. To add def-use or use-def chains, df_chain_add_problem should be called with either DF_UD_CHAIN, DF_DU_CHAIN, or both of these flags or'ed together.

After adding all the problems you want to the DF instance, calling df_analyze will cause the dataflow engine to perform all the dataflow and generate the info you have requested.

Once that is done, the info will be stored in various structures, depending on what you asked for.

Problems that generate *IN* and *OUT* sets usually have macros to access these sets, such as `DF_LIVE_IN` and `DF_LIVE_OUT`.

Problems that generate register info, reg-defs, reg-uses, or instruction info, generally have macros to access the appropriate tables, like `DF_INSNS_GET`.

The definitions of all of these macros and tables can be found in the file `df.h`.

4.2 Defining Your Own Problem

DF can solve many dataflow problems in addition to the ones defined in Section 3.1. The full harness that DF uses is somewhat complex and is there to make it very easy to use the canned problems. Only a small amount of the structure is necessary to understand if you wish to define your own problem.

The function `df_simple_iterative_dataflow` has been defined to allow simple dataflow problems defined over some part of the control flow graph to be solved. There are eight parameters to this function:

`dir` Either `DF_FORWARD` or `DF_BACKWARD`. We do not currently do bidirectional, but could add if there was a need.

`init_fun` This function of type *df_init_function* initializes the *in* and *out* sets before starting to solve the equations.

`con_fun_0` This function of type *df_confluence_function_0* is the convolution function if the block has no logical predecessors. If the value of this parameter is `NULL`, the *in set* remains at the value it was initialized to. This is useful in obscure cases to deal with no return blocks in backwards problems. This case can never happen in a forwards problem because such a block would not appear reachable.

`con_fun_n` This function of type *df_confluence_function_n* is the convolution function if the block has one or more logical predecessors.

`trans_fun` This function of type *df_transfer_function* is the transfer function through the block.

`blocks` A bitmap that defines which blocks are to be processed.

`postorder` An array of int that contains the basic blocks in `blocks` in postorder.

`n_blocks` The number of blocks in postorder.

5 Incremental Dataflow

Some of the api of FLOW and the original implementation of DF is based on the assumption that reasonable algorithms exist for performing dataflow analysis incrementally.

The first algorithms for incremental dataflow were the phd dissertations of Ryder [6] and one of the authors of this paper, Zadeck [8]. This was followed other for a period of about 15 years. In that time the community failed to develop any algorithms that were clearly superior to well engineered optimizations that did not rely on being able to update dataflow results.

The problem can be best illustrated with in the live variables problem (but correspondingly similar problems exist for all dataflow problems). There are a large number of correct solutions for the dataflow equations, there is only a

single minimal solution. For the live variables problem, as is true for any set union bit vector problem, the minimal correct solution is the one with the smallest total number of one bits in the solution bit vectors.

`update_life_info_in_dirty_blocks` will generally find a correct solution.[5] However, for changes where uses are deleted, defs are added and/or edges are moved, this function will generally not find the minimal solution: i.e., it finds solutions that contain extra one bits in some vectors.

For illustration purposes, let us take the case where there are several uses of a particular pseudo register *r* and we wish to remove some of these uses.

Finding the minimal solution is trivial if the program has no loops. The problems in finding a solution arise at join points (live variables is a backwards problem: the join points are the conditional branches in the control flow graph.) Since the solution at the bottom of join point *b* is the union of the solutions at the top of the successor of *b*, the question that must be asked when one of the difficult changes is made, is what caused one bits in those successor blocks. If any of the bits in the successors are derived from the uses of *r* that remain, then the bit for *r* at the bottom of *b* is set. However, if all of the bits were derived from instances that were deleted, the bit for *r* at the bottom of *b* must be cleared.

There are two ways to approach designing an algorithm to solve this problem: one can either clear all of the bits in all of the blocks associated with *r* or one can build auxiliary datastructures to hold the information for where the bits come from. The first approach means that one is solving the offline algorithm whenever one makes the change.

[5] Those cases where it fails to find a correct solution are bugs that are generally easy to fix.

The second approach is the one that was pursued by the incremental dataflow community in various forms. For the live variables problem, the information necessary to track changes is equivalent to the information provided by the more expensive reaching uses problem. However, to keep the information in the reaching uses up to date when structural changes are made to the flow graph requires even more expensive path information. While many interesting datastructures were investigated, the overhead involved in keeping these up to date was rarely worth the trouble. This result, coupled with the fact that most optimizations can be implemented without the need for incremental updates basically killed the area.

For these reasons, we decided that it was time to face the fact that incremental dataflow was not going to happen and have reimplemented DF without a non working incremental API. There are a few passes such as if conversion that have required extensive revision, but for most passes the changes have been simple and mechanical.

6 Demand Driven Backend Passes

In addition to improving the quality of the dataflow analysis of the backend, it has been our goal to improve the modularity of the backend. It is highly desirable to be able to reorder the passes in a compiler. We have addressed a number of these issues in the dataflow rewrite.

There are two ways to achieve modularity in a compiler: (1) implement a set of invariants that must be true at the end of each pass or (2) make sure that each pass that needs some invariant to be true applies the steps to make it that way.

One of the reasons that the backend is difficult to deal with is that many of the dependencies

between the phases do not fall into either of these categories. The components of FLOW are at the core of many of these problems and as we have replaced flow with DF, we have done so in a manner that is consistent with one of the above categories.

- At the beginning of any part of the back-end that has a correct control flow graph, any of the DF problems can be computed. Furthermore, it is expected that each pass, compute exactly the dataflow problems that it needs for itself and at the end of that pass, the flow information is discarded.[6]

- Dead code elimination is only performed if the phase where the analysis is being done, requires the program to be clear of dead code. Any phase that my produce dead code leaves that code.

  There are, at higher optimization levels, passes near the end that do a comprehensive job of cleaning up the dead code.

- Like dce, CFG Cleanup now is only performed if the phase before phases that would be inhibited by having an untidy control flow graph. The long term plan with this is to change this from a phase that eight different modes to one that has two modes: a lightweight one that is part of other passes, and a heavy weight one that is powerful and not particularly inexpensive.

While this cleanup removes many of the inter-pass dependencies, some remain, in particular there are still many issues with the way register information is built and maintained throughout the pass stream.

[6]The only exception to this are `REG_DEAD` and `REG_UNUSED` notes that are left until the next pass that uses them cleans them up.

7 Status and Results

The work so far has concentrated on been replacing the use of `flow.c` with dataflow provided by `df.c`. Currently, the code for most of our work is on the `dataflow-branch` in the GCC SVN repository. On this branch, `flow.c` is not used at all for global liveness calculation, or dead code elimination. All passes, except combine, use dataflow instances to do their work. This work should begin soon.

8 Acknowledgments

We would like to than everyone who has helped with the dataflow branch. This includes, Steven Bosscher, David Edelsohn, Jan Hubicka, Richard Sandiford, Ian Lance Taylor.

References

[1] F. E. Allen. Control flow analysis. *SIGPLAN Notices*, 5, July 1970.

[2] D. C. Atkinson and W. G. Griswold. Implementation techniques for efficient data-flow analysis of large programs. *Proc. of the 2001 International Conf. on Software Maintenance*, November 2001.

[3] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[4] S. L. Graham and M. N. Wegman. A fast and usually linear algorithm for global flow analysis. *J. ACM*, 23(1):172–202, January 1976.

[5] M. S. Hecht and J. D. Ullman. A simple algorithm for global data flow analysis problems. *SIAM J. Computing*, 4(4):519–532, December 1975.

[6] B. G. Ryder. Incremental data flow analysis. *Conf. Rec. Tenth ACM Symp. on Principles of Programming Languages*, pages 167–176, January 1983.

[7] R. E. Tarjan. Testing flow graph reducibility. *J. Computer and System Sciences*, 9:355–365, December 1974.

[8] F. K. Zadeck. Incremental data flow analysis in a structure program editor. *Proc. SIGPLAN'84 Symp. on Compiler Construction*, pages 132–143, June 1984. Published as *SIGPLAN Notices* Vol. 19, No. 6.