An Efficient Approach to Data Flow Analysis in a Multiple Pass Global Optimizer

Suneel Jain
Carol Thompson

Hewlett-Packard
19447 Pruneridge Avenue
Cupertino, CA 95014

ABSTRACT

Data flow analysis is a time-consuming part of the optimization process. As transformations are made in a multiple pass global optimizer, the data flow information must be updated to reflect these changes. Various approaches have been used, including complete recalculation as well as partial recalculation over the affected area. The approach presented here has been designed for maximum efficiency. Data flow information is completely calculated only once, using an interval analysis method which is slightly faster than a purely iterative approach, and which allows partial recomputation when appropriate. A minimal set of data flow information is computed, keeping the computation and update cost low. Following each set of transformations, the data flow information is updated based on knowledge of the effect of each change. This approach saves considerable time over complete recalculation, and proper ordering of the various optimizations minimizes the amount of update required.

1. Introduction

This paper describes the data flow analysis implementation in a machine-specific multiple pass optimizer. The major contribution of this work is the methods which have been employed to improve the speed of the entire optimization process. These are:

1. Use of a small set of data flow information to support a number of global optimizations.
2. Use of interval analysis to both speed up the process of data flow computations, and to allow partial recalculation where appropriate.
3. Incremental update of data flow information between components of the optimization process, based on knowledge of the nature of the transformations being made.

Data flow update for each interval is based on the existing data flow information for that interval. As a result, the update for each interval is done independently of the update for other intervals. The cost of this update method is O(n*c) where n is the number of intervals in the graph, and c is the number of changes. Unlike methods which propagate a change outward, this cost is independent of how much of the graph is actually affected by a transformation. However, this constant cost is offset by the straightforward manner in which it can be implemented. The update process for each interval consists of testing and setting bits in a bit vector, which can be done very efficiently.

2. Related Work

The optimization strategy of the PL.8 compiler [Ausl 82] has many similarities to this implementation. They also implemented optimization using a used a low level intermediate representation.

Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation
Atlanta, Georgia, June 22–24, 1988

However, their data flow analysis was done using Allen/Cocke interval analysis, and the data flow information was updated between passes by complete recalculation. The approach to data flow analysis presented here differs in that it incorporates structural analysis [Shar 80] to speed up the computation of data flow information, and incremental update of that information between passes to save additional compilation time.

Incremental update of data flow information has been investigated mostly from the point of view of incremental compilers. In an incremental compiler, current data flow information can be used to produce optimized code, as well as to provide the user with information on program behavior. Paull and Ryder [Paul 83], [Paul88] present an incremental update algorithm for Allen/Cocke interval analysis [Alle 76], with a worst case update cost of O(n*c) for reducible digraphs, where n is the number of nodes, and c is the number of changes. Pollock [Poll 85] presents an incremental update algorithm designed to support optimization, using a flow graph of dags and a history of optimization transformations. Zadeck took another approach [Zade 84], using graph splitting to incrementally update cluster problems, exhibiting O(n+e)*c worst case behavior, where n is the number of nodes in the graph, and e is the number of edges. L. and K. Ottenstein [Otte 84] also address the problem of incremental update, presenting an approach designed to support program slicing.

All of these approaches use an outward propagation of program changes, so that the actual update cost is proportional to the area affected. However, the cost of these update methods is high both in terms of the amount of information required, as well as the amount of work that must be done at each step in the update process.

When it is expected that the effects of a change will be fairly localized (as one might expect with incremental compilation), such approaches may be desirable, since the update cost can then be expected to be rather limited. However, the problem of data flow update between optimization passes does not exhibit this same locality, as transformations may be sprinkled throughout the flow graph. Another difference is that the transformations made by an optimizer are of a restricted set, and therefore easily characterized, unlike the transformations made by a program edit. Utilizing this knowledge, data flow update can be tailored to the specific transformations of each pass, making the process more efficient.

3. Background

The data flow analysis approach described here has been developed as part of an optimizer targeted for the HP Precision Architecture, developed in the Spectrum Program at Hewlett-Packard [Birn 85]. Optimizations that are supported include machine-specific as well as traditional data flow based optimizations. A major goal in the design of the data flow analysis was that it should be as fast as possible, since the optimizer would be integrated with production compilers. In order to meet this objective, a data flow analysis package has been designed which will support a number of optimizations with a simple set of data flow information represented efficiently. An adaptation of Sharir's approach to interval analysis is used to support data flow calculation, giving both a realization of the control flow of the program as well as an efficient calculation method [Shar 80]. Because optimization is done at the level of machine language, data flow information is gathered for both memory and register resources. Both forward and backward data flow calculation are supported.

The low level representation upon which optimizations are performed contains a doubly-linked list of machine instructions [Cout 86b] [John 86]. Each instruction entry in the graph contains opcode, source, and target information as well as additional information required by the optimizer. All operands, including both memory and registers, are specified using *resource IDs*. Registers are chosen from an infinite set, and register assignment is performed by the optimizer. A value numbering scheme is used by the code generators to ensure

that identical operations always receive the same unique target resource ID. Resource IDs are also used to convey aliasing information to the optimizer. Each resource ID may represent a reference to a single resource (register or memory), or to one of a number of resources [Cout 86a].

As local data flow information is gathered, the resource IDs are replaced with *sequence numbers*. Sequence numbers can be expressed as <instruction, resource ID> pairs. Each sequence number is used exactly once, and represents either a definition or a use of a particular resource in an instruction. Data flow information is expressed as sets (bit vectors) of sequence numbers. To save space, bit vectors are implemented as a linked list of data blocks representing 64 contiguous elements each. Definition and use sequence numbers are clustered in different partitions to decrease the number of data blocks in each bit vector.

In addition to the <instruction, resource ID> pairs, there is a special sequence number for each resource ID called the *any-ref* sequence number. This sequence number is not associated with any particular instruction and is used to convey more general information about a given resource ID, and which is not associated with a particular instruction.

4. Interval Analysis

Once code generation is complete, the instruction graph is partitioned into basic blocks. These comprise the basic unit for which local data flow information is calculated. In addition, these basic blocks become the base units for the interval structure. The interval analysis approach rediscovers the control flow of the code, and builds a hierarchy of control structures, each such structure being represented as an interval.

The first step in performing interval analysis is to calculate the set of dominators for each basic block. The standard approach to calculating dominators [Aho 77] is to initialize the dominators set for each node, except the first, to contain all nodes. However, since these sets are usually sparse, it is more efficient to initialize the set for each node to contain only itself. These sets are then propagated by iterating over the nodes in reaching depth-first order. At each node, the intersection is taken of the dominator sets of the predecessors *which have been visited during some iteration*. This continues until there are no changes.

Once dominators have been computed, the interval structures are identified. Sharir's approach recognized single-entry/single-exit loops, if-then and if-then-else constructs. All other control structures were contained in proper intervals (loops with a single entry and possible multiple exits), or improper intervals (single-entry loops which contain irreducible flow graphs). For recognized control structures such as loops or if-then intervals, no iteration is required during global data flow analysis. Instead, data flow information is propagated using a formula specific to each interval. Data flow information for both proper and improper intervals is calculated in an iterative fashion, but can be limited in the case of proper intervals.

In adapting Sharir's approach to our needs, a control structure has been added for switch/case statements. It supports the Pascal-like case statement as well as the switch statement in C, which allows fall-through from one case to another. This structure consists of a set of nodes with the following properties (figure 1):

1. The first node has three or more successors, or sister nodes.
2. Each sister node has, in addition to the first node, at most one other predecessor which is also a sister node.
3. Each of the sister nodes has a single successor, which is either another sister node, or the single successor node to the switch/case interval.

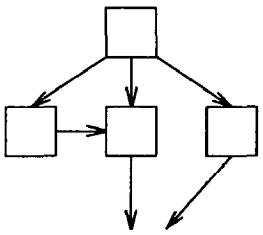

figure 1 : Switch Interval

The second contribution to Sharir's approach has been to limit the scope of improper intervals. In his approach, the effect of an irreducibility is contained only within the next enclosing loop. In this approach, an improper interval as been limited to the smallest group of nodes which contains:

1. One node which dominates all other nodes in the group, and
2. All nodes which lie on a path between the dominating node and any member of the group.

This approach often produces a smaller improper interval, thus limiting the scope of iterative computation (figure 2).

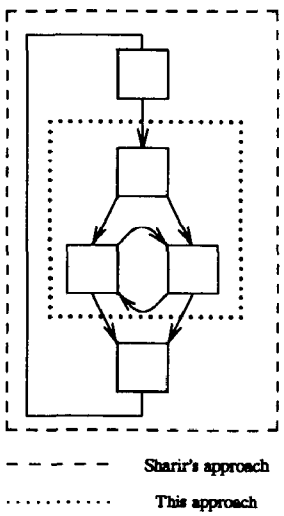

figure 2 : Improper Interval

5. Local Data Flow Calculation

Local data flow information is gathered in a forward pass over each basic block. Certain local optimizations are performed during this pass, including constant propagation, common subexpression elimination, redundant load elimination, and peephole optimizations. The following local information is computed for definitions [John 86]:

M_GEN (Might GENerate) — The set of definitions within this basic block which may reach the end of this basic block.

W_D_KILL (Will Definition KILL) — The set of definitions outside this basic block which define resources that are definitely defined in this basic block.

and the following local information for uses:

M_USE (Might USE) — The set of uses within this basic block which have no definite preceding definition in this basic block.

W_U_KILL (Will Use KILL) — The set of uses outside this basic block which use resources that are definitely defined in this basic block.

There are two types of definitions, *might definitions* and *will definitions*. A might definition of a resource indicates that the resource may or may not be absolutely defined, while a will definition indicates that the resource will definitely be defined. Might definitions occur due to either aliasing or conditionally executed (skipped) instructions.

6. Global Data Flow Calculation

The next step in data flow analysis is to propagate the local information out to the outermost interval. This is done using specific formulae for each type of interval (see figure 3). This produces local data flow information (M_GEN, W_D_KILL, M_USE, and W_U_KILL) for each interval.

Example: If-Then Interval

Outward propagation:

$$F_{REACH}[\text{If-Then}] = (F_{REACH}[\text{Then}] \circ F_{REACH}[\text{If}]) \ V \ F_{REACH}[\text{If}]$$

$$F_{NEED}[\text{If-Then}] = (F_{NEED}[\text{If}] \circ F2_{NEED}[\text{Then}]) \ V \ F_{NEED}[\text{If}]$$

where V denotes functional join and o denotes functional composition

Inward propagation:

$$\text{REACH\_TOP[If]} = \text{REACH\_TOP[If-Then]}$$

$$\text{REACH\_TOP[Then]} = F_{REACH}[\text{If}](\ \text{REACH\_TOP[If-Then]}\ )$$

$$\text{NEED\_BOT[If]} = F_{NEED}[\text{Then}](\ \text{NEED\_BOT[If-Then]}\ )$$

$$\text{NEED\_BOT[Then]} = \text{NEED\_BOT[If-Then]}$$

figure 3 : data flow propagation

Global data flow information is then propagated inward from the outermost interval (see example). The following information is computed:

REACH_TOP (REACH at TOP) — The set of sequence numbers which represent definitions which may be defined along some path between the beginning of the procedure and the top of this interval.

NEED_BOT (NEED at BOTtom) — The set of sequence numbers which represent uses which might be reached by a definition at the bottom of this interval.

This information is computed using functions of the form

$$F = < A,B > \text{ such that } F(X) = A \cap X \cup B$$

These functions are as follows:

$$F_{NEED} = < \neg W\_U\_KILL , M\_USE >$$

$$F_{REACH} = < \neg W\_D\_KILL , M\_GEN >$$

In the NEED_BOT set, the presence of the *any-ref* sequence number for a particular resource indicates that there exists some use of that sequence number which is exposed, i.e. the resource is live at the end of this interval.

These basic data flow sets are sufficient to support the data flow based optimizations which have been implemented, with the exception of Common Subexpression Elimination. Traditionally, this optimization depends upon the calculation of available expressions. Rather than generate a third type of data flow information with a different domain (resource IDs rather than sequence numbers), an approach was sought which could be incorporated into the data flow computations for reaching definitions.

The *any-ref* numbers are used in the M_GEN, W_D_KILL, and REACH_TOP sets to indicate the undefined state of a particular resource. A resource becomes undefined when a resource upon which its definition is dependent is redefined. The motivation for tracking the undefined state of a resource, rather than its availability, is that this information can be propagated along with the REACH_TOP information, using set union operations. Thus, if the resource x were defined by the following expression,

$$x = y + z$$

then any definition of the resource y would be a "definition" of UNDEF(x). If the UNDEF (or any-ref) sequence number is not set for a particular

resource, then that resource can be considered to be available, and therefore a candidate for common subexpression elimination. Although this information is implemented in conjunction with the Reaching Definitions, conceptually there exist separate data flow sets and functions for the UNDEF information:

$$F_{UNDEF} = < \neg W\_DEF, M\_KILL >$$

W_DEF (Will DEFine) — the set of resources which have definitions which will definitely reach the end of this basic block.

M_KILL (Might KILL) — the set of resources that may be invalidated in this basic block by the redefinition of resources upon which they depend.

Note that the domain of these sets and functions is the set of resource numbers as represented by the *any-ref* sequence numbers, not sequence numbers as such.

7. Use of Data Flow Information

The REACH_TOP and NEED_BOT sets provide data flow information for basic blocks, which are the lowest level in the interval structure hierarchy. However, most global optimization components require data flow information at each instruction. A traversal within the basic block coupled with the REACH_TOP and the NEED_BOT sets at the basic block boundary are used to derive this information whenever needed. The different sets computed are outlined below.

Reaching Definitions: The set of definitions for a resource R that may reach an instruction I.

Exposed Uses: The set of uses for a resource R that are upward exposed at an instruction I.

Available Resources: The set of resources available at an instruction I. Availability of a resource R is derived from the dependency information for R and the UNDEF bit for R in the REACH_TOP of the basic block containing I.

Def-Use Webs: The definitions and uses of a resource R are partitioned into disjoint sets called webs. For each use in a web, all the definitions that can reach it are in the web and for each definition in the web, all its uses are in the same web. The REACH_TOP and the NEED_BOT sets are used to build webs. The def-use webs are used for register allocation, store to copy optimization and unused definition elimination.

8. Data Flow Update

All the global optimizations use the data flow information outlined in the previous section. Since the same data flow sets are used by all components, they need to be kept accurate as various optimizations are done. The local data flow sets are not updated, since they are not used by any of the optimizations. The data flow based optimizations performed are common subexpression elimination (CSE), store to copy optimization, unused definition elimination, loop invariant code motion (LICM), induction variable elaboration (IVE) [Cock 77], register web-based optimizations and register allocation using graph coloring [Chai 82].

No data flow update is required after register allocation, which is performed last. Unused definition elimination maintains the correctness of the data flow and thus does not require update. Sequence numbers for the instructions deleted in the process are invalidated, and ignored in the data flow sets. Register web-based optimization changes certain instructions to simpler instructions. The sequence numbers of the original instruction are retained and thus the data flow information remains valid. The ordering of the other optimizations is done to minimize data flow update. Common Subexpression Elimination is the only optimization that uses the UNDEF sets. This optimization is performed first so that UNDEF information is never updated. CSE is also the only component which does not use NEED_BOT information. The NEED_BOT sets are computed after CSEs, so that they do not

have to be updated.

As mentioned earlier, the approach used in this optimizer is to incrementally update the data flow sets in each component based on the particular transformations made. This approach requires additional work to characterize the effect of each transformation, so that the data flow can be updated accurately. The benefit is to drastically reduce the time taken by the update. Also, the time taken by the update becomes proportional to the amount of optimization done by a particular component.

The incremental update algorithms for the various components are given below.

8.1 Update after Common Subexpression Elimination

This component removes redundant definitions when an earlier result is still available. For each instruction that is deleted, the set of reaching definitions for the instruction target is computed. The definitions in this set that have also been eliminated are recursively replaced by their reaching definitions. These are called the *recursive reaching definitions*. The update algorithm is given below:

```
compute the recursive reaching definitions for all
instructions deleted;
for each interval I in the procedure do
   for each instruction that is deleted do
      T ← sequence number of the instruction target;
      if T is in REACH_TOP[I] then
         replace T by its recursive reaching
         definitions in REACH_TOP[I];
      end if
   end for
end for
```

8.2 Update after Store to Copy Optimization

This optimization promotes certain memory resources to registers. Each def-use web of the memory resource is independently considered for the optimization. The web may contain special definitions and uses to denote the modification or use of the memory resource by a procedure call. The same mechanism is also used to represent a definition at procedure entry and a use at procedure exit. The optimization changes store instructions in the web to copy instructions, deletes load instructions, adds a store before any call that may use the memory resource and adds a load instruction after any call that may modify the memory resource.

When a store instruction is converted into a copy, the sequence number of the memory resource is changed to be a sequence number of the register target. Since this sequence number is already in the data flow sets, no update is needed. The update for the other transformations is given below:

```
for each interval I in the procedure do
   for each load instruction deleted do
      if the memory use is in NEED_BOT[I] then
         add all the exposed uses of the load
         target to NEED_BOT[I];
      end if
   end for
   for each store instruction added before a call
         or at procedure exit do
      if the special memory use is
            in NEED_BOT[I] then
         add the use of the load target in the
         store, to NEED_BOT[I];
      end if
   end for
   for each load instruction added after a call
         or at procedure entry do
      if the special memory definition is
            in REACH_TOP[I] then
         add the definition of the load target
         to REACH_TOP[I];
      end if
   end for
end for
```

8.3 Update after Loop Invariant Code Motion

This optimization moves computations within a loop that yield the same result for every iteration,

to the loop preheader. This movement of instructions affects the data flow only inside the loop interval. Outside the loop interval the data flow information remains accurate. The update is done after the optimization of each loop.

```
for each instruction moved out do
    for each sub-interval I of the loop do
        add the definition of the instruction
        target to REACH_TOP[I];
        add all the exposed uses of the target
        to NEED_BOT[I];
        if any of the operands of the instruction
        are not live on exit from the loop, remove
        the use of the operands from NEED_BOT[I];
    end for
    add the exposed uses of the target
    to NEED_BOT[loop preheader];
end for
```

8.4 Update after Induction Variable Elaboration

This optimization replaces multiplication operations within a loop by iterative addition operations. A new resource is created to hold the result of the multiplication and updated appropriately.

A different approach is used for update after IVE than for other components. Since IVE introduces definitions and uses of new resources, it is very difficult to do the data flow update for each transformation incrementally. Further, the effects of most of the new instructions and temporaries added are limited to the loop interval. Thus the data flow information for the loop interval is completely recomputed. This gives acceptable results since the recomputation overhead is limited to the loop interval.

For new instructions added to the loop preheader, the operand uses need to be exposed to the corresponding definitions. If the definitions are not within the preheader, some additional update is required outside the loop interval. For each such new use, there is an existing exposed use of the same resource in the loop. The new use is added to the NEED_BOT wherever the earlier use was in the NEED_BOT set.

9. Results

The time taken for data flow update was measured for a wide variety of sample programs. We present here measurements of the time taken by data flow update for each component relative to the time taken for complete data flow computation. The time for data flow computation includes the time for calculating the local data flow sets, propagation of data flow and calculating global data flow sets. The time for building the interval structure and assigning sequence numbers is not counted, since that is not required for recomputation of data flow information.

The various programs in the table were compiled with optimization using the FORTRAN, C and Pascal compilers containing the common optimizer. Gprof[Grah 82] was used to obtain the timings for different parts of the optimizer. Several runs of the compiler were made for each test file. The profiling results for all the runs of the various files in each set were summed up. Using this data, the ratio of the data flow update time for different components to the data flow computation time was taken.

The data in the table shows that the time taken for data flow update is much smaller than the time it would have taken to recompute the information. The time taken for data flow update in the benchmarks is much higher than for the operating system and the compiler code. This is because the amount of optimizations for the benchmarks is much larger as indicated by the code size reduction numbers. The benchmarks are very loop intensive, which accounts for the high ratios for update after Loop Invariant Code Motion. The ratio for update after Induction Variable Elaboration is zero for the compiler code case because no strength reduction was done in that code.

Program	Code Size Reduction	Ratio of Data Flow Update time to Data Flow Calculation			
		CSE	Store/Copy	LICM	IVE
C benchmarks [1]	46%	0.46	0.02	0.54	0.31
FORTRAN benchmarks [2]	41%	0.26	0.11	0.60	0.02
O.S. code [3]	17%	0.14	0.10	0.05	0.02
Compiler code [4]	15%	0.29	0.07	0.07	0.00

[1] The Ackerman, quicksort, puzzle, subscript puzzle, sieve and dhrystone benchmarks.

[2] The Linpack and Whetstone benchmarks.

[3] Two files from an operating system source, written in Pascal with system extensions.

[4] Four files from the C compiler source, written in C.

Table 1 : Data Flow Update Timings

10. Conclusion

A unified approach to data flow analysis has been presented which provides an efficient framework for data flow based optimizations. Using a simple set of data flow information, a number of optimizations have been implemented. Utilization of interval analysis allows rediscovery of control flow information at a low level, as well as providing an efficient method of calculation. Between optimization components, update of data flow information has been implemented using a combination of localized recalculation and direct update of data flow sets based on the transformations performed. As demonstrated, this incremental approach to data flow update offers a significant improvement in efficiency over recalculation.

11. Acknowledgements

The work described in the paper was done at HP's Computer Language Lab and HP Laboratories. The authors gratefully acknowledge significant contributions made to the design and implementation of the data flow analysis by Peter Canning, Debbie Coutant, Mark Scott Johnson, Terrence Miller and Karl Pettis.

12. References

[Aho 77] A.V. Aho and J.D. Ullman, *Principles of Compiler Design*. Addison-Wesley, 1977.

[Alle 76] F.E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure", *Communications of the ACM*, March 1976.

[Ausl 82] M. Auslander and M. Hopkins, "An Overview of the PL.8 Compiler", *Proc. of the SIGPLAN Symp. on Compiler Construction*, June 1982.

[Birn 85] J.S. Birnbaum and W.S. Worley Jr., "Beyond RISC: High-Precision Architecture", *Hewlett-Packard Journal*, 36:8, August 1985.

[Chai 82] G.J. Chaitin, "Register Allocation and Spilling via Graph Coloring", *Proc. of the SIGPLAN Symp. on Compiler*

Construction, June 1982.

[Cout 86a] D.S. Coutant, "Retargetable High-Level Alias Analysis", *Conf. Record of the 13th ACM Symp. on Principles of Programming Languages*, January 1986.

[Cout 86b] D.S. Coutant, C.L. Hammond, and J.W. Kelley, "Compilers for the New Generation of Hewlett-Packard Computers", *Hewlett-Packard Journal*, **37**:1, January 1986.

[Cock 77] J. Cocke and K. Kennedy, "An Algorithm for Reduction of Operator Strength", *Communications of the ACM*, **20**:11, November 1977.

[Grah 82] S.L. Graham, P.B. Kessler and M.K. McKusick, "gprof: a call graph execution profiler", *Proc. of the SIGPLAN Symp. on Compiler Construction*, June 1982.

[John 86] M.S. Johnson and T.C. Miller, "Effectiveness of a Machine-Level, Global Optimizer", *Proc. of the SIGPLAN Symp. on Compiler Construction*, June 1986.

[Otte 84] K.J. Ottenstein and L.M. Ottenstein, "The program dependence graph in a software development environment," *Proc. of the ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments*, April 1984.

[Paul 83] M.C. Paull and B.G. Ryder, "Incremental data flow analysis algorithms", DCS-TR-131, Rutgers University, July 1983.

[Paul 88] B.G. Ryder and M.C. Paull, "Incremental data flow analysis", *ACM Transactions on Programming Languages and Systems*, **10**:1, January 1988.

[Poll 85] L.L. Pollock and M.L. Soffa, "Incromint - An INCRemental Optimizer for Machine INdependent Transformations", *Proc. of SOFTFAIR II - A Second Conference on Software Development Tools, Techniques, and Alternatives*, December 1985.

[Shar 80] M. Sharir, "Structural Analysis: A New Approach To Flow Analysis in Optimizing Compilers", *Computer Languages*, **5**, Pergamon Press Ltd., 1980.

[Zade 84] F.K. Zadeck, "Incremental Data Flow Analysis in a Structured Program Editor", *Proc. of the ACM SIGPLAN Symposium on Compiler Construction*, June 1984.