How to Analyze Large Programs Efficiently and Informatively

Dhananjay M. Dhamdhere*
dmd @ kailash.ernet.in

Barry K. Rosen†
bkr @ watson.ibm.com

F. Kenneth Zadeck‡
fkz @ cs.brown.edu

Elimination of partial redundancies is a powerful optimization that has been implemented in at least three important production compilers and has inspired several similar optimizations. The global data flow analysis that supports this family of optimizations includes some bidirectional problems. (A bidirectional problem is one in which the global information at each basic block depends on both control flow predecessors and control flow successors.) This paper contributes two ways to simplify and expedite the analysis, especially for large programs.

- *For each global data flow question, we examine only the places in the program where the question might have an answer different from a trivial default answer. In a large program, we may examine only a small fraction of the places conventional algorithms would examine.*
- *We reduce the relevant bidirectional problems to simpler unidirectional problems. These bidirectional problems can be solved by applying a quick correction to a unidirectional approximation.*

1 Introduction

Elimination of partial redundancies (*EPR*) [MR79] is a powerful optimization that has been implemented in at least three important production compilers (COMPASS Compiler Engine, MIPS, and IBM PL.8) and has inspired several similar optimizations [Cho88, JD82a, JD82b]. While these optimizations have been useful in practice, they account for a significant portion of the time and space devoted to optimization. This paper contributes two ways to simplify and expedite the data flow analysis that supports this family of optimizations, which we call *EPR-like*.

*Dept. of Computer Science and Engineering, Indian Institute of Technology, Powai, Bombay 400 076 INDIA

†Mathematical Sciences Dept., IBM Research Division, P.O. Box 218, Yorktown Heights, NY 10598 USA

‡Computer Science Dept., P.O. Box 1910, Brown University, Providence, RI 02912 USA This author was supported in part by the Office of Naval Research and the Defense Advanced Research Projects Agency (under Contracts N00014-83-K-0146 and ARPA order 6320, and N00014-91-J-4052 and ARPA Order 8225) and in part by the National Science Foundation under grant CCR-9015988.

ACM SIGPLAN '92 PLDI-6/92/CA
© 1992 ACM 0-89791-476-7/92/0006/0212...$1.50

- For each global data flow question, we examine only the places in the program where the question *might* have an answer different from a trivial default answer. For reasons explained later in this section, the technique is called *slotwise analysis* and is also useful for some less powerful but more common optimizations. For EPR-like optimizations, where most expressions are not redundant and move only a short distance, the potential savings from using a sparse method such as slotwise analysis can be substantial.
- We show how to reduce the bidirectional problems posed by the various EPR-like optimizations to simpler unidirectional problems. (A bidirectional problem is one in which the global information at each basic block depends on both control flow predecessors and control flow successors.) These bidirectional problems can be solved by first solving a unidirectional approximation and then applying a quick correction.

These contributions are described below and more fully explained in the following sections, which also discuss relevant previous work in detail.

Standard algorithms for data flow analysis [ASU86] process a program of size n in $O(n^2)$ bit-vector steps (worst case) or $O(n)$ bit-vector steps (typical cases). In the most natural formulations of many problems, the lengths of the bit vectors are proportional to the size of the program and each bit-vector step takes $O(n)$ time. For example, a program of size n has about n candidates for redundancy elimination. Each candidate deserves its own *slot* in a bit vector. (Slots are also known as "components" or "elements" or "positions".) With long bit vectors, standard algorithms are $O(n^3)$ at worst and $O(n^2)$ at best. The algorithm presented here is $O(n^2)$ at worst. For some problems, our algorithm is effectively $O(n)$ despite the fact that n vectors of length n would have a total of n^2 slots.

More precisely, the time devoted to the i-th slot is $O(p_i)$, where $p_i \leq n$ is the size of the part of the program visited while computing the i-th slot of each of the bit vectors at *some* of the basic blocks in the program. Each slot is computed only in its own relevant part of the program. Elsewhere, it is known to take on a given default value. Our method is thus a member of the broad family of *sparse* algorithms. Let p be the average of p_i as i varies over n slots. Then the total time is $O(pn)$, with $p = n$ at worst. For some problems, p is typically a small fraction of n. In particular, for EPR-like problems, experience with the COMPASS Compiler Engine suggests that bit vectors typically have the default value for about 99% of the bits.[1]

On a machine with 32-bit words, treating each slot separately will be faster than standard bit-vector methods

[1] We thank Bob Morgan (personal communication, November 7, 1990) for this information.

whenever $32p < n$, and in some other cases as well. Details are given in §2, which also discusses the few other sparse algorithms in data flow analysis.

One important application is treated in detail here: placement analysis to support elimination of partial redundancies. (Moving invariant code out of loops is an important special case of this optimization.) The standard equations for this problem are *bidirectional*: along a control flow edge (x, y), some of the information flows *forward* (from x to y) while some of it flows *backward* (from y to x). Most efficient analysis algorithms deal only with unidirectional flow, as in the forward-only flow of ordinary redundancy elimination. We show how to perform placement analysis by a unidirectional approximation followed by a quick correction. This unidirectional approximation uses our slotwise analysis.

Solving an arbitrary bidirectional problem would require multiple corrections, but placement analysis and the other EPR-like problems are not arbitrary. The complicated equations have some simple properties that imply quick convergence, if one is careful about where to start. Details are in §3, which is formulated to be read *either* from a bit-vector viewpoint *or* from a general lattice-theoretic viewpoint. An example is worked in §4. The techniques applied in §4 to one particular formulation of placement analysis are equally applicable to other formulations and to the other EPR-like optimizations [Cho88, JD82a, JD82b]. Finally, §5 explains how our method of placement analysis can be exploited by a compiler that detects *second-order* effects: elimination of partial redundancies involving one expression can create new opportunities involving other expressions.

2 Classical Bit-Vector Problems

Being a sparse algorithm, slotwise analysis for each bit-vector slot visits only the nodes where the bit in that one slot might be different from a trivial default value. Depending on the problem in question, the default is either 0 or 1. To handle both cases simultaneously, we review three ways to classify problems.

1. Information may flow either forward or backward along edges. In each classical problem, the flow is in only one of these possible directions. Section 2.1 treats forward flow in detail. The backward case is an easy transformation of the forward case, as §2.2 explains.

2. At a join node (in a forward problem) or at a branch node (in a backward problem), previous algorithms need to *merge* together information propagated to the node along several edges. The merge operation (denoted $\sqcap$) is either bitwise AND (denoted $\wedge$) or bitwise OR (denoted $\vee$). Let $\perp$ be the bit value that *controls* the merge: for any x, $x \sqcap \perp = \perp$. Thus, $\perp = 0$ if $\sqcap$ is $\wedge$ but $\perp = 1$ if $\sqcap$ is $\vee$. In both cases, let $\top$ be the other bit value. The merge rule defines a partial ordering of the bit vectors, with $x \sqsubseteq y$ whenever each slot with $\top$ in x has $\top$ in y.

3. At the `Entry` node (in a forward problem) or at the `Exit` node (in a backward problem), each slot has a given *default value*. For example, the problem of available expressions has the default $EntryInfo = 0 = \perp$ because nothing is available at `Entry`.[2] On the other hand, the problem of reaching definitions has the default $EntryInfo = 0 = \top$ because nothing reaches `Entry`. The same bit 0 has different significance in these two problems because the merge operators are different.

2.1 Forward Problems

Associated with each edge (x, y) is a *flow function* $f = Flow(x, y)$ that tells how to transform a bit vector ξ at x to a bit vector η at y when control flows from x to y. The equations to be solved are

$$Info(\texttt{Entry}) = EntryInfo$$

and, for each node $y \neq \texttt{Entry}$,

$$Info(y) = \prod_{x \in Pred(y)} [Flow(x, y)](Info(x)).$$

The desired solution to these equations is the one that could be obtained by iterating from an initial guess of all $\top$ bits.

By solving for each slot in $Info$ separately, we simplify the flow functions. There are only four ways to map from $\{0,1\}$ to $\{0,1\}$, and only three of them make sense in data flow analysis. Along any edge (x, y), the flow function is one of those displayed in Figure 1. For example, consider whether an expression E is available on entry to a basic block. The code within the block x may START availability of E by computing E and not assigning to any operand of E thereafter. Otherwise, x may STOP availability of E by assigning to an operand of E or may just PROPAGATE to y whatever was true on entry to x. With only three possibilities to consider, we can examine the flow functions to help decide the order in which to visit nodes. Standard algorithms, on the other hand, consider n slots at once and cannot enumerate 3^n possibilities.

function	behavior	comment
START	$0 \mapsto 1$ $1 \mapsto 1$	constant at 1
STOP	$0 \mapsto 0$ $1 \mapsto 0$	constant at 0
PROPAGATE	$0 \mapsto 0$ $1 \mapsto 1$	passes arg. along

Figure 1. Possible flow functions on a single bit.

In order to treat problems with $\perp = 0$ and problems with $\perp = 1$ simultaneously, it is convenient to stipulate that LOWER is whichever of START, STOP has value $\perp$ and RAISE is whichever of START, STOP has value $\top$. Edges are considered to be PROPAGATE edges by default. Local analysis for the slot of current interest flags the exceptional (RAISE and LOWER) edges and puts them in two lists. At the end of global analysis for the current slot, these lists are traversed and the flags are reset, so as to be ready for local analysis of the next slot. Similar resetting is done for other information. In particular, the global bit of information for each basic

[2] This standard assumption is adopted here for ease of presentation but is not important to our method. More extensive optimization is sometimes possible with other assumptions. For example, an expression whose only variables are local temporaries may be considered available at `Entry`. Without interprocedural information, on the other hand, the standard assumption is definitely needed for any expression whose only variables are globals or parameters.

block defaults to the given value $EntryInfo$ associated with the Entry node. This value may be either $\top$ or $\bot$, but we assume it is the same for each slot. The general case is easily handled by applying our method twice, once for slots with $EntryInfo = \top$ (*high* information) and once for slots with $EntryInfo = \bot$ (*low* information).

2.1.1 High Information at Entry

Initialize the set N_1 of visited nodes to hold all targets of LOWER edges. Initialize a worklist to hold these same nodes. Whenever a node is taken off the worklist, every successor z reached by a PROPAGATE edge is examined. If z is not already in N_1, then z is put in N_1 and put on the worklist.

Let N_0 be the full set of nodes. Then $Info(x) = \top$ for $x \in N_0 - N_1$ and $Info(x) = \bot$ for $x \in N_1$. Thus all the $Info$ bits have been determined at cost $O(\|N_1\|)$, where the *extended size* $\|...\|$ of a set of nodes counts edges that touch the nodes as well as the nodes themselves.

2.1.2 Low Information at Entry

Initialize the first set N_1 of visited nodes to hold all targets of RAISE edges. Initialize a worklist to hold these same nodes. Whenever a node is taken off the worklist, every successor z reached by a PROPAGATE edge is examined. If z is not already in N_1, then z is put in N_1 and put on the worklist.

Let N_0 be the full set of nodes. We already know that $Info(x) = \bot$ for $x \in N_0 - N_1$, so there is no need to examine these nodes. Nodes in N_1 might have either $\bot$ or $\top$. These nodes are classified by a second worklist search. Initialize the second set N_2 of visited nodes to hold every node $y \in N_1$ such that, for some inedge $e = (x, y)$, either e LOWERs information or $x \notin N_1$ and e does not RAISE information. Initialize a worklist to hold these same nodes y. Whenever a node is taken off the worklist, every successor z reached by a PROPAGATE edge is examined. If z is in N_1 but is not already in N_2, then z is put in N_2 and put on the worklist. Once the second search is complete, we know that $Info(x) = \top$ for $x \in N_1 - N_2$ and $Info(x) = \bot$ for $x \in N_2$. Thus all the $Info$ bits have been determined at cost $O(\|N_1\|)$, with about twice as much work as in the easier case of high information at entry.

2.2 Backward Problems

With both forward and backward problems, it is customary to say that the global information "at" a basic block is what is known on *entry* to the block, so the flow function associated with an edge from x to y is determined by the code inside x and has nothing to do with the code inside y. After the flow functions have been found, backward problems are solved by the mirror image of the method for forward problems. Whenever a node is taken off the worklist, its predecessors (rather than successors) are examined. The Exit node plays the role of Entry. The given value $ExitInfo$ may be either $\top$ or $\bot$. Just as with forward problems, $\top$ needs one search while $\bot$ needs two.

2.3 Discussion

After discussing proper accounting for lengths of bit vectors when analyzing various algorithms for live (also known as "upwards-exposed") variables, Kou [Kou77] proposed that live variables be found by the backwards version of the simple worklist algorithm in §2.1.1. Apart from more elaborate (but still quadratic) algorithms oriented toward incremental analysis [MR90, Zad84], Kou's algorithm has had little influence. Standard algorithms are effectively quadratic in practice, and the 32-way parallelism of standard bit vector operations is still attractive, even when the vectors are long.

The way compilers use information about available expressions differs in three ways from the way they use information about live variables for tasks like register allocation:

1. *The information in any one slot can be computed, used, and discarded before computing the information in any other slot.* To eliminate redundant computations of an expression E, we only need availability information for E. To build an interference graph for register allocation, on the other hand, we need liveness information for all variables.

2. *The information in many of the slots is the default bit value at many of the basic blocks in a large program.* Many expressions E are unavailable throughout much of the program.

3. *The default value is* $\bot$. The information at Entry (since this is a forward problem) is 0 in every slot, and 0 is the controlling value for the merge operation.

Differences 1 and 2 make a slotwise approach much more attractive, while difference 3 makes the previous slotwise approach [Kou77] inapplicable. Our new algorithm in §2.1.2 handles difference 3. Difference 1 has the interesting consequence that using the information for one expression may improve the information for another expression. Details are in §5.

The following subsections compare slotwise analysis with two other nontraditional approaches.

2.3.1 Incremental Analysis

Incremental analysis has a focus quite unlike that of slotwise (or any other sparse) analysis. An incremental algorithm assumes that the data flow information (and perhaps some auxiliary information) has already been computed everywhere. It tries to respond to a change in the input with work related more to the corresponding change in the output than to the size of the whole problem. A sparse algorithm, on the other hand, has no previous solution of the problem to update: it starts from scratch.

2.3.2 Building Sparse Graphs

Choi, Cytron, and Ferrante [CCF91] generalized static single assignment form [CFR+91] to derive a sparse representation of any given problem. The original graph is replaced by a collection of smaller graphs such that many of the flow functions in each graph are either constants or the identity function. The combined cost of constructing all of the sparse graphs and solving the problem separately on each of them may be less than the cost of solving the problem directly.

Building sparse graphs is much more complicated than slotwise analysis and is asymptotically slower. When an analytic mistake inherited from [CFR+89] is corrected [CFR+91], the worst-case complexity for constructing each sparse graph is $O(n^3)$. Slotwise analysis is $O(n^2)$ for solving the entire problem.[3] Most bit-vector problems are more amenable to either slotwise or traditional analysis than to building sparse graphs.

One important potential advantage of sparse analysis, however, is its algebraic generality. The information at each

[3] The pessimism in worst-case bounds is more severe for building sparse graphs than for slotwise analysis, so the real difference may be less than the worst-case bounds suggest.

$$Info(y) = \mathcal{F}(y) \sqcap \mathcal{B}(y) \tag{1}$$

$$\mathcal{F}(y) \stackrel{\text{def}}{=} \begin{cases} EntryInfo & \text{if } y = \texttt{Entry} \\ \prod_{x \in Pred(y)} [ForwFlow(x,y)](Info(x)) & \text{if } y \neq \texttt{Entry} \end{cases} \tag{2}$$

$$\mathcal{B}(y) \stackrel{\text{def}}{=} \begin{cases} ExitInfo & \text{if } y = \texttt{Exit} \\ \prod_{z \in Succ(y)} [BackFlow(y,z)](Info(z)) & \text{if } y \neq \texttt{Exit} \end{cases} \tag{3}$$

Figure 2. The information at y is the merge of auxiliary information propagated forward and backward to y.

slot in a vector is not required to be a bit. For data flow analysis problems like degrees of availability [Ros81], where *bit* vectors are inappropriate but slots can be partitioned to yield many identity or constant flow functions, it is likely that building sparse graphs would be useful.

3 Bidirectional Problems

As defined here, bidirectional problems are a natural generalization of the classical unidirectional problems. Section 4 shows that this generalization covers some important practical problems. As with any formal definition, however, the possibility remains that an intuitively "bidirectional" problem may not fit the definition. Indeed, one such problem is considered briefly at the end of this section. Our formal definition is *an* appropriate one, and it would be futile to agonize over *the* appropriate one.

Notations like $\sqcap$ or $\sqsubseteq$ in this section may be read in two ways:

1. The information assigned to each node is a pair of bits, and notations from §2 are overloaded in the natural way. The Boolean operations applied by a flow function may $\wedge$ or $\vee$ the argument with local information, but may not negate the argument. This reading covers placement analysis (§4.1) and the other analyses that support other EPR-like optimizations [Cho88, JD82a, JD82b].

2. The information assigned to each node is an element of a lattice[4] where every descending chain is of finite length and there are designated top ($\top$) and bottom ($\bot$) elements. The flow functions are *monotonic*: if $\xi \sqsubseteq \eta$ then $f(\xi) \sqsubseteq f(\eta)$.

Informally, the word "bidirectional" is appropriate for any problem where the information at a node depends on both predecessors and successors. Just as classical unidirectional analysis restricts attention to the simple way that $Info(y)$ depends on $Info(x)$ for $x \in Pred(y)$ in §2.1, our bidirectional analysis restricts attention to the simple way that $Info(y)$ depends on $Info(x)$ for $x \in Pred(y)$ and on $Info(z)$ for $z \in Succ(y)$ in Figure 2. Formally, a *bidirectional problem* associates *two* flow functions $ForwFlow(x,y)$ and $BackFlow(x,y)$ with each edge (x,y). The equations to be solved are as shown in Figure 2, and the desired solution to these equations is the one that could be obtained by iterating from an initial guess of $Info(y) = \top$ for all y.

For arbitrary bidirectional problems, iteration is much less attractive than for unidirectional problems. When the nodes are visited in a well-chosen order [HU75], unidirectional bit-vector problems can be solved in just three or four passes on the graphs that occur in practice. The corresponding strategy for bidirectional problems would be to visit nodes alternately in postorder and in reverse postorder. Even an *acyclic* graph with n nodes needs $\Omega(n)$ passes in the worst case because a change from $\top$ to $\bot$ can propagate alternately forward and backward along edges. The known bidirectional problems of practical importance, however, are all easily shown to satisfy the simple algebraic conditions explained in §3.1, despite the superficial complexity of the specific equations for each problem. Moreover, the problems need only be solved for control flow graphs where certain edges have been split, as explained in §3.2. Edge splitting and algebra combine to make the important bidirectional problems easy to solve, as explained in §3.3. An intermediate step approximates the actual problems as unidirectional bit-vector problems. These problems (and some auxiliary unidirectional problems that are solved to determine their flow functions) have the properties (discussed in §2.3) that make the slotwise method preferable over standard methods.

3.1 Algebraic Assumptions

We list three possible assumptions about the flow functions in Figure 2. Consider any two outedges (x,y) and (x,z) from a node x. Then

$$BackFlow(x,y) = BackFlow(x,z). \tag{4}$$

Consider any backward function b. Then all ξ, η satisfy

$$b(\xi \sqcap \eta) = b(\xi) \sqcap b(\eta). \tag{5}$$

Consider any backward function b, any forward function f, and any result ξ of $\sqcap$-ing together some values of forward functions. Then all η satisfy

$$\begin{aligned} \eta \sqsubseteq\ & f(b(\eta)) = f(b(\eta) \sqcap \xi) \\ =\ & f(b(\eta) \sqcap EntryInfo). \end{aligned} \tag{6}$$

The *clustering*(4) and *distributivity*(5) assumptions are familiar from unidirectional data flow analysis, where they are often made, seldom really used, and sometimes false. In this paper, they are both used and true. The assumption

[4] If the word "lattice" is unfamiliar, then reading the notations in terms of pairs of bits is recommended. Though lattices have been put to good use in data-flow analysis, it is difficult to write an introduction to lattice theory for these purposes [Ros81].

that forward functions are *largish* (6) has no obvious intuitive significance. It just happens to be true for the important bidirectional problems.

3.2 Edge Splitting

We consider a simple transformation of the control flow graph underlying the bidirectional problem in Figure 2. To *split* a control flow edge (x, y) is to replace (x, y) by two new edges (x, u) and (u, y), where u is a new node representing an empty basic block. (In the course of optimization, some code may be moved into u.) Without any splitting, placement analysis would yield weaker information that misses chances to improve the compiled code [MR79, Fig. 6] [RWZ88, p. 18]. Once the principle of splitting has been accepted, there are still the questions of which edges to split and when to split them. Our splitting criteria are simple and independent of the optimizations to be performed:

- Any edge that runs directly from a branch node to a join node is to be split.
- Splitting is to be done *before* analysis begins.

Many papers have considered some form of edge splitting, usually with other splitting criteria. Dhamdhere and Isaac [DI80], for instance, considered edge splitting when placing code in light of execution-frequency information. For elimination of partial redundancies, the splitting criteria given above were used by Rosen, Wegman, and Zadeck [RWZ88, p. 17] and may have been used earlier. Sorkin [Sor89] proposed (incorrectly) that splitting to create loop preheaders would suffice. The fact that splitting is more widely needed was implicitly recognized by Morel and Renvoise [MR79, Fig. 6].

Alternatively, one can decide in the course of analysis which edges to split. This strategy was proposed independently, for different variations on the original placement analysis [MR79], by Dhamdhere [Dha88] and by Drechsler and Stadel [DS88]. Analysis predicts what *will* hold at each old node, then determines which edges need to be split in order to validate the predictions. For an ambitious compiler that tries to seize many of the opportunities that one optimization creates for another, splitting as needed complicates the compiler, especially if the compiler writer is unwilling to bear the expense of a separate call on the memory allocator for each split edge. Once code has been moved into a new block created by splitting for the sake of one optimization, the new block effectively becomes an old block for later optimizations, and a new edge to (from) the new block may need to be split later.

Because old basic blocks may become empty in the course of optimization, a late optimization that removes empty blocks is needed anyway. Our splitting criteria make the graph somewhat larger,[5] but much of the splitting we do will be done eventually, if not by one optimization then by another. Occasionally, we split an edge but never put anything into the new basic block. In this case, the edge will be restored as a byproduct of removing empty basic blocks.

[5] For example, we effectively add a dummy `else` to each `if-then` without one.

3.3 Some Problems Are Easy

The bidirectional problem in Figure 2 is *mostly backward* if the information at each node y satisfies

$$Info(y) \;=\; \mathcal{F}_C(y) \sqcap \mathcal{B}_A(y), \tag{7}$$

where $\mathcal{B}_A$ and $\mathcal{F}_C$ are defined as follows. Replacing $Info(z)$ with $\mathcal{B}(z)$ in (3), we get an ordinary backward problem that can be solved to yield the *backward approximation* $\mathcal{B}_A$ to $Info$. Then we visit each node y in any convenient order and use $\mathcal{B}_A(x)$ in place of $Info(x)$ in (2) to compute the *forward correction* $\mathcal{F}_C(y)$.

Solving a mostly backward problem is only slightly more work than solving the backward approximation, which can share storage with the final answer. The following result shows that placement analysis and other important bidirectional problems are indeed mostly backward.

Theorem 1 *Consider any bidirectional problem (Figure 2) such that (thanks to edge splitting) no edge runs from a branch node to a join node. Suppose the forward functions are largish and the backward functions are clustered and distributive. Then the problem is mostly backward.*

Proof. For each node z, let

$$\mathcal{M}(z) \stackrel{\text{def}}{=} \mathcal{F}_C(z) \sqcap \mathcal{B}_A(z).$$

Because $Info(z) \sqsubseteq \mathcal{B}_A(z)$ (and thus $Info(z) \sqsubseteq \mathcal{F}_C(z)$), we already have $Info(z) \sqsubseteq \mathcal{M}(z)$. To show that $\mathcal{M}(z) \sqsubseteq Info(z)$, we show that every edge (x, y) has

$$\mathcal{M}(y) \sqsubseteq f(\mathcal{M}(x)) \quad \text{where} \quad f = ForwFlow(x, y); \tag{8}$$
$$\mathcal{M}(x) \sqsubseteq b(\mathcal{M}(y)) \quad \text{where} \quad b = BackFlow(x, y). \tag{9}$$

Because $Info$ is the largest way to assign information to nodes such that similar $\sqsubseteq$ relations hold, these imply $\mathcal{M}(z) \sqsubseteq Info(z)$. Proving (8) and (9) thus implies that the problem is mostly backward.

Consider any edge (x, y). Let f and b be as in (8) and (9). Thanks to clustering and distributivity, the definition of $\mathcal{B}_A$ implies

$$\mathcal{B}_A(x) \;=\; b\left(\prod_{z \,\in\, Succ(x)} \mathcal{B}_A(z)\right) \tag{10}$$

In particular, $\mathcal{B}_A(x)$ has the form $b(\eta)$. Moreover, $\mathcal{F}_C(x)$ is either $EntryInfo$ or the result of $\sqcap$-ing together some values of forward functions. Thus

$$\begin{array}{lcll} \mathcal{F}_C(y) & \sqsubseteq & f(\mathcal{B}_A(x)) & \text{by definition of } \mathcal{F}_C \\ & = & f(\mathcal{B}_A(x) \sqcap \mathcal{F}_C(x)) & \text{because } f \text{ is largish} \\ & = & f(\mathcal{M}(x)) & \text{by commutativity} \end{array}$$

But $\mathcal{M}(y) = (\mathcal{F}_C(y) \sqcap \ldots) \sqsubseteq \mathcal{F}_C(y)$, so $\mathcal{M}(y) \sqsubseteq f(\mathcal{M}(x))$ and (8) holds.

For (9), the argument depends on whether or not y is a join node. Suppose first that y is a join node. Thanks to edge splitting, $Succ(x) = \{y\}$. Thus (10) implies $\mathcal{B}_A(x) = b(\mathcal{B}_A(y))$. Similarly, any other predecessor z of y

has $\mathcal{B}_A(z) = b(\mathcal{B}_A(y))$. Therefore

$$\begin{aligned}
\mathcal{B}_A(y) &= \prod_{z \in Pred(y)} \mathcal{B}_A(y) \\
&\qquad \text{because } \sqcap \text{ is idempotent}^6 \\
&\sqsubseteq \prod_{z \in Pred(y)} ForwFlow(z,y)[b(\mathcal{B}_A(y))] \\
&\qquad \text{because } ForwFlow(z,y) \text{ is largish} \\
&= \prod_{z \in Pred(y)} ForwFlow(z,y)(\mathcal{B}_A(z)) \\
&\qquad \text{because } \mathcal{B}_A(z) = b(\mathcal{B}_A(y)) \\
&= \mathcal{F}_C(y) \qquad \text{by definition of } \mathcal{F}_C
\end{aligned}$$

Therefore

$$\begin{aligned}
\mathcal{M}(x) &\sqsubseteq \mathcal{B}_A(x) && \text{by definition of } \mathcal{M} \\
&= b(\mathcal{B}_A(y)) && \text{because } Succ(x) = \{y\} \\
&= b(\mathcal{F}_C(y) \sqcap \mathcal{B}_A(y)) && \text{by } \mathcal{B}_A(y) \sqsubseteq \mathcal{F}_C(y) \\
&= b(\mathcal{M}(y)) && \text{by definition of } \mathcal{M}
\end{aligned}$$

and (9) holds if y is a join node.

Now suppose that y is not a join node. Then

$$\begin{aligned}
\prod_{z \in Succ(x)} \mathcal{B}_A(z) &\sqsubseteq f b\left(\prod_{z \in Succ(x)} \mathcal{B}_A(z)\right) \\
&\qquad \text{because } f \text{ is largish} \\
&= f(\mathcal{B}_A(x)) && \text{by (10)} \\
&= \mathcal{F}_C(y) && \text{by } Pred(y) = \{x\}
\end{aligned}$$

Therefore (10) implies $\mathcal{B}_A(x) \sqsubseteq b(\mathcal{F}_C(y))$. But $\mathcal{B}_A(x) \sqsubseteq b(\mathcal{B}_A(y)$ also, yielding

$$\begin{aligned}
\mathcal{B}_A(x) &\sqsubseteq b(\mathcal{F}_C(y)) \sqcap b(\mathcal{B}_A(y)) \\
&= b(\mathcal{F}_C(y) \sqcap \mathcal{B}_A(y)) = b(\mathcal{M}(y)).
\end{aligned}$$

Therefore $\mathcal{M}(x) \sqsubseteq \mathcal{B}_A(x) \sqsubseteq b(\mathcal{M}(y))$ and (9) also holds if y is not a join node. □

Corollary 1 *Under the hypothesis of Theorem 1, suppose also that y is either a join node or a node with $\mathcal{B}_A(y) = \bot$. Then $Info(y) = \mathcal{B}_A(y)$ and the forward functions along inedges of y have no effect on the solution.*

Proof. By the theorem, $Info(y) = \mathcal{F}_C(y) \sqcap \mathcal{B}_A(y)$. Suppose first that y is a join node. The argument for (9) in the proof of the theorem shows that $\mathcal{B}_A(y) \sqsubseteq \mathcal{F}_C(y)$ and hence that $Info(y) = \mathcal{B}_A(y)$. Now suppose instead that $\mathcal{B}_A(y) = \bot$. Then $Info(y) = \ldots \sqcap \bot = \bot = \mathcal{B}_A(y)$. In both cases, $Info(y)$ is found without applying any forward functions along inedges of y. These functions were already known to have no effect on $Info(z)$ for any $z \neq y$, so they have no effect at all on the solution. □

Drechsler and Stadel [DS88, p. 638] implicitly derived another corollary of Theorem 1 for one particular version of placement analysis. If enough of the basic blocks are empty, then the forward correction can be shown to be unnecessary at the nonempty blocks. A compiler using this corollary needs a separate round of edge splitting for each EPR-like optimization, to replenish the supply of empty blocks. With Theorem 1, on the other hand, we sometimes need the forward correction but never need to split an edge created by earlier splitting.

Dhamdhere and Patil [DP90, Dha91] considered a variant of placement analysis where forward-flowing information is merged with $\vee$ while backward-flowing information is merged with $\wedge$. They showed that this variant could be solved by a backward approximation followed by a variant of interval-based forward data flow analysis. In our result, on the other hand, the merge is the same in both directions, as is more common. The forward correction is a simple traversal in any convenient order. Our result also applies to irreducible control flow and to *any* bidirectional problem (as defined here) that satisfies the conditions explained in §3.1. Similar conditions, with the roles of forward and backward functions reversed, suffice to make a problem be mostly forward.

4 Worked Example

In this section we first present the details of placement analysis [MR79] as later refined by [Cho83, JD82a] and then work through a small example in detail.

4.1 Placement Analysis

We begin by reviewing of some auxiliary unidirectional problems whose solutions are treated like local information by the actual placement analysis. For most of this section, we consider a single expression E and the problem of where best to place the computations of E in the given program.

4.1.1 Availability

For each basic block x, let AVGEN(x) be 1 if the code in x generates the availability of E by computing E and not assigning to any operand of E thereafter. Let NONE(x) be 1 if x assigns to none of the operands of E. It is customary in [MR79]-inspired analysis to associate two global availability bits with each node y:

$$\text{AVIN}(y) = \begin{cases} 0 & \text{if } y = \texttt{Entry}; \\ \bigwedge_{x \in Pred(y)} \text{AVOUT}(x) & \text{if } y \neq \texttt{Entry}; \end{cases}$$

$$\text{AVOUT}(y) = \text{AVGEN}(y) \vee [\text{AVIN}(y) \wedge \text{NONE}(y)].$$

Substituting for AVOUT(x) in the equation for AVIN(y) yields equation (11) in Figure 3. This equation involves only the AVIN bits and exemplifies §2.1 with $\bot = 0$ and low information at `Entry`.

4.1.2 Partial Availability

The same local information used in §4.1.1 can also be used to determine *partial* availability, which says that E is available along some (perhaps not all) paths to a node y:

$$\text{PAVIN}(y) = \begin{cases} 0 & \text{if } y = \texttt{Entry}; \\ \bigvee_{x \in Pred(y)} \text{PAVOUT}(x) & \text{if } y \neq \texttt{Entry}; \end{cases}$$

$$\text{PAVOUT}(y) = \text{AVGEN}(y) \vee [\text{PAVIN}(y) \wedge \text{NONE}(y)].$$

Substituting for PAVOUT(x) in the equation for PAVIN(y) yields equation (12) in Figure 3. This equation involves only

[6] For all ξ, $\xi \sqcap \xi = \xi$.

$$\mathrm{AVIN}(y) = \begin{cases} 0 & \text{if } y = \texttt{Entry} \\ \bigwedge_{x \in Pred(y)} \mathrm{AVGEN}(x) \vee [\mathrm{AVIN}(x) \wedge \mathrm{NONE}(x)] & \text{if } y \neq \texttt{Entry} \end{cases} \tag{11}$$

$$\mathrm{PAVIN}(y) = \begin{cases} 0 & \text{if } y = \texttt{Entry} \\ \bigvee_{x \in Pred(y)} \mathrm{AVGEN}(x) \vee [\mathrm{PAVIN}(x) \wedge \mathrm{NONE}(x)] & \text{if } y \neq \texttt{Entry} \end{cases} \tag{12}$$

$$\mathrm{PPIN}(y) = \begin{cases} 0 & \text{if } y = \texttt{Entry} \\ \left(\mathrm{PAVIN}(y) \wedge (\mathrm{CB4}(y) \vee [\mathrm{PPOUT}(y) \wedge \mathrm{NONE}(y)]) \wedge \bigwedge_{x \in Pred(y)} [\mathrm{PPOUT}(x) \vee \mathrm{AVOUT}(x)] \right) & \text{if } y \neq \texttt{Entry} \end{cases} \tag{13}$$

$$\mathrm{PPOUT}(y) = \begin{cases} 0 & \text{if } y = \texttt{Exit} \\ \bigwedge_{z \in Succ(y)} \mathrm{PPIN}(z) & \text{if } y \neq \texttt{Exit} \end{cases} \tag{14}$$

$$\mathrm{INSERT}(y) \stackrel{\text{def}}{=} \mathrm{PPOUT}(y) \wedge \sim \mathrm{AVOUT}(y) \wedge \sim (\mathrm{PPIN}(y) \wedge \mathrm{NONE}(y)) \tag{15}$$

$$\mathrm{DELETE}(y) \stackrel{\text{def}}{=} \mathrm{PPIN}(y) \wedge \mathrm{CB4}(y) \tag{16}$$

Figure 3. Placement analysis in a fairly conventional formulation.

$$Info(y) = \mathcal{F}(y) \wedge \mathcal{B}(y) \tag{17}$$

$$\mathcal{F}(y) \stackrel{\text{def}}{=} \begin{cases} \langle 0, 1 \rangle & \text{if } y = \texttt{Entry} \\ \bigwedge_{x \in Pred(y)} [ForwFlow(x,y)](Info(x)) & \text{if } y \neq \texttt{Entry} \end{cases} \tag{18}$$

$$\mathcal{B}(y) \stackrel{\text{def}}{=} \begin{cases} \langle [NodeBack(y)](0), 0 \rangle & \text{if } y = \texttt{Exit} \\ \bigwedge_{z \in Succ(y)} [BackFlow(y,z)](Info(z)) & \text{if } y \neq \texttt{Exit} \end{cases} \tag{19}$$

$$[ForwFlow(x,y)](\xi) \stackrel{\text{def}}{=} \eta \quad \text{where} \quad \left\langle \begin{matrix} \eta_{in} \\ \eta_{out} \end{matrix} \right\rangle = \left\langle \begin{matrix} \xi_{out} \vee \mathrm{AVOUT}(x) \\ 1 \end{matrix} \right\rangle \tag{20}$$

$$[BackFlow(y,z)](\zeta) \stackrel{\text{def}}{=} \eta \quad \text{where} \quad \left\langle \begin{matrix} \eta_{in} \\ \eta_{out} \end{matrix} \right\rangle = \left\langle \begin{matrix} [NodeBack(y)](\zeta_{in}) \\ \zeta_{in} \end{matrix} \right\rangle \tag{21}$$

Figure 4. Equations (13) and (14) from placement analysis reformulated in terms of pairs of bits, using the functions $NodeBack(y)$ from bits to bits defined by $[NodeBack(y)](\beta) \stackrel{\text{def}}{=} \mathrm{PAVIN}(y) \wedge (\mathrm{CB4}(y) \vee [\beta \wedge \mathrm{NONE}(y)])$.

the PAVIN bits and exemplifies §2.1 with $\perp = 1$ and low information at Entry. Indeed, finding the PAVIN bits by the method of §2.1.1 is precisely the first step in finding the AVIN bits by the method of §2.1.2. Traditional algorithms, on the other hand, would not exploit the similarity between partial availability and availability.

4.1.3 Placement Possible Bits

One more bit of local information is used here. Let CB4(x) be 1 if E is computed in x before any assignments to operands of E. The global *placement possible* bits have the more complicated equations (13) and (14) in Figure 3.[7] Substituting for PPOUT(...) in the equation for PPIN(y) would not be helpful, but the somewhat more elaborate formal manipulations in §4.1.5 help reveal the direct applicability of §3 and the indirect applicability of §2.

4.1.4 Insertion and Deletion Points

Once the PPIN(...) and PPOUT(...) bits have been computed, it is easy to determine where to insert new computations of the expression and where to delete old ones that become fully redundant after the insertions. The last two equations in Figure 3 do this.

4.1.5 Pairs of Bits

When several expressions are considered, PPIN(y) and PPOUT(y) become two bit vectors, each of which has a slot for each expression. This representation is convenient for performing many operations, but not for reasoning about the results. Mathematically, it is more convenient to consider a single vector whose slots hold pairs of bits $\langle \text{PPIN}(y), \text{PPOUT}(y) \rangle$. The bitwise notations $(\wedge, \vee, \sqcap, \perp, \top, \sqsubseteq)$ from §2 are overloaded in the natural way, so as to apply to pairs of bits as well as to single bits. Unlike a flow function on a bit vector of length 2, however, a flow function applied to a pair of bits $\xi = \langle \xi_{in}, \xi_{out} \rangle$ may use both argument bits for each result bit:

$$f(\xi) = \eta \quad \text{where} \quad \left\langle \begin{matrix} \eta_{in} \\ \eta_{out} \end{matrix} \right\rangle = \left\langle \begin{matrix} f_{in}(\xi_{in}, \xi_{out}) \\ f_{out}(\xi_{in}, \xi_{out}) \end{matrix} \right\rangle.$$

Equations (13) and (14) in Figure 3 can now be written in terms of pairs of bits, where $\sqcap$ is $\wedge$ and $\perp$ is $\langle 0, 0 \rangle$. A pair $Info(y)$ is associated with each node y. This information depends on a pair $\mathcal{F}(y)$ flowing forward from predecessors of y and a pair $\mathcal{B}(y)$ flowing backward from successors of y, as shown in Figure 4 and discussed below.

The derivation of Figure 4 from Figure 3 is typical of what is needed for the analyses that support EPR-like optimizations. The equations in Figure 4 are of interest only because they have been chosen to be equivalent to equations (13) and (14) in Figure 3:

$$Info(y) = \langle \text{PPIN}(y), \text{PPOUT}(y) \rangle.$$

Consider any node $y \neq$ Entry. The local information associated with y determines the function $NodeBack(y)$ defined in the figure caption, and this function tells how to use PPOUT(y) in computing PPIN(y). Thus, (13) for $y \neq$ Entry can be rewritten as

$$\text{PPIN}(y) = [NodeBack(y)](\text{PPOUT}(y)) \wedge \bigwedge_{x \in Pred(y)} [\bullet\bullet\bullet].$$

To model this with (17), we want

$$\begin{aligned} \mathcal{F}(y)_{in} &= \bigwedge_{x \in Pred(y)} [\text{PPOUT}(x) \vee \text{AVOUT}(x)]; \\ \mathcal{B}(y)_{in} &= [NodeBack(y)](\text{PPOUT}(y)). \end{aligned}$$

To get the $\mathcal{F}(y)_{in}$ desired, we choose each $[ForwFlow(x,y)_{in}]$ as displayed in (20). To get the $\mathcal{B}(y)_{in}$ desired if $y \neq$ Exit, we observe that $NodeBack(y)$ distributes over the $\wedge$ in (14). Then we choose each $[BackFlow(y,z)_{in}]$ as displayed in (21). To get the $\mathcal{B}(y)_{in}$ desired if $y =$ Exit, we apply $NodeBack(y)$ to the value PPOUT(y) = 0 and obtain PAVIN(y) $\wedge$ CB4(y) as the appropriate $ExitInfo_{in}$ in (19).

Consider any node $y \neq$ Exit. We proceed as above, but now the formulas happen to be simpler. In particular, (14) for $y \neq$ Exit can be rewritten as

$$\text{PPOUT}(y) = 1 \wedge \bigwedge_{z \in Succ(y)} \text{PPIN}(z).$$

To model this with (17), we want

$$\begin{aligned} \mathcal{F}(y)_{out} &= 1; \\ \mathcal{B}(y)_{out} &= \bigwedge_{z \in Succ(y)} \text{PPIN}(z). \end{aligned}$$

To get the $\mathcal{F}(y)_{out}$ desired if $y \neq$ Entry, we choose each $[ForwFlow(x,y)_{out}]$ as displayed in (20). To get the $\mathcal{F}(y)_{out}$ desired if $y =$ Entry, we choose $EntryInfo_{out} = 1$ in (18). To get the wanted $\mathcal{B}(y)_{out}$, we choose each $[BackFlow(y,z)_{out}]$ as displayed in (21).

Two choices remain in Figure 4. To model (13) for $y =$ Entry, we choose $EntryInfo_{in} = 0$ in (18). To model (14) for $y =$ Exit, we choose $ExitInfo_{out} = 0$ in (19).

4.1.6 Algebraic Properties

Now that the PP equations have been cast into the general form of Figure 2, we can verify the algebraic assumptions in §3.1. In the definition (21) of the backward flow functions, the Boolean operations applied to ζ_{in} are independent of the outedge chosen. These operations also distribute over $\wedge$, so (4) and (5) hold. For (6), consider also the definition (20) of the forward flow functions. When a forward function is applied after a backward function, the result $\omega = f(b(\eta))$ has

$$\left\langle \begin{matrix} \omega_{in} \\ \omega_{out} \end{matrix} \right\rangle = f\left(\left\langle \begin{matrix} \bullet\bullet\bullet \\ \eta_{in} \end{matrix} \right\rangle\right) = \left\langle \begin{matrix} \eta_{in} \vee \bullet\bullet\bullet \\ 1 \end{matrix} \right\rangle.$$

Thus $\eta \sqsubseteq f(b(\eta))$. Moreover, because f ignores the *in*-component of its argument, any ξ with $\xi_{out} = 1$ has $f(b(\eta)) = f(b(\eta) \wedge \xi)$. In particular, $EntryInfo$ and all values of forward functions have $\xi_{out} = 1$, so (6) follows.

4.2 The Worked Example

Figure 5 contains a small program fragment and its control flow graph. Code motion guided by placement analysis eliminates partial redundancies, as shown in Figure 6. Local data flow information for this analysis is in Figure 7. Figure 8 shows the placement information found by the steps detailed in the rest of this section. In each column in Figure 8, the "–" entries represent nodes where bits are known to have

[7] The original equations [MR79] are even more complicated; we have incorporated later refinements [Cho83, JD82a].

the default value *because* the nodes were never visited in computing that column. While the number of such nodes is small here, it should be quite large in practice. Had there been more of the program between Entry and node 1 or between 6 and Exit in the example, our algorithm would not have visited any of those nodes.

We compute PAVIN by the algorithm in §2.1.1, using the substituted form in (12). For the computation of PAVIN, the controlling bit value is $\perp = 1$ and we have high information ($\top = 0$) at Entry. Thus, LOWER = START and any edge with AVGEN = 1 at its source node is a LOWER edge. Similarly, RAISE = STOP and any edge with AVGEN = 0 and NONE = 0 at its source node is a RAISE edge. The remaining edges are PROPAGATE edges.

The set of LOWER edges is {(4,5), (4,6)}; the set of RAISE edges is {(2,4)}. The set N_1 is initialized to {5, 6}. After propagation, 1, 2, 3, 4, 7, and Exit are also in N_1. Thus, PAVIN is 1 at nodes 1, 2, 3, 4, 5, 6, 7, Exit and 0 elsewhere.

We compute AVIN by the algorithm in §2.1.2, using the substituted form in (11). For the computation of AVIN, the controlling bit value is $\perp = 0$ and we have low information at Entry. Thus, RAISE = START and any edge with AVGEN = 1 at its source node is a RAISE edge. Similarly, LOWER = STOP and any edge with AVGEN = 0 and NONE = 0 at its source node is a LOWER edge. The remaining edges are PROPAGATE edges.

The set of RAISE edges is {(4,5), (4,6)}; the set of LOWER edges is {(2,4)}. From the calculation of PAVIN, we already know that N_1 is {1, 2, 3, 4, 5, 6, 7, Exit}. The set N_2 is initialized to {1, 4}. After propagation, 2 and 3 are also in N_2. Thus, AVIN is 1 at nodes {5, 6, 7, Exit} and 0 elsewhere.

The next step is to attack the PPIN, PPOUT system. By applying the transformations in §3.3 to equations (18) and (19) in Figure 4, we obtain the forward correction (22) to the backwards approximation (23) in Figure 9.

Substitution for *BackFlow* in (23) yields a system where all of the bits $\mathcal{B}_A()_{in}$ can be computed before computing any of the bits $\mathcal{B}_A()_{out}$ (see (24) and (25) in Figure 9). Problem (24) can be solved by the backwards version of the algorithm in §2.1.2. Since $\perp = 0$, RAISE = START and LOWER = STOP. The RAISE edges are { (4,5), (4,6) }; the LOWER edges are {(Entry,1), (2,4)}. The remaining edges are PROPAGATE edges.

We have low information at Exit because CB4(Exit) = 0. We initialize N_1 as {4}, then add 1, 3, 5, 6, and 7 to it. We initialize N_2 as {1}, then add 6 and 7 to it. Thus, the set of nodes y with $\mathcal{B}_A(y)_{in} = 1$ is { 3, 4, 5 }. The set of nodes z with $\mathcal{B}_A(z)_{out} = 1$ is { 2, 3, 5 }.

Substitution for *ForwFlow* in (22) yields a system where the bits $\mathcal{F}_C()_{out}$ are already known (see (26) and (27) in Figure 9). Thus, the set of nodes y with $\mathcal{F}_C(y)_{in} = 1$ is { 4, 5, 6, 7, Exit}. The equations for INSERT and DELETE in Figure 3 tell us to insert new computations in nodes 2 and 3 while deleting the old computation in node 4. The resulting program is indeed as shown in Figure 6.

5 Second Order Effects

Traditional EPR [MR79] performs placement analysis for all expressions "simultaneously" with a bit-vector algorithm but then rearranges the computations for each separate expression E. Several implementors have noticed that moving one expression may provide an opportunity to move other expressions. Slotwise analysis is well suited to detecting and exploiting such *second-order effects* without excessive work.

```
1   repeat
1     if P
2       then A ← 0
4     repeat
4       ... ← A + B
4     until Q
6   until R
```

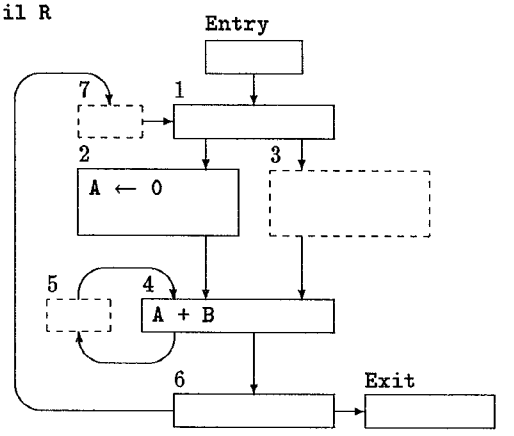

Figure 5. A sample program and its control flow graph. Edge splitting inserted the dashed line nodes.

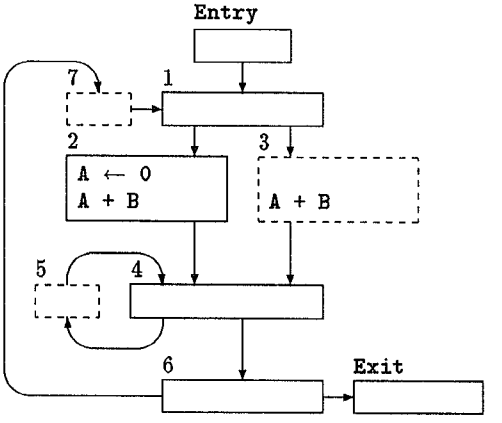

Figure 6. Effect of code motion on Figure 5.

Node	NONE	AVGEN	CB4
Entry	1	0	0
1	1	0	0
2	0	0	0
3	1	0	0
4	1	1	1
5	1	0	0
6	1	0	0
7	1	0	0
Exit	1	0	0

Figure 7. Local information for Figure 5.

Node	PAVIN	PAVOUT	AVIN	AVOUT	$\mathcal{B}_A()_{in}$	$\mathcal{B}_A()_{out}$	$\mathcal{F}_C()_{in}$	INSERT	DELETE
Entry	-	0	-	0	-	0	0	0	0
1	1	1	0	0	0	0	0	0	0
2	1	0	0	0	-	1	0	1	0
3	1	1	0	0	1	1	0	1	0
4	1	1	0	1	1	0	1	0	1
5	1	1	1	1	1	1	1	0	0
6	1	1	1	1	0	0	1	0	0
7	1	1	1	1	0	0	1	0	0
Exit	1	1	1	1	-	0	1	0	0

Figure 8. Global information for Figure 5. Default values (shown as "-") are 0 for PAVIN, AVIN, $\mathcal{B}_A()_{in}$.

$$\mathcal{F}_C(y) = \begin{cases} \langle 0, 1 \rangle & \text{if } y = \texttt{Entry} \\ \bigwedge_{x \in Pred(y)} [ForwFlow(x,y)](\mathcal{B}_A(x)) & \text{if } y \neq \texttt{Entry} \end{cases} \tag{22}$$

$$\mathcal{B}_A(y) = \begin{cases} \langle \text{PAVIN}(y) \wedge \text{CB4}(y), 0 \rangle & \text{if } y = \texttt{Exit} \\ \bigwedge_{z \in Succ(y)} [BackFlow(y,z)](\mathcal{B}_A(z)) & \text{if } y \neq \texttt{Exit} \end{cases} \tag{23}$$

$$\mathcal{B}_A(y)_{in} = \begin{cases} \text{PAVIN}(y) \wedge \text{CB4}(y) & \text{if } y = \texttt{Exit} \\ \bigwedge_{z \in Succ(y)} \text{PAVIN}(y) \wedge (\text{CB4}(y) \vee [\mathcal{B}_A(z)_{in} \wedge \text{NONE}(y)]) & \text{if } y \neq \texttt{Exit} \end{cases} \tag{24}$$

$$\mathcal{B}_A(y)_{out} = \begin{cases} 0 & \text{if } y = \texttt{Exit} \\ \bigwedge_{z \in Succ(y)} \mathcal{B}_A(z)_{in} & \text{if } y \neq \texttt{Exit} \end{cases} \tag{25}$$

$$\mathcal{F}_C(y)_{in} = \begin{cases} 0 & \text{if } y = \texttt{Entry} \\ \bigwedge_{x \in Pred(y)} [\mathcal{B}_A(x)_{out} \vee \text{AVOUT}(x)] & \text{if } y \neq \texttt{Entry} \end{cases} \tag{26}$$

$$\mathcal{F}_C(y)_{out} = 1 \tag{27}$$

Figure 9. Application of Theorem 1 to the equations in Figure 4.

We deal with the expressions separately *throughout* EPR, using a worklist. Before explaining the details, we review the (sometimes tacit) assumptions about intermediate code that underlie data flow analysis in general and placement analysis in particular.

To a first approximation, the intermediate code consists of assignments of one of two kinds. The simpler kind is just a *copy*:

$$\langle variable_{\text{dest}} \rangle \leftarrow \langle variable_{\text{source}} \rangle.$$

The more complex kind is a *computation*:

$$\langle variable \rangle \leftarrow \langle expression \rangle,$$

where $\langle expression \rangle$ is often, but not always, the application of a binary operator to a pair of variable or constant operands. The actual syntax of expressions is irrelevant here, so long as the operands are visible and so long as any possible changes to variables of interest are displayed as assignments to those variables.

Various practical complications can be handled by allowing the left-hand side of a computation to be a tuple of variables, each of which is assigned from an expression in a tuple on the right-hand side. For example, the target machine might not separate computing `X + Y` from computing a condition code to support branching on the sign of `X + Y`. Doing one entails doing the other, and both are moved or inserted or deleted at once. For a target machine

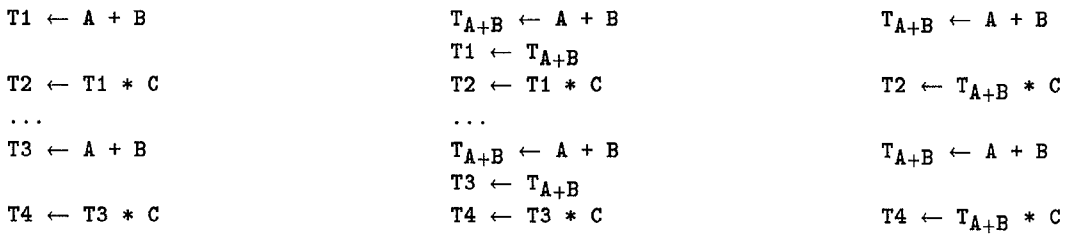

Figure 10. Code that exploits the modifications to take advantage of second-order effects.

that does addition this way, the intermediate code actually manipulated by EPR might include assignments like

$$\langle \texttt{A}, \texttt{CC} \rangle \leftarrow \langle \texttt{X} + \texttt{Y}, \texttt{Condition}(\text{``+''}, \texttt{X}, \texttt{Y}) \rangle,$$

where `A`, `X`, `Y` are numerical variables and `CC` is a condition code variable. Many more examples are in [CFR+91, §3.1]. Each "expression" considered by placement analysis is actually a tuple of expressions that appears on the right-hand side of a computation.

Once one sees that the major oversimplification in the first approximation *is* the presumption that all tuples are of length 1, it becomes fairly safe (as well as quite convenient) to use the approximation. Optimizations that seem to be restricted to simple computations like `A` ← `X+Y` are usually more general, but involve some small loops over the components of tuples when written out in general form. Omitting these loops simplifies the presentation in the following subsections. We also assume that local (within one basic block) redundancies have already been removed in the obvious way. Thus, a block may compute an expression E at most once before all assignments to operands of E and at most once after all assignments to operands of E. (There may also be computations of E interspersed with assignments to operands of E.) Section 5.1 deals with ordinary intermediate code. Section 5.2 deals with intermediate code in SSA form [CFR+91], which permits more extensive optimization in exchange for introducing another kind of copy operation and many more variables.

5.1 Ordinary Intermediate Code

Consider any expression E computed in the intermediate code, perhaps in several different basic blocks. This expression forms one slot for placement analysis. Initially, the worklist contains all of the slots. On each iteration of the algorithm, a slot is taken from the worklist and the equations in Figure 3 are solved for that one slot. If some INSERT or DELETE bits are nonzero, then the intermediate code is changed, as described below (this may entail adding another slot to the worklist). The algorithm terminates when the worklist empties.

If some of the INSERT or DELETE bits for the current slot E are nonzero, then the intermediate code is changed as follows:

1. A new unique temporary name $\texttt{T}_E$ is created.

2. Consider any basic block where AVGEN = 1 and DELETE = 0. The last old computation $V \leftarrow E$ in the block is replaced by a new computation $\texttt{T}_E \leftarrow E$ followed by a copy $V \leftarrow \texttt{T}_E$.

3. Consider any basic block where DELETE = 1. The first old computation $V \leftarrow E$ in the block is replaced by a copy $V \leftarrow \texttt{T}_E$.

4. Consider any basic block where INSERT = 1. A new computation $\texttt{T}_E \leftarrow E$ is inserted into the block, after any assignments to operands of E that may occur in the block.

The example in Figure 10 illustrates several points. On the left of this figure is an intermediate code fragment. Initially there are three slots, one for each of the expressions `A + B`, `T1 * C`, and `T3 * C`. If the slots for the last two expressions happen to be taken off the worklist first, then no changes are made to the intermediate code until the slot for `A + B` is considered. The middle part of the figure shows the results of processing the `A + B` slot. Two copy statements are added. The code on the right shows the results of copy propagation [ASU86]. Now the expression $\texttt{T}_{\texttt{A+B}}$ forms a new slot that is put on the worklist and leads to further optimization when its turn comes.

In this example, the copy propagation changes the expressions `T1 * C` and `T3 * C` to $\texttt{T}_{\texttt{A+B}} * \texttt{C}$. Not only must a new slot be created for the new expression, but also the slots for `T1 * C` and `T3 * C` must be added to the worklist, since the deletion (caused by copy propagation) of one instance of the expression may change the placement of the other instances of the expression.

5.2 SSA Form Intermediate Code

If the intermediate code is in static single assignment (SSA) form [CFR+91], then EPR can be more extensive though more complex. We plan to submit a longer version of this paper, with details for SSA-form intermediate code, to a journal. The rest of this section outlines some of the issues.

When a program is in SSA form, each variable is assigned a value at exactly one point in the program text. If the original intermediate code has several assignments to the same variable V, then each of them becomes an assignment to a new unique variable V_i. To preserve the original flow of values, SSA form introduces a new kind of copy operation at some of the join nodes.

A *ϕ-function* at a join node y has the form

$$var_{\text{dest}} \leftarrow \phi(var_{\text{source1}}, var_{\text{source2}}, \ldots),$$

where there is one operand for each control flow inedge of y. The operands are listed in the same (arbitrary) order used to list the inedges. If control passes to y along the K-th inedge (x, y), then the value of the corresponding operand $var_{\text{source}K}$ is copied to var_{dest} and the other operands are ignored.

With SSA form, copy propagation can always eliminate every ordinary copy $var_{dest} \leftarrow var_{source}$ from one variable to another. All uses of var_{dest}, including uses as operands of ϕ-functions, can safely be replaced by uses of var_{source}. Details are in [RWZ88], where it is shown that the resulting expressions act like "global value numbers" – if two occurrences of expressions E_1 and E_2 in the original program become occurrences of the very same expression $E_1' \equiv E_2'$ with SSA variables, then redundancy elimination can put those two occurrences in the same slot, even though the original E_1 and E_2 may look different.

Eliminating redundancies between two expressions that look different but compute the same value is useful. Unfortunately, translation to SSA form sometimes hides other opportunities for EPR when it replaces one expression E, computed in many different places with operands that have many different values, by many different computations of expressions $E', E'', \ldots$ with different SSA variables as the operands. Rosen, Wegman, and Zadeck [RWZ88] try to seize the new opportunities without losing the old ones, but their algorithm is complicated and restricted to reducible control flow. Moreover, there are some cases where the original EPR [MR79] moves an invariant out of a loop with multiple exits but [RWZ88] does not.[8] These cases suggest that SSA-based EPR should be done more along the lines of §5.1, but with some provision for putting several different expressions $E', E'', \ldots$ in the same slot whenever SSA form would hide old opportunities.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[CCF91] J. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. *Conf. Rec. Eighteenth ACM Symp. on Principles of Programming Langs.*, pages 55–66, January 1991.

[CFR+89] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. *Conf. Rec. Sixteenth ACM Symp. on Principles of Programming Langs.*, pages 25–35, January 1989.

[CFR+91] 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 Langs. and Systems*, 13(4):451–490, October 1991.

[Cho83] F. C. Chow. A portable machine-independent global optimizer – design and measurements. Technical Report 83-254 (PhD Thesis), Computer Systems Laboratory, Stanford U. Stanford, CA, December 1983.

[Cho88] F. C. Chow. Minimizing register usage penalty at procedure calls. *Proc. SIGPLAN'88 Symp. on Compiler Construction*, pages 85–94, June 1988. Published as *SIGPLAN Notices* Vol. 23, No. 7.

[8] An invariant expression E may be computed along some, but not all, paths through the loop body. It may also be computed along all paths starting from some, but not all, of the loop exits. If the end result is that E is computed along all paths after entering the loop, then [MR79] replaces all these computations of E by a new computation in the loop preheader.

[Dha88] D. M. Dhamdhere. A fast algorithm for code movement optimization. *SIGPLAN Notices*, 9(3):243–273, August 1988.

[Dha91] D. M. Dhamdhere. Practical adaptation of the global optimization algorithm of morel and renvoise. *ACM Trans. on Programming Langs. and Systems*, 13(2):291–294, April 1991.

[DI80] D. M. Dhamdhere and J. R. Isaac. A composite algorithm for strength reduction and code movement optimization. *Int. J. of Computer and Information Sci.*, 23(10):172–180, 1980.

[DP90] D. M. Dhamdhere and H. Patil. An efficient algorithm for bidirectional data flow analysis. Technical Report TR-016-90, Dept. of Computer Sci. and Eng., Indian Inst. of Technology, 1990. Revision to appear in *ACM Trans. on Programming Langs. and Systems.*

[DS88] K.-H. Drechsler and M. P. Stadel. A solution to a problem with Morel and Renvoise's "Global Optimization by Suppression of Partial Redundancies". *ACM Trans. on Programming Langs. and Systems*, 10(4):635–640, October 1988.

[HU75] M. S. Hecht and J. D. Ullman. A simple algorithm for global data flow analysis problems. *SIAM J. Computing*, 4(4):519–532, Dec. 1975.

[JD82a] S. M. Joshi and D. M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization (part I). *Int. J. of Computer Math.*, pages 22–41, 1982.

[JD82b] S. M. Joshi and D. M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization (part II). *Int. J. of Computer Math.*, pages 111–126, 1982.

[Kou77] L. T. Kou. On live-dead analysis for global data flow problems. *J. ACM*, 24(3):473–483, July 1977.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Comm. ACM*, 22(2):96–103, February 1979.

[MR90] T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. *Conf. Rec. Seventeenth ACM Symp. on Principles of Programming Langs.*, pages 184–196, January 1990.

[Ros81] B. K. Rosen. Degrees of availability as an introduction to the general theory of data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*, chapter 2, pages 55–76. Prentice Hall, 1981.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Langs.*, pages 12–27, January 1988.

[Sor89] A. Sorkin. Some comments on "A Solution to a Problem with Morel and Renvoise's 'Global Optimization by Suppression of Partial Redundancies' ". *ACM Trans. on Programming Langs. and Systems*, 11(4):666–668, October 1989.

[Zad84] 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.