A Simple Graph-Based Intermediate Representation

Cliff Click
cliffc@hpl.hp.com

Michael Paleczny
mpal@cs.rice.edu

Abstract

We present a graph-based intermediate representation (IR) with simple semantics and a low-memory-cost C++ implementation. The IR uses a directed graph with labeled vertices and ordered inputs but unordered outputs. Vertices are labeled with opcodes, edges are unlabeled. We represent the CFG and basic blocks with the same vertex and edge structures. Each opcode is defined by a C++ class that encapsulates opcode-specific data and behavior. We use inheritance to abstract common opcode behavior, allowing new opcodes to be easily defined from old ones. The resulting IR is simple, fast and easy to use.

1. Introduction

Intermediate representations do not exist in a vacuum. They are the stepping stone from what the programmer wrote to what the machine understands. Intermediate representations must bridge a large semantic gap (for example, from FORTRAN 90 vector operations to a 3-address add in some machine code). During the translation from a high-level language to machine code, an optimizing compiler repeatedly analyzes and transforms the intermediate representation. As compiler *users* we want these analyses and transformations to be fast and correct. As compiler *writers* we want optimizations to be simple to write, easy to understand, and easy to extend. Our goal is a representation that is simple and light weight while allowing easy expression of fast optimizations.

This paper discusses the intermediate representation (IR) used in the research compiler implemented as part of the author's dissertation [8]. The parser that builds this IR performs significant parse-time optimizations, including building a form of *Static Single Assignment* (SSA) at parse-time. Classic optimizations such as *Conditional Constant Propagation* [23] and *Global Value Numbering* [20] as well as a novel global code motion algorithm [9] work well on the IR. These topics are beyond the scope of this paper but are covered in Click's thesis.

The intermediate representation is a graph-based, object-oriented structure, similar in spirit to an operator-level *Program Dependence Graph* or *Gated Single Assignment* form [3, 11, 17, 18]. The final form contains all the information required to execute the program. The graph edges represent use-def chains. Analyses and transformations directly use and modify the use-def information. The graph form is a single-tiered structure instead of a two-tiered *Control-Flow Graph* (CFG) containing basic blocks (tier 1) of instructions (tier 2). Control and data dependencies have the same form and implementation. Optimizations such as constant propagation and value numbering use the same support code for both control and data values. The model of execution closely resembles the familiar CFG model.

Considerable care was taken in the design to allow for a very compact and easily edited implementation. Moving from the use of a value to the definition of a value requires a single pointer dereference. The implementation makes heavy use of C++'s features for code reuse, squeezing a lot of functionality in a few lines of code.

In Section 2, we introduce the graph-based intermediate representation. In Section 3, we give the model of execution for our representation. In Section 4, we present our C++ implementation. Section 5

This work has been supported by ARPA through ONR grant N00014-91-J-1989 and The Center for Research on Parallel Computation (CRPC) at Rice University, under NFS Cooperative Agreement Number CCR-9120008.

Authors' addresses: Cliff Click, Cambridge Research Office, Hewlett Packard Laboratories, One Main Street, 10th Floor, Cambridge, MA 02142. Michael Paleczny, Rice University, CITI/CRPC - MS 41, 6100 South Main, Houston, TX 77005-1892.

IR'95, 1/95, San Francisco, California, USA. © 1995 ACM

shows our experimental results for parse-time optimization. In Section 6, we look at related work, contrasting our intermediate representation with other published representations.

2. The Design

This section presents the structure of our intermediate representation, a directed graph with labeled vertices and ordered inputs. To help the reader understand the relationship between the program-as-a-graph and the classic CFG/basic block/instruction representation, we assume the program is in a simple canonical form. All expressions are simple assignments with at most one primitive operation on the right as in "$a := b + c$". The program is in SSA form, with subscripts on different versions of the same name.

2.1 Nodes and Values

The vertices will be called nodes. Edges are unlabeled and the order of output edges leaving a node is not important. The label on a node determines the kind of operation, or program primitive, the node represents. The inputs to a node are inputs to the node's operation. Each node defines a value based on its inputs and operation, and that value is available on all output edges.

Values include the typical machine word integers, as well as pointers, floating point values, and function pointers. Values also include abstract quantities like the state of memory, I/O with the outside world or control. Memory (state) and I/O (i/o) are discussed later; the control value will be covered shortly. Special values will be written underlined to help distinguish them in the text.

Nodes will be referred to with lower case letters like *x* and *a*. Node operations will be written in SMALLCAPS style. The operation at a node is referred to as *x.opcode*. Node inputs will use array syntax, so that input 0 to a node *x* is written *x*[0]. Since edges are unlabeled, we never refer to them directly; *x*[0] refers to the node that defines the first input to node *x*. The only operation we perform on output edges is to iterate all the nodes they are inputs to: forallusers *y* of *x* do *something*.

An expression "$a := b + c$" is represented by a 2-input node labeled with an ADD. The node names *a*, *b*, and *c* are for our reference only. They are not used for program semantics but are useful for output. We show the graph representation in Figure 1. Nodes are shown as a shadowed box with rounded corners. Edges carrying data values are shown as light arrows. In our written notation then, $a[0] \equiv b$, $a[1] \equiv c$, and $a.opcode \equiv$ ADD.

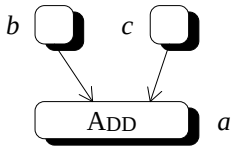


Figure 1 Expression "$a := b + c$" as a graph

2.2 Basic Blocks and REGION Nodes

Traditional representations have two distinct levels. At the top level, the CFG contains basic blocks. At the bottom level, each basic block contains instructions. In the past, this distinction has been useful for separation of concerns. CFGs deal with control flow and basic blocks deal with data flow. We handle both kinds of dependences with the same mechanism, removing this distinction to simplify our representation.

We replace basic blocks with a special REGION node [11]. A REGION node takes a control value from each predecessor block as inputs and produces a merged control as an output. Figure 2 shows the change from basic blocks to REGION nodes. Edges carrying control information are shown with thick arrows. Each primitive node takes in a control input to indicate which basic block the primitive is in. *This edge is not always required.* Removing it enables a number of global optimizations but requires a more complex serialization operation for output [8]. Such optional control edges will be shown as dashed arrows.

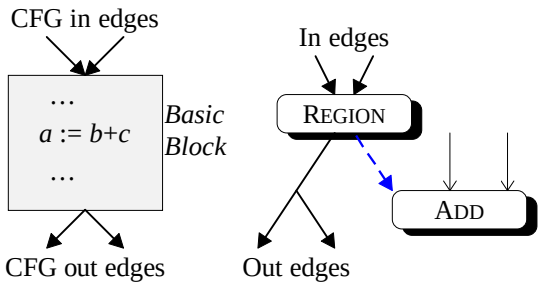


Figure 2 Basic blocks and REGION Nodes

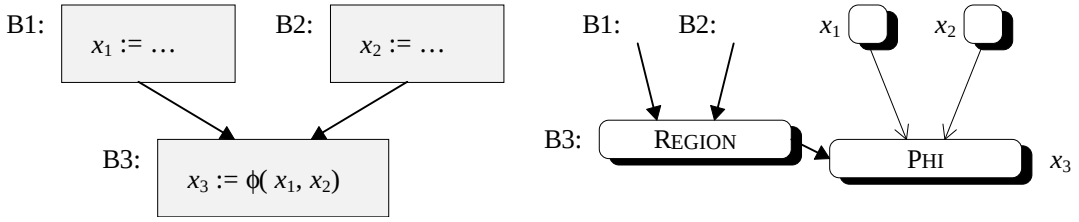


Figure 3 PHI Nodes for merging data

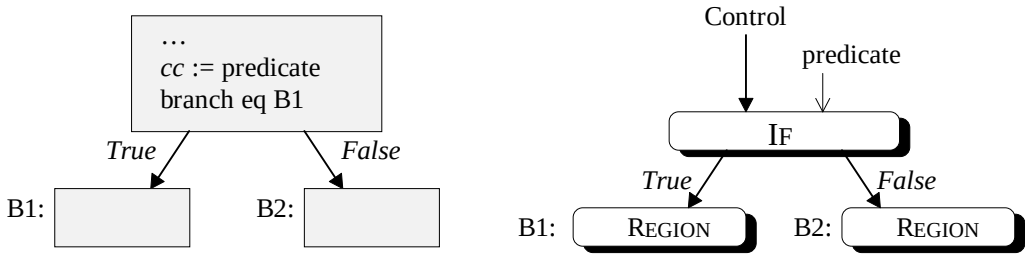


Figure 4 An example IF construct

2.3 PHI Nodes

Since our program is in SSA form, our graph representation uses ϕ-functions represented by PHI nodes. PHI nodes input several data values and output a single selected data value.

CFG edges determine which values are merged in the ϕ-functions. Without those CFG edges our intermediate representation is not *compositional*[1] [4, 20]. We need to associate with each data input to a PHI node the control input from the corresponding basic block. Doing this directly means that PHI nodes would have a set of pairs as inputs. One element of the pair would be a data dependence and the other a control dependence. This is an ungainly structure with complicated semantics. Instead, we borrow some ideas from Ballance, *et al*. and Field [3, 14].

The control input comes from the REGION node defining control for the PHI node. The other inputs to the PHI node are **aligned** with the REGION node's control inputs. The *i*th data input to the PHI node matches the *i*th control input to the REGION node. In essence, we split the paired inputs between the PHI node and the REGION node. We show a Phi node in Figure 3.

Note that these nodes have no run-time operation and do not correspond to a machine instruction. They exist to mark places where values merge and are required for correct optimization. When machine code is finally generated, the PHI nodes are folded back into normal basic blocks and CFG behavior.

2.4 IF Nodes and Projections

We replace conditional instructions with IF nodes as shown in Figure 4. In the basic-block representation, the predicate sets the condition codes (the variable named *cc*) and the *branch* sends control to either block **B1** or block **B2**. With explicit control edges, the IF node takes both a control input and a predicate input. If the predicate is true, the IF node supplies control to the true basic block's REGION node. Otherwise, control is sent to the false basic block's REGION node. As shown in Figure 4, IF nodes have labeled output edges.

Intuitively, some operations produce more than one value. Examples include instructions that set a condition code register along with computing a result (*i.e.*, subtract) and subroutine calls (which set at least the result register, the condition codes, and memory). One method for handling this is to label outgoing edges with the kind of value they carry. This requires edges to carry information and has a direct (negative) impact on our implementation. We prefer to keep our edges very lightweight.

Instead, multiple-value instructions produce the same tuple for each output edge. Then we use PROJECTION nodes to strip out the piece of the tuple

[1] In essence, we would require information not local to an instruction. A non-compositional representation is difficult to transform correctly because changing an instruction may require information not directly associated with the instruction.

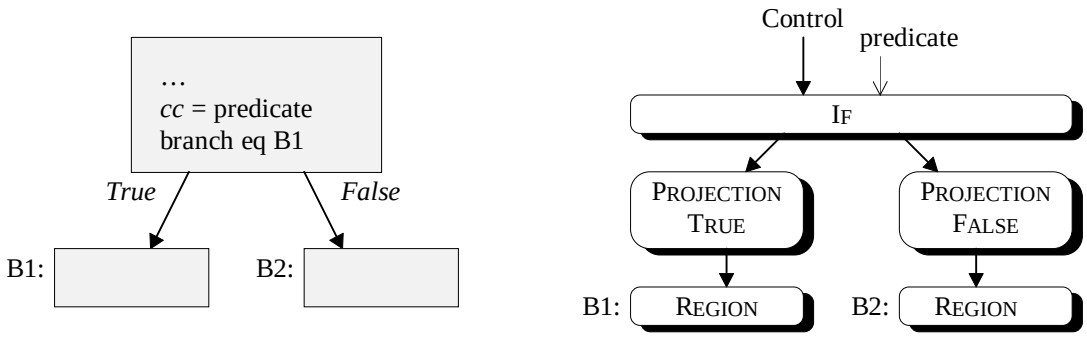


Figure 5 Projections following an IF Node

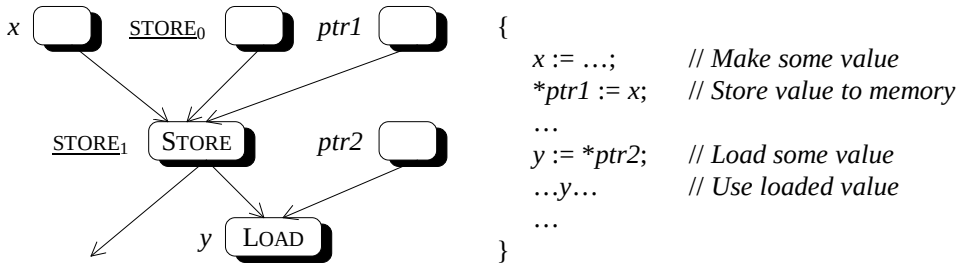


Figure 6 Treatment of memory (STORE)

that we want, giving distinct names to the different results. The PROJECTION has a field specifying which piece of the input tuple to project out. PROJECTIONs have no corresponding run-time operation (*i.e.*, they "execute" in zero cycles).

Figure 5 shows both the structure of a PROJECTION node and how it is used with an IF node. An IF node takes in control and a predicate and produces two distinct outputs: the true control and the false control. The IF produces a tuple of those two values. A PROJECTION-TRUE strips out the true control and a PROJECTION-FALSE strips out the false control.

2.5 Compound Values: Memory and I/O

We treat memory like any other value, and call it the STORE. The START node and a PROJECTION-STORE node produce the initial STORE. LOAD nodes take in a STORE and an address and produce a new value. STORE nodes take in a STORE, an address, and a value and produce a new STORE. PHI nodes merge the STORE like other values. Figure 6 shows a sample treatment of the STORE.

The lack of anti-dependences[2] is a two-edged sword. Between STORE's we allow LOAD nodes to reorder. However, some valid schedules (serializations of the graph) might overlap two STOREs, requiring that all of memory be copied. Our serialization algorithm treats memory like a type of unique machine register with infinite spill cost. The algorithm schedules the code to avoid spills if possible, and for the STORE it always succeeds.

This design of the STORE is very coarse. A better design would break the global STORE into many smaller, unrelated STORE's. Every independent variable or array would get its own STORE. Operations on the separate STORE's could proceed independently from each other. We could also add some understanding of pointers [7].

Memory-mapped I/O (*e.g.*, **volatile** in C++) is treated like memory, except that both READ and WRITE nodes produce a new I/O state. The extra dependence (READs produce a new I/O state, while LOADs do not produce a new STORE) completely serializes I/O. At program exit, the I/O state is required,

[2] An *anti-dependence* is a dependence from a *read* to a *write*. For the STORE, an anti-dependence is from a LOAD to a STORE.

however, the STORE is not required. Non-memory-mapped I/O requires a subroutine call.

The START node produces the initial control as well as initial values for all incoming parameters, memory, and the I/O state. The START node is a classic multi-defining instruction, producing a large tuple. Several PROJECTION nodes strip out the various smaller values.

We treat subroutines like simple instructions that take in many values and return many values. Subroutines take in and return at least the control token, the STORE, and the I/O state. They also take in any input parameters and may return an additional result.

3. A Model of Execution

Since our graph representation lacks basic blocks and the CFG, we needed to rethink our model of execution. In the CFG, control moves from basic block to basic block with the next block determined by the last instruction of the current block. Within a basic block, control moves serially through the instructions. Like the execution model for CFGs, our execution model has two distinct sections. We have two distinct subgraphs embedded in our single graph representation. Optimizations make no distinction between the subgraphs; only the functions used to approximate opcodes differ.

The **control** subgraph uses a Petri net model. A single control token moves from node to node as execution proceeds. This reflects how a CFG works, as control moves from basic block to basic block. Our model restricts the control token to REGION nodes, IF nodes, and the START node. The starting basic block is replaced with a START node that produces the initial control. Each time execution advances, the control token leaves the node it is currently in. The token moves onward, following the outgoing edge(s) to the next REGION or IF node. If the token reaches the STOP node, execution halts. Because we constructed the graph from a CFG, we are assured only one suitable target (REGION, IF, STOP) exists on all the current node's outgoing edges.

The **data** subgraph does not use token-based semantics. Data nodes' outputs are an immediate reflection of their inputs and operation. There is no notion of a "data token"; this is not a Petri net. Data values are available in unlimited amounts on each output edge. Intuitively, whenever a node demands a value from a data node, it follows the input edge to the data node and reads the value stored there. In an acyclic graph, changes ripple from root to leaf "at the speed of light". When propagation of data values stabilizes, the control token moves on to the next REGION or IF node. We never build a graph with a loop of only data-producing nodes; every loop has either PHI or REGION nodes.

3.1 Mixing the Subgraphs

Our two subgraphs interact at two distinct node types: PHI nodes and IF nodes. The PHI reads in both data and control, and outputs a data value. Unlike other data nodes, PHI nodes change outputs only when their REGION node is visited. When the control token moves into a REGION, the PHI nodes at that REGION latch the data value coming from the matching control input. The control token is not consumed by the PHI.

In Figure 3, when the control token moves from B1 to the REGION, the PHI latches the value from node x_1. If instead the control comes from B2, the PHI would latch x_2's value.

IF nodes take in both a data value and a control token, and produce either the {control, ∅} tuple or the {∅, control} tuple. In Figure 5, the following projections produce either the control token or no value.

3.2 A Loop Example

Figure 7 shows what a simple loop looks like. Instead of a basic block heading the loop, there is a REGION node. The REGION node merges control from outside the loop with control from the loop-back edge. There is a PHI node merging data values from outside the loop with data values around the loop. The loop ends with an IF node that takes control from the REGION at the loop head. The IF passes a true control back into the loop head and a false control outside the loop.

Irreducible loops are handled very naturally. A second REGION (and a second set of PHI nodes) marks the second entry into the loop. Unstructured control flow is also handled naturally; REGION nodes mark places where control flow merges and IF nodes are places where control flow splits.

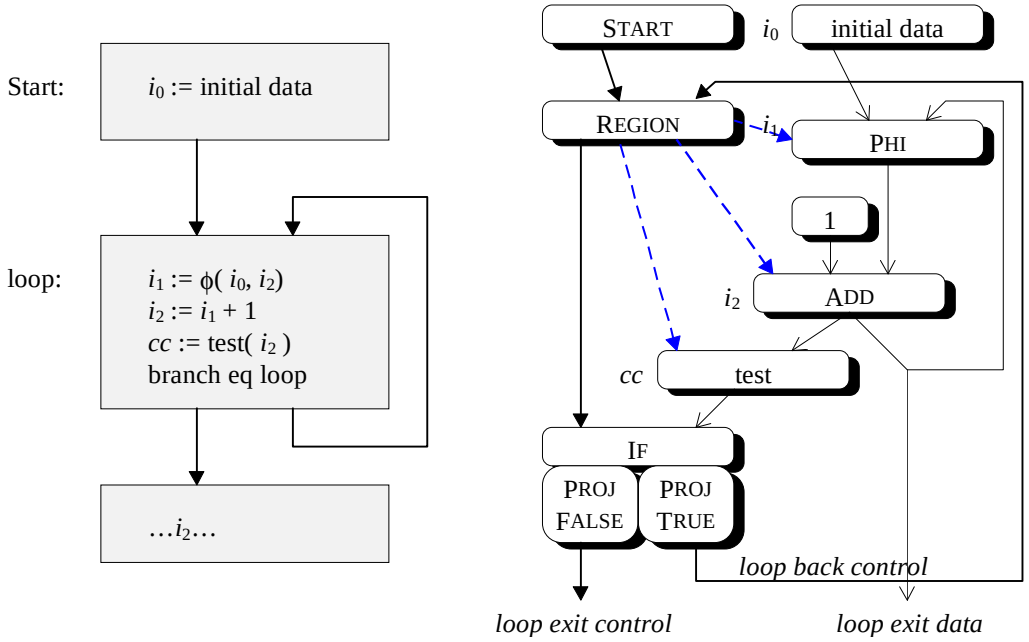


Figure 7 An example loop

4. Implementation

We used C++ for our implementation. We felt this gave us a better match between the abstract design and the concrete implementation than a previous C version. The most obvious occurrence of this was coding operation-dependent behavior in various analyses. The C version used large **switch** statements to branch on the operation. This spread out the semantics of any individual operation into many disjoint sections of code. Adding a new operation was painful; each **switch** had to be altered individually.

In the C++ version we give each operation its own class, inheriting from the base class Node. This lets us use **virtual** functions to specify operation-dependent behavior. Each operations' semantics are localized in the individual classes. Adding a new operation consists of inheriting from some similar existing class and overriding any changed behavior. The classes representing primitive operations typically inherit behaviors like the number of inputs, required input and output values (*e.g.*, integer *vs.* floating point), default constant folding, algebraic identities, and printing functions.

4.1 The Node Class

Instances of class Node contain almost no data; operation-specific information is in classes that inherit from Node. We show a slice of the class hierarchy in Figure 8. Class AddNode defines instances of algebraic rings, including the additive and multiplicative identity, commutativity, and associativity. The class IntAddNode merely provides the identity elements (integers 0 and 1) and a unique string name used in printing ("`iADD`"), and inherits all other behaviors. Finally, an integer add is an instance of class IntAddNode.

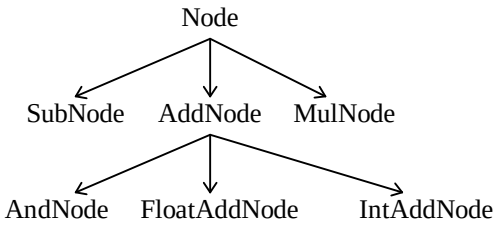
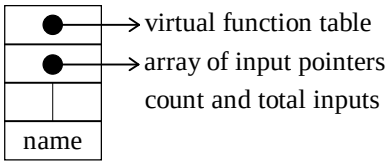


Figure 8 Part of the class Node hierarchy

Class Node, shown in Figure 9, holds only the virtual function pointer, edge pointers and the destination variable name. Notice that the Node object does not contain an opcode field of any sort. The virtual function pointer is unique to each class; we use it as the node's opcode. If we need to perform an opcode-specific function, we use a virtual function call.

4.2 Lightweight Edges

In the abstract design, edges flow from defining node to using node (*i.e.*, def→use edges). The pointers in the implementation define the edge direction back-




```
class Node {
public:
    Node **inputs;                          // Array of input pointers
    short cnt, total;                       // Count of semantic and precedence input pointers
    int name;                               // Name as a symbol table index
    Node(short cnt, short total, Node **inputs); // Constructor, with input count
    // Inline array syntax for accessing the inputs to a Node
    Node *&operator[]( int idx ) const { assert(idx<total); return inputs[idx]; }
    virtual const char *Name() = 0;         // Printable opcode name
};
```

Figure 9 Object layout and code for class Node

wards (*i.e.*, use→def edges). As shown in Figure 10, the concrete implementation of this graph allows convenient traversal of a design edge from sink to source (use to def) instead of from source to sink (def to use). Our edges are implemented as pointers to Nodes.

Some nodes require more fields. A CON node (which defines a literal constant) needs to hold the value of the constant being defined and has no other inputs. A PROJECTION has one input and a field to specify which piece of the incoming tuple to project out. As part of code generation we include add-immediate and load-with-offset operations. In Figure 11, these offsets and constants are kept in class-specific fields instead of using a large C-style **union**.

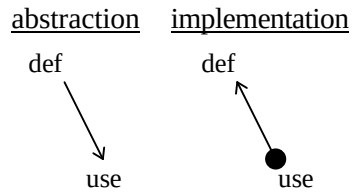


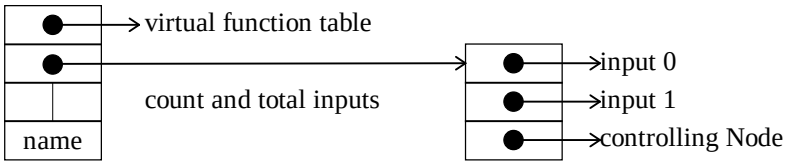
Figure 10 The implementation of dependence edges

The number of edges carrying semantic meaning is generally unchanged throughout compilation (*i.e.*, divide has exactly 2 inputs). However, the compiler may find it convenient to add precedence edges (*e.g.*, for scheduling or dependence analysis purposes). We handle this requirement by using a counted array of Node pointers. If we need another edge and the array is full, we reallocate the array to larger storage. Usually, we know the edge count ahead of time and the edge pointer array is kept in the Node structure. Class SubNode, shown in Figure 12, is an example of a 2-input, 1-result instruction. It inherits the virtual function pointer from Node and adds 2 use-def pointers and the control pointer.

Although not required for correct semantics, def-use edges are necessary for efficient forward analysis. We implement them in essentially the same manner as precedence edges. For clarity we do not show their fields. We build def-use edges in a fast batch pass just prior to doing global forward analysis. After the analysis, we discard the def-use edges because we do not wish to maintain them while transforming the pro-

```
class IntConNode : public Node {        // Class of Nodes that define integer constants
    virtual const char *Name() { return "iLDI"; }
    const int con;                      // The constant so defined
};
class ProjNode : public Node {          // Class of projections of other Nodes
    virtual const char *Name() { return "Projection"; }
    Node *src[1];                       // The tuple-producing Node
    const int field;                    // The desired field out of the incoming tuple-value
};
```

Figure 11 Definition for the IntConNode and ProjNode classes




```
class SubNode : public Node {                  // Class of 2-input, 1-result instructions
    Node *src[3];                              // Usually 3 inputs total to an Sub
    SubNode(Node *in0, Node *in1, Node *control) : Node(3,3,src)
    { inputs[0] = in0;  inputs[1] = in1;  inputs[2] = control; };
};
```

Figure 12 Object layout and code for a 2-input Node

```
int Node::hash()                                   // Hash opcode and inputs
    { int sum = (int)Name; for( int i=0; i<cnt; i++ ) sum += (int)input[i]; return sum;}
int IntConNode::hash() { return Node::hash()+con; }    // Constant is part of hash function

Node *Node::Identity() { return this; }            // Return a pre-existing equivalent Node, if any
Node *IntSubNode::Identity()                       // Integer-subtract-specific implementation
    { return src[1]→is_constant(0) ? src[0] : this; }  // Subtract of a zero is just the other input
```

Figure 13 Virtual hash and identity functions

gram with the results from the analysis.

4.3 Virtual Support Functions

We do the local value-numbering optimization using a hash table, hashing the Nodes by opcode and inputs. The opcode is ideally the class virtual function table pointer, which C++ does not let us access directly. Instead, we use the address of the virtual Name() function, which is unique to each class that defines an opcode. The hash table lookup must be able to compare nodes based on class and inputs. Differently classed nodes have different hash functions and different compare semantics. For example: addition is commutative; two ADD nodes are equal if their inputs match in any order. Sample code for virtual hash functions is presented in Figure 13.

Other common behaviors are also defined as virtual functions. In particular, algebraic identities (subtract of zero, multiply by 1) are handled this way. A call to the virtual Identity(), also shown in Figure 13, returns a node equivalent to the one passed in. This function is used in both local and global value numbering.

Each node is a self-contained C++ object, that contains all the information required to determine how the node interacts with the program around it. The major field in a node is the opcode, represented as the virtual function table pointer. An object's class determines how it propagates constants, handles algebraic identities, and finds congruences with other nodes. To make the intermediate representation understand a new kind of operation we need only to define a new class. The new class inherits fields for inputs and supplies functions for algebraic identities and hash table support. We do not need to make any changes to the optimization code itself.

4.4 SSA *vs.* Names

While SSA form is useful for optimization, the compiler must eventually emit code without ϕ-functions. This requires assigning node results to variable names. We use an infinite set of *virtual register numbers* for our names, relying on a register allocator to reduce the number used to what is really available. Our destination names are stored in the **name** field; source names are found by following use-def edges.

Instead of removing the ϕ-functions, we make them merge exactly the same set of names. By leaving the ϕ-functions in place we get to keep our use-def information. The ϕ-functions behave like input ports to basic blocks.

We start by assigning unique integers to every name field, thus maximally renaming values, then insert a COPY node on every ϕ-function input. We give the COPY the same destination name as the ϕ-function

```
class Arena {                          // Arenas are linked lists of large chunks of heap
    enum { size = 10000 };             // Chunk size in bytes
    Arena *next;                       // Next chunk
    char bin[size];                    // This chunk
    Arena( Arena *next ) : next(next) {}        // New Arena, plug in at head of linked list
    ~Arena() { if( next ) delete next; } // Recursively delete all chunks
};
class Node {                           // Base Node class
    static Arena *arena;               // Arena to store nodes in
    static char *hwm, *max, *old;      // High water mark, limit in Arena
    static void grow();                // Grow Arena size
    void *operator new( size_t x )     // Allocate a new Node of given size
    { if( hwm+x > max ) Node::grow(); old := hwm; hwm := hwm+x; return old; }
    void operator delete( void *ptr)   // Delete a Node
    { if( ptr = old ) hwm := old; }    // Check for deleting recently allocated space
};
Arena *Node::arena := NULL;            // No initial Arena
char *Node::hwm := NULL;               // First allocation attempt fails
char *Node::max := NULL;               // … and makes initial Arena
void Node::grow()                      // Get more memory in the Arena
{
    arena := new Arena(arena);         // Grow the arena
    hwm := &arena→bin[0];              // Update the high water mark
    max := &arena→bin[Arena::size];    // Cache the end of the chunk as well
}
```

Figure 14 Fast allocation with arenas

making the inputs to a ϕ-function all have the same destination name. Occasionally, we cannot naively insert COPYs because the clobbered destination name is also a source name; these cases require a temporary variable.

This COPY insertion method is very naive. A robust compiler requires a coalescing phase afterwards. It is possible to combine the COPY insertion pass and the coalescing pass so that COPYs are only inserted when values cannot be renamed (instead of inserting the COPYs then having the coalesce phase remove them by renaming) [2].

4.5 Fast Node Allocation

Each time we make a new node we call the default **operator new** to get storage space. This in turn calls **malloc** and can be fairly time consuming. In addition, optimizations frequently delete a newly created object, requiring a call to **free**. We speed up these frequent operations by hooking the class specific **operator new** and **delete** for class Node. Our replacement operators use an *arena* [15]. Arenas hold heap-allocated objects with similar lifetimes. When the lifetime ends, we delete the arena freeing all the contained objects in a fast operation. The code is given in Figure 14.

Allocation checks for sufficient room in the arena. If sufficient room is not available, another chunk of memory is added to the arena. If the object fits, the current high water mark[3] is returned for the object's address. The high water mark is then bumped by the object size. The common case (the object fits) amounts to an inlined test and increment of the high water marker.

Deallocation is normally a no-op (all objects are deleted at once when the arena is deleted). In our case, we check to see if we just allocated the deleted memory. If it was, the **delete** code pushes back the high water marker, reclaiming the space for the next allocation. The very fast allocation-deallocation cycle allows a parse-time optimizer to quickly perform naive node creation which may be reversed by an immediately following optimization.

[3] The high water mark is the address one past the last used byte in a memory chunk.

5. Parse-Time Optimizations

With use-def information we can do optimization *while parsing*. Parse-time optimizations remove simple constant computations and common subexpressions early on, before the slower global optimizations. This reduces the peak memory required by the optimizer and speeds the global optimizations that follow.

Parse-time optimizations must be pessimistic (as opposed to optimistic), because we do not know what the not-yet-parsed program will do. They do not find as many constants or common subexpressions as the global techniques, particularly around loops. For this reason they are not a replacement for global analysis or optimistic transformations, which require def-use information generally not available during parsing.

Our pessimistic analysis requires only use-def information, which we can gather as we parse the code. The compiler looks at (and changes) a fixed-size instruction "window" of the intermediate representation [11]. This window looks over the program graph instead of sequential instructions, providing the compiler with access to related instructions far away in the program text. Code outside the window is not affected by the transformation; code inside the window is transformed without knowledge of what is outside the window. Thus our pessimistic analysis is essentially a local, or peephole, analysis.

5.1 Optimizing Through a Peephole

The optimization works as follows: every time the parser builds a new node but before the node is referenced by any other node, the parser attempts to replace the new node with an existing node that computes the same value. Nodes are queried via virtual functions to determine if they compute a constant, are an algebraic identity on some other node, or can be value-numbered (equal operation on equal inputs) to some other node. If possible, new nodes are deleted and replaced with references to constants or prior nodes.

During value-numbering we ignore the control input to computing nodes that do not cause exceptions. This allows value-numbering to find nodes in different control regions to be the same. Since the prior node that replaces a new node may not dominate uses of the new node we need a pass of *Global Code Motion (GCM)* to discover a legal ordering of instructions. GCM moves code out of loops and into more control dependent regions. GCM relies solely on dependence information and does not need any prior legal ordering of instructions. GCM orders instructions by setting the control input to some REGION node, thus selecting a basic block. Other dependences determine the order within the basic block.

We compared the parse-time optimizations against a global optimization that combines *Conditional Constant Propagation (CCP)* [23], *Global Congruence Finding (GCF)* [1], and *Global Value Numbering* [20]. The combination is stronger than iterating these optimizations to a fixed point (*i.e.*, the combination finds at least as many constants and equivalent expressions) and is discussed in Click's thesis [8]. Both kinds of optimization are followed by a pass of GCM and *Dead Code Elimination (DCE)*.

5.2 Experimental Method

We converted a large test suite into a low-level intermediate language, ILOC [5, 4]. The ILOC produced by translation is very naive and is intended to be optimized. We then performed several optimizations at the ILOC level. We ran the resulting ILOC on a simulator to collect execution cycle counts for the ILOC virtual machine. All applications ran to completion on the simulator and produced correct results.

ILOC is a virtual assembly language for a virtual machine. The machine has an infinite register set and 1 cycle latencies for all operations, including LOADs, STOREs and JUMPs. ILOC is based on a load-store architecture and includes the usual assortment of logical, arithmetic and floating-point operations.

The simulator is implemented by translating ILOC files into a simple form of C. The C code is then compiled on the target system using a standard C compiler, linked against either the FORTRAN or C libraries and executed as a native program. The executed code contains annotations that collect statistics on the dynamic cycle count.

The test suite consists of a dozen applications with a total of 52 procedures. The procedures and their static instruction sizes are given in Figure 15. All the applications are written in FORTRAN, except for **cplex**. **Cplex** is a large constraint solver written in C. **Doduc**, **tomcatv, matrix300** and **fpppp** are from the Spec89 benchmark suite. **Matrix300** performs various matrix multiplies. Since our machine model does not include a memory hierarchy (single cycle LOAD latencies), we can not measure the effects of memory access patterns in the inner loops. Thus we choose **matrix300** over **matrix1000** for faster testing times, even cutting the matrix size down to 50. The remaining procedures come from the Forsythe, Malcom and Moler suite of routines [15].

Program	Routine	Static Cycles
doduc	x21y21	113
doduc	hmoy	162
doduc	si	166
doduc	coeray	370
doduc	dcoera	544
doduc	drigl	612
doduc	colbur	775
doduc	integr	844
doduc	ihbtr	919
doduc	cardeb	938
doduc	heat	1,059
doduc	inideb	1,081
doduc	yeh	1,084
doduc	orgpar	1,490
doduc	subb	1,537
doduc	repvid	1,644
doduc	drepvi	1,839
doduc	saturr	1,861
doduc	bilan	2,014
doduc	supp	2,035
doduc	inithx	2,476
doduc	debico	2,762
doduc	prophy	2,843
doduc	pastem	3,500
doduc	debflu	3,941
doduc	ddeflu	4,397
doduc	paroi	4,436
doduc	iniset	6,061
doduc	deseco	11,719

Program	Routine	Static Cycles
cplex	xload	120
cplex	xaddrow	434
cplex	chpivot	655
cplex	xielem	1,560
fmin	fmin	438
fpppp	fmtgen	601
fpppp	fmtset	704
fpppp	gamgen	835
fpppp	efill	1,200
fpppp	twldrv	15,605
fpppp	fpppp	22,479
matrix300	saxpy	94
matrix300	sgemv	285
matrix300	sgemm	646
rkf45	rkf45	169
rkf45	fehl	505
rkf45	rkfs	1,027
seval	seval	174
seval	spline	1,054
solve	solve	291
solve	decomp	855
svd	svd	2,289
tomcatv	main	2,600
urand	urand	235
zeroin	zeroin	336

Figure 15 The test suite

5.3 Results

We measured optimization time with and without the peephole pass. The total time across all applications for peephole optimization followed by global analysis was 108.8 seconds. Global analysis alone was 116.1 seconds. Time to parse the input and print the output was the same in either case. Using the peephole pass saved 6.3% on total compile times.

We ran the optimizer with and without the global optimizations, to compare the quality of just the peephole optimizer (and the global code motion algorithm) against a stronger global analysis. The results are in Figure 16. The first two columns list the application and procedure name. The next two columns give the cycles spent in the ILOC virtual machine and the percentage of optimized cycles. A higher percentage means more cycles were spent relative to the global optimization cycles. We then show the cycles and percent optimized cycles for just using parse-time optimization and GCM. As a reference point, we also ran a pass of GCF, PRE [18,12], and CCP.

The total improvement for using global optimization instead of just parse-time optimization was -1.9%, and the average improvement was -0.8%. As the numbers show, the peephole optimizer does very well. Global analysis gets only a few percentage points more improvement. Rather suprisingly, this means that most constants and common subexpressions are easy to find.

Program	Routine	Combined, GCM		Parse, GCM		GCF/PRE/CCP		no optimization
doduc	bilan	524,407	100%	536,432	102%	580,878	111%	3,137,306
doduc	cardeb	168,165	100%	168,165	100%	183,520	109%	883,190
cplex	chpivot	4,133,854	100%	4,156,117	101%	4,097,458	99%	10,865,979
doduc	coeray	1,487,040	100%	1,487,040	100%	1,622,160	109%	2,904,112
doduc	colbur	751,074	100%	751,074	100%	763,437	102%	1,875,919
doduc	dcoera	921,068	100%	921,068	100%	981,505	107%	1,814,513
doduc	ddeflu	1,292,434	100%	1,329,989	103%	1,485,024	115%	4,393,525
doduc	debflu	683,665	100%	689,059	101%	898,775	131%	3,614,716
doduc	debico	478,760	100%	479,318	100%	529,876	111%	3,058,222
solve	decomp	627	100%	642	102%	641	102%	1,714
doduc	deseco	2,165,334	100%	2,182,909	101%	2,395,579	111%	11,436,316
doduc	drepvi	680,080	100%	680,080	100%	715,410	105%	2,174,395
doduc	drigl	295,501	100%	295,501	100%	301,516	102%	1,031,658
fpppp	efill	1,927,529	100%	2,046,512	106%	1,951,843	101%	4,166,782
rkf45	fehl	134,640	100%	134,640	100%	135,960	101%	386,496
fmin	fmin	908	100%	908	100%	876	96%	1,707
fpppp	fmtgen	926,059	100%	926,059	100%	973,331	105%	2,390,998
fpppp	fmtset	1,072	100%	1,072	100%	1,074	100%	3,760
fpppp	fpppp	26,871,102	100%	26,871,102	100%	26,883,090	100%	115,738,146
fpppp	gamgen	134,226	100%	134,227	100%	158,644	118%	728,455
doduc	heat	578,646	100%	578,646	100%	613,800	106%	1,393,512
doduc	ihbtr	73,290	100%	75,708	103%	71,616	98%	180,276
doduc	inideb	850	100%	863	102%	898	106%	4,505
doduc	iniset	56,649	100%	56,649	100%	56,684	100%	170,147
doduc	inithx	2,599	100%	2,602	100%	2,776	107%	11,342
doduc	integr	389,004	100%	389,004	100%	426,138	110%	1,717,290
doduc	orgpar	23,692	100%	23,692	100%	24,985	105%	76,980
doduc	paroi	531,875	100%	536,685	101%	637,325	120%	2,554,850
doduc	pastem	589,076	100%	625,336	106%	721,850	123%	2,233,270
doduc	prophy	534,736	100%	553,522	104%	646,101	121%	3,818,096
doduc	repvid	486,141	100%	486,141	100%	507,531	104%	1,983,312
rkf45	rkf45	1,475	100%	1,475	100%	1,575	107%	3,675
rkf45	rkfs	56,024	100%	56,024	100%	66,254	118%	149,051
doduc	saturr	63,426	100%	63,426	100%	59,334	94%	134,664
matrix300	saxpy	13,340,000	100%	13,340,000	100%	13,340,000	100%	47,540,000
matrix300	sgemm	8,664	100%	8,664	100%	8,767	101%	21,320
matrix300	sgemv	496,000	100%	496,000	100%	536,800	108%	1,667,800
doduc	si	9,924,360	100%	9,924,360	100%	9,981,072	101%	20,415,860
seval	spline	882	100%	882	100%	937	106%	3,719
doduc	subb	1,763,280	100%	1,763,280	100%	1,763,280	100%	3,336,840
doduc	supp	2,564,382	100%	2,564,382	100%	2,567,544	100%	4,859,994
svd	svd	4,438	100%	4,596	104%	4,542	102%	12,895
tomcatv	tomcatv	240,688,724	100%	248,494,642	103%	243,582,924	101%	1,287,422,939
fpppp	twldrv	76,120,702	100%	76,291,270	100%	81,100,769	107%	288,657,989
urand	urand	550	100%	550	100%	563	102%	1,243
doduc	x21y21	1,253,980	100%	1,253,980	100%	1,253,980	100%	4,311,762
cplex	xaddrow	16,345,264	100%	16,345,264	100%	16,487,352	101%	39,109,805
cplex	xielem	19,735,495	100%	19,738,262	100%	19,791,979	100%	49,891,205
cplex	xload	3,831,744	100%	3,831,744	100%	3,899,518	102%	10,172,013
doduc	yeh	342,460	100%	342,460	100%	379,567	111%	713,299
zeroin	zeroin	739	100%	740	100%	729	99%	1,441

Figure 16 Procedure vs optimization, cycle counts on the ILOC virtual machine

6. Related Work

As program representations have evolved they have gained an executable model; they include all the information necessary to execute a program. Also, restrictions on the evaluation order, expressed as edges in the graph, are lifted. The later representations are both more compact and more complete and allow more optimizations. Our representation is one more step along these lines. It has an executable model and has fewer restrictions on the evaluation order than any of the previous models. This lifting of restrictions gives analysis and optimization algorithms more freedom to discover facts and reorganize code from disjoint sections of the original program.

Ferrante, Ottenstein and Warren present the *Program Dependence Graph* in [11]. The PDG is more restrictive than our representation in that control edges are added to every node. Also, the PDG lacks the control information required at merges for an execution model. Cartwright and Felleisen extend the PDG to the *Program Representation Graph* adding valve nodes to the PDG [4]. The PRG has a model of execution. Selke's thesis gives a semantic framework for the PDG [20]. The PDG has fewer restrictions than our representation where control edges are required. We would like to extend the combined optimization algorithm so it understands *control dependence*.

Cytron, Ferrante, Rosen, Wegman and Zadeck describe an efficient way to build the SSA form of a program [10] which assigns only once to each variable. The SSA form is a convenient way to express all data dependences, but it also lacks control information at ϕ-functions.

Alpern, Wegman and Zadeck present the *value graph* [1]. The value graph is essentially an SSA form of the program expressed as a directed graph. The authors extend the value graph to handle structured control flow, but do not attempt to represent complete programs this way. The value graph lacks control information at ϕ-functions and therefore does not have a model of execution.

Ballance, Maccabe and Ottenstein present the *Program Dependence Web* [3]. The PDW is a combination of the PDG and SSA form and includes the necessary control information to have a model of execution. It also includes extra control edges that unnecessarily restrict the model of execution. The PDW includes μ and η nodes to support a demand-driven data model. Our representation supports a data-driven model and does not need these nodes. The PDW is complex to build, requiring several phases.

Pingali, Beck, Johnson, Moudgill and Stodghill present the *Dependence Flow Graph* [18]. The DFG is executable and includes the compact SSA representation of data dependences. The DFG has switched data outputs that essentially add the same unnecessary control dependences found in the PDW. The DFG includes anti-dependence edges to manage the store; our representation does not require these. The DFG includes a denotational semantics and has the *one step Church-Rosser* property.

Paul Havlak has done some recent work on the thinned *Gated Single Assignment* form of a program [17]. This form is both executable and compact. Currently, the GSA is limited to reducible programs. The GSA can find congruences amongst DAGs of basic blocks not found using our representation. We can find congruences amongst loop invariant expressions not found using the GSA. It is not clear if the forms can be combined, or if one form is better than another. A promising implementation of thinned GSA exists and has been used on a large suite of FORTRAN applications with good results.

Weise, Crew, Ernst, and Steensgaard present the *Value Dependence Graph* [22]. The VDG is similar in spirit to our representation. It represents an independent line of research, performed in parallel with our own. Weise, et al. stress the VDG for research in partial evaluation, slicing and other optimizations; they have built visual display tools to help transformations on the VDG. We stress compilation speed and code quality; our compiler is much faster and includes a larger set of classic optimizations.

John Field gives a formal treatment of graph rewriting on a representation strongly resembling our representation. We hope to use his semantics with only minimal modifications [14].

7. Conclusions

Our intermediate representation is a compact and lightweight graph containing the essential information for both program execution and optimization. Data dependences are represented using use-def edges, as in an SSA graph, while control dependences become

edges to REGION Nodes. Providing control information at PHI Nodes makes our model compositional which we found helpful for implementing fast local optimizations.

The consistent treatment of control and data edges simplifies the intermediate representation and implementation of the compiler. We take advantage of C++'s inheritance mechanisms and construct a separate class for each different opcode. Opcode-specific information, such as literal fields, is defined in each class. A node's opcode is represented by the C++ virtual function table pointer. Use-def edges are simple pointers to other Nodes and control edges are pointers to REGION Nodes. Virtual functions define opcode semantics providing, for example, a node specific hash function for value numbering.

To produce a faster optimizer, we moved some of the work into the front end. We reasoned that inexpensive peephole optimizations done while parsing would reduce the size of our intermediate representation and the expense of later optimization phases. We feel that this approach was successful, and benefited from the availability of use-def information and the static single assignment property while parsing.

Bibliography

[1] B. Alpern, M. Wegman, and F. Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, 1988.

[2] V. Bala. Private conversation about the KSR register allocator, Dec. 1994.

[3] R. Ballance, A. Maccabe, and K. Ottenstein. The program dependence web: A representation supporting control-, data- and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN '90 Conference on Programming Languages Design and Implementation*, June 1990.

[4] P. Briggs. The Massively Scalar Compiler Project. Unpublished report. Preliminary version available via ftp://cs.rice.edu/public/preston/optimizer/shared.ps. Rice University, July 1994.

[5] P. Briggs and T. Harvey. Iloc '93. Technical report CRPC-TR93323, Rice University, 1993.

[6] R. Cartwright and M. Felleisen. The semantics of program dependence. In *Proceedings of the SIGPLAN '89 Conference on Programming Languages Design and Implementation*, June 1989.

[7] D. Chase, M. Wegman, F. Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Languages Design and Implementation*, June 1990.

[8] C. Click, *Combining Analyses, Combining Optimizations*. Ph.D. thesis, Rice University, 1995. Preliminary version available via ftp://cs.rice.edu/public/cliffc/thesis.ps.gz.

[9] C. Click. Global code motion, global value numbering. Submitted to PLDI '95.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth ACM Symposium on the Principles of Programming Languages*, Jan. 1989.

[11] J. Davidson and C. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, Oct. 1984.

[12] K. Drechsler and M. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, Oct. 1988.

[13] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July, 1987.

[14] J. Field. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 98–107, 1990.

[15] G. Forstyhe, M. Malcom, and C. Moler. *Computer Methods for Mathematical Computations.* Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[16] D. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software Practice and Experience*, 20(1):5–12, Jan. 1990.

[17] P. Havlak, *Interprocedural Symbolic Analysis*. Ph.D. thesis, Rice University, 1994.

[18] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, Feb. 1979.

[19] R. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. Technical Report TR-90–1152, Cornell University, 1990.

[20] B. Rosen., M. Wegman, and F. Zadeck, Global Value Numbers and Redundant Computations. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, Jan. 1988.

[21] R. Selke, *A Semantic Framework for Program Dependence*. Ph.D. thesis, Rice University, 1992.

[22] D. Weise, R. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: Representation without taxation. In *Proceedings of the 21st ACM SIGPLAN Symposium on the Principles of Programming Languages*, 1994.

[23] M. Wegman and F. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181-210, April 1991.