From Quads to Graphs: An Intermediate Representation's Journey

Cliff Click

CRPC-TR93366-S
October 1993

Center for Research on Parallel Computing
Rice University
P.O. Box 1892
Houston, TX 77251-1892

From Quads to Graphs: An Intermediate Representation's Journey

CLIFF CLICK
Rice University

We follow the slow mutation of an intermediate representation from a well known quad-based form to a graph-based form. The final form is similar to an operator-level *Program Dependence Graph* or *Gated Single Assignment* form. The final form contains all the information required to execute the program. What is more important, the graph form *explicitly* contains *use-def* information. Analyses can use this information directly without having to calculate it. Transformations modify the use-def information directly, without requiring separate steps to modify the intermediate representation and the use-def information.

Keywords: Intermediate representations, optimizing compilers, constant propagation

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 change. Our goal is a representation that is simple and light weight while allowing easy expression of fast optimizations.

This article chronicles the slow mutation of an intermediate representation from a well known quad-based form to a graph-based form. The final form is similar to (although not exactly equal to) an operator-level *Program Dependence Graph* or *Gated Single Assignment* form.[1, 2, 3, 4] The final form contains all the information required to execute the program. What is more important, the graph form explicitly contains *use-def* information. Analyses can use this information directly without having to calculate it. Transformations modify the use-def information directly without requiring separate steps to modify the intermediate representation and the use-def information. The graph form is a simple single-tiered structure instead of a two-tiered *Control-Flow Graph* containing basic blocks (tier 1) of instructions (tier 2). This single tier is reflected in our algorithms.

One of our philosophies for making a compiler run fast is to do as much of the work as possible as early in the compile as possible. This leads us to attempt strong peephole optimizations in a one-pass front end. Our final representation allows peephole optimizations that, under certain circumstances, are as strong as *pessimistic* conditional constant propagation.

This paper is **not** about building a complete optimizer, parser, or a compiler front-end. We assume that the reader can write a compiler front-end. This paper **is** about the design decisions that led us from a more traditional intermediate representation to the current graph-based representation. This transition is a continuous and on-going process.

Throughout this exposition, we use C++ to describe the data structures and code; a working knowledge of C++ will aid in understanding this paper. The later sections contain bits of C++ code sliced directly out of our compiler (with some reformatting to save space).

This work has been supported by ARPA through ONR grant N00014-91-J-1989.

1.1 Pessimistic vs Optimistic

We divide analyses (and transformations) into two broad categories. Pessimistic analyses assume the worst (conservatively correct) and try to prove better. Optimistic analyses assume the best and try to prove their assumptions. The two techniques can produce the same results *except at loops*. Optimistic analyses assume that the not-yet-proven facts coming in from a loop's back edge hold; these assumptions might be enough to allow their proof. Pessimistic techniques cannot make any assumptions around a loop's back edge. They only use what they already *know* to be true. This might not be enough to allow the proof of a fact they do not already have.

During constant propagation, for instance, an optimistic analysis assumes that values flowing around a loop's back edge are exactly equal to constants flowing into the loop. If this assumption is proven correct, a constant is found. If the assumption is disproven, then no harm is done; the loop body need only be re-analyzed with the more conservative (but correct) information.

With a pessimistic constant propagator, values flowing around the back edge of a loop are assumed to not be constants. When these non-constants are merged with values flowing into the loop, the analyzer can only determine that non-constant values are in the loop. This matches the analyzer's assumption that non-constant values are flowing around the back edge. Thus a pessimistic analyzer cannot find constants that flow around a loop.

Without loops, however, both kinds of analysis can view the code with all the facts present. Thus both kinds of analysis can find an equivalent set of facts. To do this efficiently the analyses need to visit the code in topological order; information about a particular value must be gathered before that value is used. If the code is visited out of order some of the analysis has to proceed without all the relevant facts. We see that the pessimistic analysis can usefully proceed in a *one-pass* algorithm[1] because, in the absence of knowledge, conservative assumptions are made. An optimistic analyzer requires that we re-visit the out-of-order code to validate the facts we assumed there.

1.2 Optimizations in the Front End

Since we can do pessimistic analysis in a one-pass algorithm we can do it *while we are parsing*. As the front end parses expressions, the analyzer assigns the expression a conservative value and tries to prove a better result based on previously parsed expressions. As long as the parser reads the code in topological order, the pessimistic analysis is as good as any optimistic analysis. We observe that the parser visits code built with `if/then/else` structures in topological order. At loop heads or with unstructured code, our pessimistic analysis makes conservatively correct assumptions.

Our pessimistic analysis requires only *use-def*[2] information, which we gather as we parse the code. The compiler looks at (and changes) a fixed-size region of the intermediate representation. Code outside the region is not effected by the transformation; code inside the region is transformed without knowledge of what is outside the region. This analysis resembles a strong form of constant folding or peephole optimization.

By doing these transformations during the parse phase we lower our peak size and remove work from later optimization phases.[3] Peak size is reduced because some transformations replace new instructions with existing ones; storage for the new instructions is reused during parsing. Later optimizations have less work to do because we have done some work up front.

1.3 Overview

In Section 2, we start with a traditional CFG with basic blocks of instructions. In Section 3, we move this representation into *Static Single Assignment* (SSA) form with explicit use-def information. In Section 4, we make control dependencies an explicit part of our representation. In Section 5, we use C++'s inheritance to give more structure to our instructions and to speed up their creation and deletion. In Section 6, we

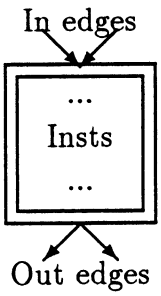
[1] A one-pass algorithm touches (references, changes) each element of its input once.
[2] Values are used and defined at program points. Use-def chains are links at a value's use site to the value's definition sites.
[3] Preliminary experimentation shows peak size is almost cut in half, with a corresponding decrease in time spent optimizing.

drop the CFG altogether, replacing it with instructions that deal with control. In Section 7, we explore the power of our peephole optimizations. In Section 8, we look at how we handle a variety of problematic instruction features, including subroutines, memory, and I/O. In Section 9, we look at the pro's and con's of removing all control information from data computations. In Section 10, we look at the more powerful optimistic transformations on the final form of the intermediate representation.

As we mutate the intermediate representation, we also mutate our pessimistic transformation technique to match. In the end the pessimistic transformation is simple, fast, and powerful. It loses information relative to the same optimistic analysis only at loops and unstructured code.

2 In the Beginning...

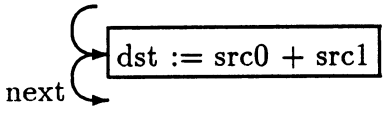
...there is the *Control-Flow Graph*. The CFG is a directed graph whose nodes are basic blocks and whose edges denote the flow of control. An implementation of basic blocks is given in Figure 1 where the basic block is shown as a double-hulled box. The CFG contains a unique Start block with no inputs and a unique Stop block with no outputs. Each basic block contains an ordered list of instructions. Each instruction is a *quad*: an operation code (*opcode*), a destination variable, and two source variables. While two source variables are common, actual instructions may have zero to three sources. An implementation of quads is described in Figure 2 and is displayed in a single-hulled box.




```
class Block {
  //Sequence of instructions in the basic block
  class Inst *first;
  //Set of output edges
  class Edge_Set *out;
  //Set of input edges
  class Edge_Set *in;
};
```

Figure 1 Class definition for a Basic Block

In the quad's concrete representation, the opcode typically takes a small number of bytes, generally between 1 and 4. Each variable is renamed to a fairly dense set of integers and a machine integer is used to represent the variable. The integer is an index into a symbol table that is not shown. The source variables and opcode, taken together, are called an *expression*. The quad may also contain a pointer to the next quad in the basic block.




```
class Inst {
  //Enumeration of all possible opcodes
  enum Opcode { Add, Copy, If, Return };
  Opcode op;          //The opcode specific to this instruction
  int dst;            //Destination variable
  int src0, src1;     //Source (input) variables
  Inst *next;         //Next instruction in basic block
};
```

Figure 2 Class definition for an Instruction

Loosely stated, the semantics of a basic block are as follows:

- Each instruction is executed in order.

- The execution of an instruction consists of
  - reading all source variables,
  - performing the primitive function specified by the opcode on those sources, and
  - assigning the result back into the destination variable.
- The last instruction of a normal basic block, instead of writing to a destination variable, may read the *conditional code* register and decide which of several following basic blocks is to be executed.
- The last instruction of the Stop block must contain the RETURN opcode. A RETURN instruction returns its inputs as the program's answer.
- The first instruction of the Start block can use sources specified from outside the program; these are the program's inputs.

Other extensions to the intermediate representation will be made as needed. In particular, subroutine calls, memory (LOAD,STORE), and I/O issues will be dealt with further along.

2.1 Pessimistic Transformations on Quads

This quad representation lacks any use-def information, i.e., given an instruction that uses a variable, what instruction assigned the value being used now? Prior to using use-def information, peephole transformations relied on a fixed-size window of instructions to be inspected and transformed. Fixed-size window techniques are very weak because they rely on the instruction order to keep variable uses close to variable definitions. We give examples using this straw-man for purposes of presentation only. All modern techniques rely on use-def information, either local (within a basic block) or global (within a procedure)[6].

We can integrate parsing (and quad generation) with pessimistic optimizations. As the program is parsed, the front end generates basic blocks and instructions. Each time an instruction is generated the front end immediately inspects the instruction in the context of previous instructions in this block. If any peephole optimizations apply, we transform the instructions in the context window. Figure 3 shows the code for dropping redundant copies and replacing the add of a zero with a copy. Use-def information allows us to use the same technique on instructions that are not adjacent sequentially.

2.2 Use-Def Information

Clearly use-def information would greatly improve our peephole optimizations. Use-def information is useful for many optimizations. Storing it in the quad along with the source variable names is redundant. Just the source variable name is nearly enough information to generate use-def chains. We need only hunt backwards in the quads to find the defining instruction. However, hunting backwards can be slow (linear in the size of the program). Worse, hunting backwards is ambiguous; several definitions can reach the same use. To correct this problem we need to put our program in *Static Single Assignment* form.

3 Static Single Assignment

Putting the program in SSA form eliminates the ambiguous reaching definitions.[7] In a normal program the same variable can be assigned several times, sometimes along different control paths (basic blocks). Converting to SSA form solves this problem by inserting ϕ-functions[4] at the head of certain basic blocks then renaming all variables. The ϕ-functions are treated like normal instructions; the opcode distinguishes the ϕ from some other function. Some sample code in SSA form is given in Figure 4.

After the renaming step, every variable is assigned exactly once.[5] Since expressions only occur on the right-hand side of assignments, every expression is associated with a variable. There is a one-to-one

[4] ϕ-functions merge the results of several assignments to the same variable.
[5] This is the property for which the term *static single assignment* is coined.

```
parse()
{
  int dst, src0, src1;
  //No previous instruction in context window
  Inst *prev = NULL;
  //parse and compute quad information
  ...;
  //Make a new quad from parsed information
  Inst *quad = new Inst(opcode, dst, src0, src1);
  //Peephole optimize with previous instruction
  quad = peephole(quad,prev);
  //If instruction not removed
  if( quad ) {
    //Insert instruction into block
    prev→next = quad;
    prev = quad;
  }
  ...;
}
```

```
Inst *peepquad( Inst *quad, Inst *prev )
{
  //Replace "z=0; y=x+z" with "z=0; y=x"
  if( (quad→op == Add) &&
      (quad→src1 = prev→dst) &&
      (prev→op == Zero) ) {
    quad→op = Copy;
    quad→src0 = prev→dst;
  }
  //Drop copies of self
  if( (quad→op == Copy) &&
      (quad→dst == quad→src0) ) {
    delete quad;
    return NULL;
  }
  return quad;
}
```

Figure 3 Peephole optimizations while parsing

Normal	SSA
int $x \leftarrow 1$;	int $x_0 \leftarrow 1$;
do {	do { $x_1 \leftarrow \phi(x_0, x_3)$;
if($x \neq 1$)	if($x_1 \neq 1$)
$x \leftarrow 2$;	$x_2 \leftarrow 2$;
	$x_3 \leftarrow \phi(x_1, x_2)$;
} while(pred())	} while(pred())
return(x);	return(x_3);

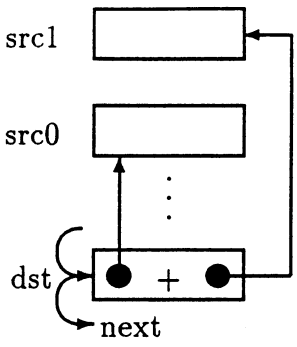
Figure 4 Sample code in SSA form

correlation between variables and expressions. Therefore, the variable name can be used as a direct map to the expression that defines it. In our implementation, we want this mapping to be fast.

In the instruction's concrete representation there are fields that hold the source variables' names (as machine integers). To speed up the variable-to-definition mapping, we replace the variable name with a pointer to the variable's defining instruction (actually a pointer to the instruction's concrete representation as a data structure). Performing the mapping from variable name to defining instruction now requires a single pointer lookup. In this manner, use-def information is explicitly encoded. Figure 5 describes the new instruction format.

We now have an abstract mapping from variables and expressions to instructions and back. Instructions no longer need to encode the variable name being defined (they use the abstract mapping instead) so the `dst` field can be removed. However, the front end still needs to map from variable names to instructions while parsing. The variable names are mapped to a dense set of integer indices. We use a simple array to map the integer indices to instructions. We update the mapping after any peephole optimizations.[6]

[6]The astute reader may notice that if we always insert instructions into the basic block's linked list and the peephole optimizer returns a previous instruction, we will attempt to put the same instruction on the linked list twice, corrupting the list. We correct this in Section 4.




```
class Inst {
  //Enumeration of all possible opcodes
  enum Opcode { Add, Copy, If, Return, Phi };
  //The opcode specific to this instruction
  Opcode op;
  //Source (input) variables
  Inst *src0, *src1;
  //Next instruction in basic block
  Inst *next;
};
```

Figure 5 Instruction definition with direct mapped inputs

Figure 6 shows the new parser interface. Since we are using use-def information instead of a context window, we no longer need a **prev** variable to hold the context window.

```
parse()
{
  int dst, src0, src1;                    //The instruction variables
  Inst **map;                             //A map from variable to instruction
  ...;                                    //parse and compute quad information
  Inst *i0 = map[src0];                   //Map variable names to instructions
  Inst *i1 = map[src1];
  Inst *quad = new Inst(opcode, i0, i1);  //Make a new quad from parsed information
  quad = peephole(quad);                  //Peephole optimize with use-def information
  map[dst] = quad;                        //Update map for new instruction
}
```

Figure 6 Parser interface for SSA programs

3.1 Building SSA Form

We use a conservative method to build SSA form. Our method does not require analyzing the entire program, which is not available while we are still parsing the program. Each definition of a variable is renamed to be the address of the definition's instruction. Each original variable has been mapped to an integer index. We use an array lookup on the index to find the most recent definition (and SSA name). When we find a new definition for an existing variable we update the mapping array.

At the start of each basic block we insert ϕ-functions whenever we *might* merge two definitions of the same variable. Inside nested **if/then/else** structures we will have parsed all paths leading to the merge point. It is a simple matter to inspect all variables for changing definitions and insert ϕ-functions as required. At loop heads or at labels we must assume all variables are defined on the paths we have not yet parsed. This requires us to insert a number of redundant ϕ-functions. While this algorithm is clearly $O(n^2)$, in practice it is quite fast.

3.2 Pessimistic Optimizations with Use-Def Information

Use-def information is now explicitly coded in the intermediate representation. With this information, we can analyze related instructions regardless of their sequential order. Our analysis is much stronger (and simpler). We give code for doing some peephole optimizations using the use-def information in Figure 7.

Instead of returning a **NULL** when an instruction is unused we always return some replacement instruction. The replacement instruction can be a previously defined one. Using a previously defined instruction instead of making a new one shrinks our code size. There are many possible optimizations; we give code for only these few:

Removing copies: We use the original value instead of the copy, making the copy instruction dead.

Adding two constants: In general, anytime the inputs to a primitive are all constants, the primitive can be evaluated at compile time to yield a constant result.[7]

Adding a zero: Several primitives have an identity element that converts the primitive into a copy.

Value-numbering: This finds instances of the same expression and replaces them with copies. The method used is a hash-table lookup where the key is computed from the inputs and the opcode. Because we do not have any control information explicitly encoded in the instructions, we may find two equivalent expressions where neither instruction dominates the other. In this case, replacing either instruction with a copy is incorrect. To fix this we require the hash-table be flushed at the end of each basic block. The equivalent instructions we find are restricted to the same basic block (i.e., *local* value-numbering).

Subtracting equal inputs: Subtracting (comparing) the results of the same instruction is always zero (equal). Similarly merging equal inputs always yields that input.

3.3 Progress

We clearly have made some progress. We have dropped a field from our instruction format (the **dst** field), collected use-def information for later passes, and strengthened, sped up, and clarified our peephole optimization. However, we can do better. We still have a fixed order for instructions within a basic block, represented by the **next** field. Yet when a basic block is executed, all instructions in the block are executed. For superscalar or dataflow machines the instructions should be allowed to execute in any order, as long as their input dependencies are met. To correct this we need to think about how instructions are sequenced.

4 Control-Flow Dependence

In the representation described so far, basic blocks contain an ordered list of instructions. In some sense, this represents *def-use* control information. The basic block "defines" control when the block is executed, and the first instruction "uses" that control, then "defines" it for the next instruction in the block. This represents the normal serial execution of instructions in a block. However, when the block is executed, *all* instructions in the block are executed. Control should be defined for all instructions simultaneously without an order dependence. Removing the serializing control dependence allows instructions in the block to be executed out of order, as long as their other data dependences are satisfied. In addition, we want to be consistent with our dependence representation and explicitly have use-def information instead of def-use information.

The ordered list of instructions in our basic blocks is implemented as a linked list. Each instruction contains a pointer to the next instruction. We replace this **next** pointer with a pointer to the basic block structure itself. We treat this pointer like the source of another input to the instruction: the *control* source. At this point each instruction has zero to three data inputs and one control input. Figure 8 describes the new instructions. In the example we only show one basic block, but the data inputs could be located in any basic block.

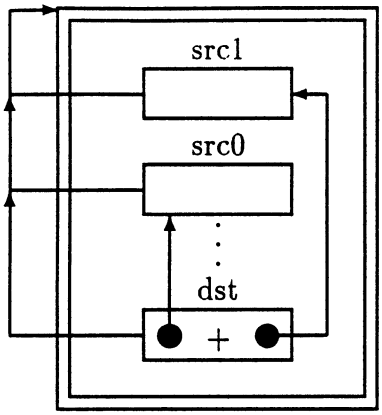
[7] For now, we store constants in the **src0** input and the **src1** input is ignored. We correct this in Section 5.

```
Inst *peephole( Inst *quad )
{
  Inst *src0 = quad→src0;                    //Dismember the quad for easy access
  Inst *src1 = quad→src1;
  //Do value numbering via a hash table lookup
  Inst *v_n = hash_lookup( hash( quad→op, src0, src1 ) );
  if( v_n ) {                                //Hit in the hash table?
    delete quad;                             //Yes: toss this quad and
    return v_n;                              //return previous quad in same block
  }
  //Check for peephole optimizations specific to an opcode
  switch( quad→op ) {
  case Copy:                                 //Copy opcode?
    delete quad;                             //Always remove Copy quads
    quad = src0;                             //and replace them with their copied input
    break;

  case Add:                                  //Add opcode?
    if( src0→op == Constant ) {              //Left input is a constant?
      int con0 = (int)(src0→src0);           //Break out constant for easy access
      if( src1→op == Constant ) {            //Right input is also a constant?
        delete quad;                         //Replace quad with a constant
        quad = new Inst(Constant, con0 + (int)(src1→src0) );
      }
      if( con0 == 0 ) {                      //Adding a zero?
        delete quad;                         //Same as a Copy of the right input
        quad = src1;
      }
    } else if( src1→op == Constant ) {       //Right input is a constant?
      if( (int)(src1→src0) == 0 ) {          //Adding a zero on the right?
        delete quad;                         //Same as a Copy of the left input
        quad = src0;
      }
    }
    break;

  case Subtract:                             //Subtract opcode?
    if( src0 == src1 ) {                     //Subtracting equivalents?
      delete quad;                           //Then replace with a constant zero
      quad = new Inst( Constant, 0 );
    }
    break;
  }
  return quad;                               //No more optimizations
}
```

Figure 7 Peephole optimizations for SSA programs




```
class Inst {
  //Enumeration of all possible opcodes
  enum Opcode { Add, Copy, If, Return, Phi };
  //The opcode specific to this instruction
  Opcode op;
  //Source (input) variables
  Inst *src0, *src1;
  //Basic block that controls this instruction
  Block *control;
};
```

Figure 8 Instruction definition with explicit control

4.1 More on ϕ-Functions

We still need the edges in the CFG to determine which values are merged in the ϕ-functions. Without those CFG edges our intermediate representation is not *compositional.*[8] [8, 9] We need to associate with each data input to a ϕ-instruction the control input from the corresponding basic block. Doing this directly means that ϕ-instructions would have a set of pairs as inputs. One element of the pair would be the data dependence and the other control dependence. This is a rather ungainly structure with complicated semantics. Instead, we borrow some ideas from Ballance, et al. and Field.[2, 10]

We split the ϕ-instruction into a set of SELECT instructions and a COMPOSE instruction, each with simpler semantics. The SELECT function takes two inputs: a data input and a control input. The result it computes depends on the control. If the control is not executed (i.e., the source basic block is not executed), no value is produced. Otherwise the data value is passed through directly. The COMPOSE instruction takes all the results from the SELECT instructions as inputs. The COMPOSE instruction passes through the data value from the one SELECT instruction that produces a value.[9] See Figure 9 for an example.

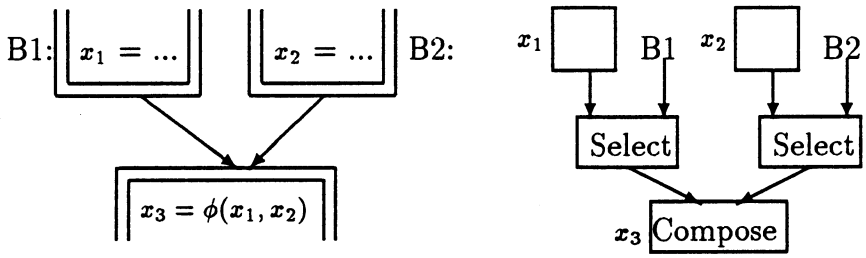


Figure 9 Select/Compose for merged data

Note that these instructions have no run-time operation. They do not correspond to a machine instruction. They exist to help the compiler understand the program semantics. When machine code is finally generated, the SELECT/COMPOSE sequences are folded back into normal basic blocks and CFG behavior.

[8] In essence, we would require information not local to an instruction. A non-compositional representation is difficult to transform correctly; changing an instruction may require information not directly associated with the instruction.
[9] Because control can reach the COMPOSE through only one prior basic block at a time.

4.2 Cleanup

At this point our instructions have use-def information for both data and control. We also have enough information to keep our representation compositional (and thus easy to modify, for example, a dead path into a merge point). However, our simple **Inst** class is having difficulty gracefully abstracting a variety of different instructions. We look at this and some other software engineering issues in the next section.

5 Engineering Concerns

At this point we notice that we have several different kinds of instructions, each with different numbers of inputs. COMPOSE instructions might have any number of inputs, a NEGATE instruction has only one, and a CONSTANT instruction (which defines a simple integer constant) needs to hold the value of the constant being defined and has no other inputs. To handle all these differences we break up the definition of an instruction into separate classes, all inheriting from the base class **Inst**. Figure 10 illustrates the new base class and some inherited classes.

We are using a functional programming style. All our objects are created and initialized but never modified. To get compiler support for this programming style, we insert the appropriate **const** qualifiers in the class definitions.

```
class Inst {
  //Enumeration of all possible opcodes
  enum Opcode { Add, Copy, If, Return, Select, Compose, Constant };
  const Opcode op;                          //The opcode specific to this instruction
  Block *const control;                     //Basic block that controls this instruction
  Inst( Opcode opnew, Block *c ) : op(opnew), control(c) {}
};

class ConstInst : public Inst {
  const int constant;                       //The specific constant
  ConstInst( int con, Block *c ) : constant(con), Inst( Constant, c ) {}
};

class CopyInst : public Inst {
  Inst *const src;                          //Source (input) variable
  CopyInst( Inst *src, Block *c ) : src(src), Inst( Copy, c ) {}
};

class AddInst : public Inst {
  Inst *const src0, *const src1;            //Source (input) variables
  AddInst( Inst *src0, Inst *src1, Block *c ) : src0(src0), src1(src1), Inst( Add, c ) {}
};

class ComposeInst : public Inst {
  const int max;                            //Number of inputs
  SelectInst **const srcs;                  //Array of input pointers
  ComposeInst( int max, Block *c ) : max(max), srcs(new Inst*[max]), Inst( Compose, c ) {}
};
```

Figure 10 Definitions for the new **Inst** classes

5.1 Virtual Optimizations

In the peephole function in Figure 7, our C++ code does a **switch** on a field that is unique to each object class. In a complete implementation, the **switch** statement would get quite large. Additionally, the semantics for a single opcode get broken into separate sections; one code section for the class definition and another for the peephole optimizations. We prefer to keep all of an opcode's semantics in one place: the class member functions. In Figure 11, we break up the peephole function into separate **virtual** functions for each opcode.

To make the hash table lookup work, we must be able to hash and compare instructions. Differently classed instructions have different hash functions and different compare semantics. For example: Add is commutative; two Add instructions are equal if their inputs match in any order. Code for virtual hash and compare functions is given in Figure 12.

5.2 Faster Malloc

Each time we make a new instruction we are calling the default **operator new** to get storage space. This in turn calls **malloc** and can be fairly time consuming. In addition, the peephole 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 **Inst**. Our replacement operators use an *arena* [11]. Arenas hold heap-allocated objects with similar lifetimes. When the lifetime ends, the arena is deleted freeing all the contained objects in a fast operation. The code is given in Figure 13.

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[10] 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 a 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 the deleted memory was recently allocated. If it was, the **delete** code pushes back the high water marker, reclaiming the space for the next allocation.

5.3 Control Flow Issues

With these changes the overall design of our intermediate representation becomes clear. Each instruction is a self-contained C++ object. The object contains all the information required to determine how the instruction interacts with the program around it. The major field in an instruction is the opcode. An opcode's class determines how instructions propagate constants, handle algebraic identities, and find congruences with other instructions. To make the intermediate representation understand a new kind of operation we need to define a new opcode and class. The class requires data fields for instruction's inputs, and functions for peephole optimization and value-numbering. We do not need to make any changes to the peephole or value-numbering code itself. Finally, we made instruction creation and deletion very fast.

So far, we have avoided many issues concerning basic blocks and CFGs. Our focus has been on the instructions within a basic block. Yet control flow is an integral part of any program representation. CFGs have a two-layered structure. This two-layered structure is reflected in all our analysis algorithms: we run some algorithm over the blocks of a CFG and then we do something to each block. In the next section we look at how we can remove this distinction (and simplify our algorithms).

6 Two Tiers to One

Our representation has 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.[11] CFGs deal with control flow and basic blocks deal with data flow. We want to handle both kinds

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

[11] The block/instruction dichotomy was adopted to save time and space when doing bit-vector dataflow analysis. We want to do *sparse* analysis because it is asymptotically faster.

```
class Inst {
  ...;                                          //Previous definition
  Inst *peephole();                             //Peephole opts common to all Insts
  virtual Inst *vpeephole() { return this; }    //Default action: no optimization
  virtual int hash() const = 0;                 //All derived classes must define
  virtual int operator == ( Inst const * ) const = 0;
};

Inst *Inst::peephole()                          //Common optimization is value-numbering
{
  Inst *v_n = hash_lookup( );                   //Hash on object-specific key
  if( v_n ) { delete this; return v_n; }        //Return any hit
  return vpeephole();                           //Otherwise do class-specific optimizations
}

class CopyInst : public Inst {
  ...;                                          //Previous definition
  Inst *vpeephole();                            //Class specific optimizations
  int hash() const { return (int)src+op; }      //Hash function depends on source and opcode
  int operator ==( Inst const * ) const;        //Equality test on objects
};

Inst *CopyInst::vpeephole()                     //Copy instructions are always thrown away
{ Inst *oldsrc = src;                           //Save copied value
  delete this;                                  //Delete self
  return oldsrc;                                //Return copied value
}

class AddInst : public Inst {
  ...;                                          //Previous definition
  Inst *vpeephole();                            //Class specific optimizations
  int hash() const { return (int)src0 + (int)src1 + op; }
  int operator ==( Inst const * ) const;        //Equality test on objects
};

Inst *AddInst::vpeephole()
{
  if( src0->op == Constant ) {                  //Left input is a constant?
    ...;                                        //Same code as before
  } else if( src1->op == Constant ) {           //Right input is a constant?
    if( (ConstInst*)src1->con == 0 ) {          //Adding a zero?
      Inst *oldsrc = src0;
      delete this;                              //Delete self
      return oldsrc;                            //Return the "x" from "x+0"
    }
  }
  return this;                                  //Otherwise, no optimization
}
```

Figure 11 Virtual peephole functions

```
int CopyInst::operator == ( Inst const *x ) const
{
  if( x→op ≠ Copy ) return 0;              //Equal if same class of object
  return src == ((CopyInst*)x)→src;        //and same input
}

int AddInst::operator == ( Inst const *x ) const
{
  if( x→op ≠ Add ) return 0;               //Equal if same kind of object and
  AddInst const *a = (AddInst*)x;          //inputs match in either order (commutative)
  return (((src0 == a→src0) && (src1 == a→src1)) ||
          ((src1 == a→src0) && (src0 == a→src1)));
}
```

Figure 12 Hash table support functions

```
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 Inst {
  static Arena *arena;                     //Arena to store instructions 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 Inst of given size
  { if( hwm+x > max ) Inst::grow(); old = hwm; hwm+=x; return old; }
  void operator delete( void *ptr )        //Delete an Inst
  { if( ptr == old ) hwm = old; }          //Check for deleting recently allocated space
  ...;                                     //Other member functions
};

Arena *Inst::arena = NULL;                 //No initial Arena
char *Inst::hwm = NULL;                    //First allocation attempt fails
char *Inst::max = NULL;                    //... and makes initial Arena

void Inst::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 13 Fast allocation with arenas

of dependences with the same mechanism. So we remove this distinction to simplify our representation.

Let us start with the instructions. Abstractly, consider each instruction to be a node in a graph. Each input to the instruction represents an edge from the defining instruction's node to this instruction's node (i.e., def→use edges). The edge direction is backwards from the pointers in the instruction (i.e., use→def edges). This is not a contradiction; we are defining an abstract graph. As shown in Figure 14, the concrete implementation of this graph allows convenient traversal of an edge from sink to source (use to def) instead of from source to sink (def to use).

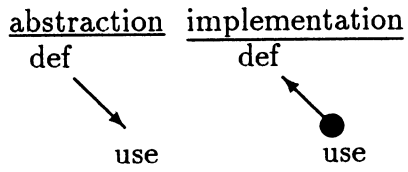
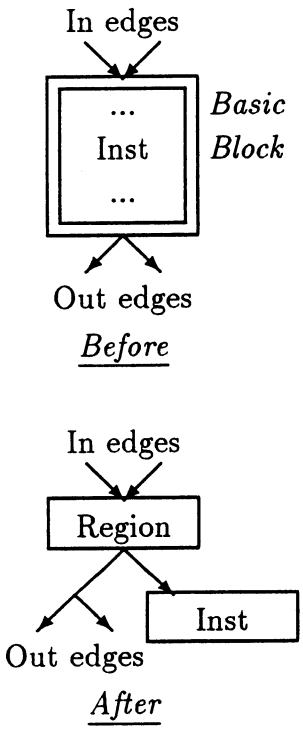


Figure 14 The implementation of dependence edges

Every instruction takes a control input from the basic block that determines when the instruction is executed. If the input is an edge in our abstract graph, then the basic block must be a node in the abstract graph. So we define a REGION instruction [3] to replace a basic block. A REGION instruction takes control from each predecessor block as inputs and produces a merged control as an output.




```
class Inst {
  enum Opcode { Negate, Add, Copy, If, Return, Select,
                Compose, Constant, Region };
  const Opcode op;
  Inst( Opcode opnew ) : op(opnew) {}
  ...;                                    //Other member functions
};

class AddInst : public Inst {
  Inst *const control;                    //Controlling instruction
  Inst *const src0, *const src1;          //Source (input) variables
  AddInst( Inst *src0, Inst *src1, Inst *block )
  : src0(src0), src1(src1), control(block), Inst( Add ) {}
  ...;                                    //Other member functions
};

class RegionInst : public Inst {
  const int max;                          //Number of control inputs
  Inst **const srcs;                      //Array of control inputs
  RegionInst( int max ) : max(max), srcs(new Inst*[max]), Inst( Region ) {}
  ...;                                    //Other member functions
};
```

Figure 15 Explicit *control* dependence

Since REGION instructions merge control, they do not need a separate control input to determine when they execute. So the control input field is moved from the **Inst** definition to the class-specific area for each inherited class. Figure 15 shows the change in the base class and some inherited classes.

If the basic block ends in a conditional instruction, that instruction is replaced with an IF instruction.

Figure 16 shows how an IF instruction works. In the original representation, the predicate sets the condition codes (the variable named *cc*) and the *branch* sends control to either block **B1** or block **B2**. With the explicit control edges, the IF instruction takes a control input and a predicate input. If the predicate is $\mathcal{T}rue$, the IF instruction sends control to the true basic block's REGION instruction. Otherwise control is sent to the false basic block's REGION instruction.

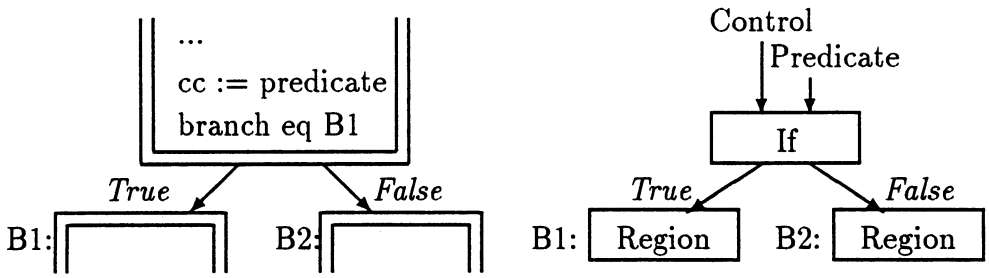


Figure 16 An example IF construct

6.1 A Model of Execution

Having lost basic blocks and the CFG, what is our model of execution? We take our cues from the design history of the intermediate representation. Like the execution model for quads, our 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 approximation each opcode 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. The control token is restricted to existing in REGION instructions, IF instructions, and the START instruction. The Start basic block is replaced with a START instruction that produces the initial control token. Each time execution advances, the control token leaves the instruction it currently is in. The token moves onward, following the outgoing edge(s) to the next REGION or IF instruction. If the token reaches the STOP instruction, execution halts. Because we constructed the graph from a CFG, we are assured only one suitable target instruction (REGION, IF, STOP) exists on all the current instruction's outgoing edges.

The **data** subgraph does not use token-based semantics. Data nodes' outputs are an immediate reflection of their inputs and function (opcode). 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 an instruction demands a value from a data instruction, it follows the use-def edge to the data instruction, 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 instruction. We never build a graph with a loop of only data-producing nodes; every loop has either COMPOSE or REGION instructions.

Our two subgraphs intermix at two distinct instruction types: COMPOSE/SELECT instructions and IF instructions. The COMPOSE/SELECT combination reads in both data and control, and outputs a data value. The control token is not consumed by a SELECT. Instead the SELECT checks for the presence of the control token. The output of the SELECT is either a copy of the data input, or a special "no-value" ($\top$ in a constant-propagation lattice) depending on the presence or absence of the control token. COMPOSE instructions output either their previous value, or the one data value present.

IF instructions take in both a data value and the control token, and hold onto either a $\mathcal{T}rue$ or a $\mathcal{F}alse$ control token. This token is available to only half of the IF instruction's users, and eventually exits to only one of two possible successors. In Section 8.1 we modify the IF instruction so that it behaves more like other control handling instructions: we give it exactly two successors, only one of which receives the control token.

Figure 17 shows what a simple loop looks like. Instead of a basic block heading the loop, there is a REGION instruction. The REGION instruction merges control from outside the loop with control from the loop-back edge. There is a COMPOSE instruction (and matching SELECT instructions) merging data values from outside the loop with data values around the loop. The loop ends with an IF instruction 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.

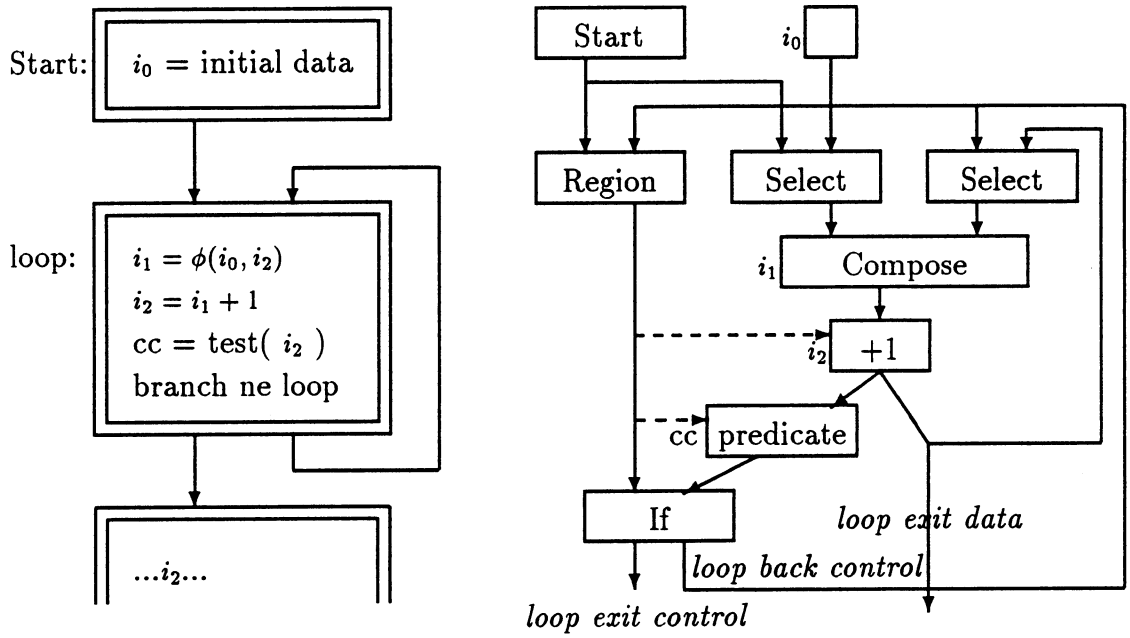


Figure 17 An example loop

6.2 Control-Based Optimizations

Having control as an explicit input means we can use control in our optimizations. An IF instruction with a constant test produces control out only one of two edges. The "live" control edge is a direct copy of the IF's input control. For the "dead" control edge, we use a special value, **NULL**, as the instruction at the edge's head. The code to do this is presented much later, in Figure 24, after we discuss how to handle the IF's two different outputs.

In the peephole code for ADD instructions, we can check to see if the code we are parsing is dead and delete it immediately. This can happen, for instance, when debugging code is compiled out, or for library calls where the programmer passes in constants for various option flags. Example code is presented in Figure 18.

Note that we are returning **NULL** as the defining instruction for unreachable code. This means any attempt at optimizing unreachable code uses the **NULL** and fails. This is intuitively correct: unreachable code never executes, so we should never try to use such code. For brevity's sake we skip the test for missing control inputs in future examples. SELECT instructions that use these **NULL** data values also have **NULL** control inputs and are therefore unreachable also. Only COMPOSE instructions need to test for **NULL** inputs (from SELECT instructions); such inputs represent dead paths, and the input can be removed from the COMPOSE.

REGION and COMPOSE instructions can be optimized in a similar fashion. **NULL** (dead) inputs can be removed. REGION and COMPOSE instructions with only one input are just copies of that input, and can be removed. Doing these optimizations during parsing requires the front end to know when no more

```
//Sample parse-time dead code
//Not debugging
const int Debug = 0;
...
//Compile-time constant test
if( Debug ) {
  //Debug code
  ...;
}
```

```
Inst *AddInst::vpeephole()
{
  if( !control ) {          //Instruction is unreachable?
    delete this;            //Delete unreachable instruction
    return NULL;            //NO instruction generates this value
  }
  ...;                      //Same code as before
  return this;              //Return instruction replacing this one
}
```

Figure 18 Parse-time unreachable code elimination

control paths can reach a particular merge point.[12] For merge points created with structured code (i.e., an if/then/else construct), all the control paths reaching the merge point are known. After parsing all the paths to the merge point, REGION and COMPOSE instructions can be optimized. For merge points created with labels, the front-end cannot optimize the merge point until the label goes out of scope. In general, this includes the entire procedure body.

6.3 Value Numbering and Control

If we encode the control input into our value numbering's hash and key-compare functions we no longer match two identical instructions in two different basic blocks. This means we no longer need to flush our hash table between basic blocks. However, we are still doing only local value numbering. Ignoring the control input (and doing some form of global value numbering) is covered in Section 9.

6.4 A Uniform Representation

At this point we are using the same basic **Inst** class to represent the entire program. Control and data flow are represented uniformly as edges between nodes in a graph. From this point on, we refine the graph, but we do not make any major changes to it.

Having made the paradigm shift from quads to graphs, what have we gained? The answer lies in the next section where we look at the generic code for peephole optimizations. This code applies uniformly to all instruction types. Adding a new instruction type (or opcode) does not require any changes. This peephole code is powerful; while the front end is parsing and generating instructions, the peepholer is value-numbering, constant folding, and eliminating unreachable code.

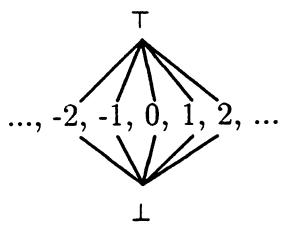
7 Types and Pessimistic Optimizations

Our previous **vpeephole** functions combined both constant folding and identity-function optimizations. As we see in Section 10, conditional constant propagation cannot use identity-function optimizations and requires only some type of constant finding code. So we break these functions up into **Compute** for finding constants, and **Identity** for finding identity functions. Any constants found by **Compute** are saved in the instruction as a *type*.

A type is a set of values. We are interested in the set of values, or type, that an instruction might take on during execution. We use a set of types arranged in the lattice with $\top$, constants, and $\bot$.[13] Figure 19 shows the lattice and the class structure to hold lattice elements. For the control-producing instructions we use $\top$ and $\bot$ to represent the absence (unreachable) and presence (reachable) of control.

[12] During the parsing stage, we cannot yet be in SSA form. We do not have ϕ-functions to mark places to insert COMPOSE instructions.

[13] The lattice used by our compiler is much more complex, allowing us to find ranges and floating point constants in addition to integer constants.




```
class Type {
  //Height in lattice
  enum { Top, Constant, Bottom } height;
  //Actual constant for middle height
  int con;
};
```

Figure 19 **Type** lattice and concrete data structure

Code for finding identity functions and computing a new **Type** is given in Figure 20. If **Identity** finds that instruction x is an identity function of some other instruction y, then x is deleted and y is returned as x's replacement. Deleting x and returning y only works if nothing references x (else we have a dangling pointer to x). So the **Identity** code can only be used when we have recently created x, during parsing. Since one of our goals is parse-time optimizations, this is not a problem.

7.1 Putting it Together: Pessimistic Optimizations

Our next peephole optimizer works as follows:

- **Compute** a new **Type** for the instruction.
- If the instruction's type is a constant, replace the instruction with a CONSTANT instruction. We delete **this** before creating the new instruction, so our fast **operator new** and **delete** can reuse the storage space. This also means we need to save the relevant constant before we delete **this**.
- Value-number the instruction, trying to find a previously existing instruction. If one exists, delete this instruction and use the old one. We don't need to run **Identity** on the old instruction, since **Identity** already ran on the old instruction before it was placed in the hash table.
- Next, try to see if this instruction is an identity function on some previous instruction.
- If we did not find a replacement instruction we must be computing a new value. Place **this** new instruction in the value-numbering hash table.
- Finally, we return the optimized instruction.

7.2 Defining Multiple Values

We have realized one of our design goals: the code for the peephole optimizations is fairly simple and straightforward. In our experience this peephole code reduces our peak memory size (and running times) by half.[14]

However, in this representation we still have the question of the IF instructions. IF instructions produce two separate results. The IF's users are partitioned into two groups, depending on which result they can access. So far, no other instruction has this behavior. In the next section we present several instruction types that produce more than one value and we find a uniform solution to selecting which result an instruction uses.

[14] Results of a preliminary study on some of the Spec benchmark codes. Complete results will be available when the compiler is completed.

```
class Inst {
  //Previous definition, withOUT vpeephole.
  Type type;                    //Runtime values
  virtual void Compute();       //Compute new Type
  virtual Inst *Identity();     //Return equivalent
};

//Default: not an identity function
Inst *Inst::Identity() { return this; }

Inst *AddInst::Identity()
{
  //We assume no identity function
  Inst *tmp = this;
  if( (src0→op == Constant) &&
      !(ConstInst*)src0→con ) {
    tmp = src1;                 //Add of zero
    delete this;                //Use addend
  }
  //Same code for src0 and src1 reversed.
  ...;
  //Return instruction we are equal to
  return tmp;
}

Inst *ComposeInst::Identity()
{ if( !max ) return this;       //No inputs?
  //Get the first datum being merged
  Inst *tmp = srcs[0]→data;
  //For all remaining inputs
  for( int i = 1; i<max; i++ ) {
    //Get datum being merged
    Inst *data = srcs[i]→data;
    //Merging unequal data means: not an identity
    if( tmp ≠ data ) return this;
  }
  delete this;
  return tmp;                   //Merging all equals
}
```

```
//The default instruction does not change type
void Inst::Compute() { }

//Constants are just themselves
void ConstInst::Compute()
{ type.height = Constant;
  type.con = constant;
}

void AddInst::Compute()
{
  //If either input is undefined
  if( (src0→type.height == Top) ||
      (src1→type.height == Top) )
    type.height = Top;      //then we are undefined
  else                      //either input is unknown
    if( (src0→type.height == Bottom) ||
        (src1→type.height == Bottom) )
      type.height = Bottom;
    else {
      type.height = Constant;
      type.con = src0→type.con + src1→type.con;
    }
}

//Lattice meet over all inputs
void ComposeInst::Compute()
{ for( int i = 0; i<max; i++ ) {
    SelectInst *sel = srcs[i];
    switch( sel→type.height ) {
    case Top: break;        //Top, no change
    case Constant:          //Constants must match
      if( (type.height == Constant) &&
          (type.con == sel.con) )
        break;
      if( type.height == Top ) {
        type = sel.type;
        break;
      }
    case Bottom:            //Fall to bottom
      type.height = Bottom;
      return;
    }
  }
}
```

Figure 20 Finding identities, computing a new **Type**

```
Inst *Inst::peephole()
{
  Inst *tmp = this;                    //Possible replacement instruction
  Compute();                           //Compute a new Type
  if( type.height == Constant ) {      //Compute a constant?
    int con = type.con;                //Save constant before delete
    delete this;                       //Nuke self
    tmp = new ConstInst(con);          //Use constant instead of self
  }
  Inst *v_n = tmp->hash_lookup();      //Hash on object-specific key
  if( v_n ) { delete tmp; return v_n; } //Return any hit
  tmp = tmp->Identity();               //Find any identity function
  if( tmp == this ) hash_install();    //New instruction for hash table
  return tmp;                          //Return optimized instruction
}
```

Figure 21 Peephole optimizations

8 More Engineering Concerns

In the original quad-based implementation, several instruction types defined 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). Previously these instructions have been dealt with on an *ad hoc* basis. We use a more formal approach.

Having a single instruction, such as an IF instruction, produce multiple distinct values (the true control and the false control) is problomatic. When we refer to an instruction, to which value are we referring? We fix this by making such multi-defining instructions produce a tuple value. Then we use PROJECTION instructions to strip out the piece of the tuple that we want. Each PROJECTION instruction takes in the tuple from the defining instruction and produces a simple value.

These PROJECTION instructions have no run-time operation (i.e., they "execute" in zero cycles). When expressed in machine code, a tuple-producing instruction is a machine instruction with several results (i.e., a subroutine call or a subtract that computes a value and sets the condition codes). The PROJECTION instructions, when expressed as machine code, merely give distinct names to the different results.

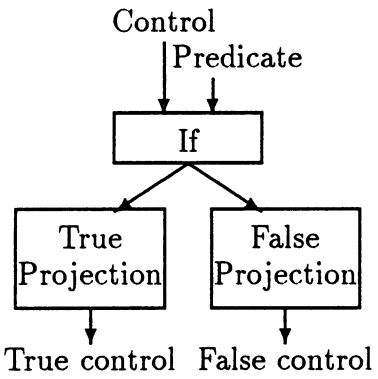
Computing a new **Type** for a PROJECTION instruction is the responsibility of the tuple-producer. The **Compute** code for a PROJECTION instruction "passes the buck" by passing the PROJECTION on to the **Compute** code for the tuple-producer, letting the tuple-producer determine the PROJECTION's **Type** and using that result. Since non-tuple-producing instructions should never be the target of a PROJECTION, the default is an error as shown in Figure 22. The **Identity** code is handled similarly.

```
class Inst {
  //Default Type computation for Projections
  virtual void Compute( Projection * ) { abort(); }
  //Default identity-function-finding for Projections
  virtual Inst *Identity( Projection * ) { abort(); }
};
```

Figure 22 Default peephole optimizations on PROJECTION instructions

8.1 IF *Instructions*

An IF instruction takes in control and a predicate and produces two distinct outputs: the true control and the false control. We implement this by having the IF instruction produce a tuple of those two values. Then a TRUE-PROJECTION instruction strips out the true control and a FALSE-PROJECTION instruction strips out the false control. The basic block that would execute on a true branch is replaced by a REGION instruction that takes an input from the TRUE-PROJECTION instruction. Similarly, the basic block that would execute on a false branch is replaced by a REGION instruction that takes an input from the FALSE-PROJECTION instruction. Figure 23 shows both the structure of a PROJECTION instruction and how it is used with an IF instruction. Figure 24 shows the code for an IF instruction.




```
class Projection : public Inst {
  //The tuple-producing input
  Inst *const src;
  //The field desired from the tuple
  const int field;
  Projection( Inst *src, int field )
  : src(src), field(field), Inst( Projection ) {}
  //Computation done by tuple-producer
  void Compute() { src→Compute(this); }
  Inst *Identity() { return src→Identity(this); }
};
```

Figure 23 Projections following an IF Instruction

8.2 *The* START *Instruction*

The START instruction needs to produce the initial control. It also needs to produce initial values for all incoming parameters, memory, and the I/O state. The START instruction is a classic multi-defining instruction, producing a large tuple. Several PROJECTION instructions strip out the various smaller values.

8.3 *Memory and I/O*

Memory is treated like any other value, and is called the *Store*. The initial *Store* is produced by the START instruction and a MEMORY-PROJECTION instruction. LOAD instructions take in a *Store* and an address and produce a new value. STORE instructions take in a *Store*, an address, and value and produce a new *Store*. The *Store* is merged in COMPOSE instructions like other values. Figure 25 shows a sample treatment of the *Store*.

The lack of anti-dependencies[15] is a two-edged sword. Between STOREs we are allowed to reorder LOAD instructions. 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* up into many smaller, unrelated *Stores*. Every independent variable or array would get its own *Store*. Operations on the separate *Stores* could proceed independently from each other. We could also add some understanding of pointers [12].

Memory-mapped I/O (e.g., **volatile** in C++) is treated like memory, except that both READ and WRITE instructions produce a new I/O state. The extra dependency (READs produce a new I/O state,

[15] 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.

```
class IfInst : public Inst {
  Inst *const control;                    //Controlling instruction
  Inst *const data;                       //Test data
  IfInst( Inst *c, Inst *d ) : control(c), data(d), Inst( If ) {}
};

void IfInst::Compute( Projection *proj )
{ proj→type.height = Top;                 //Assume unreachable
  if( control→type.height == Bottom ) {   //We are executable?
    switch( data→type.height ) {
    case Top: break;                      //Testing an undefined variable
    case Constant:                        //Testing a constant
      if( !data→type.con ∧ proj→field )   //Got test flipped?
        break;                            //Must be unreachable
    case Bottom:                          //Testing an unknown variable
      proj→type.height = Bottom;          //Must be reachable
      break;
    }
  }
}

Inst *IfInst::Identity( Projection *proj )
{ Inst *tmp = proj;                       //Assume no identities
  if( (data→op == Constant) &&            //Matching constant input?
      (!(ConstInst*)data→con ∧ proj→field) ) {
    tmp = control;                        //Then control falls through
    delete proj;                          //And we do not need this Projection
  }
  return tmp;
}
```

Figure 24 IF Instruction and optimizations

while LOADs do not produce a new *Store*) completely serializes I/O. At program exit, the I/O state is required. However, the *Store* is not required. Non-memory-mapped I/O requires a subroutine call.

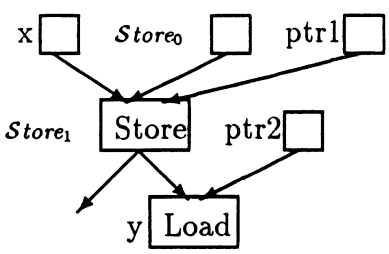
8.4 Subroutines

Subroutines are treated 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 often return a result (they already return three other results). The peephole (and other) optimization calls can use any interprocedural information present.

8.5 PROJECTION *Instructions*

With the definition of PROJECTION instructions a major gap in our model is filled. We now have concrete code on how the peepholer finds and removes unreachable code. We also see how to handle memory, subroutines, and instructions that define many values.

So far, every data instruction includes a control input that essentially defines what basic block the instruction belongs to. But in many cases we do not care what block an instruction is placed in, as long as it gets executed after its data dependencies are satisfied and before any uses. Indeed, on a superscalar or VLIW machine, we may want to move many instructions across basic block boundaries to fill idle slots on multiple functional units. In the next section we look at a simple solution to this problem: removing




```
{
  x = ...;          //Make some value
  *ptr1 = x;        //Store value in memory
  ...
  y = *ptr2;        //Load some value
  ...y...           //Use it
}
```

Figure 25 Treatment of memory (*Store*)

the control input.

9 Removing Control Information

In our model, we require every data computation to have a control input to determine when that data computation should execute. We can remove the control input from the data computations, and rely solely on the data dependencies. Removing the control input has some advantages and drawbacks. Pros:

- Fewer edges in the graph which means a smaller graph, less work to build and manipulate.
- Value numbering works by finding congruent sub-graph sections, where congruent is defined as "equal functions on congruent inputs" [13]. Removing the control edges removes an input and therefore a source of incongruence; our pessimistic value numbering becomes as strong as global value numbering [14].
- With the control input missing, only data inputs remain. Computations no longer have a notion of a basic block that they belong in. Schedulers that do code motion across basic blocks do not need to discover how much freedom an instruction has; that information is explicit.

Cons:

- With the control information removed, data instructions are not defined to be in a particular basic block. Finding a good, correct serial order (i.e., finding a basic block) for them is hard. If they are simply scheduled when all data inputs are available, the code has a massive amount of speculative execution. With the massive speculative execution comes long live ranges, high register pressure, and too much spill code. In addition, if there are not enough functional units, the code is slower.
- Global value numbering can be unprofitable. Common expressions on unrelated paths might be combined. Since an expression must be hoisted above all uses, it might be hoisted above a path that avoids all uses (i.e., a path is lengthened). Partial redundancy elimination only hoists common expressions when no path is lengthed.
- Not all data instructions can have their control inputs removed. Divide instructions have an implicit jump on a divide-by-zero condition. This change of control needs to be modeled. One way to do it is to leave the control input to the divide instruction. LOAD and STORE instructions also have an implicit test for a bad address, with a jump to a fault handler.

9.1 The Impact

Our preliminary results indicate that removing the control input with a naive scheduler gives bad results. The naive scheduler places instructions too early. Many more instructions are scheduled for speculative execution than we have registers or functional units for. We have a lot of spill code early on, completely

swamping the beneficial effects of finding more congruent expressions. We plan on implementing a stronger scheduler and this may change the impact.

Having presented our intermediate representation in great length, did we make analyses easier to code? In the next section we look at a sample optimistic analysis.

10 Optimistic Transformations

Optimistic transformations, such as *Sparse Conditional Constant Propagation* (SCCP), make "optimistic assumptions" that they must prove correct.[5] They may need to analyze the entire program to validate a single guess. Because of this we need to keep some information about our current assumptions at each instruction. This information is stored in the **Type** field, and the **Type** field is set by the previously defined **Compute** code.

Another requirement for this global analysis that we avoided with the pessimistic analysis is real def-use edges. So far, all of our optimizations can be performed given only an instruction and what it directly uses (i.e., use→def edges). For optimistic transformations we assume all instructions are $\top$ (undefined) and all code is unreachable. Starting with START, we modify these assumptions as we discover they are incorrect. When we discover an instruction defines a value other than $\top$, we must inspect the assumptions at all instructions that use that value. Hence we need def-use edges.

Because we need the def-use edges for a global (batch), algorithm, we find them all at once and put them into a large array. We put the def-use edges for a single instruction in a sequential section of this array. To access the edges for an instruction we need the section start and length in the instruction. Each instruction gets two new fields: **def_use_edge** and **def_use_cnt**.

We find the def-use edges by walking the graph's use-def edges. To do the graph walking we need a visit flag, a use-def edge count, and a function that accesses use-def edges by index. We include the new definitions for **Inst** in Figure 26.

```
class Inst {
  static Inst **DefUse;                  //Array of def-use edges
  static int EdgeCnt;                    //Number of use-def edges
  static int visit_goal;                 //Goal color for visits
  ...                                    //Other member functions
  int use_def;                           //Number of use-def edges
  Inst *operator[](int idx) const;       //Access use-def edge by index
  Node **def_use_edge;                   //Start of def-use edges
  int def_use_cnt, tmp;                  //Number of def-use edges
  int visit;                             //Visit flag for DFS walking
  Type type;                             //Set of possible run-time values
};
```

Figure 26 New fields for def-use edges

We build the def-use edges in Figure 27. We require the STOP instruction and the number of use-def edges as inputs.[16] We build an empty array to hold the def-use edges. We walk the graph once to count the def-use edges out of each **Inst**. During this walk we also initialize all **Types** to $\top$. On the second pass, we store both the edge values into the array and the start of the array section into the **Inst**. Since we start all walks from the STOP instruction and only travel along use-def edges we never visit dead code. Since we never visit dead code, we never write it out; we have effectively done a round of dead code elimination before running SCCP.

[16] If we do not keep a running count of use-def edges as they are made, we must make another pass to find the total number of use-def edges.

```
static Inst **du_tmp;
Inst **Build_Def_Use( Inst *stop, int EdgeCnt )
{
  Inst **DefUse = du_tmp = new Inst *[EdgeCnt];
  visit_goal = !visit_goal;                  //Goal color for visit flops
  stop->bld_du_cnt();                        //Count def-use edges per Inst
  visit_goal = !visit_goal;                  //Goal color for visit flops
  stop->bld_du_edge();                       //Setup section of def-use edges per Inst
  return DefUse;
}

void Inst::bld_du_cnt()
{
  if( visit == visit_goal ) return;          //Been here already
  visit = visit_goal;                        //Do not come here again
  type.height = Top;                         //Initialize everything to Top
  for( int i=0; i<use_def; i++ ) {           //For all inputs
    Inst *use = (*this)[i];                  //Get a value we use
    use->tmp = ++use->def_use_cnt;           //Value is being used by us
    use->bld_du_cnt();                       //Recursively cover the graph
  }
}

void Inst::bld_du_edge()
{
  if( visit == visit_goal ) return;          //Been here already
  visit = visit_goal;                        //Do not come here again
  defuse = du_tmp;                           //Start of our array section
  du_tmp += def_use_cnt;                     //Bump so next Inst gets different section
  for( int i=0; i<use_def; i++ ) {           //For all inputs
    Inst *use = (*this)[i];                  //Get a value we use
    use->bld_du_edge();                      //Make value assign own array section
    use->def_use_edge[--u->tmp] = this;      //We use what the value defines
  }
}
```

Figure 27 Building def-use edges

Next we run SCCP. We put the START instruction on our worklist.[17] Then we enter a simple loop where we pull an instruction from the worklist, compute a new **Type** for it, and if the **Type** changed, we put all users of that instruction back on the worklist. When the worklist empties, the job is done.

10.1 The Payoff

Here then, is the result we have been working for. This expression of *Sparse Conditional Constant Propagation* is very clear and simple.

```
void SCCP( Inst *start )
{
  start→type.height = Bottom;              //Start is executable
  worklist_push(start);                    //Push the Start instruction
  while( !worklist_empty() ) {             //While work to do...
    Inst *x = worklist_pop();              //Get some work
    Type old = x→type;                     //Save previous Type
    x→Compute();                           //Compute new Type
    if( old ≠ x→type ) {                   //Changed?
      for( int i=0; i<x→def_use_cnt, i++ )//Yes! For all users...
        worklist_push(x→def_use_edge[i]); //...put user on worklist
    }
  }
}
```

Figure 28 *Sparse Conditional Constant Propagation*

11 Summary

To produce a faster optimizer, we decided to do some of the work in the front end. We reasoned that cheap peephole optimizations done while parsing would reduce the size of our intermediate representation and the expense of later optimization phases. To do useful peephole optimizations we need use-def information and the static single assignment property.

We made our front end build in SSA form. Because we cannot analyze the entire program while we parse it, we had to insert redundant ϕ-functions. We observed that variable names were a one-to-one map to program expressions. So we replaced the variable names with pointers to their concrete representations. At this point the variable name defined by any expression becomes redundant, so we removed the name (the `dst` field) from our **Insts**. We also observed an implicit flow of control within basic blocks, which we made explicit (and thus subject to optimizations). We discovered, while trying to write peephole versions of unreachable code elimination, that our model was not compositional. We fixed this by bringing in control dependences to the ϕ-functions and breaking up ϕ-functions into SELECT and COMPOSE instructions.

We took advantage of C++'s inheritance mechanisms and restructured our **Insts** into separate classes for each kind of instruction. We also plugged in specialized allocate and delete functions.

At this point we noticed that our basic blocks' structures held no more information than a typical instruction (i.e., a variety of dependence edges). So we replaced the basic block structures with REGION instructions. Our same peephole mechanism we had been using all along now allowed us to do unreachable code elimination in addition to the regular constant folding and value numbering optimizations.

[17] Actually we put all zero-input instructions on the worklist. There are a variety of ways to deal with constants, none of which are mentioned here.

We broke up the per-instruction peephole optimizations into constant folding and identity finding functions. The constant folding functions are being used by the global optimizations. As we progressed we discovered convenient mechanisms to handle instructions that defined several values, subroutines, memory, and I/O.

As an experiment we removed the control dependence from most of the data-computing instructions. This gives our value numbering optimization more power, but requires us to have a less naive scheduler. Since many modern CPUs require a powerful scheduler already we do not believe this to be a drawback.

Finally we implemented SCCP and were pleased with the simple and clear way we could express it.

We have presented the outline of an intermediate representation being used in a research compiler. Several optimizations have been implemented using this representation. Those optimizations have been easier to write and maintain than versions written for an earlier intermediate representation. The representation is fairly complete, handling various messy details like subroutine calls, I/O, and instructions that define multiple values in a uniform fashion. Control and data optimizations are also handled in a uniform fashion.

Our preliminary results indicate the resulting optimizer runs quite fast, on par with several commercial compilers. We expect our compiler, with some hand tuning and a touch of assembler, can double in speed.

11.1 Acknowledgements

I would like to thank my advisor, Keith Cooper, for allowing me explore compiler design with a different slant. I also wish to thank Michael Paleczny for many hours of pondering how an intermediate representation defines a program and Chris Vick for being a wonderous sounding board.

REFERENCES

[1] Paul Havlak. Construction of gated single-assignment form. submitted to SIGPLAN '93, 1993.

[2] R. A. Ballance, A. B. Maccabe, and Karl J. 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 Language Design and Implementation*, 1990.

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

[4] R. Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. Technical Report TR-90-1152, Cornell University, 1990.

[5] Mark N. Wegman and F. Kenneth Zadeck. Constant propogation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

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

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth ACM Symposium on the Principles of Programming Languages*, 1989.

[8] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. In *Proceedings of the ACM SIGPLAN '89 Conference on Program Language Design and Implementation*, 1989.

[9] Rebecca P. Selke. *A Semantic Framework for Program Dependence.* PhD thesis, Rice University, 1992.

[10] John 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.

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

[12] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990.

[13] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, 1988.

[14] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, 1988.