The Advantages of Machine-Dependent Global Optimization

Manuel E. Benitez and Jack W. Davidson

Department of Computer Science
University of Virginia
Charlottesville, VA 22903 U. S. A.

Abstract. Using an intermediate language is a well-known, effective technique for constructing interpreters and compilers. This paper describes a retargetable, optimizing compilation system centered around the use of two intermediate languages (IL): one relatively high level, the other a low level corresponding to target machine instructions. The high-level IL (HIL) models a stack-based, hypothetical RISC machine. The low-level IL (LIL) models target machines at the instruction-set architecture level. All code improvements are applied to the LIL representation of a program. This is motivated by the observation that most optimizations are machine dependent, and the few that are truly machine independent interact with the machine-dependent ones. This paper describes several 'machine-independent' code improvements and shows that they are actually machine dependent. To illustrate how code improvements can be applied to a LIL, an algorithm for induction variable elimination is presented. It is demonstrated that this algorithm yields better code than traditional implementations that are applied machine-independently to a high-level representation.

1 Introduction

A retargetable, optimizing compiler must perform a comprehensive set of code improvements in order to produce high-quality code for a wide range of machines. A partial list of code improvements that must be included in the compiler's repertoire is:

- register assignment and allocation,
- common subexpression elimination,
- loop-invariant code motion,
- induction variable elimination,
- evaluation order determination,
- constant folding,
- constant propagation,
- dead code elimination,
- loop unrolling,
- instruction scheduling, and
- inline function expansion.

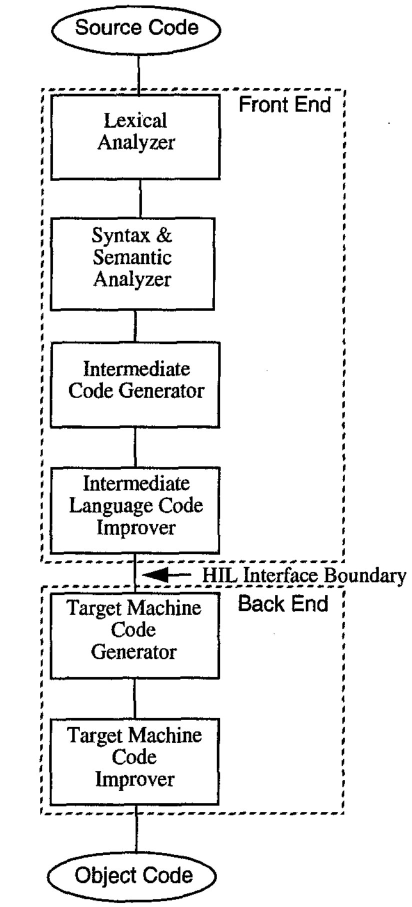
This list of code improvements traditionally is divided into two groups: those that are considered to be *machine-independent* and those that are *machine-dependent*. Machine-independent code improvements are those that do not depend on any features or characteristics of the target machine. Examples of code improvements included in this group are constant folding, dead code elimination, and constant propagation. Because of their machine-independence, these code improvements are often applied to the high-level intermediate language representation of the program.

The proper application of machine-dependent code improvements, on the other hand, requires having specific information about the target machine. Obviously, code improvements such as register allocation and instruction scheduling are machine dependent. In the case of register allocation, the types and number of registers available affect the promotion of data to registers. Similarly, effective instruction scheduling requires information about the operation of the target machine's pipeline. Somewhat less obvious, but no less machine dependent are the code improvements inline function expansion and loop unrolling. Inline function expansion can be performed most effectively when details of the target machine's instruction cache is available. Similarly, the amount of loop unrolling performed depends on the number of target machine registers available, characteristics of the instruction pipeline as well as the size of the instruction cache.

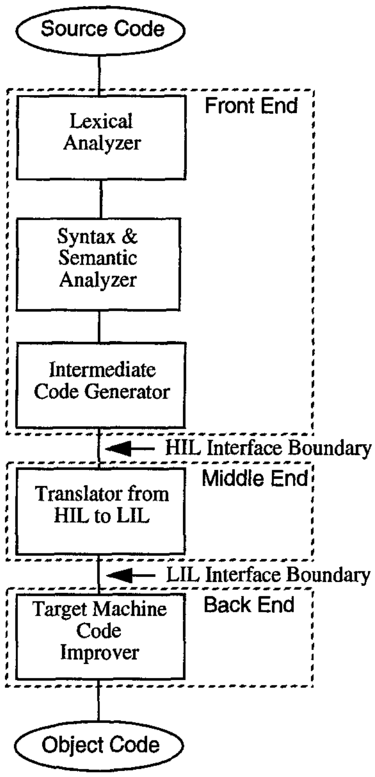
The belief that some code improvements are machine-independent and some are machine-dependent and the use of a single high-level intermediate representation results in a compiler with a structure shown in Figure 1a. Unfortunately, most code improvements are not machine-independent, and the few that truly are machine independent interact with those that are machine dependent causing phase-ordering problems. For example, dead code elimination is machine independent. However, it interacts with machine-dependent code improvements such as inline function expansion. Expanding functions inline exposes new opportunities for dead code elimination by effectively propagating constants across calls. Hence, essentially *there are no machine-independent code improvements.* In Section 2, some code improvements that typically are viewed as being machine independent are examined and shown to be machine dependent. This section also provides examples of how true machine-independent code improvements interact with machine-dependent ones. Section 3 describes a compiler structure that uses two intermediate languages: a high level intermediate language (HIL) that serves to isolate the language-dependent portion of the compiler from target machine details, and a low-level intermediate language (LIL) that supports the application of global code improvements. Section 4 contains a detailed description of an induction variable elimination algorithm that operates on a low-level representation of a program. The algorithm is largely machine independent, and requires no modification when the compiler is retargeted, yet it generates superior code when compared to a traditional HIL implementation. Section 5 evaluates the effectiveness of the LIL implementation of induction variable elimination on a set of representative benchmark programs.

2 The Case for Machine-Dependent Global Optimization

To illustrate the point that all code improvements are effectively machine dependent, consider constant propagation. This deceptively simple code improvement involves propagating a constant that has been assigned to a variable (a definition) to points in the


Source Code
Front End
Lexical Analyzer
Syntax & Semantic Analyzer
Intermediate Code Generator
Intermediate Language Code Improver
HIL Interface Boundary
Back End
Target Machine Code Generator
Target Machine Code Improver
Object Code

a) Structure resulting from use of a single HIL.



b) Structure resulting from use of a HIL and a LIL.

Figure 1. Structure of two compiler organizations.

program where the variable is used and the definition reaches. After constant propagation is performed often the assignment to the variable becomes useless and can be eliminated. Additionally, knowing the value of a constant at a particular point in the program permits other code improvements to be performed. Constant propagation typically is considered to be a machine-independent code improvement and is performed in the machine-independent front end portion of the compiler. Unfortunately, constant propagation is not machine independent. To be done most effectively, characteristics of the target machine must be known.

To explain some of the machine-dependent complications that arise when performing constant propagation, consider the C code in Figure 2a. Assuming that variable y is not modified elsewhere in the function, should the value 10.0 be

```
void foo()
{
      double y;

      y = 10.0;
      ...
      baz (y);
      ...
      bar (y);
      ...
}
```

a) Constant propagation is not worthwhile.

```
main()
{
      double y;

      y = 10.0;
      foo(y);
      ...
      foo(y + 32.56);
      ...
}
```

c) Constant propagation is worthwhile.

```
_foo:
      ...
      # load 10.0
      sethi  %hi(L20),%o0
      ldd    [%o0+%lo(L20)],%f0
      # store in y
      std    %f0,[%fp-8]
      ...
      # call baz with y
      call   _baz,2
      ldd    [%fp-8],%o0
      ...
      # call bar with y
      call   _bar,2
      ldd    [%fp-8],%o0
      ...
L20: .double  0r10.0
```

b) SPARC assembly code for fragment in Figure 2a.

```
_main:
      ...
      sethi  %hi(L20),%o0
      # call foo with 10.0
      call   _foo,2
      ldd    [%o0+%lo(L20)],%o0
      ...
      sethi  %hi(L21),%o0
      # call foo with 42.56
      call   _foo,2
      ldd    [%o0+%lo(L21)],%o0
      ...
L20: .double  0r10.0
L21: .double  0r42.56
```

d) SPARC assembly code for fragment in Figure 2c.

Figure 2. Code illustrating complications of machine-independent constant propagation.

propagated to each of the uses of the variable? If constant propagation is performed at a high-level, before details of the target machine are known, the choice is simple; propagate the constant. However, the correct action depends on the target machine and how the constant is being used. On the SPARC architecture three factors affect whether a floating-point constant should be propagated. First, there is no direct data path between the fixed-point registers and the floating-point registers. Moving a value from one register set to another requires going through memory. Second, the SPARC calling sequence requires that the first 24 bytes of arguments be passed in the fixed-point registers `%o0` through `%o5` regardless of their type. Third, the only way to load a floating-point constant into a register is by fetching it from global memory (i.e., there is no load immediate for floating-point values). With the current conventions this requires two instructions.

Figure 2b contains the SPARC assembly code generated for the C code fragment in Figure 2a using Sun's optimizing compiler with the highest level of optimization. Recall that call and branch instructions on the SPARC have a single-instruction delay slot. In this example, the constant was not propagated. Indeed, if it had, inferior code would have been produced. The load double instructions in the call delay slots would each be a two instruction sequence to load the constant from global memory. The astute reader might argue that the compiler, in this case, should not have propagated the constant, but rather allocated it to a floating-point register, and used the register at each point in the code where `y` is referenced. Unfortunately, this would result in even poorer code. Because the only path from a floating-point register to a fixed-point register is through memory, this approach would have required storing the contents of the floating-point register in memory and reloading it in the appropriate output registers. Because of these complications and because it performs constant propagation at a high-level, it appears Sun's SPARC compiler is forced to follow the simple rule: never propagate floating-point constants.

Is it always best not to propagate floating-point constants on the SPARC? To answer this question, consider the C code fragment in Figure 2c. Here it would be beneficial to propagate the constant. If the constant is propagated, constant folding can be done for the argument in the second call to `foo` and the resulting constant can be loaded directly into `%o0` and `%o1`. The code is shown in Figure 2d. This code is 50% smaller than the code that would be produced without constant propagation.

As another example, consider the code improvement loop-invariant code motion (LCM). Again, many compilers perform this transformation at a high-level under the assumption that machine-specific information is not required. However, this is not the case. Consider the code fragment in Figure 3a. `limit` is an external global variable and `update` is an external function that can potentially alter the value of `limit`. Figure 3b shows typical HIL for this code. With this representation, there is no visible loop invariant code. The code generated for the SPARC is shown in Figure 3c. An inspection of this code reveals that the computation of `limit`'s address was loop invariant.

It is tempting to say that this problem can be solved by changing the HIL so that computation of addresses is decoupled from the actual reference. Figure 3d shows a HIL version of the code using this approach. Now the computation of the address of `limit` is visible, and a code improver operating on the HIL would move it out of the loop. This code is shown in Figure 3e. Unfortunately, this still does not yield the best possible machine code. On the SPARC, the calculation of the address of a global requires two instructions. However, it is possible to fold part of the address calculation into the instruction that does the memory reference. By taking into account the machine's addressing modes and the costs of instructions, a code improver that operates on a LIL representation produces the code of Figure 3f. While the loops of Figure 3e and Figure 3f have the same number of instructions, the overall code in Figure 3f is one instruction shorter. If this is a function that is called many times, the impact on execution time will be noticeable.

```
extern int limit;

void find(value)
int value;
{
     extern void update();

     while (value < limit)
        update();
}
```

a) Code with loop-invariant address calculation.

```
FUNC    void  find
LABEL   16
LOAD    int   value   PARAM
LOAD    int   limit   EXTERN
JMPGE   int   17
CALL    void  update  EXTERN
GOTO    16
LABEL   17
EFUNC   void  find
```

b) HIL for a.

```
_find:
   save   %sp,-96,%sp
L16:
   sethi  %hi(_limit),%o0
   ld     [%o0+%lo(_limit)],%o0
   cmp    %i0,%o0
   bge    L17
   call   _update
   ba     L16
L17:
   ret
   restore
```

c) SPARC code generated from b.

```
FUNC    void  find
LABEL   16
ADDR    int   value   PARAM
DEREF   int
ADDR    int   limit   EXTERN
DEREF   int
JMPGE   int   17
CALL    void  update  EXTERN
GOTO    16
LABEL   17
EFUNC   void  find
```

d) HIL with address calculations exposed.

```
_find:
   save   %sp,-96,%sp
   sethi  %hi(_limit),%l0
   add    %l0,%lo(_limit),%l0
L16:
   ld     [%l0],%o0
   cmp    %i0,%o0
   bge    L17
   call   _update
   ba     L16
L17:
   ret
   restore
```

e) SPARC code generated from d with LCM.

```
_find:
   save   %sp,-96,%sp
   sethi  %hi(_limit),%l0
L16:
   ld     [%l0+%lo(_limit)],%o0
   cmp    %i0,%o0
   bge    L17
   call   _update
   ba     L16
L17:
   ret
   restore
```

f) SPARC code with machine-dependent LCM.

Figure 3. Example illustrating the machine dependence of loop-invariant code motion (LCM).

As a last example, consider dead code elimination (DCE). This transformation is truly machine independent. That is, any code that will never be executed should always be eliminated no matter what the target machine. Unfortunately, machine-dependent code improvements create opportunities for DCE, and therefore, to be most effective,

```
void daxpy(n, da, dx, incx, dy, incy)
int n, incx, incy;
double da, dx[], dy[];
{
     if (n <= 0)
        return;
     if (da = 0.0)
        return;
     if (incx != 1 || incy != 1) {
        /* Code for unequal increments or */
        /* equal increments other than one */
        }
     else
        for (i = 0; i < n; i++)
           dy[i] = dy[i] + da * dx[i];
}
```

Figure 4. *daxpy* routine from *linpack.*

DCE should also be performed at the machine level. To see this, first consider the machine-dependent code improvement inline code expansion. This transformation replaces calls to functions with the body of the called function. It eliminates call/return overhead, may improve the locality of the program, and perhaps most importantly, can enable other code improvements which includes, among others, dead code elimination. Inline code expansion is machine dependent because the decision to inline depends on the characteristics of the target machine. One important consideration is the size of the instruction cache [11]. Inlining a function into a loop and possibly causing the loop to no longer fit in the cache can result in a serious drop in performance. To illustrate how DCE interacts with inlining, consider the *daxpy* function from the well-known *linpack* benchmark. The code is shown in Figure 4. Generally, all calls to *daxpy* set `incx` and `incy` to one. Thus, by first performing inlining and constant propagation (both machine dependent code improvements), a dead code eliminator that operates on LIL after inline code expansion and constant propagation will eliminate the code for handling increments that are not both one.

3 A HIL/LIL Compiler Organization

These observations lead to the conclusion that more effective code improvement can be performed if all transformations are done on a low-level representation where target machine information is available. To accomplish this requires two intermediate representations: a HIL that isolates, as much as possible, the language-dependent portions of the compiler from the target machine specific details, and a LIL that supports applying code improvements. The use of two intermediate languages yields a compiler with the structure shown in Figure 1b. It is significantly different from that of the traditional, single intermediate language representation shown in Figure 1a. The influence of the use of two intermediate languages is pervasive—affecting the design of the HIL, as well as the code generation algorithms used in the front end. The following sections discuss these effects.

3.1 The High-Level Intermediate Language

In most modern compilers the front end is decoupled from the back end through the use of an intermediate representation. The goal is to make the front end machine independent so that it can be used for a variety of target architectures with as little modification as possible. One popular approach is to generate code for an abstract machine. Well-known abstract machines include P-code (used in a several Pascal compilers) [12], U-code (used in the compilers developed by MIPS, Inc. for the R2000/ R3000 family of microprocessors) [3], and EM (used in the Amsterdam compiler kit) [17, 18]. In the quest for efficiency, the abstract machine often models the operations and addressing modes found on the target architectures. For a retargetable compiler, with many intended targets, this can yield a large and complex abstract machine. Such abstract machines have been termed 'union' machines as they attempt to include the union of the set of operators supported on the target architectures [6]. The Berkeley Pascal interpreter, for example, has 232 operations [10].

There is an equally compelling argument for designing a small, simple abstract machine. Small, simple instruction sets are faster and less error prone to implement than a large complex instruction set. Abstract machine designers have long recognized this dilemma. In 1972, Newey, Poole, and Waite [13] observed that

> 'Most problems will suggest a number of specialized operations which could possibly be implemented quite efficiently on certain hardware. The designer must balance the convenience and utility of these operations against the increased difficulty of implementing an abstract machine with a rich and varied instruction set.'

Fortunately, applying all code improvements to the LIL removes efficiency considerations as a HIL design issue. The abstract machine need only contain a set of features roughly equivalent to the intersection of the operations included in typical target machines. The result is a small, simple abstract machine. Such abstract machines are termed 'intersection' machines. The analogy between union/intersection abstract machines and CISC/RISC architectures is obvious.

There are other reasons for preferring a small abstract machine instruction set. First, a small instruction set is more amenable to extension. Adding additional operations to support a new language feature (for example, a new opcode to support the pragma feature of ANSI C was recently added to the CVM instruction set) is generally not a problem. However, for large instruction sets, this may cause problems. For example, many abstract machines have over 200 operations. Adding more operations may require changing the instruction format (a byte code may not be sufficient). Second, intersection machines are more stable. If a machine appears with some new operation, the operation must be added to the union machine. The intersection machine, on the other hand, need only be changed if the new operation cannot be synthesized from the existing operations. Third, if the compiler is to be self-bootstrapping (a lost art), a small intermediate language can significantly reduce the effort to bootstrap [15, 13]. For additional justification for preferring a small, simple abstract machine over a large, complex one see [6].

The HIL described here is called CVM (C Virtual Machine), and it supports most imperative languages, although it was motivated mainly by the desire to support variants of the C language (K&R C, ANSI C, and C++). The CVM instruction set contains 51 executable instructions and 17 pseudo operations. Similar to the abstract machines mentioned above, CVM is a stack architecture as opposed to a register architecture. CVM is stack-oriented for a couple of reasons. First, algorithms for generating code for a stack machine are well understood and easy to implement. Second, it was important to be able to specify the semantics of operation of CVM. This is done operationally through an interpreter. Implementing an interpreter for a stack-based machine is quite simple, easy to understand, and reasonably efficient [8].

3.2 The Low-Level Intermediate Language

The LIL representation of a program is what will be manipulated by all code improvement algorithms. Thus, while it is necessary that the LIL encode machine-specific details so that the code improvement algorithms can produce better code, it must be done in such a way that the implementation of the algorithms does not become machine-dependent.

The LIL representation described here is based on register transfer lists (RTLs), which are derived from the Instruction Set Processor (ISP) notation developed by Bell and Newell. Essentially, RTLs describe the effects of machine instructions and have the form of conventional expressions and assignment's over the hardware's storage cells. Each RTL corresponds to a single target machine instruction in the same way that each traditional assembly code line corresponds to a single instruction. Unlike assembly language syntax, which varies from machine to machine, the RTL specification of an operation is identical across machines. For example, the following list shows register-to-register add instructions on various machines using their assembly language syntax:

`a`	`r1=r1,r2`	-- IBM RS/6000
`ar`	`1,2`	-- IBM 370
`add`	`1,2`	-- DecSystem 10
`ix1`	`x1+x2`	-- CDC 6600
`add`	`%o2,%o1,%o1`	-- Sun SPARC
`add.l`	`d2,d1`	-- Motorola 68020
`addl2`	`r2,r1`	-- VAX-11
`addu`	`$1,$1,$2`	-- MIPS R3000

Using the RTL notation, each of these instructions is represented by the following RTL:

```
r[1] = r[1] + r[2];
```

In contrast to the assembly language specification of the instruction, the RTL unambiguously describes the action of the instruction. Thus, the above RTL clearly indicates which register receives the result of the addition operation. Assembly language instructions, on the other hand, use conventions to designate source and destination operands.

Most machines have instructions that perform several actions. These are specified using lists of transfers (hence the name register transfer *lists*). A common occurrence in some machines are instructions that perform an arithmetic operation and set bits in a condition code register to indicate some information about the result (e.g. equal to zero, negative, overflow, etc.). Such multi-effect instructions are specified using RTLs as:

```
r[1] = r[1] + r[2]; Z = (r[1] + r[2]) == 0;
```

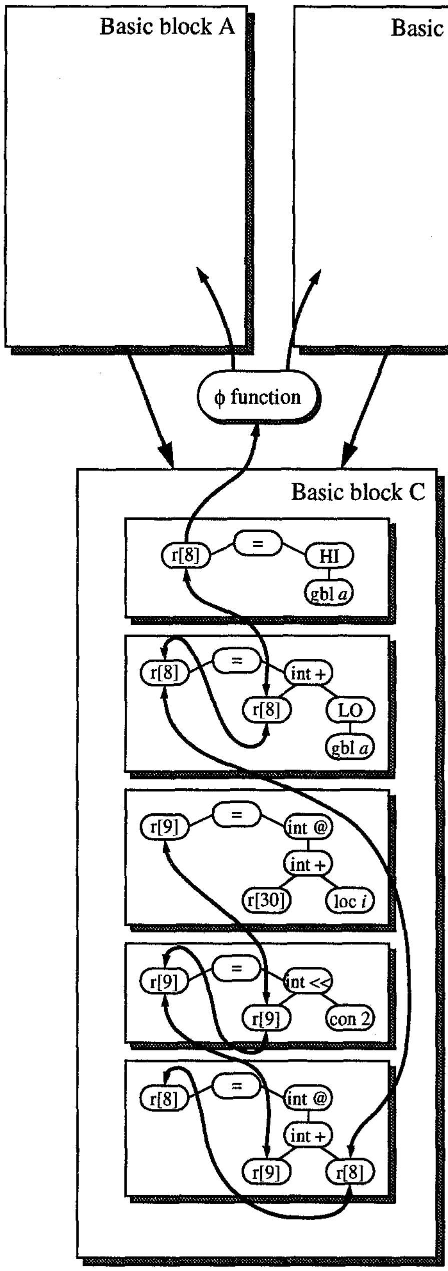
Each register transfer in the list is performed concurrently. Thus, the above RTL specifies the same addition operation as the previous RTL and also sets the zero bit in the condition code register (specified by `Z`) if the result of the operation is zero or clears it, otherwise.

RTLs have been used successfully to automate machine-specific portions of a compiler such as instruction selection, common subexpression elimination, and evaluation order determination [5, 6, 7] These are all local transformations that do not require information beyond that contained in a basic block. To better support global code improvements such as loop-invariant code motion, induction variable elimination, constant propagation, loop unrolling, and inline function expansion, we represent RTLs using a binary tree structure that allows each component of a register transfer (e.g. a register, a memory reference, an operation, a constant, etc.) to contain information that is specific to that component. For example, tree node representing operators include a type specifier and register nodes contain a pair of pointers linking them to the next and previous reference of the register. This LIL is more than a language, it is a representation that allows global code improvements algorithms to take into account the characteristics of the target machine in a machine-independent fashion.

A simplified diagram of a program fragment represented using this LIL is shown in Figure 5a. Details are shown only for basic block C and references to registers `r[8]` and `r[9]`. The representation consists of a control-flow graph of basic blocks. Associated with each basic block is a list of RTLs that represent the machine instructions that will be executed when flow of control passes through this basic block. The corresponding RTLs in string form and in SPARC assembly language for the code in this basic block are shown in Figure 5b and Figure 5c, respectively. Also associated with each basic block is information about which loops the basic block is a member of. This information includes the location of the preheader block of the loop (if one exists), dominance relations, induction variable information, and loop-invariant values.

For each reference to a register or a memory location, a def-use chain is maintained. Thus, from any reference the code improver can find either the previous reference or the next reference. Previous reference involving merging flow can be found through ϕ functions [4]. These functions and the def-use links form the SSA form for the code. This form is used to determine a canonical value for each component of the code. These are similar to value numbers [1] and are used to perform common-subexpression elimination as well as code motion.

For each memory reference, information about the memory partition affected by the reference is maintained [2]. This structure is used to hold information that is vital for



a) Internal LIL representation

```
r[8]=HI[_a];
r[8]=r[8]+LO[_a];
r[9]=M[r[8]+i];
r[9]=r[9]<<2;
r[8]=M[r[9]+r[8]];
```

b) RTL code for basic block C

```
sethi  %hi(_a),%o0
add    %o0,%lo(_a),%o0
ld     [%i6 + i],%o1
sll    %o1,2,%o1
ld     [%o1 + %o0],%o0
```

c) SPARC assembly code for basic block C

Figure 5. Schematic of the internal LIL representation.

performing induction variable elimination (IVE). For instance, if the memory reference is via an induction variable, information about the induction variable such as the scale (sometimes called the *cee* value) and displacement (sometimes called the *dee* value) is

```
int cmp(a, b)
int a[], b[];
{
     int i;

     for (i = 0; i < 100; i++)
        if (a[i] != b[i])
           return(1);
     return(0);
}
```

a) C code with two induction expressions

```
_cmp:
1.    add    %o0,400,%o2
L16:
2.    ld     [%o0],%o3
3.    ld     [%o1],%o4
4.    cmp    %o3,%o4
5.    bne    L17
6.    add    %o0,4,%o0
7.    add    %o1,4,%o1
8.    cmp    %o0,%o2
9.    bl     L16
10.   mov    0,%o0
11.   retl
L17:
12.   mov    1,%o0
13.   retl
```

b) SPARC code produced by a high-level code improver

```
_cmp:
1.    sub    %o1,o0,%o1
2.    add    %o0,400,%o2
L16:
3.    ld     [%o0],%o3
4.    ld     [%o0 + %o1],%o4
5.    cmp    %o3,%o4
6.    bne    L17
7.    add    %o0,4,%o0
8.    cmp    %o0,%o2
9.    bl     L16
10.   mov    0,%o0
11.   retl
L17:
12.   mov    1,%o0
13.   retl
```

c) SPARC code produced by a low-level code improver

Figure 6. High-level versus low-level code improvement.

maintained. This structure also contains information that allows the code improver to resolve potential aliasing problems.

The above structure is very flexible and supports the implementation of all common, and some not so common, code improvements. The following section describes one of these code improvements in detail with emphasis on how it is accomplished in a machine-independent way, yet takes into account machine-dependent information.

4 Machine-Dependent Induction Variable Elimination

An induction variable is a variable that is used to induce a sequence of values. In the code of Figure 6a, the variable i is an induction variable because it is being used to induce a series of addresses (those of the array elements). If this is the only use of the variable, it can be beneficial to eliminate the induction variable altogether and just compute the sequence of addresses. The sequence of values being computed from the induction variable is called the *induced sequence.*

Simple induction variables are used to compute induced sequences of the form $scale \times i + displacement$, where i is the basic induction variable. In the example in Figure 6a, the sequences being computed are $4 \times i + a$ and $4 \times i + b$, where a and b are the starting addresses of the arrays. Using well-known algorithms [1], the induction variable `i` can be eliminated and be replaced by the computation of the induced sequence of the addresses. The SPARC code produced by a code improver operating on a HIL is shown in Figure 6b. Notice that the sequences of addresses are being computed using two registers. The sequence for referencing `a` is being computed in `%o0`, and the sequence for `b` is being computed in register `%o1`. As argued in Section 2, no code improvement is really machine independent. Better IVE can be performed if it is done on a LIL where target machine information is available. The loop in Figure 6c is one instruction shorter than the loop in Figure 6b. On the SPARC, machine-dependent IVE results in one instruction being saved for every induction expression that can be computed via a difference from the basic induction variable. A systematic inspection of source code shows that approximately 22 percent of loops with induction variables contain multiple references using the same basic induction variable. Figure 7 contains a high-level description of the algorithm that is used to perform IVE on RTLs.

As the algorithm is explained, one key point should be kept in mind: the algorithm is machine-independent! That is, no changes are necessary to it when a new machine is accommodated. The algorithm obtains needed machine-dependent information via calls to a small set of machine-dependent routines that are constructed automatically from a description of the target architecture. These calls are underlined. This is a subtle point, but very important. It is possible to implement code improvements in a machine-independent way, yet take into account machine-dependent information.

Basic information needed to perform IVE is collected (lines 2-4). This includes the loop invariant values, the basic induction variables, and the induction expressions. The induction expressions are expressions that involve the use of the same basic induction variable. The list of induction expressions include the basic induction variables. This information is stored with the loop information that is accessible from each block in the loop.

The ***while*** loop starting at line 5 processes each of the induction expressions. For a particular induction expression, all induction expressions that depend on that one are collected into a list at lines 8 and 9. This list is sorted by the machine-dependent routine, *OrderInductionExprs* according to the capabilities of the target machine. For example, if the target machine allows only positive offsets in the displacement addressing mode, it is best to have the expression in order of increasing value. On the other hand, if the machine has a limited range of offset, yet supports both negative and positive offsets, the list should be ordered so that expressions in the middle of the range are first so smaller offsets can be employed.

```
1  proc ImproveInductionExprs(LOOP) is
2     LOOP.InvariantVals = FindLoopInvariantVals(LOOP)
3     LOOP.InductionVars = FindInductionVars(LOOP, LOOP.InvariantVals)
4     LOOP.InductionExprs = FindInductionExprs(LOOP, LOOP.InductionVars, LOOP.InvariantVals)
5     while $LOOP.InductionExprs \neq \emptyset$ do
6        IND = FirstItem(LOOP.InductionExprs)
7        $EXPR = \emptyset$
8        for each E where $E \in LOOP.InductionExprs \wedge E.Family = IND.Family \wedge E.Scale = IND.Scale$ do
9           $EXPR = EXPR \cup E$
10       endfor
11       OrderInductionExprs(EXPR)
12       IND = FirstItem(EXPR)
13       R = NewRegister(ADDRESS_TYPE)
14       if $LOOP.Preheader = \emptyset$ then
15          BuildPreheader(LOOP)
16       endif
17       InsertCalculation(LOOP.Preheader, "$R = IND.Family \times IND.Scale + IND.Displacement$")
18       for each E where $E \in EXPR$ do
19          DIFF = CalculateDifferenceExpression(E.Displacement, IND.Displacement)
20          UPDATED = FALSE
21          if DIFF = 0 then
22             NEW = ReplaceExpression(E.Inst, E, "R")
23             if IsValidInstruction(NEW) then
24                UPDATED = TRUE
25             endif
26          endif
27          if $\neg UPDATED \wedge IsLiteralConstant(DIFF)$ then
28             NEW = ReplaceExpression(E.Inst, E, "R + DIFF")
29             if IsValidInstruction(NEW) then
30                UPDATED = TRUE
31             endif
32          endif
33          if $\neg UPDATED \wedge IsLoopInvariant(DIFF, LOOP.InvariantVals)$ then
34             DR = NewRegister(ADDRESS_TYPE)
35             NEW = ReplaceExpression(E.Inst, E, "R + DR")
36             if IsValidInstruction(NEW) then
37                UPDATED = TRUE
38                InsertCalculation(LOOP.Preheader, "DR = E.Displacement - IND.Displacement")
39             endif
40          endif
41          if UPDATED then
42             ReplaceInstruction(E.Instruction, NEW)
43          endif
44          if $UPDATED \vee (DIFF = 0)$ then
45             LOOP.InductionExprs = LOOP.InductionExprs - E
46          endif
47       endfor
48    endwhile
49 endproc
```

Figure 7. High-level description of machine-dependent induction variable elimination algorithm.

To accommodate computing the induced sequences, a preheader is added to the loop at line 14 if one does not exist, and the machine instructions needed to generate the first value of the sequence are inserted in the preheader. This routine is machine-dependent because it must generate the LIL code that represents the target machine instructions needed to compute the value. The ***for*** loop at line 18 processes the induction expressions (including the first one selected outside the loop). Lines 21 through 40 of this loop determine, for each induction expression, the best way to compute the expression for the target machine. The difference between the displacement value of first induction expression selected and the current one is determined at line 19. If the difference is zero, then the register holding the induction value can be used. The routine, *ReplaceExpression*, replaces the reference to the expression with the reference to the register containing the induction value. This new instruction is checked to see whether it is valid on the target machine by the call to *IsValidInstruction* at line 23.

If the difference was not zero, the difference is checked at line 27 to see if it is a literal constant. If it is, then this expression can potentially be computed using a displacement address mode. An RTL expression is constructed at line 28 and is substituted for the expression. This new instruction is checked to see whether it is a valid machine operation. Whether it is a target machine instruction depends on the addressing modes supported by the target machine and the size of the displacement.

If the previous two alternatives do not succeed, the difference is checked to determine if it is loop invariant. If it is, then the induction expression potentially can be computed by adding the difference to the basic induction variable. If the target machine supports this addressing mode, a calculation is placed in the loop preheader to compute the difference. In the example in Figure 6c, the difference between the starting addresses of a and b is calculated by the instruction at line 1 of Figure 6c. The induction expression is replaced with this register plus register computation.

If one of the alternatives succeeds, then *UPDATED* will be true, and the instruction that used the induction expression will be replaced by the new instruction at line 41. If this induction expression can be calculated from the induction expression selected at line 12, it is removed from the list of induction expressions. If not, it will be handled by a subsequent iteration of the ***while*** loop. That is, a single register will be allocated to be used to induce the sequence. After the algorithm completes, a pass is made over the loop and instruction selection is repeated on all changed instructions. This ensures that the most efficient target machine instructions are used. Again, this is an advantage of applying code improvements to a LIL. This pass also updates use-def chain information.

The algorithm is guaranteed to terminate because the initial set of induced expressions in *LOOP.InductionExprs* is finite and because line 45 removes at least one item from this set each time through the ***while*** loop. Note that *IND* represents one of the elements in the set *EXPR* upon entry to the ***for*** loop at line 18. Consequently, there will be at least one case where the difference value calculated at line 19 will be zero. Thus, even if *UPDATED* is not set for any of the items in the *EXPR* set, the condition in line 44 will be true for at least the item with the zero difference.

Name	Description	Source	Type	Lines
cache	Cache simulation	User code	I/O, Integer	820
compact	Huffman file compression	UNIX utility	I/O, Integer	490
diff	Text file comparison	UNIX utility	I/O, Integer	1,800
eqntott	PLA optimizer	SPEC benchmark	CPU, Integer	2,830
espresso	Boolean expression translator	SPEC benchmark	CPU Integer	14,830
gcc	Optimizing compiler	SPEC benchmark	CPU, Integer	92,630
li	LISP interpreter	SPEC benchmark	CPU, Integer	7,750
linpack	Floating-point benchmark	Synthetic benchmark	CPU, FP	930
mincost	VLSI circuit partitioning	User code	CPU, FP	500
nroff	Text formatting	UNIX utility	I/O, Integer	6,900
sort	File sorting and merging	UNIX utility	I/O, Integer	930
tsp	Traveling salesperson problem	User code	CPU, Integer	450

Table I. Benchmark programs.

An astute reader will note that this algorithm does not necessarily produce the best code sequences for all possible architectures. For example, on a machine with an indexed displacement mode [*base_reg* + *index_reg* + *displacement*], the algorithm would not realize that only one index register needs to be incremented and that the difference between the base address of two arrays is not needed. This deficiency, however, can be easily overcome by adding a test for this addressing mode similar to those at lines 27 and 33. This additional test does not prevent the algorithm from working on architectures that do not have an indexed displacement mode, but produces better code for the machines that do. This extensibility greatly simplifies the task of retargeting the compiler and often allows the effort invested in improving the code for one architecture to be amortized across many machines.

5 Results

A retargetable optimizing C compiler has been constructed with the structure shown in Figure 1b that operates on the LIL described in Section 3.2. The compiler is fully operational for six architectures.† These are:

- VAX-11
- Intel 80386
- MIPS R2000/R3000
- Motorola 68020
- Sun SPARC
- Motorola 88100

To determine the effectiveness of machine-dependent IVE, the SPARC architecture was chosen as the platform to run experiments. A set of experiments were performed using the benchmark programs described Table I. This set of programs includes the four C programs in the SPEC suite [16] along with some common Unix utilities and user code. Together the programs comprise approximately 130,000 lines of source code.

†Actually over ten different architectures have been accommodated, but only these six are maintained.

Program Name	Percent Speedup with Machine-Independent IVE (Column A)	Percent Speedup with Machine-Dependent IVE (Column B)	Column b - Column a (Column C)
cache	-2.39	-0.08	2.31
compact	0.58	3.74	3.16
diff	-3.26	0.56	3.82
eqntott	-4.05	3.68	7.53
espresso	-0.78	-8.51	-7.73
gcc	-1.39	-0.88	0.51
iir	25.30	40.00	14.70
li	-5.20	-0.80	4.40
linpack	-7.43	1.34	8.86
mincost	-1.97	3.49	5.46
nroff	-3.46	0.80	4.26
sort	0.37	4.02	3.65
tsp	3.98	4.27	0.29

Table II. Comparison of the effectiveness of machine-independent and machine-dependent induction variable elimination on the SPARC2.

The first experiment determined the overall effectiveness of IVE. The programs in Table I were compiled with and without IVE enabled. For the runs with IVE enabled the machine-dependent aspects of the algorithm were disabled effectively making it mimic a high-level machine-independent implementation of IVE. The resulting executables were run five times on a lightly loaded SPARC2 and an average execution time was computed. From this average, the speedup due to machine-independent IVE was computed (see the Column A of Table II). Surprisingly, most programs slowed down with machine-independent IVE enabled. Because the effect was most pronounced for *linpack*, the code for this program was examined to determine what was happening. Most of *linpack*'s execution time is spent in *daxpy*. Comparison of the two versions of this loop revealed why machine-independent IVE ran slower. Without IVE, the loop was 9 instructions long. With machine-independent IVE, the loop was also 9 instructions, but the preheader contained instructions that copied the addresses of the arrays to temporaries, and computed the value needed to test against for loop termination $(dx + 400 \times n)$. Because the routine is called tens of thousands of times during the course of a run, the extra overhead lowered performance. For one program, *iir*, machine-independent IVE showed a large benefit. Inspection of this code revealed that this was because IVE produced an opportunity for recurrence detection and optimization [2] to take effect, and a large percent of the benefit was from this improvement. These results confirm that it is difficult to apply code improving transformations to a HIL because the cost/benefit analysis is so dependent on the target machine.

To determine the effectiveness of machine-dependent IVE, the same programs were compiled and run, but this time the machine-dependent aspects of the IVE algorithm were enabled. Column B of Table II shows the speedup when machine-dependent IVE was performed compared to when no IVE was performed. The improvement due to machine-dependent IVE is similar to that reported elsewhere in the literature averaging two or three percent [14].

The one anomaly in Column B is the serious loss of performance for *espresso*. Using a measurement tool called *ease* [9], the execution behavior of the three versions of this program was examined. First, it was observed that several of the routines that were called frequently had loops with very low iteration counts (50% of the loops in these routines had iteration counts of less than two). This explained why IVE was producing poor results. The preheader overhead was not being offset by savings in the loops. However, this did not explain why machine-dependent IVE, with smaller preheader loop overhead, ran slower than machine-independent IVE. The measurement tool revealed that the version of the program produced by compiling the program with machine-dependent IVE performed fewer instructions (less preheader overhead), but more memory references than the version produced by compiling it with machine-independent IVE. Inspection of the optimized loops showed that because the loop was tighter (i.e. fewer instructions), the scheduler had, in order to fill the delay slot of the branch at the end of the loop, resorted to using an annulled branch and had placed a load in the delay slot and replicated it in the preheader. Apparently, these extra (useless) loads caused performance to suffer.

Column C shows the performance difference in machine-independent IVE and machine-dependent IVE. For all but the anomalous *espresso*, performing IVE at a low level where machine-specific information is available appears to be worthwhile, and performs better than machine-independent IVE. Experience with the compiler indicates that other code improvements yield similar benefits when applied at a low-level.

These experiments show, in general, that any single code improvement will affect only a subset of the programs to which it is applied. For some programs the effect will be small and for others it will be large. Thus, a good optimizing compiler uses a collection of code improvements where each transformation produces a small benefit most of the time and a large benefit occasionally. The results also show how difficult it is to measure the effects of a code improvement. Each code improvement can affect what another does and it sometimes difficult to isolate the effect of a single transformation.

To determine the compilation times between a production compiler structured as shown in Figure 1a and one using the structure in Figure 1b, the amount of time spent in the middle end was measured and compared against the amount of time required to perform the entire compilation process. Obtaining these measurements for the benchmark suite shown in Table I revealed that the extra translation step from HIL to LIL increases the compilation time of the compiler by an average of 3.1%. This value ranged from a low of 1.9% for the *linpack* program to a high of 5.5% for *li*. This slight increase in compilation time is the primary disadvantage of using a structure that utilizes both an HIL and a LIL.

6 Summary

To be applied most effectively, most global optimizations require information about the target machine. For those few transformations where this is not true, it is likely that they interact with those that do and thus, effectively, they are also machine dependent. This paper has described the structure of a compiler that is designed so that code improvements can be applied when machine-specific information is available. The compiler has two intermediate representations: one that is a target for intermediate code generation, and a second one that is designed to support the machine-specific application of global code improvements such as code motion, induction variable elimination, and constant propagation.

Using one transformation as an example, this paper showed that it is possible to implement global code improvements that operate on a LIL representation of the program and that it is beneficial to do so. The implementation of the algorithm is itself kept machine-independent by carefully isolating the access to target-specific information via a few routines that can be generated automatically from a specification of the target architecture. The results presented show that the benefits of such a structure are worth the effort in spite the modest compilation time penalty that it incurs.

Acknowledgments

This work was supported in part by National Science Foundation grant CCR-9214904.

References

1. Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, 1986.
2. Benitez, M. E., and Davidson, J. W., "Code Generation for Streaming: an Access/ Execute Mechanism", *Proceedings of the Fourth International Symposium on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991, pp. 132—141.
3. Chow, F. C., *A Portable Machine-Independent Global Optimizer—Design and Measurements*, Ph.D. Dissertation, Stanford University, 1983.
4. Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K., "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph", *ACM Transactions on Programming Languages and Systems*, **13**(4), October 1991, pp. 451—490.
5. Davidson, J. W. and Fraser, C. W., "Register Allocation and Exhaustive Peephole Optimization", *Software—Practice and Experience*, **14**(9), September 1984, pp. 857—866.
6. Davidson, J. W. and Fraser, C. W., "Code Selection Through Peephole Optimization", *Transactions on Programming Languages and Systems*, **6**(4), October 1984, pp. 7—32.

7. Davidson, J. W., "A Retargetable Instruction Reorganizer", *Proceedings of the '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986, pp. 234—241.
8. Davidson, J. W. and Gresh, J. V., "Cint: A RISC Interpreter for the C Programming Language", *Proceedings of the ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, St. Paul, MN, June 1987, pp. 189—198.
9. Davidson, J. W. and Whalley, D. B., "A Design Environment for Addressing Architecture and Compiler Interactions", *Microprocessors and Microsystems*, **15**(9), November 1991, pp. 459—472.
10. Joy, William N. and McKusick, M. Kirk, "Berkeley Pascal PX Implementation Notes Version 2.0—January, 1979", Department of Engineering and Computer Science, University of California, Berkeley, January 1979.
11. McFarling, S., "Procedure Merging with Instruction Caches", *Proceedings of the ACM SIGPLAN '91 Symposium on Programming Language Design and Implementation*, Toronto, Ontario, June 1991, pp. 71—79.
12. Nelson, P. A., "A Comparison of PASCAL Intermediate Languages", *Proceedings of the SIGPLAN Symposium on Compiler Construction*, Denver, CO, August 1979, pp. 208—213.
13. Newey, M. C., Poole, P. C., and Waite, W. M., "Abstract Machine Modelling to Produce Portable Software—A Review and Evaluation", *Software—Practice and Experience*, **2**, 1972, pp. 107—136.
14. Powell, M. L., "A Portable Optimizing Compiler for Modula-2", *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984, pp. 310—318.
15. Richards, M., "The Portability of the BCPL Compiler", *Software—Practice and Experience*, **1**(2), April 1971, pp. 135—146.
16. Systems Performance Evaluation Cooperative, Waterside Associates, Fremont, CA, 1989.
17. Tanenbaum, A. S., Staveren, H. V., and Stevenson, J. W., "Using Peephole Optimization on Intermediate Code", *Transactions on Programming Languages and Systems*, **4**(1), January 1982, pp. 21—36.
18. Tanenbaum, A. S., Staveren, H. V., Keizer, E. G., and Stevenson, J. W., "A Practical Tool Kit for Making Portable Compilers", *Communications of the ACM*, **26**(9), September 1983, pp. 654—660.