Quick Compilers Using Peephole Optimization

JACK W. DAVIDSON AND DAVID B. WHALLEY

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

SUMMARY

Abstract machine modeling is a popular technique for developing portable compilers. A compiler can be quickly realized by translating the abstract machine operations to target machine operations. The problem with these compilers is that they trade execution efficiency for portability. Typically the code emitted by these compilers runs two to three times slower than the code generated by compilers that employ sophisticated code generators. This paper describes a C compiler that uses abstract machine modeling to achieve portability. The emitted target machine code is improved by a simple, classical rule-directed peephole optimizer. Our experiments with this compiler on four machines show that a small number of very general hand-written patterns (under 40) yields code that is comparable to the code from compilers that use more sophisticated code generators. As an added bonus, compilation time on some machines is reduced by 10 to 20 percent.

KEY WORDS: Code generation Compilers Peephole optimization Portability

INTRODUCTION

A popular method for building a portable compiler is to use abstract machine modeling[1-3]. In this technique the compiler front end emits code for an abstract machine. A compiler for a particular machine is realized by constructing a back end that translates the abstract machine operations to semantically equivalent sequences of machine-specific operations. This technique simplifies the construction of a compiler in a number of ways. First, it divides a compiler into two well-defined functional units—the front and back end. Second, by carefully designing the abstract machine to support a single source language, code generation in the front end is often trivial. Third, because the front end is largely machine independent, a compiler for a new machine is realized simply by supplying a new back end.

While abstract machine modeling simplifies the construction of a retargetable compiler, the emitted code is usually substantially inferior to the code produced by a compiler constructed using a sophisticated code generator. This paper describes a portable C compiler that uses abstract machine modeling for portability, and a simple rule-directed peephole optimizer to produce reasonable object code. The compiler has been ported to four machines. The interesting aspect of this compiler is that for all four machines (including a complex machine with a large number of addressing modes and operations), no more than 40 peephole optimization rules were required to produce code that was comparable to the code emitted by compilers that used sophisticated code generators. Furthermore, for three of

the four machines the compiler was 10 to 20 percent faster than the more sophisticated ones.

FRONT END

The front end of the C compiler is called *vpcc* (Very Portable C Compiler)[4]. It supports the full C programming language as defined by Kernighan and Ritchie[5]. *vpcc* emits code for a hypothetical machine called CVM (C Virtual Machine). CVM's design was motivated, to a large extent, by the principles and arguments used to design RISCs[6]. This is most evident by the reduced number of machine operations. CVM contains 46 simple operators. Like P-code for Pascal[2] , CVM was specifically designed for the implementation of C compilers and interpreters. It is *not* a general-purpose abstract machine. A complete description of the C Virtual Machine appears in Appendix I.

Unlike most real RISC machines, CVM is a stack architecture as opposed to a register architecture. The architecture provides instructions for manipulating two distinct stacks. Most instructions manipulate values found on the evaluation stack or E-stack. A second stack, called the C-stack, is used to support the C calling sequence. Only five instructions (`PUSHA`, `PUSHV`, `STARG`, `CALL`, and `RET`) access this stack. The separate stacks permit back ends for pure stack machines or for register machines to be constructed easily. The next section discusses the mapping of the CVM onto a real machine.

Using a small abstract machine language in the front end has a number of advantages[7]. Code generation in *vpcc* is trivial. The reduced nature of the abstract machine means that the code generator can forgo the case analysis typically necessary to emit special case operators. The code is verbose, but simple to produce. The simplicity of the abstract machine also reduces the effort required to map the abstract machine onto the target machine. Because the front end emits code for a virtual machine, retargeting the front end for a new machine requires relatively little effort. A few ''defined'' constants must be changed to reflect the sizes of integers, floating and double precision numbers, and pointers. In addition, conditional compilation is used to select the order of evaluation of arguments to functions that best matches the calling sequence supported by the target machine. Figure 1b shows the CVM code produced by *vpcc* for the simple function in Figure 1a that computes the length of a string.

```
int slen(s)
char *s;
{
   int i = 0;
   while (*s++)
      i++;
   return (i);
}
```

Figure 1a. String length function

```
SEG     1                                               # text segment
FUNC    FTN int slen    EXTDEF                          # begin function slen
DCL     PTR char        s       PARAM   4       1       # declare parameter s
BGNBLK  2                                               # begin block 2
DCL     int     i       AUTO    4       2               # declare local i
BGNSTMT 4                                               # begin source stmt 4
CON     int     SNULL   0                               # push 0
ADDR    int     i       AUTO    2                       # push address of i
=       int                                             # store
EPDEF                                                   # end of procedure prologue code
LABEL   14                                              # label 14
BGNSTMT 5
ADDR    PTR char        s       PARAM   1               # push address of s
@       PTR char                                        # dereference s
DUP     PTR char                                        # duplicate top of stack
CON     int     SNULL   1                               # push 1
VCONV   int     PTR char                                # convert to pointer
+       PTR char                                        # add
ADDR    PTR char        s       PARAM   1               # push address of s
=       PTR char                                        # store
@       char                                            # dereference
CON     char    SNULL   0                               # push 0
JEQ     char    15                                      # if equal jump to label 15
BGNSTMT 6
ADDR    int     i       AUTO    2                       # push address of i
@       int                                             # dereference i
CON     int     SNULL   1                               # push 1
+       int                                             # add
ADDR    int     i       AUTO    2                       # push address of i
=       int                                             # store
GOTO    14                                              # jump to label 14
LABEL   15                                              # label 15
BGNSTMT 7
ADDR    int     i       AUTO    2                       # push address of i
@       int                                             # dereference i
RETURN  int                                             # return value to caller
EFUNC   4                                               # end of function
```

Figure 1b. Example of CVM code emitted by vpcc for code in Figure 1a

BACK END

The back end translates the CVM operators to machine-specific assembly language. Because each CVM operator represents, in most cases, a simple operation, the mapping from CVM instructions to target machine instructions is usually trivial. There are two strategies that can be used to map CVM code onto the target machine. One strategy

is to map both the E-stack and C-stack of the CVM onto the run-time stack of the target machine. A back end for the Western Electric 32100 was constructed using this approach[8]. There are a couple of advantages to this strategy. First, the mapping of the CVM operators to target machine instructions is straightforward. Second, this approach does not require a register allocator. The disadvantage is that for register machines the resulting code is inefficient because all references to the E-stack result in target machine code that references memory.

For machines with an adequate number of allocable registers (more than four), the second strategy is to map the E-stack onto the target machine's registers. While the mapping is a bit more difficult, the code generated is substantially more efficient. Figure 2 shows the assembly code produced for the string length function by a VAX-11 back end that uses this strategy.

```
        .text                   /* change to text segment */
        .align 1                /* align at next byte */
        .globl _slen            /* declare _slen as global */
_slen:  .word slen.r            /* mask containing registers used in slen */
        subl2 $slen.,sp         /* adjust stack pointer for locals in slen */
        .set s.,4               /* define symbol s. to be 4 */
        .set i.,-4              /* define symbol i. to be -4 */
        /* BGNSTMT 4 */
        movl $0,r0              /* load 0 in r0 */
        moval i.(fp),r1         /* load address of i. in r1 */
        movl r0,(r1)            /* store r0 in location pointed at by r1 */
L14:    /* BGNSTMT 5 */
        moval s.(ap),r0         /* load address of s. in r0 */
        movl (r0),r0            /* load value of s. in r0 */
        movl r0,r1              /* move r0 to r1 */
        movl $1,r11             /* load 1 in r11 */
        addl2 r11,r1            /* add r11 to r1 */
        moval s.(ap),r11        /* load address of s. in r11 */
        movl r1,(r11)           /* store r1 in location pointed at by r11 */
        movb (r0),r0            /* load byte at location pointed to by r0 to r0 */
        movb $0,r1              /* load a 0 byte in r1 */
        cmpb r0,r1              /* compare bytes in r0 and r1 */
        jeql L15                /* if they are equal jump to L15 */
        /* BGNSTMT 6 */
        moval i.(fp),r0         /* load address of i. in r1 */
        movl (r0),r0            /* load value of i in r0 */
        movl $1,r1              /* load 1 in r1 */
        addl2 r1,r0             /* add r1 to r0 */
        moval i.(fp),r1         /* load address of i. in r1 */
        movl r0,(r1)            /* store r0 in location pointed to by r1 */
        jbr L14                 /* jump to L14 */
L15:    /* BGNSTMT 7 */
        moval i.(fp),r0         /* load addr i. in r0 */
        movl (r0),r0            /* load value of i in r0 */
        ret                     /* return to caller */
L13:    ret                     /* return to caller */
        .set slen.,4            /* define symbol slen. to be 4 */
        .set slen.r.,0x800      /* define register save mask to save r11 */
```

Figure 2. Assembly code emitted by VAX-11 back end

This back end maps the CVM's E-stack onto registers `r11` through `r6`, and `r0` and `r1`. For this function, only three registers (`r0`, `r1`, and `r11`) were required to compile the function. The back end uses registers `r0` and `r1` when possible as they do not need to be saved across function calls. If `r0` or `r1` are alive when a call is encountered their contents are copied to a register that will be saved. If possible, the peephole optimizer will replace the load of `r0` or `r1` and the copy instruction with a load directly into the destination register of the copy instruction. There is the possibility that the depth of the E-stack will exceed the number of registers available. If this occurs, the back end issues an error message that identifies the C source statement that caused the problem. This permits the offending statement to be split into two simpler statements. In practice, exceeding the number of available registers rarely occurs. During more than a year of use, the VAX-11 back end has never run out of registers. We have used the strategy above on a number of machines with success. These include machines with different types of registers (e.g. Motorola 68000), and machines where the registers have special uses (e.g. Intel 8086).

By changing the implementation of the CVM operators that manipulate the E-stack, other strategies are possible. For example, if the target machine had a small number of registers, but supports operations that permit operands to be either registers or memory references, the back end could easily use the run-time stack when the registers were exhausted. Similarly one could use a dedicated area in the activation record to simulate the E-stack. The choice of implementation depends on the target machine characteristics and the effort one is willing to invest in implementing the appropriate CVM instructions.

A back end consists of a set of 46 functions, one for each CVM instruction. To retarget the compiler for a new machine, each of these functions must be modified to emit assembly language that performs the indicated function. For most machines, the modification is as simple as changing the *sprintf* statements that construct the assembly code statements.

PEEPHOLE OPTIMIZER

Using the technique of emitting code for an abstract machine and expanding it into assembly language for a particular machine yields a compiler that is quickly and easily retargeted for a new machine. There are, however, two serious problems. One is that the code produced by such a compiler is quite slow. For example, the code produced by *vpcc* and a back end for the VAX-11 runs 1½ to 2 times slower than the code produced by the Berkeley 4.3BSD

C compiler. The second is that the compiler runs slowly. This is because the assembly phase of the compiler takes a long time due to the verbosity of the emitted code. The slow compilation times are particularly serious if the compiler is to be used by students in which case fast compilation is often more important than fast execution. To solve these problems, a simple, rule-directed peephole optimizer was constructed to improve the naive machine code emitted by the compiler. The optimizer operates by applying a set of ‘‘optimization’’ rules to the generated code.

Optimization Rules

An optimization rule consists of two parts; a pattern to match and a pattern to replace the matched portion of the input. For example, the following rule for the VAX-11

```
"moval  %1,r%2",        /* pattern to match */
"movl   (r%2),%3"
=>
"movl   %1,%3"          /* pattern to replace */
```

replaces a load-address and load-indirect instruction with a single equivalent instruction. The `%i` notation, where `i` is a digit, specifies a pattern that matches everything up to the occurrence of the next pattern character. In addition, a limited form of regular expressions can be used with the `%i` notation. For example, `%i[bwld]` specifies a pattern that matches one of the single characters ‘b’, ‘w’, ‘l’, or ‘d’. All occurrences of `%i` must denote the same string. For instance, in the above example the `%2` in the first line of the pattern and the `%2` in the second line of the pattern must match the same string.

A rule can also include a semantic action that performs some check on or conversion of the input. In the rule

```
"j%0    L%1",
"jbr    L%2",
"L%1:"          invert(%0)
=>
"j%0    L%2",
"L%1:"
```

`invert` reverses the sense of the condition specified by the branch. For example, the rule above would transform the VAX-11 code sequence

```
        ...
        jeql    L1
        jbr     L2
L1:
        ...
```

to

```
        ...
        jne     L2
L1:
        ...
```

The ability to include simple semantic functions as part of a rule often permits a single rule to handle several cases.

In the previous example, the rule handles all six conditional branches.

Applying Rules

Each assembly language instruction emitted by the back end is inserted at the end of a doubly-linked list. After an instruction is added, the back end invokes the peephole optimizer. All matching and replacements occur at a point in the linked list marked by the ‘‘cursor’’. The individual lines of the pattern-to-match portion of a rule are matched against the input in reverse order. For example, when the VAX-11 rule

```
"movl   %1,r%2",
"addl2  r%2,%3"
=>
"addl2  %1,%3"
```

is applied, the `addl2` portion of the rule is matched against the instruction at the cursor. If the pattern matches, matching continues with the previous line of the rule and the previous instruction in the linked list. If a failure occurs at any point during pattern matching, the next rule in the list of rules, if any, is applied. If no rules match, control returns to the back end which inserts the next instruction at the end of the linked-list, advances the cursor, and the optimizer is reinvoked.

If the entire pattern portion of a rule matches the input, the instructions at the cursor that correspond to the pattern-to-match portion of the rule are deleted, and the replacement pattern with all `%i`’s instantiated are inserted in the linked list. The cursor is set to point to the instruction that corresponds to the last line of the replacement pattern and matching continues by applying the first rule again. This permits any new patterns introduced by the replacement to be optimized.

For the previous string length function, Figure 3 shows the code produced by *vpcc* with the peephole optimizer enabled. In terms of instruction selection, the emitted code cannot be improved substantially.

```
        .text                       /* change to text segment */
        .align 1                    /* align at next byte */
        .globl _slen                /* declare _slen as global */
_slen:  .word slen.r                /* mask containing registers used */
        subl2 $slen.,sp             /* adjust stack pointer for locals */
        .set s.,4                   /* define symbol s. to be 4 */
        .set i.,-4                  /* define symbol i. to be -4 */
        /* BGNSTMT 4 */
        clrl i.(fp)                 /* store 0 in i. */
L14:    /* BGNSTMT 5 */
        movl s.(ap),r0              /* load s. in r0 */
        addl3 $1,r0,s.(ap)          /* add 1 to r0 and store back in s. */
        tstb (r0)                   /* test value at location pointed to by r0 */
        jeql L15                    /* if equal to zero jump to L15 */
        /* BGNSTMT 6 */
        incl i.(fp)                 /* add 1 to i. */
        jbr L14                     /* jump to L14 */
L15:    /* BGNSTMT 7 */
        movl i.(fp),r0              /* load i. into r0 */
        ret                         /* return to caller */
L13:    ret                         /* return to caller */
        .set slen.,4                /* define symbol slen. to be 4 */
        .set slen.r,0x000           /* define save mask to save no registers */
```

Figure 3. Code produced by vpcc for a VAX-11

Compiling Rules

The peephole optimizer was originally implemented as an interpreter. The interpretative application of the rules to the input, however, resulted in compilers that ran too slow. Consequently, a translator that compiles rules into C code was implemented. This simple change resulted in a ninefold reduction in the CPU time required to optimize a program.

The compilation of a rule to C code that applies the pattern to the input is straightforward. The following VAX-11 rule that optimizes a move of zero to a clear instruction

```
"mov%0 $0,%1"
=>
"clr%0 %1"
```

is compiled into

```
pat0:
   f = t;
   p = f->line;
   if (*p++ != 'm' || *p++ != 'o' || *p++ != 'v')
      goto pat1;       /* try next rule */
   for (s = arg[0]; (*s++ = c = *p++) && c != ' '; )
      ;
   *(s-1) = '\0';
   if (c != ' ' || *p++ != '$' || *p++ != '0' || *p++ != ',')
      goto pat1;       /* try next rule */
   for (s = arg[1]; *s++ = *p++; )
      ;
   /* make replacement */
   f = t;
   p = f->line;
   *p++ = 'c';
   *p++ = 'l';
   *p++ = 'r';
   for (s = arg[0]; *p++ = *s++; )
      ;
   *(p-1) = ' ';
   for (s = arg[1]; *p++ = *s++; )
      ;
   goto pat0;  /* rule successful, start from beginning */
pat1:
   /* code for subsequent rules follow */
```

Simple character comparisons are used to match portions of the pattern that contain fixed characters. The `%i` portions of a pattern are matched by scanning forward for the character that follows the `%i`. The scanned characters are copied into the array `arg[i]`. This permits these strings to be matched against the input or used in the replacement pattern. In the example above, the substrings matching `%0` and `%1` are copied into `arg[0]` and `arg[1]` respectively which are used the create the replacement instructions.

In the above example, if the rule does not match the next rule is applied. In practice, this is often unnecessary. Consider the following rule for the Motorola 68020.

```
"lea a%0@(%1),a%2"
"movl a%2@,%3"
=>
"movl a%0(%1),d%4"
```

This rule replaces a load effective address instruction followed by a load indirect with a load direct instruction. If during matching the '`m`' of the `movl` pattern fails to match the input, all subsequent rules where the first pattern begins with '`m`' will also fail. Rather than waste time attempting to match these rules, the rule compiler determines the next rule to attempt when a rule fails. On the VAX, measurements of several benchmark programs revealed that on average 3.87 rules are attempted for a successful optimization, while 8.18 rules out of a possible 39 were tried on an unsuccessful attempt to perform an optimization. These numbers indicate that the compile-compile time analysis performed by the rule compiler is worthwhile.

On the VAX, the 39 optimization rules were compiled into a C routine consisting of 6025 lines which resulted in an object file of 20816 bytes. Overall, the sizes of the text, data, and bss segments of the compiler are 92160, 25600, and 66400 bytes respectively, resulting in a total size of 184160 bytes. The system C compiler's text, data, and bss segments are 88064, 15360, and 158944 bytes respectively, resulting in a total size of 262368 bytes. On the Sun-3/75, the 40 68020 optimization rules resulted in a 7277 line C program that was compiled into a 21516 byte object file. Overall, the text, data, and bss sections of the compiler are 114688, 32768, and 72168 bytes respectively resulting in a total size of 218724 bytes. The Sun-3/75 C compiler text, data, and bss sections measured 155648, 32768, and 67268 bytes respectively. The total size of this compiler was 255684 bytes. Compiling optimization rules into code does not seem to seriously affect the size of the compiler.

RESULTS

During the course of experimenting with *vpcc* and the optimizer on a VAX-11, it was discovered that a compiler using a surprisingly small number of rules (39) was able to generate code that was as good as the code generated by the standard C compiler which incorporates a more sophisticated code generator. This compiler, *pcc*, uses a tree-matching algorithm to produce assembly code for the target machine[9,10]. Table 1 contains data comparing the compilation and execution times of these compilers. In order to compare only the code generators, in these runs as well as the runs reported in Table 2, the optimization phases of the vendor-supplied compilers (-O option for the *pcc*-based compilers) was disabled. In analyzing these results, our first thought was that the orthogonality and regularity of the VAX-11 instruction set and assembly language allowed a small number of rules to capture a large percentage of the optimizations necessary to produce good code. To test whether this was true or not, C compilers for three additional machines were constructed using the same approach. The new machines used were the Sun-3/75 (Motorola 68020; 40 rules), the Tektronix Workstation (National Semiconductor 32032; 33 rules), and an AT&T 6300 (Intel 8086; 22 rules). Table 1 also compares these ports of *vpcc* with the C compilers from the vendors. Appendix II contains the complete rule set for the Tektronix Workstation. The Sun-3/75 C compiler is the one distributed with SUNOS 3.0. The Tektronix C compiler is distributed with UTek, while the compiler used on the AT&T 6300 is the Microsoft C compiler Version 4.0. The Sun and Tektronix compilers appear to be derivatives of *pcc*. Its code generation algorithms are well documented[9,10]. The Microsoft C compiler uses a code generator that

was developed in-house[11].

	Program	Compile Time			Execution Time		
		cc	vpcc	vpcc/cc	cc	vpcc	vpcc/cc
VAX-11	puzzle	8.5	7.4	0.87	10.45	10.13	0.97
	grep	13.2	12.3	0.93	5.60	6.00	1.07
	matmult	2.7	2.6	0.96	3.42	3.83	0.89
	od	18.9	17.5	0.93	7.30	7.10	0.97
	merge	3.0	2.9	0.97	19.63	19.26	0.98
	qsort	5.3	4.2	0.79	0.95	0.95	1.00
	bubble	2.5	2.4	0.96	8.33	8.57	1.03
	Average			0.92			0.99
Sun3/75	puzzle	4.9	3.5	0.71	5.40	5.18	0.96
	grep	6.3	5.4	0.86	3.30	3.40	1.03
	matmult	1.8	1.3	0.72	2.22	2.35	1.07
	od	9.6	8.1	0.84	3.60	3.50	0.97
	merge	2.2	1.3	0.59	10.93	9.95	0.91
	qsort	3.3	2.2	0.67	0.53	0.50	0.94
	bubble	1.7	1.2	0.71	4.55	4.80	1.05
	Average			0.73			0.99
Tektronix	puzzle	17.7	13.1	0.77	27.20	24.15	0.89
	grep	20.9	19.1	0.91	8.80	8.90	1.01
	matmult	5.4	4.7	0.87	11.80	11.20	0.95
	od	32.2	29.2	0.91	15.30	15.40	1.01
	merge	6.0	4.7	0.78	58.48	59.41	1.02
	qsort	11.1	8.7	0.78	2.50	2.48	0.99
	bubble	4.6	4.6	1.00	23.25	23.75	1.02
	Average			0.86			0.98
AT&T 6300	puzzle	34.6	30.4	0.88	20.6	17.4	0.84
	grep	45.4	46.8	1.03	3.7	3.9	1.05
	matmult	16.2	16.8	1.04	9.4	10.1	1.07
	od	60.6	57.2	0.94	56.3	47.3	0.84
	merge	17.8	17.6	0.99	8.1	7.3	0.90
	qsort	24.3	23.5	0.97	2.8	3.0	1.07
	bubble	16.8	17.0	0.96	54.6	49.9	0.91
	Average			0.97			0.95

Table 1. Comparison of Compilation and Execution Times

The programs used to produce Table 1 were the ones that were used to determine the optimization rules for each machine. To determine if the rules deduced from the training set were sufficient, a second set of programs were

selected and benchmarked. The execution times of these programs and comparisons against the native C compilers are presented in Table 2. These numbers indicate that the small number of rules deduced from the benchmark programs seem sufficient to produce a relatively good C compiler.

VAX-11

Program	Execution Time		
	cc	vpcc	vpcc/cc
cache	15.3	14.9	0.97
lex	49.8	53.3	1.07
keyword	2.0	1.9	0.95
patgen	5.8	5.9	1.02
m4	2.3	2.2	0.96
dhryst	35.4	34.2	0.97
mincost	48.2	47.6	0.99
acker	5.4	5.1	0.94
sieve	3.0	3.2	1.07
Average			0.99

Sun3/75

Program	Execution Time		
	cc	vpcc	vpcc/cc
cache	8.3	7.8	0.94
lex	26.5	28.9	1.09
keyword	1.9	2.1	1.11
patgen	3.5	3.4	0.97
m4	1.2	1.1	0.92
dhryst	23.0	20.6	0.90
mincost	26.9	23.8	0.88
acker	1.9	1.1	0.58
sieve	1.7	1.8	1.06
Average			0.85

Tektronix

Program	Execution Time		
	cc	vpcc	vpcc/cc
cache	34.5	34.5	1.00
lex	94.4	96.1	1.02
keyword	4.2	4.5	1.07
patgen	14.5	14.0	0.97
m4	3.7	3.8	1.03
dhryst	78.7	78.3	0.99
mincost	94.8	91.2	0.96
acker	6.6	6.1	0.92
sieve	7.58	7.73	1.01
Average			1.00

AT&T 6300

Program	Execution Time		
	cc	vpcc	vpcc/cc
cache	28.9	29.2	1.01
lex	137.0	148.9	1.09
keyword	4.7	4.6	0.98
patgen	10.3	10.2	0.99
m4	2.7	2.6	0.97
dhryst	29.3	31.3	1.07
mincost	259.2	259.0	1.00
acker	6.9	7.2	1.04
sieve	2.6	2.3	0.88
Average			1.00

Table 2. Comparison of Execution Times

The code produced by the *vpcc*-based compiler and peephole optimizer could be further improved if there were some mechanism for combining instructions that were not physically adjacent. Our examinations of the emitted code and previous studies[7] have shown that this would result in further improvements in the emitted code and faster compilation times. In order to perform such optimizations, the optimizer would need to perform some flow analysis as well as identify dead variables. The extra mechanisms required did not seem consistent with our goal of a quick, simple way to produce a reasonable compiler.

DISCUSSION

Caveats

Building a compiler using the techniques described here is a straightforward way to implement a fast compiler that produces reasonably good code. There are a number of situations where such compilers can be used. In an educational environment, compilation speed is more important than execution speed. Further, educational settings are characterized by a wide variety of hardware and budgets for the purchase of software are limited. The techniques described in this paper are also useful when experimenting with new languages or developing new machines. The ability to quickly build a compiler permits experiments to be performed or software to be quickly bootstrapped onto the new machine. It is *not* the approach to use to build a highly optimizing compiler.

While the approach is both conceptually and in practice simple, there are a few caveats that should be mentioned. One problem is that optimization rules can sometimes interact resulting in the production of poor code. These interactions can be avoided by a simple modification of certain rules. For example, on some machines a typical interaction is caused by a rule to convert a move of constant zero to a clear instruction and other rules involving move instructions. Consider the VAX-11 rules

```
"mov%0 $0,%1"
=>
"clr%0 %1"
```

and

```
"mov%0 %1,r%2",
"mov%0 r%2,%3"
=>
"mov%0 %1,%3"
```

If the following instructions

```
movl $0,r3
movl r3,i.(fp)
```

are emitted by the code generator, the move of zero is optimized into a clear instruction (recall that all optimizations take place at the end of the linked list). When the second instruction is emitted and added to the linked list and the peephole optimizer invoked, the rule above that would seem to apply is no longer applicable.

There are two solutions to this problem. The obvious solution is to add extra rules to handle these special situations. For example, the rule

```
"clr%0 r%1",
"mov%0 r%1,%2"
=>
"clr%0 %2"
```

would handle the previous situation. Our solution, which requires no extra rules, is to delay some optimizations by forcing them to not match until more context is available. The previous situation can be handled by specifying the rule to handle clear instructions as

```
"mov%0 $0,%1",
"%2"
=>
"clr%0 %1",
"%2"
```

and placing this rule after the one that optimizes move instructions. The clear rule cannot be matched until the second instruction is seen, which permits any rules listed first to be applied.

To help determine what actions the optimizer is taking, the rule compiler can be directed to emit code in the rule matching routines that records the number of times a rule is successfully applied. This information can be used to identify rules that are not being used, and it is also useful for ordering the rules.

A second, but less serious, problem with using this approach is that it is often tempting to spend too much time sifting through the emitted code looking for new rules. Our experience is that this is done more for amusement than necessity. There is a certain satisfaction in discovering new and clever optimization rules even though they may be applied infrequently. The most useful optimization rules are generally obvious and can be determined by using knowledge of how the code generator emits code.

Comparison with Other Work

Lamb[12] describes a peephole optimizer similar to the one described here. This optimizer was used to improve the code emitted by a prototype Ada® compiler that generated code for the VAX-11. The optimization rules were compiled into a set of subroutines that matched the various pattern portions of a rule. On the VAX, there were 53 rules that resulted in 237 subroutines. He noted that extra rules were often necessary to handle similar situations. For example, the VAX optimizer required several rules to handle the various types of clear instructions. Our language for specifying rules handles this type of situation simply and efficiently. While a production version of Lamb's optimizer was never constructed, it was felt that the optimizer could be made to run quite fast. Our experi-

®Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

ment seems to confirm this.

Davidson and Fraser[13] describe a fast rule-directed peephole optimizer where the rules are inferred automatically by running a training set through a peephole optimizer that is driven by a machine description. The advantage to this approach is that the rules are derived automatically. The disadvantage is that essentially two compilers must be built; one that operates using the machine description and one that uses the rules. The second one, however, is constructed automatically once the first one is completed. The final result is a thorough, fast peephole optimizer.

In the Amsterdam compiler kit[3] , two peephole optimizers are used to replace a traditional code generator. In a manner similar to the compiler described here, the front end emits code for an abstract machine. This machine, called EM (Encoding Machine), models a stack machine and includes a number of special-case operations (e.g., increment, decrement, clear, etc.). The EM code is improved by a peephole optimizer that replaces inefficient sequences of EM instructions with more efficient ones. This optimizer is driven by a table of pattern-replacement pairs much like the rules described here. Tanenbaum reports the use of 400 such rules in the EM peephole optimizer. The improved abstract machine code is processed by a global optimizer that performs a number of optimizations that require a larger view of the program's structure[14]. A back end translates the optimized abstract machine code to target-machine instruction sequences. A second, machine-specific peephole optimizer improves the target machine assembly code. This peephole optimizer is similar to the one that operates on EM code. The number of pattern-replacement pairs required for various machines is not reported.

SUMMARY

This paper has described a quick technique for building a simple, yet fast compiler. The compiler's code generator is implemented using a simple rule-directed peephole optimizer. Optimization rules are written in a simple notation. For the four machines studied, it was found that a small number of optimization rules were sufficient to construct compilers that produced code that compared favorably to the code emitted by compilers that used more sophisticated code generation techniques. As an added bonus, the simple compilers were smaller and generally 10 to 20 percent faster than their more complicated counterparts.

ACKNOWLEDGEMENTS

Nelson T. Fung implemented the 68020 code generator. Jonathan Watts did the original implementation of *vpcc*.

APPENDIX I

C Virtual Machine Instruction Set

Arithmetic Operators	Description
+, -, *, /, %	v1←pop; v2←pop; push(v1 ***op*** v2);
<<, >>, &, ^, |	v1←pop; v2←pop; push(v1 ***op*** v2);

Unary Operators	Description
-, ~	v←pop; push(***op*** v);
FLD ***n***	v←pop; push(extract_field(v, ***n***));
IFLD ***n***	v←pop; push(insert_field(v, ***n***));

Conversion Operators	Description
PCONV ***ot nt***	Convert pointer of type ***ot*** to pointer of type ***nt***.
VCONV ***ot nt***	Convert value of type ***ot*** to type ***nt***.

Data Movement	Description
CON ***con***	push(***con***);
ADDR ***id class level***	push(addr(***id***));
@	addr←pop; push(m[addr]);
=	addr←pop; v←pop; m[addr]←v
DUP	v←pop; push(v); push(v)
SWAP	v1←pop; v2←pop; push(v1); push(v2)
ROTATE	v1←pop; v2←pop; v3←pop; push(v2); push(v1); push(v3)
STASG ***n***	dst←pop; src←pop; strncpy(dst, src, ***n***);

Program Control and Jump	Description
CALL ***nargs argsize***	addr←pop; push(environ); push(retaddr); pc←addr;
SWITCH ***s r***	Switch stmt with starting value ***s*** and ***r*** consecutive values.
GOTO ***n***	Jump to label ***n***.
JEQ ***n***	v1←pop; v2←pop; pc←v1 == v2 ? ***n*** : pc;
JNE ***n***	v1←pop; v2←pop; pc←v1 != v2 ? ***n*** : pc;
JLT ***n***	v1←pop; v2←pop; pc←v1 < v2 ? ***n*** : pc;
JLE ***n***	v1←pop; v2←pop; pc←v1 <= v2 ? ***n*** : pc;
JGT ***n***	v1←pop; v2←pop; pc←v1 > v2 ? ***n*** : pc;
JGE ***n***	v1←pop; v2←pop; pc←v1 >= v2 ? ***n*** : pc;
FUNC ***name class***	Define start of function.
RET	v←pop; addr←pop; pop(environ); push(v); pc←addr;

Argument Transmission	Description
PUSHA	addr←pop; pass(addr);
PUSHV	v←pop; pass(v)
STARG ***n***	addr←pop; strncpy(v,addr,n); pass(v);

Pseudo Operations	Description
BGNSTMT ***n***	Begin code for statement ***n***.
FILE ***name***	Code was generated from source file ***name***.
EFUNC ***n***	End function that required ***n*** bytes for locals.
EPDEF	End of prologue code for a procedure.
DCL ***id class n level***	Define variable ***id***. It requires ***n*** bytes.
DC ***value***	Initialize a memory location with ***value***.
LABEL ***n***	Generate local label ***n***.
SEG ***n***	Signal start of segment ***n***.

Notes:

1. Each executable opcode is followed by a type indicator.
2. ***Class*** denotes the scope of the variable (i.e., local, global, etc.).

APPENDIX II

The rule set for the Tektronix Workstation (National Semiconductor 32032).

```
1.      /* load */
        "       addr %0,r%1",                                   /* addr i.(fp),r1 */
        "       mov%2 (r%1),%3"                                 /* movd (r1),r3 */
        =>
        "       mov%2 %0,%3";                                   /* movd i.(fp),r3 */

2.      /* store */
        "       addr %0,r%1",                                   /* addr i.(fp),r1 */
        "       mov%2 %3,(r%1)"                                 /* movd r3,(r1) */
        =>
        "       mov%2 %3,%0";                                   /* movd r3,i.(fp) */

3.      /* store address */
        "       addr %0,r%1",                                   /* addr i.(fp),r1 */
        "       movd r%1,%2"                                    /* movd r1,j.(fp) */
                        isvar(%2)
        =>
        "       addr %0,%2";                                    /* addr i.(fp),j.(fp) */

4.      /* register relative load */
        "       add%0[q\0]%1 $%2,r%3",                          /* addd $4,r3 */
        "       mov%4 %5(r%3),%6"                               /* movd (r3),r5 */
                        offsetexpr(%5,%7)
        =>
        "       mov%4 %2%7(r%3),%6";                            /* movd 4(r3),r5 */

5.      /* register relative store */
        "       add%0[q\0]%1 $%2,r%3",                          /* addd $4,r3 */
        "       mov%4 %5,%6(r%3)"                               /* movd r5,(r3) */
                        offsetexpr(%6,%7)
        =>
        "       mov%4 %5,%2%7(r%3)";                            /* movd r5,4(r3) */

6.      /* mov to var */
        "       mov%0[q\0]%1 %2,%3[rf]%4",                      /* movd $5,r4 */
        "       mov%5 %3[rf]%4,%6"                              /* movd r4,i.(fp) */
                        movtovar(%2,%6)
        =>
        "       mov%5 %2,%6";                                   /* movd $5,i.(fp) */

7.      /* convert to var */
        "       mov%0[xz\0]%1[dlbfw]%2[dlbfw] %3,%4[rf]%5",     /* movbd r3,r5 */
        "       mov%2[dlbfw] %4[rf]%5,%6"                       /* movd r5,i.(fp) */
                        isvar(%6)
        =>
        "       mov%0%1%2 %3,%6";                               /* movbd r3,i.(fp) */

8.      /* convert */
        "       mov%0 %1,%2[rf]%3",                             /* movb r1,r3 */
        "       mov%4[xz\0]%5[dlbfw]%6[dlbfw] %2[rf]%3,%7"      /* movbd r3,r7 */
        =>
        "       mov%4%5%6 %1,%7";                               /* movbd r1,r7 */

9.      /* memory relative load */
        "       mov%0[dbw] %1(fp),r%2",                         /* movd i.(fp),r2 */
        "       mov%3 %4(r%2[0-7]),%5"                          /* movd 4(r2),r5 */
        =>
        "       mov%3 %4-0(%1(fp)),%5";                         /* movd 4-0(i.(fp)),r5 */
```

```
10.     /* memory relative store */
        "       mov%0[dbw] %1(fp),r%2",                         /* movd i.(fp),r2 */
        "       mov%3 %4,%5(r%2)"                               /* movd r5,4(r2) */
        =>
        "       mov%3 %4,%5-0(%1(fp))";                         /* movd r5,4-0(i.(fp)) */

11.     /* load, binary operation, store */
        "       mov%0 %1,%2[rf]%3",                             /* movd r2,r4 */
        "       %4 %5,%2[rf]%3",                                /* addd r7,r4 */
        "       mov%0 %2[rf]%3,%1"                              /* movd r4,r2 */
                        typematch(%0,%4) && isbinoper(%4)
        =>
        "       %4 %5,%1";                                      /* addd r7,r2 */

12.     /* add to register as address */
        "       add%0[q\0]d %1[$\0]%2,r%3",                     /* addd r1,r3 */
        "       movd r%3,%4"                                    /* movd r3,i.(fp) */
                        (ISCONST(%1) || isreg(%2)) && isvar(%4)
        =>
        "       addr %2[b‘r%3],%4";                             /* addr r1[b‘r3],i.(fp) */

13.     /* special case for putchar */
        "       mov%0 %1,r%2",                                  /* movd r1,r2 */
        "       movb r%2,r%3",                                  /* movb r2,r3 */
        "       addr _%4,r%5",                                  /* addr _i,r5 */
        "       movd (r%5),r%6",                                /* movd (r5),r6 */
        "       movd r%6,r%7",                                  /* movd r6,r7 */
        "       addr 1[b‘r%7],(r%5)",                           /* addr 1[b‘r7],(r5) */
        "       movb r%3,(r%6)",                                /* movb r3,(r6) */
        "       movxbd r%2,r%2"                                 /* movxbd r2,r2 */
        =>
        "       movd _%4,r%2",                                  /* movd _i,r2 */
        "       addqd $1,_%4",                                  /* addqd $1,_i */
        "       movb %1,(r%2)",                                 /* movb r1,(r2) */
        "       movxbd (r%2),r%2";                              /* movxbd (r2),r2 */

14.     /* move quick */
        "       mov%0[dbw] $%1[-\0]%2[0-7],%3"                  /* movd $0,r3 */
        =>
        "       movq%0 $%1%2,%3";                               /* movqd $0,r3 */

15.     /* delete useless assignment */
        "%0",
        "       mov%1[dlbfw] %2,%2"                             /* movd r2,r2 */
        =>
        "%0";

16.     /* compare */
        "       mov%0[q\0]%1 %2,%3[rf]%4",                      /* movd r2,r4 */
        "       cmp%1 %3[rf]%4,%5"                              /* cmpd r4,r5 */
        =>
        "       cmp%0%1 %2,%5";                                 /* cmpd r2,r5 */

17.     /* compare */
        "       mov%0[q\0]%1 %2,%3[rf]%4",                      /* movd r2,r4 */
        "       cmp%5[q\0]%1 %6,%3[rf]%4"                       /* cmpd r6,r4 */
        =>
        "       cmp%5%1 %6,%2";                                 /* cmpd r6,r2 */

18.     /* special case for getchar and putchar */
        "       addr _%1,r%2",                                  /* addr _i,r2 */
        "       movd (r%2),r%3",                                /* movd (r2),r3 */
        "       addqd $-1,r%3",                                 /* addqd $-1,r3 */
        "       movd r%3,(r%2)",                                /* movd r3,(r2) */
```

```
"       cmpd r%3,$0"                                    /* cmpd r3,r0 */
=>
"       addqd $-1,_%1",                                 /* addqd $-1,_i */
"       cmpd _%1,$0";                                   /* cmpd _i,$0 */

19.     /* compare quick */
"       cmp%0[dbw] $%1[-\0]%2,%3"                       /* cmpd $1,r3 */
=>
"       cmpq%0 $%1%2,%3";                               /* cmpqd $1,r3 */

20.     /* call */
"       addr %0,r%1",                                   /* addr _foo,r1 */
"       jsr r%1"                                        /* jsr r1 */
=>
"       bsr %0";                                        /* bsr _foo */

21.     /* shift index register */
"       ashd $%0[123],r%1",                             /* ashd $2,r1 */
"       addr %2\[b'r%1],%3"                             /* addr _i[b'r1],r3 */
                        indexchr(%0,%4)
=>
"       addr %2[%4'r%1],%3";                            /* addr _i[d'r1],r3 */

22.     /* add reg to address */
"       addr %0,r%1",                                   /* addr _i,r1 */
"       addd r%1,r%2"                                   /* addd r1,r2 */
                        notindex(%0)
=>
"       addr %0[b'r%2],r%2";                            /* addr _i[b'r2],r2 */

23.     /* index register */
"       ashd $%0[123],r%1",                             /* ashd $2,r1 */
"       addd r%1,r%2"                                   /* addd r1,r2 */
                        indexchr(%0,%3)
=>
"       addr r%2[%3'r%1],r%2";                          /* addr r2[d'r1],r2 */

24.     /* add offset to addr */
"       addr _%0,r%1",                                  /* addr _i,r1 */
"       addd $%2,r%1"                                   /* addd $4,r1 */
                        notindex(%0)
=>
"       addr _%0+%2,r%1";                               /* addr _i+4,r1 */

25.     /* add quick */
"       add%0[dbw] $%1[-\0]%2[0-7],%3"                  /* addd $1,r3 */
=>
"       addq%0 $%1%2,%3";                               /* addqd $1,r3 */

26.     /* special case for getchar */
"       addr _%0,r%1",                                  /* addr _i,r1 */
"       movd (r%1),r%2",                                /* movd (r1),r2 */
"       movd r%2,r%3",                                  /* movd r2,r3 */
"       addr 1[b'r%3],(r%1)",                           /* addr 1[b'r3],(r1) */
"       movxbd (r%2),r%1",                              /* movxbd (r2),r1 */
"       andd $255,r%1"                                  /* andd $255,r1 */
=>
"       movd _%0,r%2",                                  /* movd _i,r2 */
"       movzbd (r%2),r%1",                              /* movzbd (r2),r1 */
"       addqd $1,r%2",                                  /* addqd $1,r2 */
"       movd r%2,_%0";                                  /* movd r2,_i */

27.     /* reverse compare to make quick */
```

```
"       cmp%0[dbw] %1,$%2[-\0]%3[0-7]",             /* cmpd r3,$1 */
"       b%4 L%5"                                    /* bgt L05 */
                        revcmp(%4)
=>
"       cmpq%0 $%2%3,%1",                           /* cmpd $1,r3 */
"       b%4 L%5";                                   /* blt L05 */
```

28.
```
/* add quick by minus value */
"       sub%0[dbw] $%1[0-7],%2"                     /* subd $1,r2 */
=>
"       addq%0 $-%1,%2";                            /* addqd $-1,r2 */
```

29.
```
/* negate const to move */
"       neg%0 $%1,%2"                               /* negd $1,r2 */
                        negconst(%1,%3)
=>
"       mov%0 $%3,%2";                              /* movd $-1,r2 */
```

30.
```
/* bin oper comb */
"       mov%0[q\0]%1 %2,%3[rf]%4",                  /* movd r2,r4 */
"       %5 %3[rf]%4,%6"                             /* addd r4,r6 */
                        typematch(%1,%5) && isbinoper(%5)
=>
"       %5 %2,%6";                                  /* addd r2,r6 */
```

31.
```
/* jump comb */
"       b%0 L%1",                                   /* beq L01 */
"       br L%2",                                    /* br L02 */
"L%1:"                                              /* L01: */
                        jmpflip(%0,%3)
=>
"       b%3 L%2",                                   /* bne L02 */
"L%1:";                                             /* L01: */
```

32.
```
/* elim dup reg */
"       mov%0 %1[rf]%2,%1[rf]%3",                   /* movd r2,r3 */
"       mov%0 %1[rf]%3,%4",                         /* movd r3,i.(fp) */
"%5"
                        isvar(%4) && notlastinst()
=>
"       mov%0 %1%2,%4",                             /* movd r2,i.(fp) */
"%5";
```

33.
```
/* elim reg used as addr in deref */
"       movd r%0,r%1",                              /* movd r0,r1 */
"       mov%2 (r%1),%3",                            /* movd (r1),r3 */
"%4"                                                /* addd r3,r4 */
=>
"       mov%2 (r%0),%3",                            /* movd (r0),r3 */
"%4";                                               /* addd r3,r4 */
```

REFERENCES

1. M. C. Newey, P. C. Poole and W. M. Waite, 'Abstract Machine Modelling to Produce Portable Software', *Software—Practice and Experience*, **2**, 107-136 (1972).
2. R. E. Berry, 'Experience with the Pascal P-Compiler', *Software—Practice & Experience*, **8**, 617-627 (September 1978).
3. A. S. Tanenbaum, H. V. Staveren, E. G. Keizer and J. W. Stevenson, 'A Practical Tool Kit for Making Portable Compilers', *Communications of the ACM*, **26**, 654-660 (September 1983).
4. J. S. Watts, *Construction of a Retargetable C Language Front-end*, Masters Thesis, University of Virginia, Charlottesville, VA, 1986.
5. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
6. D. A. Patterson, 'Reduced Instruction Set Computers', *Communications of the ACM*, **28**, 8-21 (January 1985).
7. J. W. Davidson and C. W. Fraser, 'Code Selection through Object Code Optimization', *Transactions on Programming Languages and Systems*, **6**, 7-32 (October 1984).
8. J. Steele, 'A Code Generator for the AT&T 3B Computers', Master's Project Report, University of Virginia, Charlottesville, VA, August 1986.
9. S. C. Johnson, 'A Tour Through the Portable C Compiler', *Unix Programmer's Manual, 7th Edition*, **2B**, Section 33 (January 1979).
10. S. C. Johnson, 'A Portable Compiler: Theory and Practice', *Proceedings of the Fifth Annual Symposium on Principles of Programming Languages*, Tucson, AZ, 97-104 (January 1978).
11. R. R. Ryan and H. Spiller, 'The C Programming Language and a C Compiler', *IBM Systems Journal*, **24**, 37-48 (1985).
12. D. A. Lamb, 'Construction of a Peephole Optimizer', *Software—Practice & Experience*, **11**, 639-647 (June 1981).
13. J. W. Davidson and C. W. Fraser, 'Automatic Inference and Fast Interpretation of Peephole Optimization Rules', *Software—Practice & Experience*, **17**, 801-812 (November 1987).
14. H. E. Bal and A. S. Tanenbaum, 'Language- and Machine-Independent Global Optimization on Intermediate Code', *Computer Language*, **11**, 105-121 (1986).