LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

A Portable Compiler

Computation Structures Group Memo 149
June 1977

Alan Snyder

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

ABSTRACT

The design of a portable C compiler that produces code for register-oriented target machines is presented. Previous work relating to the design of portable compilers is discussed, including the abstract machine approach and work on the automatic construction of generation phases. The present method, which is based on the use of a class of machine-dependent abstract machines, is summarized. The class of abstract machines is described, as is the mechanism for defining particular members of the class via target machine descriptions. The technique used to generate code for a class of abstract machines is presented. An analysis of the compiler and the overall approach is given.

Key Words and Phrases: compilers, portable compilers, portability, abstract machines, code generation, machine description, compiler-compilers.
CR Categories: 4.12

This work has been reported in an expanded form in [20]. Work reported herein was supported in part by the Bell Telephone Laboratories, Inc., the National Science Foundation Research Grant GJ-34671, IBM funds for research in Computer Science, and the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract no. N00014-75-C-0661.

1. Introduction

A portable program is one that can run on many host machines. In the case of a *portable compiler*, there is usually an additional goal: the compiler should be able to produce code for many target machines. This paper describes a portable C compiler, written in C, that has been designed to be capable of producing assembly-language code for target machines of the register-oriented class.[1] C [17] is an implementation language developed at Bell Laboratories and a descendant of the language BCPL [15]. It is a procedure-oriented language, having four primitive data types (integers, characters, and single- and double-precision floating-point), four data type constructors (pointers, arrays, functions, and structures), and a small set of simple control structures.

The ultimate goal of the development of portable compilers is to make programming languages more widely available, by reducing the effort needed to implement compilers for various languages on various machines. A more general approach was taken in the unsuccessful UNCOL project [21]. Other, more practical approaches attempt to develop techniques that reduce the effort involved in implementing compilers for particular language-machine combinations. Examples of such techniques are parser generators [1] and syntax-directed symbol processors [7]. Another approach is to develop techniques for implementing families of compilers for many source languages and one target machine. An example of such a technique is a compiler writing system with code generation primitives, such as FSL [6]. The approach taken in this work is the portable compiler, a compiler for a particular source language that can produce code for many target machines.

There are two basic approaches toward the design of portable compilers, the *machine description* approach and the *abstract machine* approach. With the machine description approach, one attempts to express code generation as a target-machine-independent process that uses information about the target machine in order to correctly generate code for that machine. The information about a target machine is called a *machine description*. If a code generation algorithm with these characteristics can be found, then only the machine description need be rewritten for each target machine. The code generation algorithm determines the class of target machines for which reasonably efficient code can be produced.

Although the machine description approach is theoretically attractive and the obvious approach for those familiar with the concept of a compiler-compiler, little progress has been made toward the goal of being able to generate a compiler from a rigorous definition of machine semantics. The work that has been done on the machine description approach is described below, in a comparison with the portable C compiler.

With the abstract machine approach, one attempts to split the process of compilation into two separate phases such that the target machine dependencies are confined to the second phase. If such a division can be found, then only the second phase of the compiler need be rewritten for each target machine. The first phase of the compiler can be regarded as translating source programs into programs for some *abstract machine*; the second phase translates abstract machine programs into the target language. The choice of the abstract machine determines the ease with which translators from the abstract machine language to target machine languages can be written

1. The initial implementation of the compiler produced code for the HIS-6000 computer [9]. An implementation for the DEC PDP-10 [4] was later developed.

and the class of target machines for which reasonably efficient code can be easily produced.

To the best of our knowledge, all previously existing portable compilers have been based on the abstract machine approach.[1] In most of these compilers [2, 5, 8, 16], the abstract machine language is essentially the usual internal representation passed between the analysis and generation phases in conventional compilers. Creating a compiler for a new target machine thus requires, in essence, the writing of a new code generator, a significant effort. In the case of the BCPL compiler, for example, modifying the compiler to produce code for a new target machine may require only three to four weeks under ideal conditions, but otherwise may require up to five months [16].

More recently, Poole and Waite [13, 14] have described abstract machines designed specifically for portability. These abstract machines are simpler and more machine-oriented than those used in the compilers described above; the translation to a target machine language can generally be performed via a set of simple macro definitions. An example of an abstract machine family designed for use in portable compilers is described in [3]; portable compilers using abstract machines designed specifically for portability are described in [12] and [23].

The major difficulty with the abstract machine approach is determining the appropriate abstract machine. A higher-level abstract machine allows the generation of more efficient code, but requires that a non-trivial translator be written for each target machine. A lower-level abstract machine allows a simple macro translation to the target language. However, unless the abstract machine corresponds closely to the target machine, either the code produced will be inefficient or the implementation will be complicated by optimization code (e.g., code for register allocation [3, 16]).

A solution to this difficulty, proposed by Poole and Waite [14], is to define a *sequence* of abstract machines, ranging from a high to a low level. For any particular target machine, one need write only a translator from the lowest-level abstract machine language to the target machine language. However, where more efficient code is desired, higher-level abstract machine instructions can be translated directly to the target language. This solution allows one to choose from a set of abstract machines. However, the chosen abstract machine will still have the disadvantages listed in the previous paragraph.

The solution used in the portable C compiler is to support a *class* of abstract machines. For each target machine, the implementer can define a particular, *machine-dependent* abstract machine that closely resembles the target machine in several significant respects. As a result, the translation from the abstract machine language to the target machine language becomes simpler (e.g., simple macro expansion), and more efficient code is produced.

The C compiler can be viewed as using the machine description approach in that the definition of a particular abstract machine, along with the macro definitions, is (at some level) a description of the target machine. The definition of the abstract machine contains such information as a description of the abstract machine registers, the sizes and alignments of the various C data types, and descriptions of the behavior of the abstract machine instructions in terms of their operand and result locations (in memory or in particular registers).

1. The Pascal-P2 compiler, discussed below, was developed concurrently with the portable C compiler.

Using this approach, the implementer creates a compiler for a new target machine by writing a machine description, which is then used to generate a set of compiler tables. These tables are combined with the machine-independent compiler procedures to produce a compiler for the new target machine. The code generator uses the tables to produce code for the abstract machine defined in the machine description. This code is then translated to target-machine assembly code using the macro definitions.

Although not an ideal solution, the method of describing the instructions of a machine by providing simple instruction sequences that interpret the instructions of an abstract machine seems to be a good compromise between the desire to minimize per-target-machine coding and the difficulty of mathematically defining a machine and utilizing such a definition in generating code. The previous work on the machine description approach by Sibley [9] and by Miller [10] can be viewed as using this solution. In Sibley's SLANG system, the machine description contained information such as the word size of the target machine, the number of index registers, and information about indexing and indirection. In Miller's proposal, the machine description was extended to allow the implementer to specify constraints on the abstract operations in terms of the permitted operand and result locations. However, neither SLANG nor Miller's model was intended to be a complete proposal, and neither resulted in a successful implementation.

The Pascal-P2 compiler [8] is an existing compiler that is parameterized by information about the target machine. This information consists of size limitations for programs and strings, plus size information for the basic data types. No information about machine registers, addressing, or instructions is used.

Section 2 of this paper describes the class of abstract C machines and the machine description used in the portable C compiler. The third section is a presentation of the C compiler's code generation algorithm. In section 4, conclusions with regard to the compiler, the compiled code, and the overall approach are discussed. Examples in this paper are taken from the HIS-6000 implementation.

2. Modeling the Target Machine

A code generation algorithm, if it is to be independent of particular target machines, requires a machine model with which to work. The C compiler's model is an abstract C machine, a machine whose instructions perform the primitive operations of the C language. The data types of this abstract machine are the primitive C data types (characters, integers, and single- and double-precision floating point), supplemented by one or more pointer classes, distinguished by their ability to resolve addresses. The basic addressable unit of the abstract machine memory is the *byte*, which holds a single character value (characters are the smallest C data type). Values of the other abstract machine data types occupy an integral number of bytes, possibly aligned in larger units of memory. The abstract machine has a set of registers, each capable of holding values of some subset of the abstract machine data types. The instructions of the abstract machine are three-address instructions. Each address may specify an abstract machine register or a location in memory; the mechanisms for referencing a memory location correspond to the primitive addressing modes in C.

In the machine description, the implementer describes the target machine in terms of this machine model by defining a *particular* abstract machine. The implementer specifies the sizes and alignments of the primitive C data types and defines the pointer classes (described in section 2.2). The implementer defines the abstract machine registers, which correspond to those registers of the target machine to be used in evaluating expressions. The implementer also specifies which registers may hold values of each of the abstract machine data types. The implementer defines the abstract machine instructions in terms of their operand/result locations and possible *side-effects* on other registers. In addition, the implementer provides a set of macro definitions that implement the abstract machine instructions on the target machine.

2.1 The Abstract Machine Language

The abstract machine language consists of the three-address abstract machine instructions, plus *keyword macros* corresponding to assembly-language pseudo-operations and instructions implementing the primitive C control structures. The three-address instructions, which perform the evaluation of C expressions, consist of an operator (called an *abstract machine operator* or AMOP) and addresses (called *references* or REFs) for the first operand, second operand (if any), and result. Most AMOPs are basic C operations that are qualified by the specific abstract machine data types of their operands. For example, in the HIS-6000 implementation there are four AMOPs corresponding to the C operator '+':

+i	integer addition
+d	double-precision floating-point addition
+p0	addition of an integer to a pointer to a byte-aligned object
+p1	addition of an integer to a pointer to a word-aligned object

In addition, there are AMOPs for data movement, data type conversion, and conditional jumps. The minimum number of AMOPs that must be defined in the machine description is about 60, depending upon the number of pointer classes defined.

It has been found useful to have some AMOPs whose definition in the machine description is optional. If the semantics of a particular AMOP can be expressed in terms of a composition of more basic AMOPs, then that AMOP can be left undefined in the machine description; the compiler can use the equivalent composition of AMOPs instead. Alternatively, the implementer can give an explicit definition, if that definition will result in better object code. Examples of optional AMOPs include assignment operators (e.g. add-to-memory), increment/decrement operators, testing for null pointers, and comparisons between pointers that are not of the pointer class with the finest resolution.

A REF is a C-oriented description of the location of an operand or result of an abstract machine instruction. A REF may specify either a register of the abstract machine or a location in memory. The possible classes of memory references include C variables of various storage classes (automatic, static, external, parameter, temporary) as well as literals and indirect references (see Table I).

2.2 Defining the Abstract Machine

The abstract machine is defined in two sections of the machine description. First, a set of definition statements defines the registers and memory of the abstract machine. Second, the behavior of the AMOPs is elaborated via the specification of the possible locations of their operands and results in the abstract machine registers and memory.

Figure 1 presents some definition statements from the HIS-6000 machine description. First, the **regnames** statement defines the eight abstract machine registers used in the HIS-6000 implementation. The registers X0 through X4 correspond to the first five of eight HIS-6000 index registers, the A and Q correspond to the accumulators, and the F register is a fictitious floating-point accumulator that corresponds to the combined A, Q, and E (exponent) registers on the HIS-6000. The fact that the F register conflicts in the target machine with the A and Q registers is

---

Table I. Abstract Machine References (REFs)

A REF specifies one of the following alternatives:

register	- a specified abstract machine register
auto	- an automatic or temporary variable, specified by its offset in the stack frame
extern	- an external variable, specified by an internal identifier number
static	- a static (internal) variable, specified by an internal static variable number
parm	- a parameter, specified by the offset of the variable or its address in the parameter list
label	- a label, specified by an internal label number
intlit	- an integer literal, specified by its value
floatlit	- a floating-point literal, specified by an internal literal number
stringlit	- a character string literal, specified by an internal string number
indirect	- a reference indirect through a pointer in a specified register; an offset (displacement) can also be specified

Fig. 1. Some HIS-6000 Machine Description Definition Statements.

```
regnames (x0,x1,x2,x3,x4,a,q,f);
conflict (a,f), (q,f);
class x(x0,x1,x2,x3,x4), r(a,q);
pointer p0(1), p1(4);
size 1(char), 4(int,float), 8(double);
align 1(char), 4(int,float), 8(double);
type int(r), char(r), float(f), double(f), p0(r), p1(x);
```

specified by the conflict statement. Of the remaining HIS-6000 index registers, two are used to hold "environment pointers" and one is used as a scratch register. These registers are not included in the machine description because they are not available for use in evaluating expressions.

The class statement is used to define classes of abstract machine registers for convenience in later specifications; in this example, the class statement defines the class X of index registers and the class R of general registers. The pointer statement is used to define classes of abstract machine pointers. Multiple pointer classes are necessary on machines that are not byte-addressed, since pointers to byte-aligned objects will be handled differently than pointers to word-aligned objects. In this example, the pointer statement defines the class P0 of byte pointers and the class P1 of word pointers. The "4" indicates that the value of a P1 pointer is always a multiple of four bytes. The fact that there are four bytes per word on the HIS-6000 is specified in the size statement; the align statement specifies the corresponding alignment restrictions. The type statement defines which registers can hold values of each of the abstract machine data types. In this example, word pointers are held in the index registers (class X), while byte pointers are held in the general registers (class R).

The definition of the abstract machine is completed in the OPLOC section of the machine description, where the implementer specifies the behavior of the abstract machine operations in terms of their OPerand/result LOCations. For example, the *location definition*

```
+d:          f,M,f;
```

specifies that the AMOP '+d' (double-precision floating-point addition) can take its first operand in the F register and its second operand in any memory location and, under these circumstances, the result is placed in the F register. The construct on the right in the location definition is called an *OPLOC*. It consists of three *location expressions*, one for the first operand, second operand, and result (reading from left to right). A location expression may specify any set of abstract machine registers or any set of memory reference classes. For example, the location expression

```
r | x
```

represents the set consisting of the general registers R and the index registers X, and the location expression

```
~ intlit
```

represents the set consisting of all memory reference classes except that of integer literals. An OPLOC may specify that the result is placed in the first or second operand location. For example, the location definition

```
+i:        r,M,l;
```

specifies that the AMOP '+i' (integer addition) takes its first operand in a general register and its second operand in any memory location, and that the result is placed in the register that contained the first operand. This location definition is equivalent to

```
+i:        a,M,a; q,M,q;
```

which explicitly lists the two alternatives. An OPLOC may also specify that the contents of certain registers are destroyed during the execution of an AMOP. For example, the location definition

```
*i:        q,M,q [a];
```

specifies that an integer multiplication destroys the contents of the A register.

2.3 Defining the Target Language

The machine description is completed by providing macro definitions for the abstract machine instructions and keyword macros. In its simplest form, a macro definition is just a character string to be substituted for the macro call during macro expansion. A macro definition may also include references to the actual arguments of the macro call, embedded macro calls, and strings whose inclusion in an expansion of the macro is conditional upon the locations of an AMOP's operands and/or result.

A macro definition for an abstract machine instruction is closely related to the location definition for the corresponding AMOP. The macro definition can assume that the actual operand/result locations appearing in an abstract machine instruction satisfy the constraints specified in the location definition. At the same time, the macro definition must produce correct code for all (meaningful) combinations of operand/result locations allowed by the location definition.

As an example, consider the HIS-6000 macro definition for '+i' (integer addition):

```
+i:        "       AD*R   *S"
```

(The *R and *S are examples of special character sequences used to refer to symbolic representations of the operation and/or operand/result locations of an abstract machine instruction. These character sequences, *O (operation), *F (first operand), *S (second operand), and *R (result), are abbreviations for calls to an implementer-defined macro that converts an AMOP opcode or a REF into the desired target language representation.) The location definition for '+i',

```
+i:        r,M,l;
```

states that the first operand must be in a general register, the second operand in memory, and that the result is left in the location of the first operand. Thus, in the instructions generated by the '+' macro, the "*R" will be replaced by either "A" or "Q", and the "*S" will be replaced by some legal memory reference expression. Examples of instructions that could be produced by this definition are

```
ADA     X       (add external variable X to A)
ADQ     0,6          (add parameter 0 to Q)
ADA     12,7    (add automatic variable to A)
ADQ     0,0     (add via index register 0)
ADA     3,DL         (add literal 3 to A)
```

A macro may also be defined by a C routine. C routine macro definitions are used when processing is needed that is beyond the capabilities of the simple macro scheme so far described. In the HIS-6000 implementation, C routine macro definitions are used to translate REFs into assembler symbols, to translate the source language representations of identifiers and floating-point literals into their assembler equivalents, to define character string literals, and to buffer characters while defining storage for variables.

3. Generating Code for an Abstract Machine

The interesting part of the compiler is the code generator since, unlike most code generators, which produce code for a fixed target language, the code generator of the C compiler is designed to produce code for a class of abstract machines. The code generator must generate a correct sequence of abstract machine instructions to carry out the operations indicated in the source program. The operand and result locations it specifies in the abstract machine instructions must conform to the location definitions provided in the machine description. Moreover, the code generator must keep track of the locations of all intermediate results and correctly administer the abstract machine registers and temporary locations.

This section describes the method used to generate code for expressions (including conditional jumps). Code generation for expressions is performed by a set of recursive routines. Each routine receives expressions in the form of trees whose interior nodes are AMOPs and whose leaf nodes are identifiers and literals. Thus, an expression can be considered to consist of a "top-level" operator along with zero or more operand expressions. At this point in compilation, type checking has already been performed, explicit conversion operators have been inserted where necessary, and any optional AMOPs not defined in the machine description have been replaced by the corresponding sequences of more basic operations.

3.1 Specifying Desired Locations

The operation of the code generation routines is basically top-down. When a call is made to generate code to evaluate an expression or a subexpression, a set of desired locations for the result of that evaluation is also specified. This specification, called a LOC, is described in Table II. Note that a particular memory location is never specified as the desired location for a result. Instead, classes of possible memory locations are specified.

Table II. Desired Location Specifications (LOCs)

A LOC specifies one of the following alternatives:

label	the "result" is a specified internal label (used only for conditional jump AMOPs)
register	the result is to be placed in one of a specified set of registers
memory	the result is to be placed in memory; a set of acceptable memory reference classes is specified (used only to select registers for pointers in indirect references)
any	the result may be left in any location acceptable for values of the particular data type

For convenience, if the LOC passed to the top-level code generation routine specifies that the result is desired in a register, then all registers not capable of holding the result value are removed from the LOC (these are registers that have been defined in the type statement of the machine description as being unable to hold values of the result data type). Similarly, if the LOC specifies memory reference classes, then for each pointer register unable to hold a pointer to the result value, the corresponding indirect memory reference class is removed from the LOC. This removal of "impossible" locations allows code generation routines to specify "any register" or "any memory class" without concern about data type restrictions.

The removal of "impossible" registers from a LOC is not performed when such action would leave no remaining acceptable registers. This situation can actually occur in certain special cases, such as return statements, where a value may be required in a register not normally used to hold values of that type.

3.2 High-Level Description

The function of the top-level code generation routine is to generate a sequence of abstract machine instructions that will evaluate a given expression and leave the result in an acceptable location, as specified by a LOC parameter. This function is performed in three steps: First, the "impossible" cases are removed from the LOC parameter, as described above. Second, code is generated for the expression, using the LOC parameter as a non-binding indication of preference. Finally, abstract machine instructions are emitted, if necessary, to move the result to an acceptable location.

The second step consists basically of testing for a set of special cases and then performing the appropriate action. Here, the LOC parameter is used only where a choice exists. The first special case is where the expression node is shared and the expression has already been evaluated; in this case, no action need be taken. Another special case is where the top-level operator is a conditional AMOP and a value is desired (as opposed to a jump, the usual case); in this case, a routine is called to emit the desired code. The other special cases involve particular top-level operators (indirection, assignment, conditional expression, and function call) and the leaves of the expression tree (identifiers and literals); for each of these cases, a special code generation routine is called. In all other cases, the "basic" code generation algorithm, described below, is performed.

3.3 The Basic Algorithm

The task of the basic code generation algorithm is to generate code for an expression whose top-level operator is an arithmetic, logical, or conditional jump operator (i.e., not one of the special cases described above). As before, a LOC argument indicates a non-binding preference for the location of the result of the expression evaluation.

The algorithm consists of six steps. First, an OPLOC is selected from the top-level operator's *location definition* in the machine description (location definitions and OPLOCs are described in section 2.2). Second, desired locations for the operands of the top-level operator are determined. Third, recursive calls are made to the top-level code generation routine to emit code to evaluate the operands into the desired locations. Fourth, code is emitted to save any registers specified in the machine description as being "clobbered" by the execution of the top-level operator. Fifth, the

exact location of the result of the expression is determined. Sixth, the abstract machine instruction for the top-level operator is emitted.

If the result location specified by the LOC parameter is a label, or if the selected OPLOC specifies that the result is left in the first or second operand location, then the exact location of the result of the expression is fixed. Otherwise, a particular register must be chosen from the set of registers specified in the result field of the OPLOC (the compiler is currently unable to handle OPLOCs that specify a set of memory references classes as the location of the result). In the search for a result register, the priorities are as follows: first, free registers that are preferred result locations; second, busy registers that are preferred result locations; third, free registers that are not preferred result locations; and fourth, busy registers that are not preferred result locations. If a busy register is selected, register contents are saved in temporary locations as necessary.

For the purposes of finding a result register, a register containing an operand is considered free and a register containing a pointer to an operand is protected. A register containing a pointer to an operand is protected because the implementation of an AMOP may alter the contents of the result register before referencing an operand in memory. An example is the following HIS-6000 code for the AMOP '+pl' (addition of an integer to a pointer to a word-aligned object):

```
LXL0    I       (load register x0 from I)
ADLX0   P       (add P to register x0)
```

This code loads[1] index register 0 with the integer I and then adds to register 0 the pointer P. If the code generated for P leaves P referenced through index register 0, and register 0 is not protected, the following incorrect code could be produced:

```
LXL0    I       (load register x0 from I)
ADLX0   0,0     (add to x0 indexed via x0)
```

This code is incorrect because the load instruction "clobbers" register 0 before P is accessed by the add instruction. However, if index register 0 is protected, index register 1 will be chosen instead to hold the result, producing the following correct code:

```
LXL1    I       (load register x1 from I)
ADLX1   0,0     (add to x1 indexed via x0)
```

After an OPLOC has been selected, recursive calls are made on the top-level code generation routine to generate code for the operands of the top-level abstract machine operator. The LOC arguments passed in these calls are taken from the operand fields of the selected OPLOC and, in the case of operators that place their result in an operand location, the desired locations for the result of the top-level operator. If there are two operands, the compiler makes sure that the two operands will not require the use of the same register (for example, by using a register to hold both one operand and a pointer to the other operand), by checking the LOCs for "overlap" and removing certain possibilities. In addition, the "more complicated operand" (in terms of having the larger tree representation) is evaluated first; this strategy tends to reduce the number of saving and

1. The code for the AMOP includes the load instruction because arbitrary C integers cannot be stored in the HIS-6000 index registers, which are only halfword registers.

restoring operations generated. In the course of generating code to evaluate an operand of a binary abstract machine operator, it may be necessary to use the register containing the already-computed value of the other operand or a pointer used to reference it, in which case code is generated to save the contents of this register in a temporary location. After generating code to evaluate both operands, code is emitted to restore the saved value to its original register.

3.4 OPLOC Selection

The purpose of OPLOC selection is to select a set of operand/result locations for the top-level operator of an expression by choosing one of the OPLOCs from the operator's location definition in the machine description. The choice of operand/result locations can affect the amount of code produced, both because of different code sequences that may be produced by the macro definition and because of additional loading, storing, and saving operations that may be required in order to set up the operands and move the result to an acceptable location. A general solution, taking into account all possible locations of operands and results, is a complex optimization problem. Instead, two sources of information are used to make a local decision. The first source is the result location preference indicated by the LOC parameter. The second source is the set of possible locations for the operand values, which can be (conservatively) determined by examining the top-level operators in the operand subexpressions. For example, if an operand is an identifier, then its location is known to be a memory reference of a particular class. Similarly, various operators may be defined in the machine description to always place their result in one of a particular set of registers. Using information of this sort, plus knowledge about current register usage, a rough estimate can be made of the number of additional load and store instructions that will be required for each OPLOC in the location definition. From the set of OPLOCs, the one with the lowest additional cost is chosen.

3.5 An Example

As an example, consider the expression "I + (J / K)." (For clarity, source language operator symbols are used in this example to represent the corresponding integer abstract machine operations.) Assume the following location definitions (the OPLOCs are numbered for reference):

+:	r,r,l;	*(1)*
	r,M,l;	*(2)*
	M,r,2;	*(3)*
/:	r1,r,l [r2];	*(4)*
	r2,r,l [r3];	*(5)*
	r3,r,l [r4];	*(6)*
	r1,M,l [r2];	*(7)*
	r2,M,l [r3];	*(8)*
	r3,M,l [r4];	*(9)*

Here *M* represents all memory reference classes and *r* represents a set of general registers consisting of *r1*, *r2*, *r3*, and *r4*. The division operator is modeling a machine instruction that produces pairs of results (quotient and remainder) in adjacent registers. For the division abstract machine operator, only the quotient is used; the other register is considered to be "clobbered" by the execution of the operator. OPLOC *(3)* is expressing the commutativity of the addition operator.

Note that one can deduce from these location definitions that both operators always leave their results in general registers.

The generation of code for the expression "I + (J / K)" begins with the selection of an OPLOC from the location definition of the '+' operator. In this case, all of the OPLOCs specify the same set of result locations (the general registers); thus, the desired location for the result of the expression does not affect the choice of OPLOCs. Instead, the choice is made on the basis of the possible locations for the operands of '+'. In this case, the first operand is a variable I, which is known to be a memory reference of a particular class. The second operand is the result of a division operator, which is known to leave its results in either *r1*, *r2*, or *r3*. On this basis, OPLOC *(3)* is chosen, because no extra operations are needed to move the operands into acceptable locations, whereas both OPLOCs *(1)* and *(2)* do require such extra operations.

Next, a recursive call is made to generate code to evaluate the subexpression "J / K." The desired locations for the result of this expression are those specified by the chosen '+' OPLOC for its second operand, namely *r*, the set of general registers. However, since the '+' OPLOC specifies that its second operand location is also the location of its result, the intersection of that location set with the set of desired locations for the result of the '+' operator is used instead, if that intersection is non-null. Thus, the following factors are used in selecting an OPLOC for the '/' operator: first, which of the possible result registers (*r1*, *r2*, *r3*) are desired result locations; second, which of the possible result registers are free; and third, which of the "clobbered" registers (*r2*, *r3*, *r4*) are free. In this particular situation, the possible location of the first operand (J) is a memory reference, and thus does not favor any of the OPLOCs. However, the second operand, which is also known to be a memory reference, favors OPLOCs *(7)*, *(8)*, and *(9)*.

Assume that the desired locations originally specified for the result of the '+' operator were registers *r2* and *r3*, and that registers *r1* and *r4* are busy. Then, the desired locations specified for the result of the '/' operator will also be registers *r2* and *r3*. Under these conditions, OPLOC *(8)* will be selected, because it is the only one that implies no extra load and store operations. Thus, the resulting code (for this hypothetical machine) will be:

```
LOAD    R2,J
DIV     R2,K
ADD     R2,I
```

3.6 OPLOC Selection Revisited

When selecting an OPLOC from a location definition, certain OPLOCs may be rejected entirely because they specify conditions that cannot be met. For example, if an OPLOC specifies (either directly or indirectly through an operand location) that the result is left in a register, but the result is desired in memory, then that OPLOC will be rejected if a temporary location is not acceptable. The OPLOC is rejected because, given a value in a register, the only general method by which the code generator can make that value into a memory reference is by saving it in a newly allocated temporary location. (Recall that a specific memory location is not provided for the result, only a set of acceptable memory reference classes.) Similarly, if the result will be in memory and is desired in memory, then that OPLOC will be rejected if there are one or more possible result memory reference classes that are not acceptable result locations. The OPLOC is rejected

because the code generator is not capable of transforming a memory reference from one class to another. Similar checking is performed on the operand location specifications in the OPLOC: If an operand is required by the OPLOC to be in memory, but not all non-indirect memory reference classes are allowed, then that OPLOC will be rejected if the operand operator is not guaranteed to place its result in an acceptable memory location, or if it can place its result in a register but temporary locations are not acceptable. These restrictions allow a location definition to contain extra OPLOCs that apply only in special cases. Such OPLOCs will be chosen only when the special cases hold.

An example of how the OPLOC selection method can be utilized in the writing of a machine description is the possible definition of the '+pl' AMOP (addition of a integer to a pointer to a word-aligned object) for the HIS-6000 presented in Fig. 2. This definition contains three different code sequences. The first is the shortest code in the general case; it requires both operands to be in memory. The second is better when the operands are in registers. The third is a special case that can be used only when the second operand is an integer literal. (The character sequence "%o(*'S)" represents an embedded macro call whose function is to return the value of the integer literal. The three code sequences are prefixed by tags that specify the operand/result locations for which they are applicable.) The described OPLOC selection method allows all three OPLOCs to be included in the location definition for '+pl'. In particular, it guarantees that the third OPLOC will be selected only if the second operand is an integer literal.

Fig. 2. An example AMOP definition.

```
+pl:        M,M,x;  x,r,l;  x,intlit,l;

(M,M,x):    "       LXL*R *S                (load integer into index register)
                    ADLX*R *F"              (add pointer to index register)

(x,r,x):    "       *SLS   18               (shift integer into left half of register)
                    ST*S   TEMP             (store integer from register into TEMP)
                    ADLX*R TEMP"            (add TEMP to pointer in index register)

(x,intlit,x):   "   EAX*R %o(*'S),*R"       (effective address to index register)
```

4. Discussion and Conclusions

This paper has presented a technique for the design of portable compilers and has demonstrated its practicality through the implementation of a portable C compiler. Like the Pascal-P2 compiler, and unlike the other work on the machine description approach described in section 1, the portable C compiler was designed specifically for the language being implemented. It is this restriction that contributes most to the practicality of the approach.

This section discusses the compiler, the machine model, and the code produced by the compiler, and presents conclusions on the overall approach to the design of portable compilers.

4.1 The Compiler

A major question with any portable compiler is the extent to which it is inherently less efficient than a compiler designed for one particular target machine. We attempt to answer this question for the portable C compiler by comparing it with the UNIX C compiler (see Tables III and IV). Both compilers were compiled by the UNIX C compiler[1] and run on a PDP-11/70 UNIX system. The compilers compiled the basic code generation routines of the portable C compiler, which involves processing 37469 characters on 1619 lines. The portable C compiler produced PDP-10 assembly language; the UNIX C compiler produced PDP-11 assembly language. The indicated CPU usage includes time spent in the operating system (approximately 10% of the total time).

Table III. Portable C Compiler Statistics

Compiler Phases:[1]

CC: control and error message editing
LP: preprocessor, lexical and syntactic analysis
C: code generation
M: macro expansion

Phase:	CC	LP	C	M
source code (chars/lines)				
machine-independent[2]	19154/893	99169/4291	74298/3410	11044/593
machine description tables	0	191/17	3646/176	7281/383
C routine macros	0	0	0	4662/137
memory usage (words)				
code	2.9K	11.6K	10.9K	3.3K
data	6.9K	21.7K	13.3K	7.0K
stack	0.7K	0.7K	0.7K	0.7K
total	10.4K	33.8K	24.8K	10.8K
CPU usage (sec)	2.9	13.8	10.8	19.2

Total source code: 239507 chars, 10687 lines
Total source code (compressed[3]): 122157 chars, 6560 lines
Total CPU usage: 45.7 secs, 35 lines/sec

Notes:

1. Phases CC and LP each can be configured as two separate phases in order to reduce memory requirements.
2. Not included is code common to more than one phase, which totals 20763 characters on 1011 lines.
3. Compressed by removal of comments and formatting characters.

1. In order to compile the portable C compiler, it was necessary to enlarge some of the UNIX compiler's tables. The given statistics are for this enlarged compiler.

Table IV. UNIX C Compiler Statistics

Compiler Phases:

CC: control and preprocessor
C0: lexical and syntactic analysis
C1: code generation

Phase	CC	C0	C1
source code (chars/lines)			
C source	12941/812	52818/3185	50253/2891
assembly-language tables	0	1704/159	11941/1340
memory usage (words)			
code	3.1K	6.2K	7.3K
data	2.1K	7.1K	4.5K
stack	3.0K	0.7K	0.7K
total	8.2K	13.9K	12.4K
CPU usage (sec)	0.2	0.1	0.8

Total source code: 129357 chars, 8237 lines
Total source code (compressed): 86110 chars, 6945 lines
Total CPU usage: 21.2 secs, 76 lines/sec

These statistics indicate that the portable C compiler is about 2.2 times as slow as the UNIX C compiler. This slowness is due almost entirely to the use of a macro expansion phase (a phase not likely to be present in ordinary compilers), as the compiler spends nearly half of its time (or more) in the macro expansion phase. This problem could be reduced by adopting a different method of translating the abstract machine instructions into the target language. For example, the macros could be compiled into procedures, which would then be called directly by the code generator.

In terms of source code, after removal of comments and formatting characters, the portable C compiler is about 40% larger in characters and about equal in lines, compared to the UNIX C compiler. In terms of memory usage, comparing the code generation and macro phases of the portable C compiler to the code generation phase of the UNIX compiler, we find that the portable C compiler is about two times as large in code and about four times as large in data as the UNIX C compiler. However, comparing the analysis portions of the compilers (which are mostly target-machine independent), we find ratios nearly as great. Thus, although some of the size difference is undoubtedly an inherent property of the chosen method of writing portable compilers, most appears to be due to other factors, such as functional differences (e.g. larger table sizes, better error messages, different symbol table strategies) and programming differences (e.g. different parsing methods).

In summary, the portable C compiler does involve a substantial speed and size overhead, although it is efficient enough for day-to-day use where computing resources are not at a premium. Furthermore, we believe that were one prepared to make the investment necessary to implement C on another machine, these problems would be outweighed by the relative speed with which one could bring up a working implementation. One could then concentrate on making it more efficient, having the advantages of a C compiler to work with and the ability to program in C. As an example, the initial machine description and macro definitions for the PDP-10 implementation were written and debugged by the author in a period of two days. Of course, this is an ideal case; someone not familiar with C and the compiler would require a longer time.

Moving the compiler and the associated machine description processor to new host machines can be accomplished using standard methods of portable software. An unparameterized abstract machine, called CMAC, has been developed for this purpose. A C compiler producing CMAC output has been constructed using the method described in this paper. This compiler can be used to produce CMAC versions of itself and the associated programs. To get these programs running on a new host machine, one writes simple macro definitions for the CMAC macros and implements a few, simple I/O routines. Then, one can write a machine description for the machine and construct a compiler that directly produces code for it.

4.2 The Machine Model

Aside from the restriction to register-oriented machines, there are additional assumptions in the C machine model that could make generating desired code inconvenient or even impossible without modifying the code generator. The most important such assumption is the addressing model, which assumes that all C data can be referenced via the operand fields of machine instructions. Where this condition does not hold, one may have to write more complicated macro definitions. Furthermore, the compiler has no mechanism for keeping track of "intermediate results" computed by a macro definition in the course of making an operand addressable.

An example of this problem is external variables on the IBM System/360. Because a full machine address is needed to address an external variable, one must first load a pointer to the variable into some register before the variable can be accessed. The easiest way for an implementer to handle this problem would be to define most AMOPs as being unable to accept arguments of external storage class, thus causing all external operands to be loaded into registers before use.[1] Better code would be generated if each AMOP macro loaded the necessary pointers for external operands, as the unnecessary loading of external operands into registers would be eliminated. However, the code generator would know nothing of these pointers and thus would be unable to allocate registers for them or remember their locations for possible reuse. The best solution to this problem would be to add to the code generator a default coercion from any memory class to indirect, implemented by computing in a register the address of the memory location. This coercion would be similar to the existing coercions, memory to register, register to register, and register to temporary, which are used to move operand values to acceptable locations. This solution would produce the best code, would not require extra instructions in every AMOP definition, and would provide a handle for optimizing the use of these pointers.

Another area where the addressing mode assumption causes inconvenience is the use of immediate addressing. On the HIS-6000, immediate addressing is one of the machine addressing modes. Thus, the C routine macro that converts REFs to symbolic addresses can easily check for integer literals in the proper range and produce the appropriate form of address specification. The PDP-10, on the other hand, specifies immediate addressing by special instruction opcodes. Thus, in order to use immediate addressing on the PDP-10, one must use conditional macro expansion in the relevant AMOP definitions.

---

1. The AMOPs for data movement and address-of must be defined for all relevant storage classes.

4.3 The Compiled Code

Although there are weak spots, the code produced by the compiler is good considering that almost no optimization is attempted by the compiler (not even reusing values in registers). It is certainly better than what would be produced if the abstract machine were the typical machine-independent abstract machine with one accumulator and one index register [3], given the same complexity of the macro definitions (they do not perform register allocation). Such a compiler would not be able to take advantage of the HIS-6000's two accumulators or the multiple index registers, nor would it recognize the fact that byte pointers cannot fit in the index registers.

One of the weak spots in the compiled code concerns floating-point operations. The code generator "performs" all floating-point operations in double-precision, issuing single-to-double conversion operations before using single-precision operands. It is unable to utilize the HIS-6000 machine instructions that operate on a single-precision operand in memory and a double-precision operand in the F register. Since the implementation of a single-to-double conversion is to load the single-precision operand into the F register, very poor code is produced for single-precision floating-point expressions (as opposed to very good code for double-precision expressions). One way to handle this situation would be to implement a general subtree-matching facility for optimization. With such a facility, the implementer specifies in the machine description that a particular combination of abstract machine operators (specified in the form of a tree) is to be replaced by the code generator with a new abstract machine operator; the new operator is defined by the implementer in the machine description just like any of the built-in operators. In the floating-point case, one would specify that a subtree of the form (using a LISP-like notation)

```
(double-prec-add *1 (single-to-double *2))
```

would be replaced by

```
(single-prec-add *1 *2)
```

where single-prec-add is a new abstract machine operator which would be defined to be the "FAD" instruction. This method of subtree-matching is similar to the sequence of abstract machines method in that the new abstract machine operators can be considered instructions of a higher-level abstract machine. However, using the subtree-matching method, the definition of higher-level operators is optional (thus there is no multistage translation when optimization is not desired or needed), and the implementer defines the higher-level operators to suit his needs. The subtree-matching approach to machine-dependent code optimization has been investigated by Wasilew [22].

Another weakness in the compiled code concerns array subscripting. Instead of placing the offset of an array element into an index register and performing an indexed memory reference, the code generator adds the offset to a pointer to the base of the array, producing a pointer (in an index register) which is then used to reference the array element. Thus, the code generator regards index registers only as base registers to hold pointers, and not as index registers to hold offsets. One reason for not implementing the capability of using index registers for subscripting is that the benefits are small on machines like the HIS-6000 with single-indexed instructions, where this method can be used only for external and static arrays. All other arrays require the use of an

index register just to reference the base of the array.[1] The capability of using index registers for subscripting could be implemented using the subtree-matching facility described above; one would test for subtrees of the form

(pointer-add (address-of <externlstatic>) <any>)

and replace them with a new abstract machine operator which would be defined to produce the desired code. A more satisfying solution would give the code generator more knowledge about addressing so that it could use index registers for subscripting whenever possible, based on information given in the machine description.

A third weakness in the compiled code is the use of indirection. The code generator only indirects through pointers in registers; it is unable to utilize an indirection-through-memory facility (except through a specific location that implements an abstract machine register). Again, a better understanding of addressing is what is really needed.

Although optimization is not performed by the present compiler, we believe the compiler structure provides a framework in which many machine-dependent optimizations can be expressed in a machine-independent manner. We refer the reader to a recent thesis by Newcomer [11] for an examination of this possibility.

4.4 Conclusions

The advantages of the technique presented in this paper over rewriting some or all of the generation phase are (1) that the implementer can construct a compiler that produces code for a new machine with less effort and in less time, and (2) that the implementer can be more confident in the correctness of the new compiler. Almost the entire code of the generation phase, already tested in the initial implementation, is unchanged in the new implementation. This code includes the code generation algorithm, the register management routines, and the macro processor. Furthermore, the modifications that must be made are localized in two areas, the machine description and the C routine macro definitions. The implementer is primarily concerned with the correct implementation of the individual abstract machine instructions. The interaction among these instructions, in terms of their correct ordering and the use of registers and temporary locations, is handled by the code generation algorithm and need not be of concern to the implementer. It is this reduction in the complexity of the problem that leads to the increased confidence in the results.

The advantage of the presented technique over techniques that use a single abstract machine is that improved object code efficiency can be obtained over a wider range of target machines without complicating the macro definitions. However, in order to realize this advantage, the implementer must learn to write machine descriptions.

---

1. Actually, one can perform double-indexing on the HB-6000 by using an indirect word; however, this was not recognized at the time the compiler was written.

References

1. Aho, A. V. and Johnson, S. C. LR parsing. *Computing Surveys 6*, 2 (June 1974), 99-124.
2. Basili, V. R. and Turner, A. J. A transportable extendable compiler. Rep. TR-269, Computer Science Center, University of Maryland, 1973.
3. Coleman, S. S., Poole, P. C., and Waite, W. M. The mobile programming system, JANUS. *Software Practice and Experience 4*, 1 (Jan. 1974), 5-23.
4. Digital Equipment Corp. *DecSystem10 Assembly Language Handbook*. Maynard, Mass., 1973.
5. Englund, D. and Clark, E. The CLIP translator. *Comm. ACM 4*, 1 (Jan. 1961), 19-22.
6. Feldman, J. A. A formal semantics for computer languages and its application in a compiler-compiler. *Comm. ACM 9*, 1 (Jan. 1966), 9-9.
7. Feldman, J. and Gries, D. Translator writing systems. *Comm. ACM 11*, 2 (Feb. 1968), 77-113.
8. Halstead, M. H. *Machine-Independent Computer Programming*. Spartan Books, Washington, 1962.
9. Honeywell Information Systems, Inc. *Series 6000 Macro Assembler Program*. Waltham, Mass., 1972.
10. Miller, P. L. Automatic creation of a code generator from a machine description. Rep. TR-85, Project MAC, M.I.T., Cambridge, Mass., 1971.
11. Newcomer, J. M. Machine-independent generation of optimal local code. Ph.D. Thesis, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., 1975.
12. Pasko, H. J. A pseudo-machine for code generation. Rep. CSRG-90, Computer Systems Research Group, Univ. of Toronto, 1979.
13. Poole, P. C. and Waite, W. M. Machine independent software. Proc. ACM Second Symposium on Operating Systems Principles, 1971, 19-24.
14. Poole, P. C. and Waite, W. M. Portability and adaptability. In *Advanced Course on Software Engineering*. Springer-Verlag, Berlin, 1973, 183-277.
15. Richards, M. BCPL: a tool for compiler writing and system programming. Proc. SJCC 1969, 557-566.
16. Richards, M. The portability of the BCPL compiler. *Software Practice and Experience 1*, 2 (April 1971), 135-146.
17. Ritchie, D. M., Kernighan, B. W., and Lesk, M. E. The C Programming Language. Computing Science Technical Report No. 31, Bell Laboratories, Murray Hill, N. J., 1975.
18. Richmond, G. H, ed. Pascal Newsletter. *Sigplan Notices 11*, 2 (Feb. 1976), 41-42.
19. Sibley, R. A. The SLANG system. *Comm. ACM 4*, 1 (Jan. 1961), 75-84.
20. Snyder, A. A portable compiler for the language C. Rep. TR-149, Project MAC, M.I.T., Cambridge, Mass., 1975.
21. Strong, J., et. al. The problem of programming communication with changing machines -- a proposed solution. *Comm. ACM 1*, 8 (Aug. 1958) 12-18, 9 (Sept. 1958) 9-15.
22. Wasilew, S. G. A compiler writing system with optimization capabilities for complex object structures. Ph.D. Thesis, Northwestern University, Evanston, Illinois, 1971.
23. Weber, L. B. A machine independent Pascal compiler. M.S. Thesis, Univ. of Colorado, Boulder, 1973.