Plan 9 C Compilers

Ken Thompson
ken@plan9.bell-labs.com

ABSTRACT

This paper describes the overall structure and function of the Plan 9 C compilers. A more detailed implementation document for any one of the compilers is yet to be written.

1. Introduction

There are many compilers in the series. Six of the compilers (MIPS 3000, SPARC, Intel 386, Power PC, DEC Alpha, and Motorola 68020) are considered active and are used to compile current versions of Plan 9. Several others (Motorola 68000, Intel 960, ARM 7500, AMD 29000) have had only limited use, such as to program peripherals or experimental devices.

2. Structure

The compiler is a single program that produces an object file. Combined in the compiler are the traditional roles of preprocessor, lexical analyzer, parser, code generator, local optimizer, and first half of the assembler. The object files are binary forms of assembly language, similar to what might be passed between the first and second passes of an assembler.

Object files and libraries are combined by a loader program to produce the executable binary. The loader combines the roles of second half of the assembler, global optimizer, and loader. The names of the compliers, loaders, and assemblers are as follows:

SPARC	`kc`	`kl`	`ka`
Power PC	`qc`	`ql`	`qa`
MIPS	`vc`	`vl`	`va`
Motorola 68000	`1c`	`1l`	`1a`
Motorola 68020	`2c`	`2l`	`2a`
ARM 7500	`5c`	`5l`	`5a`
Intel 960	`6c`	`6l`	`6a`
DEC Alpha	`7c`	`7l`	`7a`
Intel 386	`8c`	`8l`	`8a`
AMD 29000	`9c`	`9l`	`9a`

There is a further breakdown in the source of the compilers into object-independent and object-dependent parts. All of the object-independent parts are combined into source files in the directory `/sys/src/cmd/cc`. The object-dependent parts are collected in a separate directory for each compiler, for example `/sys/src/cmd/vc`. All of the code, both object-independent and object-dependent, is machine-independent and may be cross-compiled and executed on any of the architectures.

Originally appeared, in a different form, in *Proceedings of the Summer 1990 UKUUG Conference,* pp. 41-51, London, 1990.

3. The Language

The compiler implements ANSI C with some restrictions and extensions [ANSI90]. Most of the restrictions are due to personal preference, while most of the extensions were to help in the implementation of Plan 9. There are other departures from the standard, particularly in the libraries, that are beyond the scope of this paper.

3.1. Register, volatile, const

The keyword `register` is recognized syntactically but is semantically ignored. Thus taking the address of a `register` variable is not diagnosed. The keyword `volatile` disables all optimizations, in particular registerization, of the corresponding variable. The keyword `const` generates warnings (if warnings are enabled by the compiler's `-w` option) of non-constant use of the variable, but does not affect the generated code.

3.2. The preprocessor

The C preprocessor is probably the biggest departure from the ANSI standard.

The preprocessor built into the Plan 9 compilers does not support `#if`, although it does handle `#ifdef` and `#include`. If it is necessary to be more standard, the source text can first be run through the separate ANSI C preprocessor, `cpp`.

3.3. Unnamed substructures

The most important and most heavily used of the extensions is the declaration of an unnamed substructure or subunion. For example:

```
typedef
struct    lock
{
     int    locked;
} Lock;

typedef
struct    node
{
     int  type;
     union
     {
          double dval;
          float  fval;
          long   lval;
     };
     Lock;
} Node;

Lock*     lock;
Node*     node;
```

The declaration of `Node` has an unnamed substructure of type `Lock` and an unnamed subunion. One use of this feature allows references to elements of the subunit to be accessed as if they were in the outer structure. Thus `node->dval` and `node->locked` are legitimate references.

When an outer structure is used in a context that is only legal for an unnamed substructure, the compiler promotes the reference to the unnamed substructure. This is true for references to structures and to references to pointers to structures. This happens in assignment statements and in argument passing where prototypes have been declared. Thus, continuing with the example,

```
lock = node;
```

would assign a pointer to the unnamed `Lock` in the `Node` to the variable `lock`. Another example,

```
extern void lock(Lock*);
func(...)
{
     ...
     lock(node);
     ...
}
```

will pass a pointer to the `Lock` substructure.

Finally, in places where context is insufficient to identify the unnamed structure, the type name (it must be a `typedef`) of the unnamed structure can be used as an identifier. In our example, `&node->Lock` gives the address of the anonymous `Lock` structure.

3.4. Structure displays

A structure cast followed by a list of expressions in braces is an expression with the type of the structure and elements assigned from the corresponding list. Structures are now almost first-class citizens of the language. It is common to see code like this:

```
r = (Rectangle){point1, (Point){x,y+2}};
```

3.5. Initialization indexes

In initializers of arrays, one may place a constant expression in square brackets before an initializer. This causes the next initializer to assign the indicated element. For example:

```
enum errors
{
     Etoobig,
     Ealarm,
     Egreg
};
char* errstrings[] =
{
     [Ealarm]  "Alarm call",
     [Egreg]   "Panic: out of mbufs",
     [Etoobig] "Arg list too long",
};
```

In the same way, individual structures members may be initialized in any order by preceding the initialization with `.tagname`. Both forms allow an optional =, to be compatible with a proposed extension to ANSI C.

3.6. External register

The declaration `extern register` will dedicate a register to a variable on a global basis. It can be used only under special circumstances. External register variables must be identically declared in all modules and libraries. The feature is not intended for efficiency, although it can produce efficient code; rather it represents a unique storage class that would be hard to get any other way. On a shared-memory multi-processor, an external register is one-per-processor and neither one-per-procedure (automatic) or one–per–system (external). It is used for two variables in the Plan 9 kernel, `u` and `m`. `U` is a pointer to the structure representing the currently running process and `m` is a pointer to the per–machine data structure.

3.7. Long long

The compilers accept `long long` as a basic type meaning 64–bit integer. On all of the machines this type is synthesized from 32–bit instructions.

3.8. Pragma

The compilers accept `#pragma lib` *libname* and pass the library name string uninterpreted to the loader. The loader uses the library name to find libraries to load. If the name contains %O, it is replaced with the single character object type of the compiler (e.g., `v` for the MIPS). If the name contains %M, it is replaced with the architecture type for the compiler (e.g., `mips` for the MIPS). If the name starts with `/` it is an absolute pathname; if it starts with `.` then it is searched for in the loader's current directory. Otherwise, the name is searched from `/%M/lib`. Such `#pragma` statements in header files guarantee that the correct libraries are always linked with a program without the need to specify them explicitly at link time.

They also accept `#pragma hjdicks on` (or `yes` or `1`) to cause subsequently declared data, until `#pragma hjdicks off` (or `no` or `0`), to be laid out in memory tightly packed in successive bytes, disregarding the usual alignment rules. Accessing such data can cause faults.

Similarly, `#pragma profile off` (or `no` or `0`) causes subsequently declared functions, until `#pragma profile on` (or `yes` or `1`), to be marked as unprofiled. Such functions will not be profiled when profiling is enabled for the rest of the program.

Two `#pragma` statements allow type–checking of `print`–like functions. The first, of the form

```
#pragma varargck argpos error 2
```

tells the compiler that the second argument to `error` is a `print` format string (see the manual page *print*(2)) that specifies how to format `error`'s subsequent arguments. The second, of the form

```
#pragma varargck type "s" char*
```

says that the `print` format verb `s` processes an argument of type `char*`. If the compiler's `−F` option is enabled, the compiler will use this information to report type violations in the arguments to `print`, `error`, and similar routines.

4. Object module conventions

The overall conventions of the runtime environment are important to runtime efficiency. In this section, several of these conventions are discussed.

4.1. Register saving

In the Plan 9 compilers, the caller of a procedure saves the registers. With caller–saves, the leaf procedures can use all the registers and never save them. If you spend a lot of time at the leaves, this seems preferable. With callee–saves, the saving of the registers is done in the single point of entry and return. If you are interested in space, this seems preferable. In both, there is a degree of uncertainty about what registers need to be saved. Callee–saved registers make it difficult to find variables in registers in debuggers. Callee–saved registers also complicate the implementation of `longjmp`. The convincing argument is that with caller–saves, the decision to registerize a variable can include the cost of saving the register across calls. For a further discussion of caller– vs. callee–saves, see the paper by Davidson and Whalley [Dav91].

In the Plan 9 operating system, calls to the kernel look like normal procedure calls, which means the caller has saved the registers and the system entry does not have to. This makes system calls considerably faster. Since this is a potential security hole, and can lead to non-determinism, the system may eventually save the registers on entry, or more likely clear the registers on return.

4.2. Calling convention

Older C compilers maintain a frame pointer, which is at a known constant offset from the stack pointer within each function. For machines where the stack grows towards zero, the argument pointer is at a known constant offset from the frame pointer. Since the stack grows down in Plan 9, the Plan 9 compilers keep neither an explicit frame pointer nor an explicit argument pointer; instead they generate addresses relative to the stack pointer.

On some architectures, the first argument to a subroutine is passed in a register.

4.3. Functions returning structures

Structures longer than one word are awkward to implement since they do not fit in registers and must be passed around in memory. Functions that return structures are particularly clumsy. The Plan 9 compilers pass the return address of a structure as the first argument of a function that has a structure return value. Thus

```
x = f(...)
```

is rewritten as

```
f(&x, ...).
```

This saves a copy and makes the compilation much less clumsy. A disadvantage is that if you call this function without an assignment, a dummy location must be invented.

There is also a danger of calling a function that returns a structure without declaring it as such. With ANSI C function prototypes, this error need never occur.

5. Implementation

The compiler is divided internally into four machine-independent passes, four machine-dependent passes, and an output pass. The next nine sections describe each pass in order.

5.1. Parsing

The first pass is a YACC-based parser [Joh79]. Declarations are interpreted immediately, building a block structured symbol table. Executable statements are put into a parse tree and collected, without interpretation. At the end of each procedure, the parse tree for the function is examined by the other passes of the compiler.

The input stream of the parser is a pushdown list of input activations. The preprocessor expansions of macros and `#include` are implemented as pushdowns. Thus there is no separate pass for preprocessing.

5.2. Typing

The next pass distributes typing information to every node of the tree. Implicit operations on the tree are added, such as type promotions and taking the address of arrays and functions.

5.3. Machine–independent optimization

The next pass performs optimizations and transformations of the tree, such as converting `&*x` and `*&x` into `x`. Constant expressions are converted to constants in this pass.

5.4. Arithmetic rewrites

This is another machine–independent optimization. Subtrees of add, subtract, and multiply of integers are rewritten for easier compilation. The major transformation is factoring: `4+8*a+16*b+5` is transformed into `9+8*(a+2*b)`. Such expressions arise from address manipulation and array indexing.

5.5. Addressability

This is the first of the machine–dependent passes. The addressability of a processor is defined as the set of expressions that is legal in the address field of a machine language instruction. The addressability of different processors varies widely. At one end of the spectrum are the 68020 and VAX, which allow a complex mix of incrementing, decrementing, indexing, and relative addressing. At the other end is the MIPS, which allows only registers and constant offsets from the contents of a register. The addressability can be different for different instructions within the same processor.

It is important to the code generator to know when a subtree represents an address of a particular type. This is done with a bottom–up walk of the tree. In this pass, the leaves are labeled with small integers. When an internal node is encountered, it is labeled by consulting a table indexed by the labels on the left and right subtrees. For example, on the 68020 processor, it is possible to address an offset from a named location. In C, this is represented by the expression `*(&name+constant)`. This is marked addressable by the following table. In the table, a node represented by the left column is marked with a small integer from the right column. Marks of the form A_i are addressable while marks of the form N_i are not addressable.

Node	**Marked**
`name`	A_1
`const`	A_2
$\&A_1$	A_3
A_3+A_1	N_1 (note that this is not addressable)
$*N_1$	A_4

Here there is a distinction between a node marked A_1 and a node marked A_4 because the address operator of an A_4 node is not addressable. So to extend the table:

Node	**Marked**
$\&A_4$	N_2
N_2+N_1	N_1

The full addressability of the 68020 is expressed in 18 rules like this, while the addressability of the MIPS is expressed in 11 rules. When one ports the compiler, this table is usually initialized so that leaves are labeled as addressable and nothing else. The code produced is poor, but porting is easy. The table can be extended later.

This pass also rewrites some complex operators into procedure calls. Examples include 64–bit multiply and divide.

In the same bottom–up pass of the tree, the nodes are labeled with a Sethi–Ullman complexity [Set70]. This number is roughly the number of registers required to compile the tree on an ideal machine. An addressable node is marked 0. A function call is marked infinite. A unary operator is marked as the maximum of 1 and the mark of its subtree. A binary operator with equal marks on its subtrees is marked with a subtree mark plus 1. A binary operator with unequal marks on its subtrees is marked with the maximum mark of its subtrees. The actual values of the marks are not too important, but the relative values are. The goal is to compile the harder (larger mark) subtree first.

5.6. Code generation

Code is generated by recursive descent. The Sethi-Ullman complexity completely guides the order. The addressability defines the leaves. The only difficult part is compiling a tree that has two infinite (function call) subtrees. In this case, one subtree is compiled into the return register (usually the most convenient place for a function call) and then stored on the stack. The other subtree is compiled into the return register and then the operation is compiled with operands from the stack and the return register.

There is a separate boolean code generator that compiles conditional expressions. This is fundamentally different from compiling an arithmetic expression. The result of the boolean code generator is the position of the program counter and not an expression. The boolean code generator makes extensive use of De Morgan's rule. The boolean code generator is an expanded version of that described in chapter 8 of Aho, Sethi, and Ullman [Aho87].

There is a considerable amount of talk in the literature about automating this part of a compiler with a machine description. Since this code generator is so small (less than 500 lines of C) and easy, it hardly seems worth the effort.

5.7. Registerization

Up to now, the compiler has operated on syntax trees that are roughly equivalent to the original source language. The previous pass has produced machine language in an internal format. The next two passes operate on the internal machine language structures. The purpose of the next pass is to reintroduce registers for heavily used variables.

All of the variables that can be potentially registerized within a procedure are placed in a table. (Suitable variables are any automatic or external scalars that do not have their addresses extracted. Some constants that are hard to reference are also considered for registerization.) Four separate data flow equations are evaluated over the procedure on all of these variables. Two of the equations are the normal set-behind and used-ahead bits that define the life of a variable. The two new bits tell if a variable life crosses a function call ahead or behind. By examining a variable over its lifetime, it is possible to get a cost for registerizing. Loops are detected and the costs are multiplied by three for every level of loop nesting. Costs are sorted and the variables are replaced by available registers on a greedy basis.

The 68020 has two different types of registers. For the 68020, two different costs are calculated for each variable life and the register type that affords the better cost is used. Ties are broken by counting the number of available registers of each type.

Note that externals are registerized together with automatics. This is done by evaluating the semantics of a ''call'' instruction differently for externals and automatics. Since a call goes outside the local procedure, it is assumed that a call references all externals. Similarly, externals are assumed to be set before an ''entry'' instruction and assumed to be referenced after a ''return'' instruction. This makes sure that externals are in memory across calls.

The overall results are satisfactory. It would be nice to be able to do this processing in a machine-independent way, but it is impossible to get all of the costs and side effects of different choices by examining the parse tree.

Most of the code in the registerization pass is machine-independent. The major machine-dependency is in examining a machine instruction to ask if it sets or references a variable.

5.8. Machine code optimization

The next pass walks the machine code for opportunistic optimizations. For the most part, this is highly specific to a particular processor. One optimization that is performed on all of the processors is the removal of unnecessary ‘‘move’’ instructions. Ironically, most of these instructions were inserted by the previous pass. There are two patterns that are repetitively matched and replaced until no more matches are found. The first tries to remove ‘‘move’’ instructions by relabeling variables.

When a ‘‘move’’ instruction is encountered, if the destination variable is set before the source variable is referenced, then all of the references to the destination variable can be renamed to the source and the ‘‘move’’ can be deleted. This transformation uses the reverse data flow set up in the previous pass.

An example of this pattern is depicted in the following table. The pattern is in the left column and the replacement action is in the right column.

MOVE a−>b	(remove)
(sequence with no mention of a)	
USE b	USE a
(sequence with no mention of a)	
SET b	SET b

Experiments have shown that it is marginally worthwhile to rename uses of the destination variable with uses of the source variable up to the first use of the source variable.

The second transform will do relabeling without deleting instructions. When a ‘‘move’’ instruction is encountered, if the source variable has been set prior to the use of the destination variable then all of the references to the source variable are replaced by the destination and the ‘‘move’’ is inverted. Typically, this transformation will alter two ‘‘move’’ instructions and allow the first transformation another chance to remove code. This transformation uses the forward data flow set up in the previous pass.

Again, the following is a depiction of the transformation where the pattern is in the left column and the rewrite is in the right column.

SET a	SET b
(sequence with no use of b)	
USE a	USE b
(sequence with no use of b)	
MOVE a−>b	MOVE b−>a

Iterating these transformations will usually get rid of all redundant ‘‘move’’ instructions.

A problem with this organization is that the costs of registerization calculated in the previous pass must depend on how well this pass can detect and remove redundant instructions. Often, a fine candidate for registerization is rejected because of the cost of instructions that are later removed.

5.9. Writing the object file

The last pass walks the internal assembly language and writes the object file. The object file is reduced in size by about a factor of three with simple compression techniques. The most important aspect of the object file format is that it is independent of the compiling machine. All integer and floating numbers in the object code are converted to known formats and byte orders.

6. The loader

The loader is a multiple pass program that reads object files and libraries and produces an executable binary. The loader also does some minimal optimizations and code rewriting. Many of the operations performed by the loader are machine-dependent.

The first pass of the loader reads the object modules into an internal data structure that looks like binary assembly language. As the instructions are read, code is reordered to remove unconditional branch instructions. Conditional branch instructions are inverted to prevent the insertion of unconditional branches. The loader will also make a copy of a few instructions to remove an unconditional branch.

The next pass allocates addresses for all external data. Typical of processors is the MIPS, which can reference ±32K bytes from a register. The loader allocates the register `R30` as the static pointer. The value placed in `R30` is the base of the data segment plus 32K. It is then cheap to reference all data in the first 64K of the data segment. External variables are allocated to the data segment with the smallest variables allocated first. If all of the data cannot fit into the first 64K of the data segment, then usually only a few large arrays need more expensive addressing modes.

For the MIPS processor, the loader makes a pass over the internal structures, exchanging instructions to try to fill ‘‘delay slots’’ with useful work. If a useful instruction cannot be found to fill a delay slot, the loader will insert ‘‘noop’’ instructions. This pass is very expensive and does not do a good job. About 40% of all instructions are in delay slots. About 65% of these are useful instructions and 35% are ‘‘noops.’’ The vendor-supplied assembler does this job more effectively, filling about 80% of the delay slots with useful instructions.

On the 68020 processor, branch instructions come in a variety of sizes depending on the relative distance of the branch. Thus the size of branch instructions can be mutually dependent. The loader uses a multiple pass algorithm to resolve the branch lengths [Szy78]. Initially, all branches are assumed minimal length. On each subsequent pass, the branches are reassessed and expanded if necessary. When no more expansions occur, the locations of the instructions in the text segment are known.

On the MIPS processor, all instructions are one size. A single pass over the instructions will determine the locations of all addresses in the text segment.

The last pass of the loader produces the executable binary. A symbol table and other tables are produced to help the debugger to interpret the binary symbolically.

The loader places absolute source line numbers in the symbol table. The name and absolute line number of all `#include` files is also placed in the symbol table so that the debuggers can associate object code to source files.

7. Performance

The following is a table of the source size of the MIPS compiler.

lines	module
509	machine-independent headers
1070	machine-independent YACC source
6090	machine-independent C source
545	machine-dependent headers
6532	machine-dependent C source
298	loader headers
5215	loader C source

The following table shows timing of a test program that plays checkers, running on a

MIPS R4000. The test program is 26 files totaling 12600 lines of C. The execution time does not significantly depend on library implementation. Since no other compiler runs on Plan 9, the Plan 9 tests were done with the Plan 9 operating system; the other tests were done on the vendor's operating system. The hardware was identical in both cases. The optimizer in the vendor's compiler is reputed to be extremely good.

4.49s	Plan 9 `vc` −N compile time (opposite of −O)
1.72s	Plan 9 `vc` −N load time
148.69s	Plan 9 `vc` −N run time
15.07s	Plan 9 `vc` compile time (−O implicit)
1.66s	Plan 9 `vc` load time
89.96s	Plan 9 `vc` run time
14.83s	vendor `cc` compile time
0.38s	vendor `cc` load time
104.75s	vendor `cc` run time
43.59s	vendor `cc` −O compile time
0.38s	vendor `cc` −O load time
76.19s	vendor `cc` −O run time
8.19s	vendor `cc` −O3 compile time
35.97s	vendor `cc` −O3 load time
71.16s	vendor `cc` −O3 run time

To compare the Intel compiler, a program that is about 40% bit manipulation and about 60% single precision floating point was run on the same 33 MHz 486, once under Windows compiled with the Watcom compiler, version 10.0, in 16-bit mode and once under Plan 9 in 32-bit mode. The Plan 9 execution time was 27 sec while the Windows execution time was 31 sec.

8. Conclusions

The new compilers compile quickly, load slowly, and produce medium quality object code. The compilers are relatively portable, requiring but a couple of weeks' work to produce a compiler for a different computer. For Plan 9, where we needed several compilers with specialized features and our own object formats, this project was indispensable. It is also necessary for us to be able to freely distribute our compilers with the Plan 9 distribution.

Two problems have come up in retrospect. The first has to do with the division of labor between compiler and loader. Plan 9 runs on multi-processors and as such compilations are often done in parallel. Unfortunately, all compilations must be complete before loading can begin. The load is then single-threaded. With this model, any shift of work from compile to load results in a significant increase in real time. The same is true of libraries that are compiled infrequently and loaded often. In the future, we may try to put some of the loader work back into the compiler.

The second problem comes from the various optimizations performed over several passes. Often optimizations in different passes depend on each other. Iterating the passes could compromise efficiency, or even loop. We see no real solution to this problem.

9. References

[Aho87] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers – Principles, Techniques, and Tools,* Addison Wesley, Reading, MA, 1987.

[ANSI90] *American National Standard for Information Systems – Programming Language C*, American National Standards Institute, Inc., New York, 1990.

[Dav91] J. W. Davidson and D. B. Whalley, ''Methods for Saving and Restoring Register Values across Function Calls'', *Software–Practice and Experience,* Vol 21(2), pp. 149–165, February 1991.

[Joh79] S. C. Johnson, ''YACC – Yet Another Compiler Compiler'', *UNIX Programmer's Manual, Seventh Ed., Vol. 2A,* AT&T Bell Laboratories, Murray Hill, NJ, 1979.

[Set70] R. Sethi and J. D. Ullman, ''The Generation of Optimal Code for Arithmetic Expressions'', *Journal of the ACM,* Vol 17(4), pp. 715–728, 1970.

[Szy78] T. G. Szymanski, ''Assembling Code for Machines with Span–dependent Instructions'', *Communications of the ACM,* Vol 21(4), pp. 300–308, 1978.