A New C Compiler

Ken Thompson

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes yet another series of C compilers. These compilers were developed over the last several years and are now in use on Plan 9. These compilers are experimental in nature and were developed to try out new ideas. Some of the ideas were good and some not so good.

1. Introduction

Most C compilers consist of a multitude of passes with numerous interfaces. A typical C compiler has the following passes – pre-processing, lexical analysis and parsing, code generation, optional assembly optimisation, assembly (which itself is usually multiple passes), and loading. [Joh79a]

If one profiles what is going on in this whole process, it becomes clear that I/O dominates. Of the cpu cycles expended, most go into conversion to and from intermediate file formats. Even with these many passes, the code generated is mostly line-at-a-time and not very efficient. With these conventional compilers as benchmarks, it seemed easy to make a new compiler that could execute much faster and still produce better code.

The first three compilers built were for the National 32000, Western 32100, and an internal computer called a Crisp. These compilers have drifted into disuse. Currently there are active compilers for the Motorola 68020 and MIPS 2000/3000 computers. [Mot85, Kan88]

2. Structure

The compiler is a single program that produces an object file. Combined in the compiler are the traditional roles of pre-processor, compiler, code generator, local optimiser, 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 and loaded by a second program to produce the executable binary. The loader combines the roles of second half of the assembler, global optimiser, and loader. There is a third small program that serves as an assembler. It takes an assembler-like input and performs a simple translation into the object format.

3. The Language

The compiler implements ANSI C with some restrictions and extensions. [Ker88] If this had been a product-oriented project rather than a research vehicle, the compiler would have implemented exact ANSI C. Several of the poorer features were left out. Also, several extensions were added to help in the implementation of Plan 9. [Pik90] There are many more departures from the standard, particularly in the libraries, that are beyond the scope of this paper.

3.1. Register, volatile, const

The keywords `register`, `volatile`, and `const`, are recognised syntactically but are semantically ignored. `Volatile` seems to have no meaning, so it is hard to tell if ignoring it is a departure from the standard. `Const` only confuses library interfaces with the hope of catching some rare errors.

`Register` is a holdover from the past. Registers should be assigned over the individual lives of a variable, not on the whole variable name. By allocating registers over the life of a variable, rather than by preallocating registers at declaration, it is usually possible to get the effect of about twice as many registers. The compiler is also in a much better position to judge the allocation of a register variable than the programmer. It is extremely hard for a programmer to place register variables wisely. When one does, the code is usually optimised to a particular compiler or computer. The portability of the performance of a program with register declarations is poor.

There is a semantic feature of a declared register variable in ANSI C – it is illegal to take its address. This compiler does not catch this ‘‘mistake.’’ It would be easy to carry a flag in the symbol table to rectify this, but that seems fussy.

3.2. The pre-processor

The C pre-processor is probably the biggest departure from the ANSI standard. Most of differences are protests about common usage. Some of the difference is due to the generally poor specification of the existing pre-processors prior to the ANSI report.

This compiler does not support `#if`, though it does handle `#ifdef`. In practice, `#if` is almost always followed by a variable like ‘‘pdp11.’’ What it means is that the programmer has buried some old code that will no longer compile. Another common usage is to write ‘‘portable’’ code by expanding all possibilities in a jumble of left-justified chicken scratches.

As an alternate, the compiler will compile very efficient normal `if` statements with constant expressions. This is usually enough to rewrite old `#if`-laden code.

If all else fails, the compiler can be run with any of the existing pre-processors that are still maintained as separate passes.

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:

```
struct      lock
{
      int    locked;
} *lock;

struct      node
{
      int   type;
      union
      {
            double dval;
            float  fval;
            long   lval;
      };
      struct lock;
} *node;
```

This is a declaration with an unnamed substructure, `lock`, and an unnamed subunion. This shows the two major usages of this feature. The first 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. In C, the name of a union is almost always a non-entity that is mechanically declared and used with no purpose.

The second usage is poor man's classes. When a pointer to the outer structure is used in a context that is only legal for an unnamed substructure, the compiler promotes the type. 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(struct lock*);
func(...)
{
        ...
        lock(node);
        ...
}
```

will pass a pointer to the lock substructure.

It would be nice to add casts to the implicit conversions to unnamed substructures, but this would conflict with existing C practice. The problem comes about from the almost ambiguous dual meaning of the cast operator. One usage is conversion; for example `(double)5` is a conversion, but `(struct lock*)node` is a PL/1 ‘‘unspec.’’

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. Initialisation indexes

In initialisers of arrays, one may place a constant expression in square brackets before an initialiser. This causes the next initialiser to go in that indicated element. This feature comes from the expanded use of `enum` declarations. Example:

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

This example also shows a micro-extension – it is legal to place a comma on the last `enum` in a list. (Wow! What were they thinking?)

3.6. External register

The declaration `extern register` will dedicate a register to a variable on a global basis. It can only be used under special circumstances. External register variables must be identically declared in all modules and libraries. The declaration is not for efficiency, although it is efficient, but 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-machine 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. Goto case, goto default

The last extension has not been used, so is probably not a good idea. In a switch it is legal to say `goto case 5` or `goto default` with the obvious meaning.

4. Object module conventions

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

4.1. Register saving

In most compilers, the called program saves the exposed registers. This compiler has the caller save the registers. There are arguments both ways. With caller-saves, the leaf subroutines can use all the registers and never save them. If you spend a lot of time at the leaves, this seems preferable. In called-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. The convincing argument is that with caller-saves, the decision to registerise a variable can include the cost of saving the register across calls.

Perhaps the best method, especially on computers with many registers, would to have both caller-saved registers and called-saved registers.

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

4.2. Calling convention

Rule: ‘‘It is a mistake to use the manufacturer’s special call instruction.’’ The relationship between the (virtual) frame pointer and the stack pointer is known by the compiler. It is just extra work to mark this known point with a real register. If the stack grows towards lower addresses, then there is no need for an argument pointer. It is also at a known offset from the stack pointer. If the convention is that the caller saves the registers, then the entry point saves no registers. There is therefore no advantage to a special call instruction.

On the National 32100 computer programs compiled with the simple ‘‘jsr’’ instruction would run in about half the time of programs compiled with the ‘‘call’’ instruction.

4.3. Floating stack pointer

On computers like the VAX and the 68020, there is a short, fast addressing mode to push and pop the top of stack. In a sequence of subroutine calls within a basic block, arguments may be pushed and popped many times. Pushing arguments is, to some extent, a useful activity, but popping is just overhead. If the arguments of the first call are left on the stack for the second call, a single pop of both sets of arguments (usually an ‘‘add’’ instruction) will suffice for both calls. This optimisation is worth several percent in both space and runtime of object modules.

The only penalty comes in debugging, when the distance between the stack pointer and the frame pointer must be communicated as a program counter-dependent variable rather than a single variable for an entire subroutine.

4.4. 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. These 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. An earlier version of the compiler passed a null pointer in such cases, but was changed to pass a dummy argument after measuring some running programs.

There is also a danger of calling a function that returns a structure without declaring it as such. Before ANSI C function prototypes, this would probably be enough consideration to find some other way of returning structures. These compilers have an option that complains every time that a subroutine is compiled that has not been fully specified by a prototype, which catches this and many other errors. This is now the default and is highly recommended for all ANSI C compilers.

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. [Joh79b] All code is put into a parse tree and collected, without interpretation, for the body of a function. The later passes then walk this tree.

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

Even though it is just one pass of many, the parsing take 50% of the execution time of the whole compiler. Most of this (75%) is due to the inefficiencies of YACC. The remaining 25% of the parse time is due to the low level character handling. The flexibility of YACC was very important in the initial writing of the compiler, but it would probably be worth the effort to write a custom recursive descent parser.

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 optimisation

The next pass performs optimisations and transformations of the tree. Typical of the transforms: `&*x` and `*&x` are converted into `x`. Constant expressions are converted to constants in this pass.

5.4. Arithmetic rewrites

This is another machine-independent optimisation. 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 computer is defined as the expression that is legal in the address field of a machine language instruction. The addressability of different computers varies widely. At one end of the spectrum are the 68020 and VAX, which allow a complex array of incrementing, decrementing, indexing and relative addressing. At the other end is the MIPS, which allows registers and constant offsets from the contents of a register. The addressability can be different for different instructions within the same computer.

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 labelled with small integers. When an internal node is encountered, it is labelled by consulting a table indexed by the labels on the left and right subtrees. For example, on the 68020 computer, 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 `A1` are addressable while marks of the form `N1` are not addressable.

Node	**Marked**
`name`	`A1`
`const`	`A2`
`&A1`	`A3`
`A3+A1`	`N1` (note this is not addressable)
`*N1`	`A4`

Here there is a distinction between a node marked `A1` and a node marked `A4` because the address operator of an `A4` node is not addressable. So to extend the table:

Node	**Marked**
`&A4`	`N2`
`N2+N1`	`N1`

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

In the same bottom-up pass of the tree, the nodes are labelled 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 simple 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 jumps. This is fundamentally different than compiling an expression. The result of the boolean code generator is the position of the program counter and not an expression. 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. Registerisation

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 registerised within a routine are placed in a table. (Suitable variables are all automatic or external scalars that do not have their addresses extracted. Some constants that are hard to reference are also considered for registerisation.) Four separate data flow equations are evaluated over the routine 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 registerising. 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 registerised 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 very satisfying. 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 registerisation 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 optimisation

The next pass walks the machine code for opportunistic optimisations. For the most part, this is highly specific to a particular computer. One optimisation that is performed on all of the computers 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 relabelling 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 if 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)
(no use of `a`)	
`USE b`	`USE a`
(no use of `a`)	
`SET b`	`SET b`

Experiments have shown that it is marginally worth while 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 relabelling 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`
(no use of `b`)	
`USE a`	`USE b`
(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 organisation is that the costs of registerisation calculated in the previous pass must depend on how well this pass can detect and remove redundant instructions. Often, a fine candidate for registerisation is rejected because of the cost of instructions that are later removed. Perhaps the registerisation pass should discount a large percentage of a ‘‘move’’ instruction anticipating the effectiveness of this pass.

5.9. Writing the object file

The last pass walks the internal assembly language and writes the object file. The object file is reduces 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 machine-independent. All integer and floating numbers in the object code are converted to known formats and byte orders. This is important for Plan 9 because the compiler might be run on different computers.

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 optimisations 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, unconditional branch instructions are removed. 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. An example of this appears in a later section.

The next pass allocates addresses for all external data. Typical of computers is the 68020 which can reference ±32K from an address register. The loader allocates the address register `A6` as the static pointer. The value placed in `A6` 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 computer, the loader makes a pass over the internal structures exchanging instructions to try to fill ‘‘delay slots’’ with useful work. (A delay slot on the MIPS is a euphemism for a timing bug that must be avoided by the compiler.) 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 20% of all instructions are in delay slots. About 50% of these are useful instructions and 50% are ‘‘noops.’’ The vendor supplied assembler does this job much more effectively filling about 80% of the delay slots with useful instructions.

On the 68020 computer, 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 on each other. 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 computer, 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 has source line numbers at its disposal, but the interpretation of these numbers relative to `#include` files is not done. The loader is also in a good position to perform some global optimisations, but this has not been exploited.

7. Performance

The following is a table of the source size of the various components of the compilers.

lines	module
409	machine-independent compiler headers
975	machine-independent compiler Yacc
5161	machine-independent compiler C
819	68020 compiler headers
6574	68020 compiler C
223	68020 loader headers
4350	68020 loader C
461	MIPS compiler headers
4820	MIPS compiler C
263	MIPS loader headers
4035	MIPS loader C
3236	Crisp compiler headers
2526	Crisp compiler C
132	Crisp loader headers
2256	Crisp loader C

The following table is timing of a test program that does Quine-McClusky boolean function minimisation. The test program is a single file of 907 lines of C that is dominated by bit-picking and sorting. The execution time does not significantly depend on library implementation. Since no other compiler runs on Plan 9, these tests were run on a single-processor MIPS 3000 computer with vendor supplied software. The optimiser in the vendor supplied compiler is reputed to be extremely good. Another compiler, *lcc*, is compared in this list. *Lcc* is another new and highly portable compiler jointly written at Bell Labs and Princeton. None of the compilers were tuned on this test.

1.0s	new cc compile time
0.5s	new cc load time
90.4s	new cc run time
1.6s	vendor cc compile time
0.1s	vendor cc load time
138.8s	vendor cc run time
4.0s	vendor cc –O compile time
0.1s	vendor cc –O load time
84.7s	vendor cc –O run time
1.6s	vendor lcc compile time
0.1s	vendor lcc load time
96.3s	vendor lcc run time

Although it was not possible to directly compare *gcc* to the new compiler, *lcc* typically compiles in 50% of the time of *gcc* and the object runs in 75% of the time of *gcc*. The original *pcc* compiler is also not directly compared. It is too slow in both compilation and runtime to compete with the above compilers. Since *pcc* has not been updated to accept ANSI function prototypes, it is also hard to find test programs to form a comparison.

8. Example

Here is a small example of a fragment of C code to be compiled on the 68020 compiler.

```
int   a[10];
void
f(void)
{
      int i;

      for(i=0; i<10; i++)
            a[i] = i;
}
```

The following is the tree of the assignment statement after all machine-independent passes. The numbers in angle brackets are addressabilities. Numbers 10 or larger are addressable. The addressability, 9, for the `INDEX` operation means addressable if its second operand is placed in an index register. The number in parentheses is the Sethi-Ullman complexity. The typing information is at the end of each line.

```
ASSIGN (1) long
      INDEX <9> long
            ADDR <12> *long
                  NAME "a" 0 <10> long
            NAME "i" -4 <11> *long
      NAME "i" -4 <11> long
```

The following is the 68020 machine language generated before the registerisation pass. Note that there is no assembly language in this compiler; this is a print of the internal form in the same sense as the previous tree is a print of that internal form.

Here is some explanation of notation: `(SP)` denotes an automatic variable; `(SB)` denotes an external variable; `A7` is the stack pointer, `$4` is a constant.

```
f:    TEXT
      SUBL  $4,A7
      CLRL  i(SP)
loop: MOVL  $10,R0
      CMPL  R0,i(SP)
      BLE   ret
      MOVL  i(SP),R0
      MOVL  i(SP),a(SB)(R0.L*4)
      ADDL  $1,i(SP)
      JMP   loop
ret:  ADDL  $4,A7
      RTS
```

The following is the code after all compiling passes, but before loading:

```
f:    TEXT
      SUBL  $4,A7
      CLRL  R1
loop: MOVL  $10,R0
      CMPL  R0,R1
      BLE   ret
      MOVL  R1,a(SB)(R1.L*4)
      ADDL  $1,R1
      JMP   loop
ret:  ADDL  $4,A7
      RTS
```

The following is the code produced by the loader. The only real difference is the expansion and inversion of the loop condition to prevent an unconditional branch.

```
f:     TEXT
       CLRL  R1
loop:  MOVL  $10,R0
       CMPL  R0,R1
       BLE   ret
l1:    MOVL  R1,a(SB)(R1.L*4)
       ADDL  $1,R1
       MOVL  $10,R0
       CMPL  R0,R1
       BGT   l1
ret:   RTS
```

The compare sequence

```
       MOVL  $10,R0
       CMPL  R0,R1
```

was expanded from the single instruction

```
       CMPL  $10,R1
```

because the former is both shorter and faster. The relatively dumb loader made a second copy of the sequence without realising that the

```
       MOVL  $10,R0
```

is redundant.

9. Conclusions

The new compilers compile quickly, load slowly, and produce medium quality object code. The compilers are relatively portable, requiring but a couple weeks work to produce a compiler for a different computer. As a whole, the experiment is a success. For Plan 9, where we needed several compilers with specialised features and our own object formats, the project was indispensable.

Two problems have come up in retrospect. The first has to do with the division of labour between compiler and loader. Plan 9 runs on a multi-processor 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 will try to put some of the loader work back into the compiler.

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

10. References

Aho87. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers – Principles, Techniques, and Tools,* Addison Wesley, Reading, MA (1987).

Joh79a. S. C. Johnson, ‘‘A Tour Through the Portable C Compiler,’’ in *UNIX Programmer’s Manual, Seventh Ed., Vol. 2A,* AT&T Bell Laboratories, Murray Hill, NJ (1979).

Joh79b. S. C. Johnson, ‘‘YACC – Yet Another Compiler Compiler,’’ in *UNIX Programmer’s Manual, Seventh Ed., Vol. 2A,* AT&T Bell Laboratories, Murray Hill, NJ (1979).

Kan88. Gerry Kane, *MIPS RISC Architecture,* Prentice-Hall, Englewood Cliffs, NJ (1988).

Ker88. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language, Second Edition,* Prentice-Hall, Englewood Cliffs, NJ (1988).

Mot85. Motorola, *MC68020 32-Bit Microprocessor User's Manual, Second Edition,* Prentice-Hall, Englewood Cliffs, NJ (1985).

Pik90. Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, ‘‘Plan 9 from Bell Labs,’’ *Proc. UKUUG Conf.*, London, UK (July 1990).

Set70. R. Sethi and J. D. Ullman, ‘‘The Generation of Optimal Code for Arithmetic Expressions,’’ *J. ACM* **17**(4), pp. 715-728 (1970).

Szy78. T. G. Szymanski, ‘‘Assembling Code for Machines with Span-dependent Instructions,’’ *Comm. ACM* **21**(4), pp. 300-308 (1978).