A Retargetable Compiler for ANSI C

Christopher W. Fraser
AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974

and

David R. Hanson
Department of Computer Science, Princeton University, Princeton, NJ 08544

Research Report CS-TR-303-91

February 1991

Abstract

`lcc` is a new retargetable compiler for ANSI C. Versions for the VAX, Motorola 68020, SPARC, and MIPS are in production use at Princeton University and at AT&T Bell Laboratories. With a few exceptions, little about `lcc` is unusual — it integrates several well engineered, existing techniques — but it is smaller and faster than most other C compilers, and it generates code of comparable quality. `lcc`'s target-independent front end performs a few simple, but effective, optimizations that contribute to good code; examples include simulating register declarations and partitioning switch statement cases into dense tables. It also implements target-independent function tracing and expression-level profiling.

Introduction

lcc is a new retargetable compiler for ANSI C [2]. It has been ported to the VAX, Motorola 68020, SPARC, and MIPS R3000, and it is in production use at Princeton University and at AT&T Bell Laboratories. When used with a compliant preprocessor and library, lcc passes the conformance section of Version 2.00 of the Plum-Hall Validation Suite for ANSI C.[1]

Other reports describe lcc's storage manager [13], intermediate language [8], code generator [7], and register manager [9]. This report surveys the remaining features of lcc that may interest some readers. Chief among these are its performance, some aspects of its design that support this performance, and some features for debugging and profiling user code.

Design

With a few exceptions, lcc uses well established compiler techniques. The front end performs lexical, syntactic, and semantic analysis, and some machine-independent optimizations, which are described below. Both the lexical analyzer and the recursive-descent parser are hand-written. Theoretically, this approach complicates both future changes and fixing errors, but accommodating change is less important for a standardized language like ANSI C, and there have been few lexical or syntactic errors. Indeed, less than 15 percent of lcc's code concerns parsing, and the error rate in that code is negligible. Despite its theoretical prominence, parsing is a relatively minor component in lcc as in other compilers; semantic analysis, optimization, and code generation are the major components and account for most of the code and most of the errors.

The target-independent front end and a target-dependent back end are packaged as single program, tightly coupled by a compact, efficient interface. The interface consists of a few shared data structures, 17 functions, and a 36-operator dag language. The functions emit function prologues, define globals, emit data, etc., and most are simple. The dag language encodes the executable code from a source program; it corresponds to the "intermediate language" used in other compilers, but it is smaller than typical intermediate languages. Reference [8] describes the interface.

Code generators are generated automatically from compact, rule-based specifications [7]. Some rules rewrite intermediate code as naive assembly code. Others peephole-optimize the result. They are compiled into a monolithic hard-coded program that accepts dags annotated with intermediate code, and generates, optimizes, and emits code for the target machine. Hard-coding contributes significantly to lcc's speed.

[1] The lcc front end and a sample code generator are available for anonymous ftp from princeton.edu. The file README in the directory pub/lcc gives details.

component	code	rules	generated code generator
front end	968+7847		
back-end support	114+741		
VAX	35+170	178	5782
MIPS	40+378	104	2966
68020	42+190	145	8301
SPARC	40+290	128	3888
VAX+68020+SPARC symbol table emitters	584		
naive VAX code generator	67+578		
rule compiler	1285		

Table 1: Number of Lines in `lcc` Components.

Table 1 shows the number of lines in each of `lcc`'s components. The notation $h + c$ indicates h lines of definitions in "header" files and c lines of C code. The "back-end support" is back-end code that shared by four back ends, e.g., initialization and most of the register manager.

Target-specific files include a configuration header file, which defines parameters like the widths and alignments of the basic datatypes, target-specific interface functions, e.g., those that emit function prologues, and code generation rules, from which the code generators are generated by the rule compiler, which is written in Icon [11]. Retargeting `lcc` requires involves writing these three back-end components, which vary from 377 to 522 lines in existing back ends. In practice, new back ends are implemented by writing new rules and editing copies of an existing configuration and set of interface functions.

All of `lcc`'s production back ends use the technology summarized above and detailed in Reference [7]. The interface between the front and back end does not depend on this technology; other back ends that conform to the interface specification can be used. For example, Reference [8] details a hand-written code generator that emits naive VAX code.

While `lcc` uses well established techniques, it uses some of their more recent incarnations, each of which contributes to `lcc`'s efficiency as described below.

Lexical Analysis

The design of the input module and of the lexical analyzer and judicious code tuning of the lexical analyzer contribute to `lcc`'s speed.

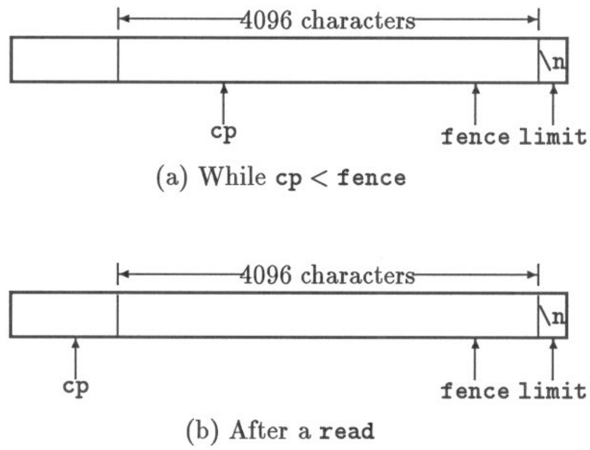

Figure 1: Input Buffering.

Input and lexical analysis use variations of the design described in Reference [20]. Since the lexical analyzer is the only module that inspects every input character, the design avoids extraneous per-character processing and minimizes character movement by scanning tokens directly out of a large input buffer.

Input is read directly from the operating system into a 4096-character buffer as depicted in Figure 1a, and `cp` and `limit` delimit the unscanned portion of the buffer. The next token is scanned by advancing `cp` across white space and switching on the first character of the token, `*cp`. `cp` is advanced as the token is recognized.

Newlines, denoted by `\n`, cannot occur within C tokens, which explains the newline at `*limit` shown in Figure 1. This newline terminates a scan for any token so a separate, per-character test for the end of the buffer is unnecessary. When a newline is encountered, an input module function is called to refill the input buffer, if necessary, and to increment the line number.

ANSI C stipulates a maximum line length of no less than 509, but few compilers insist on a specific limit. Tokens, however, can be limited to 32 characters; string literals are an exception, but they are handled as a special case.

In general, an input buffer ends with a partial token. To insure that an entire token lies between `cp` and `limit`, the end of the buffer is moved to the memory locations *preceding* the buffer whenever `cp` passes `fence`. Doing so concatenates a partial token with its tail after the next read as shown in Figure 1b. Testing if `cp` has passed `fence` is done for each token after `cp` is advanced across white space.

The important consequence of this design is that most of the input characters are accessed by `*cp` and many are never moved. Only identifiers (excluding keywords) and string literals that appear in executable code are copied out of the buffer into permanent storage.

Reference [20]'s algorithm moves partial *lines* instead of partial tokens and does so after scanning the *first* newline in the buffer. But this operation overwrites storage before the buffer when a partial line is longer than a fixed maximum. The algorithm above avoids this problem, but at the per-token cost of comparing `cp` with `fence`.

Instead of actually using `cp` as suggested above, `cp` is copied to the register variable `rcp` upon entry to the lexical analyzer, and `rcp` is used in token recognition. `rcp` is assigned to `cp` before the lexical analyzer returns. Using `rcp` improves performance and makes scanning loops compact and fast, e.g., white space is elided by

```
while (map[*rcp]&BLANK)
   rcp++;
```

`map[c]` is a mask that classifies character `c` as suggested in Reference [20]; e.g., `map[c]&BLANK` is non-zero if `c` is a white-space character (but not a newline). `lcc` generates four VAX instructions for the body of this loop:

```
        jbr L142
L141:   incl r11
L142:   cvtbl (r11),r5
        bicl3 $-2,_map[r5],r5
        jneq L141
```

`rcp` is register `r11`. Some optimizing compilers can make similar improvements locally, but not across potentially aliased assignments and calls to other, irrelevant functions.

Keywords are recognized by a hard-coded decision tree, e.g.,

```
case 'i':
   if (rcp[0] == 'f'
   && !(map[rcp[1]]&(DIGIT|LETTER))) {
      cp = rcp + 1;
      return IF;
   }
   if (rcp[0] == 'n'
   &&  rcp[1] == 't'
   && !(map[rcp[2]]&(DIGIT|LETTER))) {
      cp = rcp + 2;
      return INT;
   }
   goto id;
```

IF and INT are defined as the token codes for the keywords if and int, respectively, and id labels the code that scans identifiers. This code is generated automatically by a 50-line C program and included in the lexical analyzer during compilation.

The VAX code generated for this fragment follows; again, r11 is rcp.

```
L347:   cmpb (r11),$102
        jneq L348
        cvtbl 1(r11),r5
        bicl3 $-13,_map[r5],r5
        jneq L348
        addl3 $1,r11,_cp
        movl $77,r0
        ret
L348:   cmpb (r11),$110
        jneq L226
        cmpb 1(r11),$116
        jneq L226
        cvtbl 2(r11),r5
        bicl3 $-13,_map[r5],r5
        jneq L226
        addl3 $2,r11,_cp
        movl $5,r0
        ret
```

Thus, the keyword int is recognized by less than a dozen instructions, many less than are executed when a table is searched for keywords, even if perfect hashing is used.

As in other compilers [1], strings that must be saved (identifiers and string literals) are hashed into a table in which a string appears only once, which saves space. For performance, there are variants for installing strings of digits and strings of known length. After installation, strings are known by their addresses and the characters are accessed only for output. For example, looking a name up in the symbol table is done by hashing on the address of the name; string comparison is unnecessary.

Symbol Tables

Fast symbol table manipulation also contributes to lcc's speed. It took several versions of the symbol table module to arrive at the current one, however.

Symbols are represented with structures defined by

```
struct symbol {
   char *name;          /* symbol name */
   int scope;           /* scope level */
```

```
    ...
};
```

The symbol table module uses hash tables for symbol tables; the initial version used a single table for all scopes, i.e.,

```
struct entry {
    struct symbol sym;  /* this symbol */
    struct entry *link; /* next entry on hash chain */
};
struct table {
    struct entry *buckets[HASHSIZE]; /* hash buckets */
};
```

Symbols are wrapped in **entry** structures to keep the linkage information private to the symbol table module.

Scope entry required no code. Each new symbol was added to the head of its hash chain and thereby hid symbols with the same names, which appeared further down on the same chains. At scope exit, however, entries at the current scope level, indicated by the value of **level**, were removed from the table ***tp** by the code

```
for (i = 0; i < HASHSIZE; i++) {
    struct entry *p = tp->buckets[i];
    while (p && p->sym.scope == level)
        p = p->link;
    tp->buckets[i] = p;
}
```

Measurements revealed that this code accounted for over 5 percent of **lcc**'s execution time on typical input. This code scanned the hash buckets even for scopes that introduce no new symbols, which are common in C.

The second version of the symbol table module used a separate hash table for each scope level:

```
struct table {
    struct table *previous; /* table at lower scope */
    struct entry *buckets[HASHSIZE]; /* hash buckets */
};
```

Searching for a symbol took the same number of comparisons, but also required a traversal of the list of separate tables, e.g.,

```
struct symbol *lookup(char *name, struct table *tp) {
    struct entry *p;
    unsigned h = ((unsigned)name)&(HASHSIZE-1);
```

```
    do
        for (p = tp->buckets[h]; p; p = p->link)
            if (name == p->sym.name)
                return &p->sym;
    while (tp = tp->previous);
    return 0;
}
```

Notice that symbol names are compared by simply comparing addresses as explained in the previous section. Despite the conventional wisdom about hashing functions [16], using a power of two for **HASHSIZE** gave better performance; using a prime instead and modulus in place of masking slowed **lcc**.

This variation reduced the scope exit code to

```
tp = tp->previous
```

for table ***tp**. Unfortunately, scope entry then required allocation and initialization of a table:

```
struct table *new = (struct table *)alloc(sizeof *new);
new->previous = tp;
for (i = 0; i < HASHSIZE; i++)
    new->buckets[i] = 0;
tp = new;
```

So, the time wasted at scope exit in the first version was traded for a similar waste at scope entry in the second version.

The symbol table module in actual use avoids this waste by lazy allocation and initialization of tables. Tables include their associated scope level:

```
struct table {
    int level;              /* scope level for this table */
    struct table *previous; /* table at lower scope */
    struct entry *buckets[HASHSIZE]; /* hash buckets */
};
```

New tables are allocated and initialized only when a symbol is installed:

```
struct symbol *install(char *name, struct table **tpp) {
    unsigned h = ((unsigned)name)&(HASHSIZE-1);
    struct table *tp = *tpp;
    struct entry *p = (struct entry *)alloc(sizeof *p);

    if (tp->level < level) {
        int i;
        struct table *new = (struct table *)alloc(sizeof *new);
        new->previous = tp;
```

```
        new->level = level;
        for (i = 0; i < HASHSIZE; i++)
            new->buckets[i] = 0;
        *tpp = tp = new;
    }
    p->sym.name = name;
    p->sym.scope = tp->level;
    ...
    p->link = tp->buckets[h];
    tp->buckets[h] = p;
    return &p->sym;
}
```

Since few scopes in C, which are delimited by compound statements, declare new symbols, the lazy allocation code above is rarely executed and entry to most scopes is nearly free. The scope exit code must check before discarding a table, but remains simple:

```
if (tp->level == level)
    tp = tp->previous;
```

This design also simplifies access to separate tables. For example, the table that holds globals is at the end of the list of identifier tables; by making it the value of `globals`, symbols can be installed into it directly. In the initial implementation, a global declared at a nested scope had to be inserted in the middle of its hash chain.

Storage Management

Allocation and deallocation in early versions of `lcc` accounted for a significant portion of the total execution time. Replacing the naive use of `malloc` and `free` reduced total execution time by about 8–10 percent. As detailed in Reference [13], allocation is based on the lifetime of the objects allocated, and all objects with the same lifetime are freed at once.

This approach to storage management simplified `lcc`'s code. Initially, each object type had explicit deallocation code, perhaps replicated at several points. Some of this code was intricate, e.g., involving complex loops or recursive data structure traversals. Allocation incurred an obligation to provide the necessary deallocation code, so there was a tendency to use algorithms that avoided allocation, perhaps at the expense of time, complexity, and flexibility. And it was easy to forget deallocation, resulting in storage leaks.

The current scheme eliminated nearly all explicit deallocation code, which simplified the compiler and eliminated storage leaks. More importantly, it encouraged the use of simple applicative algorithms, e.g., in rewriting trees. The replacements cost space, but not time, since allocation and deallocation are nearly free. Besides contributing to fast compilation, the other visible benefit of this approach is that `lcc` imposes few arbitrary limits on its input; e.g., it permits any number of cases in switch statements, any number of parameters and locals, block nesting to any depth, expressions of arbitrary complexity, initializations of arbitrary size, etc. These quantities are limited only by the memory available.

Optimization

`lcc` is not properly called an "optimizing" compiler because it does no global optimization, *per se*. Its front end does, however, perform some simple, target-independent transformations that help its back ends generate good local code.

The front end eliminates local common subexpressions, folds constant expressions, and makes numerous simple transformations that improve the quality of local code [12]. Many of these improvements are simple tree transformations that lead to better addressing code.

The front end lays out loops so as to reduce the number of unconstructive branches [3], e.g., the code for

`for (`e_1`;` e_2`;` e_3`)` S

has the form

```
     goto L1
L2:  S
L3:  e3
L1:  if (e2) goto L2
```

The `goto L1` is omitted if e_2 is initially non-zero. In addition, the front end eliminates branch chains and dead branches.

The selection code for switch statements is generated entirely by the front end. It generates a binary search of dense branch tables [5], where the density is the percentage of non-default branch table entries. For example, with the default density of 0.5, a switch statement with the case values 1, 2, 6–8, 1001–1004, and 2001–2002 has the following VAX selection code. Register `r4` holds the value of the switch expression, `L3–15` label the statements for the case values above, and `L1` is the default label.

```
        cmpl r4,$1001
        jlss L17
        cmpl r4,$1004
        jgtr L16
        movl _18-4004[r4],r5
        jmp (r5)
_18:    .long L8
```

```
        .long L9
        .long L10
        .long L11
L17:    cmpl r4,$1
        jlss L1
        cmpl r4,$8
        jgtr L1
        movl _21-4[r4],r5
        jmp (r5)
_21:    .long L3
        .long L4
        .long L1
        .long L1
        .long L1
        .long L5
        .long L6
        .long L7
L16:    cmpl r4,$2001
        jlss L1
        cmpl r4,$2004
        jgtr L1
        movl _24-8004[r4],r5
        jmp (r5)
_24:    .long L12
        .long L13
        .long L14
        .long L15
```

The density can be changed by a command-line option; e.g., `-d0` yields a single branch table for each switch statement, and `-d1` requires that all branch tables be fully populated.

Finally, the front end simulates register declarations for all scalar parameters and locals that are referenced at least 3 times and do not have their addresses taken explicitly. Locals are announced to the back ends with explicitly declared `register` locals followed by the remaining locals in the order of decreasing frequency of use. Each top-level occurrence of an identifier counts as 1 reference. Occurrences in a loop, either of the then/else arms of an if statement, or a case in a switch statement each count, respectively, as 10, 1/2, or 1/10 references. These values are adjusted to account for nested control structures. The next section describes how these estimated counts may be replaced with counts from an actual profile.

This scheme simplifies register assignment in the back ends, and explicit `register` declarations are rarely necessary. For example,

```
strcpy(char *s1, char *s2) { while (*s1++ = *s2++); }
```

yields the VAX code

```
_strcpy:.word 0x0
        movl 4(ap),r4
        movl 8(ap),r5
L26:    movb (r5)+,(r4)+
        jneq L26
        ret
```

Features

`lcc` provides a few noteworthy features that help users develop, debug, and profile ANSI C programs. For example, an option causes `lcc` to print ANSI-style C declarations for all defined globals and functions. For instance, the code (adapted from Section 6.2 of Reference [14])

```
typedef struct point { int x,y; } point;
typedef struct rect { point pt1, pt2; } rect;

point addpoint(p1, p2) point p1, p2; {
   p1.x += p2.x;
   p1.y += p2.y;
   return p1;
}
int ptinrect(p, r) point p; rect r; {
   return p.x >= r.pt1.x && p.x < r.pt2.x
      && p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

generates the declarations

```
extern point addpoint(point, point);
extern int ptinrect(point, rect);
```

Editing such output can simplify conversion to ANSI C.

Another option causes `lcc` to issue warnings for declarations and casts of function types without prototypes. These include pointers to functions, which are easy to overlook when updating pre-ANSI code. For example, it is likely that `char *(_alloc)()` should be updated to be `char *(_alloc)(size_t)`.

Debugging

`lcc` supports the standard debugger symbol tables on VAXes and Suns. It also has two options of its own to assist in program debugging.

Dereferencing zero pointers is a frequent C programming error. On some systems, execution continues until the consequences cause a fault somewhere unrelated to the actual point of error. To help catch such errors, an option causes `lcc` to generate code to test for dereferencing zero pointers. If a zero pointer is detected, the offending file name and line number are reported on the standard error, e.g.,

```
null pointer dereferenced @foo.c:36
```

and the program terminates by calling the standard library function `abort`.

Some languages provide built-in facilities for tracing function calls and returns [11]. An option instructs `lcc` to generate calls to `printf` (or a user-specified equivalent) just after entry to each function and just before each return. The entry code prints the arguments and the return code prints the value returned. For example, calling the functions shown above would elicit messages like

```
addpoint#2(p1=(point){x=0,y=0},p2=(point){x=10,y=10}) called
addpoint#2 returned (point){x=10,y=10}
...
ptinrect#1(p=(point){x=-1,y=-1},
   r=(rect){pt1=(point){x=10,y=10},pt2=(point){x=310,y=310}}) called
ptinrect#1 returned 0
```

(Long lines have been folded to fit this page.) As illustrated by this output, the messages show the full details of the arguments, including structure contents. The numbers that follow function names, e.g., `#2`, are activation numbers and can help locate a specific call and its return.

These debugging options are implemented entirely in the front end and thus are available on all of `lcc`'s targets.

Profiling

`lcc` supports `prof`-style (viz. [6, `prof` command]) and `gprof`-style [10] execution profiling on VAXes and Suns. These profilers sample the location counter periodically to obtain an estimate of the percentage of total execution time spent in each function, and they report the number of calls to each function.

Heeding long-standing advice [15, 17], `lcc` also supports frequency-based profiling. An option causes `lcc` to emit counters that record the number of times each *expression* is executed, and the values of these counters are written to the file `prof.out` when the program terminates. A companion program, `bprint`, reads `prof.out` and prints the source code annotated with execution counts, e.g.,

```
...
4  main()
5  <1>{
...
```

```
12     <1>queens(0);
13     return <1>0;
14 <1>}
15
16 queens(c)
17 <1965>{
18     int r;
19
20     for (<1965>r = 0; <15720>r < 8; <15720>r++)
21         if (<15720>rows[r] && <5508>up[r-c+7] && <3420>down[r+c]){
22             <2056>rows[r] = up[r-c+7] = down[r+c] = 0;
23             <2056>x[c] = r;
24             if (<2056>c == 7)
25                 <92>print();
26             else
27                 <1964>queens(c + 1);
28             <2056>rows[r] = up[r-c+7] = down[r+c] = 1;
29         }
30 <1965>}
...
```

Execution counts are enclosed in angle brackets. The counts on the outermost braces for `queens` give the number of calls. Line 21 shows the benefit of associating a count with each expression instead of each line; the counts reveal that `up[r-c+7]` was tested only slightly more than one-third of the number of times the if statement was executed. Conditional expressions are annotated similarly.

Users sometimes report an "off-by-one" bug when they see that `r < 8` in line 20 was executed the same number of times as `r++`. These counts are a consequence of the way `lcc` lays out for loops and eliminates the test before the first iteration, as described above.

Data in `prof.out` accumulates, so it is possible to execute a program repeatedly and then have `bprint` display the cumulative frequencies. This method is particularly useful for developing test data that exercises all parts of a program: `<0>` highlights untested code.

Another option causes `lcc` to read `prof.out` and use the counts therein to compute the frequency of use of each identifier instead of using the estimates described in the previous section. Doing so may reduce the number of uses for identifiers that appear in loops that rarely executed more than once, and increase the number of uses for those that appear in then/else arms that are executed most of the time.

Complex preprocessor macros can obscure `bprint`'s presentation. It necessarily uses post-expansion source coordinates to annotate pre-expansion source files.

Profiling code also records the number of calls made from each call site, which can be used to reconstruct the dynamic call graph. `bprint` prints a line for each edge, e.g.,

```
1       queens  from main       in 8q.c:12.8
1964    queens  from queens     in 8q.c:27.11
92      print   from queens     in 8q.c:25.10
```

This output shows that all but one of the calls to `queens` was from the call at character 11 in line 27. This kind of data is particularly helpful in identifying hot spots that are caused by inappropriate calls to a function instead of inefficiencies within the function itself. Such data can also help identify functions that might profitably be replaced with two functions so that one can handle the common case more efficiently [4, Sec. 5.3].

Expression execution frequency profiling is implemented entirely by the front end. The only machine dependency is the name of the ultimate termination function in the revised `exit` function that writes `prof.out` at program termination.

The implementation is a machine-independent variation of the method described in Reference [21]. The front end generates an array of counters for each file and starts each expression with code to increment the appropriate counter. In also builds a parallel array that holds the source coordinates corresponding to each counter. At the entry point of each function, the front end generates the equivalent of

```
if (!_yylink.link) {
   extern struct _bbdata *_bblist;
   _yylink.link = _bblist;
   _bblist = &yylink;
}
_prologue(&callee);
```

A `_bbdata` structure is generated for each file:

```
static struct _bbdata {
   struct _bbdata *link;
   unsigned npoints;
   unsigned *counts;
   unsigned *coords;
   struct func *funcs;
} _yylink;
```

The `counts` and `coords` fields point the arrays mentioned above, which each have `npoints` entries. The entry point code uses the `link` field to add each file's `_bbdata` structure to the list headed by `_bblist`, which the revised `exit` function walks to emit `prof.out`.

`_prologue` accumulates the dynamic call graph. It is passed one of the `func` structures — one for each function — that appear on the list emanating from `_yylink.funcs`:

```
struct func {
   struct func *link;
   struct caller {
      struct caller *link;
      struct callsite *caller;
      unsigned count;
   } *callers;
   char *name;
   unsigned coord;
};
```

The `name` and `coord` fields give the function's name and beginning source coordinate, respectively. `callers` points to a list of `caller` structures, one for each call site. Each `caller` structure records the number of calls from the caller's `callsite`:

```
struct callsite {
   char *file;
   char *name;
   unsigned coord;
};
```

`caller` structures are allocated at execution time and point to `callsites`, which are generated by the front end at compile time.

Just before each call, the front end generates an assignment of a pointer to a `callsite` structure to the global variable `_caller`. `_prologue` uses `_caller` to record an edge in the dynamic call graph. If a record of the caller already exists, its count is simply incremented. Otherwise, a `caller` structure is allocated and prefixed to the callee's list of callers.

```
_prologue(struct func *callee) {
   static struct caller callers[4096];
   static int next;
   struct caller *p;

   for (p = callee->callers; p; p = p->link)
      if (p->caller == _caller) {
         p->count++;
         break;
      }
   if (!p && next < sizeof callers/sizeof callers[0]) {
      p = &callers[next++];
```

```
        p->caller = _caller;
        p->count = 1;
        p->link = callee->callers;
        callee->callers = p;
    }
    _caller = 0;
}
```

Profiling can be restricted to only those files of interest. The counts printed by `bprint` will be correct, but some edges may be omitted from the call graph. For example, if `f` calls `g` calls `h` and `f` and `h` are compiled with profiling, but `g` is not, `bprint` will report that `f` called `h`. The total number of calls to each function is correct, however.

Performance

`lcc` emits local code that is comparable to that emitted by the generally available alternatives. Table 2 summarizes the results of compiling and executing the C programs in the SPEC benchmarks [18] with three compilers on the four machines listed above. Configuration details are listed with each machine. `cc` and `gcc` denote, respectively, the manufacturer's C compiler and the GNU C compiler from the Free Software Foundation. The times are elapsed time in seconds and are the lowest elapsed times over several runs on lightly loaded machines. All reported runs achieved at least 97 percent utilization (i.e., the ratio of times $(user + system)/elapsed \geq 0.97$).

The entries with `-O` indicate compilation with the "default" optimization, which often includes some global optimizations. `lcc` performs no global optimizations. The `gcc` and `gcc -O` figures for `gcc1.35` on the MIPS are missing because this benchmark did not execute correctly when compiled with `gcc`.

`lcc` is faster than many (but not all [19]) other C compilers. Table 3 parallels Table 2, but shows compilation time instead of execution time. Except for the MIPS, the times are for running only the compiler proper; preprocessing, assembly, and linking time are not included. Two times are given for the MIPS because the manufacturer's `cc` front end consists of two programs; the first translates C to "u-code" and the second generates object code. Generating assembly language costs *more* than generating object code, so Table 3 gives both times for all compilers. The last row in Table 3 lists the number of non-blank lines and the total number of lines in each benchmark *after* preprocessing.

`lcc` is smaller than other compilers. Table 4 lists the sizes of the three compilers in kilobytes. Each entry is the sum of sizes of the program and data segments for the indicated compiler as reported by the UNIX `size` command.

	benchmark			
compiler	1. `gcc1.35`	8. `espresso`	22. `li`	23. `eqntott`
VAX: MicroVAX II w/16MB running Ultrix 3.1				
`lcc`	1734	2708	7015	3532
`cc`	1824	2782	7765	3569
`gcc`	1439	2757	7353	3263
`cc -O`	1661	2653	7086	3757
`gcc -O`	1274	2291	6397	1131
68020: Sun 3/60 w/24MB running SunOS 4.0.3				
`lcc`	544	1070	2591	567
`cc`	514	1005	3308	619
`gcc`	426	1048	2498	591
`cc -O`	428	882	2237	571
`gcc -O`	337	834	1951	326
MIPS: IRIS 4D/220GTX w/32MB running IRIX 3.3.1				
`lcc`	116	150	352	111
`cc`	107	153	338	100
`gcc`		188	502	132
`cc -O`	92	130	299	70
`gcc -O`		145	411	112
SPARC: Sun 4/260 w/32MB running SunOS 4.0.3				
`lcc`	196	370	790	209
`cc`	203	381	1094	275
`gcc`	186	411	1139	256
`cc -O`	150	296	707	183
`gcc -O`	127	309	788	179

Table 2: Execution Time for C SPEC Benchmarks in Seconds.

compiler	1. `gcc1.35`	8. `espresso`	22. `li`	23. `eqntott`
		benchmark		
VAX:				
`lcc`	792	237	69	36
`cc`	1878	576	174	79
`gcc`	1910	637	192	86
68020:				
`lcc`	302	90	28	15
`cc`	507	168	52	29
`gcc`	599	196	56	27
MIPS:				
`lcc`	97 195	35 63	10 24	6 16
`cc`	318 177	104 68	40 26	24 19
`gcc`	320 391	88 118	28 42	13 24
SPARC:				
`lcc`	103	38	12	8
`cc`	175	60	18	11
`gcc`	313	100	31	16
line counts	79102/250496	25717/58516	7070/22494	2680/6569

Table 3: Compilation Time for C SPEC Benchmarks in Seconds.

compiler	VAX	68020	MIPS	SPARC
`lcc`	181	244	280	276
`cc`	256	306	616	402
`gcc`	378	507	777	689

Table 4: Sizes of Compiler Executables in Kilobytes.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, Reading, MA, 1986.

[2] American National Standard Institute, Inc., New York. *American National Standards for Information Systems, Programming Language C ANSI X3.159–1989*, 1990.

[3] F. Baskett. The best simple code generation technique for while, for, and do loops. *SIGPLAN Notices*, 13(4):31–32, Apr. 1978.

[4] J. L. Bentley. *Writing Efficient Programs.* Prentice Hall, Englewood Cliffs, NJ, 1982.

[5] R. L. Bernstein. Producing good code for the case statement. *Software—Practice & Experience*, 15(10):1021–1024, Oct. 1985.

[6] Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA. *UNIX User's Manual, Reference Guide*, virtual VAX-11 version edition, Mar. 1984.

[7] C. W. Fraser. A language for writing code generators. *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 24(7):238–245, July 1989.

[8] C. W. Fraser and D. R. Hanson. A code generation interface for ANSI C. Technical Report CS-TR-270-90, Princeton University, Department of Computer Science, Princeton, NJ, July 1990.

[9] C. W. Fraser and D. R. Hanson. Simple register spilling in a retargetable compiler. *Software—Practice & Experience*, submitted.

[10] S. L. Graham, P. B. Kessler, and M. K. McKusick. An execution profiler for modular programs. *Software—Practice & Experience*, 13(8):671–685, Aug. 1983.

[11] R. E. Griswold and M. T. Griswold. *The Icon Programming Language.* Prentice Hall, Englewood Cliffs, NJ, second edition, 1990.

[12] D. R. Hanson. Simple code optimizations. *Software—Practice & Experience*, 13(8):745–763, Aug. 1983.

[13] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice & Experience*, 20(1):5–12, Jan. 1990.

[14] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, NJ, second edition, 1988.

[15] D. E. Knuth. An empirical study of FORTRAN programs. *Software—Practice & Experience*, 1(2):105–133, Apr. 1971.

[16] R. Sedgewick. *Algorithms*. Addison Wesley, Reading, MA, 1988.

[17] R. L. Sites. Programming tools: Statement counts and procedure timings. *SIGPLAN Notices*, 13(12):98–101, Dec. 1978.

[18] Standards Performance Evaluation Corp. *SPEC Benchmark Suite Release 1.0*, Oct. 1989.

[19] K. Thompson. A new C compiler. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 41–51, London, July 1990.

[20] W. M. Waite. The cost of lexical analysis. *Software—Practice & Experience*, 16(5):473–488, May 1986.

[21] P. J. Weinberger. Cheap dynamic instruction counting. *Bell System Technical Journal*, 63(8):1815–1826, Oct. 1984.