SOFIWARE—PRACTICE AND EXPERIENCE, VOL. 22(1), 89–99(JANUARY 1992)

Simple Register Spilling in a Retargetable Compiler

CHRISTOPHER W. FRASER
AT&T Bell Laboratories, 600 Mountain Avenue 2C-464, Murray Hill, NJ 07974 U.S.A.
AND
DAVID R. HANSON
Department of Computer Science, Princeton University, Princeton, NJ 08544 U.S.A.

SUMMARY

This paper describes the management of register spills in a retargetable C compiler. Spills are rare, which means that testing is a bigger problem than performance. The trade-offs have been arranged so that the common case (no spills) generates respectable code quickly and the uncommon case (spills) is less efficient but as simple as possible. The technique has proven practical and is in production use on VAX, Motorola 68020, SPARC and MIPS machines.

KEY WORDS ANSI C code generation compilers register allocation register spilling

INTRODUCTION

When register allocators run out of registers, they generate code to *spill* one or more busy registers into temporaries and code to *reload* those values when they are needed again. The trend in compiling research is increasing the sophistication — and the implementation and execution costs — of the techniques that avoid spills. [1–6]

This paper describes experience with a complementary approach. A spill manager has been simplified as much as seems consistent with application in a production compiler. It spills the register whose next use is the most distant. This well known choice is analogous to the optimal demand paging strategy that replaces pages whose next use is most distant in the execution stream. [7] The experience reported below corroborates the studies of small sample programs that support the effectiveness of simple spill managers. [8]

This approach is used in `lcc`, a new compiler for ANSI C [9] `lcc` is a retargetable compiler, and it has been ported to the VAX, Motorola 68020, SPARC, and MIPS R3000. [10] It is in production use at Princeton University and AT&T Bell Laboratories.

`lcc` performs optimizations that increase the demand for registers: the front end assigns promising variables to registers even without explicit register declarations, and the back ends

0038–0644/92/010085–15$07.50

Received 5 October 1990
Revised 30 August 1991

Table I. Spill Counts

Benchmark	Target 68020	VAX	MIPS	SPARC
001.gcc	102+9/55236	81+99/34564	96+124/68968	655+223/86039
008.espresso	21+0/13726	20+43/8019	22+48/18072	126+56/21777
022.li	0+0/2954	0+3/2139	2+3/4434	13+1/5858
023.eqntott	1+0/1378	1+21/872	1+24/2143	1+0/2647
lcc	52+1/15107	53+20/8886	56+22/17782	251+15/29944
Plum Hall	99+33/130465	13+203/72281	14+24/140653	647+3225/221214
port3	954+1/176788	791+176/86413	386+137/241650	230+1552/393322
awk	8+1/4606	9+313039	8+5/5885	141+27/9309
troff	4+0/5064	1+10/2910	2+18/7054	6+9/10061
tex	0+1/19729	0+10/13483	5+12/24582	14+17/33252

generally reserve about half of the available registers for this purpose. The front end also eliminates common subexpressions within basic blocks. More aggressive optimization —for example, global register allocation or function inlining — can increase the number of spills, but many C compilers in use today are less aggressive than lcc.

Nevertheless, spills are rare in lcc. And when spills are rare, there are good reasons to make the spill manager as simple as possible. First, it is wasteful to tune code that is seldom used. Second, test cases are hard to find and hard to isolate, so it can be hard to exercise a complex implementation thoroughly. Because spills are rare, lcc trades speed and sophistication in handling spills for a simple, fast register allocator and simple, correct spill generation. Its production register allocator is 269 1ines of C; its spill manager accounts for 109 of those lines.

One simplification that was not practical was the use of trees as the intermediate representation. There are simple, elegant, fast algorithms to give optimal allocations for pure trees [11] but the natural representation after common subexpression elimination is not trees but dags: Worse, C includes several operators — multiple assignment, augmented assignment, increment and decrement — that implicitly reuse certain values. The operators might be represented by trees, but the register allocator would still need to recognize the values with multiple uses and thus treat them as if they were dags.

Table I supports lcc 's approach to spills. It lists the number of spills and registers allocated for several benchmark programs. The first four lines are for the C programs in the SPEC benchmark, [12] and the next two lines are for lcc itself and the Plum Hall Validation Suite. port 3 is a large (over 200,000 lines) numeric library [13] generated by a Fortran-to-C translator, [14] and awk, troff, and tex are well known tools.

The notation $n + m/t$ denotes a total of $n + n$ spills and t allocations. There were n spills because the allocator simply ran out of registers, and m spills because a busy register was overwritten by a call or another instruction that overwrites fixed registers. Values of t

include both integer and floating-point register allocations. On the MIPS and Motorola 68020, allocating a double register is counted as one allocation because each half is not allocated for other purposes. On the VAX and SPARC, however, allocating a double register is counted as two allocations because the halves can be allocated separately.

Underlining highlights the few entries in Table I for which $n + m/t$ exceeds one percent. Numeric code often includes complex statements with many common subexpressions, so `port3` might be expected to show high spill counts. But $n + m/t$ topped one percent only on the VAX, which has the smallest register set of the four targets.

The highlighted `eqntott` counts are caused by the idiom `p -> x = f (p->x)`, which loads p before the call and spills it to a callee-saved register because it is also used after the call. The resulting code is better than reloading `p.` On the 68020 and SPARC, conventions place `p` in a register that is not overwritten by the call to `f.`

Table I shows more spills for the SPARC than for the other targets. Two factors account for these spills. First, under most calling conventions, some registers are saved by the caller and some by the callee; the SPARC convention has the caller save *all* busy floating-point registers. These saves show up in n for the SPARC (e.g., `port3` and Plum Hall). If callees saved some floating-point registers, these figures would drop.

Second, some nested calls force register-to-register spills. The first six arguments are passed in registers on the SPARC, and the result is returned in the first such register. Thus, for `f (x, g (y) ), g` returns its value in the register that `x` must occupy, so that register is spilled, which accounts for most of the spills in `awk,` for example. The MIPS convention also passes some arguments in registers, but its figures show no similar pattern because values are returned in a different register.

Spills cost little in `lcc.` It is impractical to isolate spill costs in substantial benchmarks or to compare `lcc` with other compilers, because different compilers vary in so many ways. It is, however, possible to compile and execute substantial benchmarks with smaller register sets; additional spills account for any increased execution time. One badly placed spill could increase execution time a lot, and many fortuitously placed spills might affect execution time little, but such timings are still suggestive.

Table II gives the execution times and program sizes for the C SPEC benchmarks on the MIPS. The first column describes the production compiler, and the rest describe *artificial* variants to suggest spill costs. Reference 10 gives a complete list of SPEC timings for the production `lcc` and for other compilers on the four targets; it shows that `lcc` generates competitive code.

The times are elapsed time in seconds and are the lowest times over several runs on a lightly loaded machine. All runs achieved at least 99 percent utilization (i.e., the ratio of times (*user* + *system,*) */elapsed* ≥ 0.99). The sizes of the program segments are in bytes. The second row for each entry gives $n + m/t$ as in Table I.

Each column lists figures for a different register configuration $i + f, I + F$, where i and f are the number of integer and floating-point temporaries, respectively, and I and F are the corresponding number of register variables. The second column artificially reduces the number of temporaries to a minimum, but the times remain nearly equal to those in the first

Table II. MIPS Sizes and Execution Times

Benchmark	Register Configuration 10+16,9+12	3+6,9+12	10+16,0+0	3+6,0+0
001.gcc	105s 805264b 96+ 124/68968	105s 806464b 252+1 22/69023	119s 912496b 96+212/893 17	122s 915600b 385+210/89601
008.espresso	149s 213232b 22+48/18072	149s 214160b 136+48/18117	210s 249952b 22+105/26307	211s 251936b 227+102/26511
022.li	352s 83712b 2+3/4434	351s 83712b 3+3/4434	414s 89808b 2+5/5747	416s 89808b 3+5/5748
023.eqntott	109s 46160b 1+24/2143	110s 46 160b 1+24/2143	154s 51024b 1+27/3192	154s 51024b 2+27/3192

column. `eqntott` appears to run slightly slower with no more spills, but these variants are *identical*, so this difference must be due toclockdrift. Likewise, clockdrift mustexplain why `li` appears to run slightly faster with *more* spills.

The third and fourth columns describe analogous but artificial configurations *without* register variables, so all spills go to memory. Even for the most drastic configuration (3 + 6), only one benchmark (`gcc`) ran more than 0.5 percent slower. For all comparable configurations, program size never increased more than one percent.

`lcc` 's approach makes it possible to tour a working spill manager for a real machine. It is drawn from a sample code generator that emits naive VAX code, i.e., it uses only the 'RISC subset' of the instruction set,[15] but a similar spiller is used in the production versions of `lcc`. Though naive, the sample code generator is complete: when used with a conforming preprocessor and library, the compiler with this code generator passes the conformance section of Version 2.00 of the Plum Hall Validation Suite for ANSI C.

CODE GENERATION INTERFACE

The interface between `lcc` 's target-independent front end and its target-dependent back ends consists of a few shared data structures, 18 functions, and a 36-operator dag language, which encodes the executable code from a source program.[15] Most of the functions are simple, e.g., they emit function prologues, define globals, lay out data, etc.

The front and back ends share symbol table entries and dag nodes:

```
typedef struct symbol *Symbol;
struct symbol {         /* symbol table entries: */
   char *name;             /* name */
   unsigned char class;    /* storage class */
   Type type;              /* data type */
};
```

```
typedef struct node *Node;
struct node {           /* dag nodes: */
   Opcode op;              /* operator */
   short count;            /* reference count */
   Symbol syms[MAXSYMS]; /* symbols */
   Node kids[MAXKIDS];     /* operands */
   Xnode x;                /* back-end's type extension */
};
```

A symbol's `name`, `class`, and `type` fields give its name, its storage class, and its type, respectively. Fields and types irrelevant to register allocation have been omitted.

In a dag node, the kids point to the operand nodes, and the `syms` point to symbol table entries for those operators that take symbols as operands. `count` holds the number of references to this node from `kids` in other nodes.

The `x` field is the back end's 'extension' to nodes, and it holds the per-node, target-dependent data that the back end needs to generate code. The sample `Xnode` is

```
typedef struct {
   int reg;         /* register number */
   unsigned rmask; /* unshifted register mask */
   Node next;       /* next node in linearized forest */
} Xnode;
```

`reg` holds the number of the register allocated to this node. `rmask` is `1` if the node needs an ordinary register and `3` if it needs a register pair. `next` points to the next node in the linearized forest, which is described below.

The `op` field holds an operator. The last character of each is a *type suffix* from Table III. For example, the generic operator `ADD` has the variants `ADDI`, `ADDU`, `ADDP`, `ADDF`, and `ADDD`.

Table III. Type Suffixes

type suffix	type
C	character
S	short
I	int
U	unsigned
P	any pointer type
F	float
D	double
V	void

Table IV lists the generic operators used in this paper, their valid type suffixes, and the number of `kids` and `syms` each uses. The type suffix denotes the type of operation to perform for most operators and the type of the result, if there is one.

Table IV. Node Operators

syms	kids	operator	type suffixes	operation
1	0	ADDRG	P	address of a global
1	0	ADDRL	P	address of a local
1	0	CNST	CSIUPFD	constant
	1	CVD	IUF	convert from double
	1	INDIR	CSIPFD	fetch
	2	ADD	IUPFD	addition
	2	LSH	IU	left shift
	2	MUL	IUFD	multiplication
	2	SUB	IUPFD	subtraction
	2	ASGN	CSIPFD	assignment

The leaf operators yield the address of a variable or the value of a constant, which is identified by `syms[0]`.

`CVD` accepts a double and yields a number. `IND IR` accepts an address and yields the value at that address. The binary operators accept two numbers and yield one. `ASGN` stores the value of `kids [ 1 ]` in the cell addressed by `kids [0]`. The operators used for their side-effects (like `ASGN`) always appear as roots in the forest of dags, and they appear in the order in which they must be executed.

REGISTER ALLOCATION

The front end passes a forest of dags to the back end. The production back ends traverse each dag to select suitable instructions, but the sample back end emits naive code so it does little during this first pass. Details appear in Reference 15.

After code selection, all back ends linearize the forest and make two passes over the resulting list, which is linked through the `x.next` fields in each node. The first pass allocates registers and the second emits the final code. For example, the linearized forest for `int i, *P; i = *p++` is shown in the right-hand columns below; the left column shows the code emitted by the naive code generator.

```
VAX instruction      #  op       count kids       syms
moval _p, r1         1.  ADDRGP   2                P
movl (r1) , r2       2.  INDIRP   2      1
movl $4, r3          3.  CNSTI    1                4
add13 r3, r2, r3     4. ADDP      1      2  3
movl r3, (r1)        5.  ASGNP    0      1  4
moval _i,r1          6.  ADDRGP   1                i
```

```
movl (r2) ,r2          7.  INDIRI  1     2
movl r2, (r1)          8.   ASGNI  0     6  7
```

The `INDIRP` node, which fetches the value of `p`, comes before node 5, which changes `p`, so the original value of `p` is available for subsequent use by node 7, which fetches the integer pointed to by that value.

`lcc` 's register allocation strategy is simple: it traverses the linearized forest and allocates registers to each node. The `count` field tells when all of the references to a node have been processed. For each node, the registers used by its children are released by `putreg`:

```
static void putreg (Node p) {
   if (p && --p-> count <= 0)
      rmask &= ~sets (p) ;
}
```

`putreg` decrements `count` and frees the register only when the last reference is removed, by clearing the appropriate bits in the global variable `rmask`, where `rmask& (l<<r)` is `1` if register `r` is busy.

Next, a register or register pair is allocated to a node by `getreg`:

```
static void getreg (Node p) {
   int r, m = optype (p–>op) == D ? 3 : 1;

   for (r = 0; r < nregs; r++)
      if ( (rmask& (m<<r) ) == 0) {
         p->x. rmask = m;
         p->x. reg = r;
         rmask | = sets (p) ;
         return;
      }
   r = spillee (p, m) ;
   spill (r, m, p) ;
   getreg (p) ;
}
```

`optype (op )` returns the type suffix of operator op. `m` is 1_2 if the result of `p->op` needs an ordinary register and 11_2 if it needs a register pair. `getreg` loops over the registers. If it finds one that's free, it sets the node's `x.reg` field to the register allocated and the `x.rmask` field to `m`. If no registers are free, `getreg` spills a register and calls itself to try again. sets (p) returns `p->x.rmask<<p->x. reg`. No optimization is attempted; a register is spilled even if its value is already available elsewhere in memory and even if it would be cheaper to recompute than to spill and reload.

SPILLS

The dag constructed by the front end minimizes the reevaluation of common subexpressions. Some spills are, in a sense, a result of the front end's eagerness to avoid reevaluation, and handling spills amounts to 'breaking the dags' generated by the front end. A node representing a common subexpression is changed so that it stores the value in a temporary, and subsequent references to that node are edited to load the value from the temporary.

Spilling involves three major steps: Identifying the registers to spill, generating the code for the spills at the correct location the output stream, and generating the code for the reloads, again at the correct locations. These steps are performed by the calls to `spillee` and `spil1` at the end of `getreg,` shown above. `spillee` identifies the register (`m` is `1`) or register pair (`m` is `3`) to be spilled on behalf of `p,` and `spill` generates the spill and reload code. Once `r` has been spilled, it is available, and the final call to `getreg` is guaranteed to succeed.

The register allocator frees registers as soon as possible. If the available registers are exhausted, it is often because there are multiple references to the nodes holding the registers, which arise from common subexpressions and from multiple assignment, augmented assignment, and the operators `++` and `--` Consider the following program.

```
double a[10], b[10];
int i;
f(){ i = (a[i]+b [i]) *(a[i]-b[i]); }
```

The initial linearized forest is shown to the left of the vertical line in the display below.

#	op	kids	syms	count	count	uses	sets
1.	ADDRGP		i	2	1		r1
2.	INDIRI	1		1	0	r1	r2
3.	CNSTI		3	1	0		r3
4.	LSHI	2 3		2	0	r2 r3	r2
5.	ADDRGP		a	1	0		r3
6.	ADDP	4 5		1	0	r2 r3	r3
7.	INDIRD	6		2	2	r3	r3 r4
8.	ADDRGP		b	1	0		r5
9.	ADDP	4 8		1	0	r2 r5	r2
10.	INDIRD	9		2	2	r2	
11.	ADDD	7 10		1	1	r3 r4	
12.	SUBD	7 10		1	1	r3 r4	
13.	MULD	11 12		1	1		
14.	CVD I	13		1	1		
15.	ASGNI	1 14		0	0	r1	

Nodes with count fields greater than 1 represent four common subexpressions: the lvalue of `i` (node 1), the addressing expression `i<<3` (node 4), and the rvalues of `a [ i ]` (node 7) and `b [ i ]` (node 10). If only registers 1–5 are available, `ralloc` runs out of registers at node 10. The linearized forest at that point is shown to the right of the vertical line in the display

above. As indicated by the non-zero `counts` and the sets column, node 1 is using `r1` and node 7 is using `r3` and `r4`. Node 10 needs a register pair, but only registers `r2` and `r5` are available. Note that the count of node 4, which is `i<<3`, has dropped to 0 because `ralloc` has processed both uses (nodes 6 and 9).

`getreg` calls `spillee` to identify a register pair to be spilled for use by node 10. `spillee` chooses the register whose next use is the most distant in the linearized forest.

For each register, `spillee` simply searches down the linearized forest for register uses and records the most distant.

```
static int spillee (Node dot, unsigned m) {
   int bestdist = -1, bestreg = 0, dist, r;
   Node q;

   for (r = 1; r < nregs - (m>>1) ; r++) {
      dist = 0;
      for (q = dot-> x.next; q &&  ! (uses (q) & (m<<r) ) ; q = q->x. next)
         dist++;
      if (dist > bestdist) {
         bestdist = dist;
         bestreg = r;
      }
   }
   return bestreg;
}
```

`spillee` is called with dot equal to the node at which the spill is required, which is node 10 in the example above. `uses(p)` returns a bit mask giving the registers used by node `p`, i.e., the registers set by `p` 's kids.

In this example, `spillee` finds `r1` used in node 15, which is at 'distance' 4, and `r3` and `r4` used in node 11 at distance 0. Consequently, `spillee` returns `r1`, which denotes both `r1` and `r2` because `m` is `3`.

Actually spilling the register chosen by `spillee` and inserting the reloads is done by spill and its associates, `genspill` and `genre loads`. In the example, these functions collaborate to 'break the dag' at node 1, which sets register `rl` (`r2` is already free). `genspill` generates a temporary and the nodes necessary to store the value computed by node 1 into the temporary. These new nodes are stitched into the linearized forest immediately after node 1. genre loads changes future *uses* of the value computed by node 1 to reload the value from the temporary instead of referencing node 1 directly. The effect is to remove the common subexpression by assigning it to a temporary.

`spill` identifies the locations at which to insert the spills by searching the linearized forest beyond dot for nodes that use the registers that are to be spilled, e.g., `r1` in the example.

```
static void spill (int r, unsigned m, Node dot) {
   int i;
```

```
    Node p = dot;

    while (p = p->x next)
       for (i = O;  i < MAXKIDS; i++)
          if (p->kids [i] && sets (p->kids [i]) &(m<<r)) {
             Symbol temp = genspill (p->kids[i]);
             rmask &= ~sets (p->kids[i]);
             genreloads(dot,  p->kids[il,  temp);
          }
}
```

`genspill` allocates the temporary, generates the spill code, and inserts it into the linearized forest right after the node that set the `spillee`. It returns the symbol table entry for the temporary so that it can be used by `genreloads` to generate the reloads.

```
static Symbol genspill (Node p) {
   Symbol temp = newtemp (AUTO, typecode (p) ) ;
   Node q = p->x.next;

   linearize (newnode(ASGN + typecode (p),
      newnode(ADDRLP, 0, 0, temp) , p, 0) ,
      &p->x. next, p->x.next) ;
   rmask &= ~1;
   for (p = p->x.next; p != q; p = p->x.next)
      ralloc(p) ;
   rmask |= 1;
   return  temp;
}
```

`typecode` is like `optype`, but maps `U` to `I` because the front-end uses `ASGNI` and `INDIRI` to store and load unsigned values. The front-end's `newtemp(class, type)` allocates a new temporary or reuses an existing one if its storage `class` and `type` code match those requested.

The spill code is an `ADDRLP` node to compute the address of the temporary and an `ASGN` node to store the value into that address. The right operand of the `ASGN` is simply `p`, the node that set the registers to spill, e.g., node 1 above. A node is allocated and initialized by the front-end function `newnode (op, 1, r, sym)`, where `op` is the operator, `l` and `r` are `kids [ 0 ]` and `kids [1]`, and `sym` is the symbol table pointer for leaf nodes. `newnode` also increments the reference counts of `1` and `r`.

Finally, `genspill` linearizes the spill code and inserts it right after `p`, which sets the spilled register. `linearize (Node p, Node *last, Node next )` linearizes the dag at `p` and inserts it into the growing list of nodes between `*last` and `next`. The manipulation of `rmask` ensures that register allocator, `ralloc`, assigns register `r0` — which is not otherwise

allocated — to the ADDRLP node to compute the address. The linearized forest after genspill returns is

```
 #  op       kids     syms   count   uses     sets
 1. ADDRGP           i        1             r1
16. ADDRLP           -4       0             r0
17. ASGNP    16  1              0     r0 r1
 2.  INDIRI  1                  0     r1       r2
...
```

Nodes 16 and 17 are the spill code. This code references node 1, so its count goes to 2 momentarily until ralloc processes the reference from the spill code. The remaining reference is from node 15. Node 16's symbol -4 is the frame offset to the temporary location.

After genspill returns the temporary, spill frees the registers set by the node and calls genre loads. genre loads searches the linearized forest after dot for uses of p.

```
static void genre loads (Node dot, Node p, Symbol temp) {
   int i;
   Node last;

   for (last = dot; dot = dot–>x.next; last = dot)
      for (i = 0;  i < MAXKIDS; i++)
         if (dot ->kids [i] == p) {
            dot->kids [i] = newnode (INDIR + typecode (P) ,
               newnode (ADDRL+P, 0, 0, temp) , 0, 0) ;
            dot->kids [i] ->count = 1;
            p->count--;
            linearize (dot ->kids [i] , &last->x.next, last->x.next) ;
            last = dot->kids [i] ;
         }
}
```

Each use of p is changed to a reload of the temporary temp. The reload code is an ADDRLP node to compute the address of the temporary and an INDIR node to load the value from that address. There is only one reference to each reload, so the INDIR 's count is set accordingly, and p- >count is decremented to reflect the change. The reload code is linearized and inserted into the linearized node list immediately *before* its use. For the example above, a reload is placed before node 15. Since the reload is beyond the current focus of register allocation, registers are allocated for these nodes by subsequent calls to ralloc.

The linearized forest after spill returns is

```
 #  op       kids     syms   count   uses     sets
 1. ADDRGP           i        0             r1
16. ADDRLP           -4       0             r0
```

```
17. ASGNP   16  1           0    r0 r1
 2.  INDIRI  1              0    r1       r2
...
14. CVDI    13             1
18. ADDRLP          -4     1
19.  INDIRP 18             1
15. ASGNI   19 14          0
```

Nodes 18 and 19 are the reload. Note that node 1's `count` has become 0; after inserting the reload, there are no unprocessed references.

Another spill occurs at node 11, which computes `a [ i ] +b [ i ] .` Registers `r1` and `r2,` which are set by node 10, are spilled, and node 12 is edited to refer to the second temporary, which is reloaded by nodes 22 and 23 inserted before node 12. The last spill occurs at node 23, which is the reload created for the second spill. `r1` and `r2,` which are set by node 11, are spilled again, and node 13 is edited to refer to the third temporary. The final linearized forest and corresponding VAX code follows; `count` fields are omitted because they're all 0, and each instruction shows which registers are used and set by each node.

```
VAX instruction      #  op       kids     syms
moval _i,r1          1. ADDRGP            i
moval -4 (fp) , r0  16. ADDRLP            -4
movl r1, (r0)       17. ASGNP    16  1
movl (r1) , r2       2. INDIRI    1
movl $3, r3          3. CNSTI             3
ashl r3, r2, r2      4. LSHI      2  3
moval _a,r3          5. ADDRGP            a
add13 r3, r2, r3     6. ADDP      4  5
movd (r3) , r3       7. INDIRD    6
moval _b, r5         8. ADDRGP            b
add13 r5, r2, r2     9. ADDP      4  8
movd (r2) , r1      10. INDIRD    9
moval -12 (fp) ,r0  20. ADDRLP            -12
movd r1, (r0)       21. ASGND    20 10
addd3 r1, r3, r1    11. ADDD      7 10
moval -20 (fp) , r0 24. ADDRLP            -20
movd r1, (r0)       25. ASGND    24 11
moval -12 (fp) ,r5  22. ADDRLP            -12
movd (r5) , r1      23. INDIRD   22
subd3 r1, r3, r1    12. SUBD      7 23
moval -20 (fp) , r3 26. ADDRLP            -20
movd (r3) , r3      27. INDIRD   26
muld3 r1, r3, r1    13. MULD     27 12
cvtdl r1, r1        14. CVDI     13
```

```
moval –4(fp), r2     18. ADDRLP          -4
movl (r2) ,r2        19.  INDIRP  18
movl r1, (r2)        15. ASGNI    19 14
```

Spills have added nodes 16-27.

EXTENSIONS

The production versions of the functions above are somewhat more general, but even they are only about 50 percent longer. The extra code accommodates multiple register sets, register variables, targeting, and nodes that set their destination register before finishing with their source registers.

More complex register sets complicate the representation of registers and the code that accesses it. The production code is shared by all targets, so it cannot include any overt machine-specific code. The sample faces only one set of 16 registers, but production targets have as many as three sets and 64 registers. The simple `rmask` above is thus extended to a vector of masks, one for each register set; the number of register sets is a configuration parameter. The register 'representation' above — a register number plus a mask — is extended to a structure that records: the register name (often a digit string); the register set; a mask (usually one or two bits) that identifies which registers in the set are occupied by *this* register.

Register variables complicate the identification of busy registers. The mechanism above — scanning the successors of the current node for children with unfreed registers — finds active expression temporaries but misses register variables that happen to be unused in the rest of the block. So the production code generators record the node or symbol to which each register has been assigned. They thus identify busy registers by scanning the register list instead of the node list.

Register variables also complicate the insertion of spills and reloads. When the register assigned to a node is spilled, the spill is inserted after the node, and the reloads are inserted before the later uses of the node. Register variables, however, may not have such nodes in the block at hand, so something else is needed. The register conventions used to date have been such that register variables are spilled only by instructions that destroy certain fixed registers. So the production spill manager puts the spills just before the instruction and the reloads just after it. Calls may be regarded as destroying a fixed set of registers; if the spill manager were used to implement a pure caller-save convention, it would spill more and might require tuning.

Targeting complicates reloading. The production compilers use targeting only for instructions that *require* a given child use a constrained set of registers. For example, return instructions often force their child into a fixed register. If the child is spilled, it must be reloaded into the same register. So the production compilers record the constraints for each targeted node, and the production `genreload` ensures that the reloads share these constraints.

In the production compilers, some nodes are implemented by a sequence of instructions. For example, on machines like the 68020 whose instructions specify at most two operands, some `ADDI` nodes compile into two instructions: the first loads one operand into the destination register and the second adds the other operand to it. The implementations of `spillee` and

`spill` above assume that the instruction or instructions that implement each node read the source registers before setting the destination register. This assumption permits the register allocator to reassign registers used by a node to that node, but it gives bad code if our `ADDI` is assigned a register not read until its second instruction. The production code generators flag such instructions and postpone register reuse until the next node.

In one respect, the production functions are simpler than those presented above. The sample's code is so naive that it can't spill a register without having a register (`r0`) to hold the address of the temporary. The production code generators can spill a register in a single instruction, so they need allocate no such register. They are thus spared analogs of the sample's `rmask` manipulations in `genspill`.

DISCUSSION

The spill manager described above evolved from one that was considerably longer. The most important simplification was the introduction of a pass to linearize the forest before allocating registers. The original system did the linearization and allocation in a single pass, so it had to deal with mixed code: linearized code when looking back to site spills and unlinearized code when looking ahead to identify the `spillee` and site reloads.

The resulting complications hid errors that — because spills are so rare — were not corrected for months, when `lcc` was used with a Fortran-to-C translator[14] to compile the complete port 3 numeric library.[13] Numeric codes in Fortran generally include more common subexpressions, which increases the number of spills. Using such code to test spill managers is strongly recommended.

The linearization uses a conventional bottom-up traversal of the dags, but it could use any ordering that obeys the constraints implicit in the dags. Indeed, preliminary work is underway to replace the linearize with a table-driven instruction scheduler.

Register-hungry numeric computations or targets with limited register sets like the Intel 386 may require register allocators more sophisticated than `lcc` 's, e.g., those based on graph coloring.[1] Experience with `lcc`, however, shows that a simple approach to register allocation and spilling is practical for systems programs written in C and for at least some numeric software, even when compiled for targets with as few as six registers devoted to expression evaluation and common subexpressions.

ACKNOWLEDGEMENTS

David Gay helped isolate the errors made when compiling the PORT3 mathematical subroutine library with the Fortran-to-C translator and `lcc`. Suggestions from Todd Proebsting improved the register allocator and spill manager, and Andrew Appel suggested using artificial register configurations to expose spill costs.

REFERENCES

1. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, 'Register allocation via coloring', *Journal of Computer Languages,* **6,** 47–57 (1981).

2. F. Chow and J. Hennessy, 'Register allocation by priority-based coloring', *SIGPLAN* Notices, **19,** 222–232 (1984).
3. D. Bernstein, D. Q. Goldin, M. C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Y. Pinter, 'Spill code minimization techniques for optimizing compilers', *SIGPLAN* Notices, **24,** 258–263 (1989).
4. R. Gupta, M. L. Soffa, and T. Steele, 'Register allocation via clique separators', *SIGPLANNotices,* **24,** 264–274 (1989).
5. P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon, 'Coloring heuristics for register allocation', *SIGPLAN Notices,* **24,** 275–284 (1989).
6. B. R. Nickerson, 'Graph coloring register allocation for processors with multi-register operands', *SIGPLAN Notices,* **25,** 40–52 (1990).
7. R. A. Freiburghouse, 'Register allocation via usage counts', *Comm. of the ACM,* **17,** 638–642 (1974).
8. W. Hsu, C. N. Fischer, and J. R. Goodman, 'On the minimization of loads/stores in local register allocation', *IEEE Trans. on Software Engineering,* **15,** 1252–1260 (1989).
9. *American National Standard for Information Systems, Programming Language C ANSI X3.159–1989,* American National Standards Institute, Inc., New York, 1990.
10. C. W. Fraser and D. R. Hanson, 'A retargetable compiler for ANSI C', *SIGPLAN Notices, 26,* to appear (1991).
11. R. Sethi and J. D. Unman, 'The generation of optimal code for arithmetic expressions', *Journal of the ACM,* **17,** 715–728 (1970).
12. Standards Performance Evaluation Corp., *SPEC Benchmark Suite Release 1.0,* October 1989.
13. P. A. Fox, A. D. Hall, and N. L. Schryer, 'The PORT mathematical subroutine library', *ACM Trans. on Mathematical Software,* **4,** 104–126 (1978).
14. S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer, 'A Fortran-to-C converter', *Technical Report 149,* Computing Science Research Center, AT&T Bell Laboratories, Murray Hill, NJ, May 1990.
15. C. W. Fraser and D. R. Hanson, 'A code generation interface for ANSI C', *Software—Practice & Experience,* **21,** to appear (1991).