Unrolling Recursions Saves Space

Andrew W. Appel*
Princeton University

CS-TR-363-92

March 10, 1992

Abstract

Partially unrolling a single-branch, non-tail recursion reduces the number of stack frames allocated. This simple transformation can speed up the program and avoid stack overflow.

Loop unrolling is a well-known technique to cut down on the per-iteration overhead of executing a loop. In a typical example,

```
for(i=0;i<N;i++)
  sum = sum + A[i]
```

unrolling the loop k times reduces the number of increments and tests from N to N/k:

```
for(i=0;i<N;i+=4)
  {sum = sum + A[i];
   sum = sum + A[i+1];
   sum = sum + A[i+2];
   sum = sum + A[i+3];
  }
for(;i<N;i++)
   sum = sum + A[i];
```

Now there is one increment, one test, and one branch for each four iterations, so that overhead—which might have been 50% of the first loop's execution time—might now be only 20%; the second program might run almost twice as fast. (This assumes that the C compiler is smart enough to evaluate $(A+1), (A+2), (A+3)$ at compile time.)

*Supported in part by NSF grant CCR-9002786.

Unrolling can also be applied to recursions. Consider a prototypical single-branched recursion:

```
f(x) =
   if p(x)
     then g(f(h(x)))
     else z(x)
```

The function f does something (h) to x, calls itself recursively, then does something else (g) before returning. If g is nontrivial, then this is not a tail-recursion.

Let's unroll this recursion four times:

```
f'(x1) =
  if p(x1)
  then let x2 = h(x1)
       if p(x2)
       then let x3 = h(x2)
            if p(x3)
            then let x4 = h(x3)
                 if p(x4)
                 then g(g(g(g(f'(h(x4))))))
                 else g(g(g(z(x4))))
            else g(g(z(x3)))
      else g(z(x2))
  else z(x1)
```

It is easily seen that f and f' compute the same function, and that I have simply unrolled the recursion.

Indeed, unrolling recursions is not a new idea. Tail recursions, of course, can be turned into loops, for which ordinary loop unrolling works. Harrison

[3] completely unrolls non-tail recursions into a "forward" loop (effectively, N applications of h in my example) and a "backward" loop (N applications of g); he stores local variables in an array of size (at least) N. The unrolling I describe here is not so drastic: I merely expand k recursive calls in the body of the original loop, leaving a recursive function.

Now what have I accomplished? The discerning reader will see that I haven't even eliminated any tests or branches; it's as if I had unrolled the original `for` statement as follows:

```
while (i<N)
     {sum = sum + A[i];
      i = i+1;
      if (i>=N) break;
      sum = sum + A[i];
      i = i+1;
      if (i>=N) break;
      sum = sum + A[i];
      i = i+1;
      if (i>=N) break;
      sum = sum + A[i];
      i = i+1;
     }
```

This loop won't run significantly faster than the original one.

But consider the memory usage of f on a deep recursion. One frame will be kept on the stack for each call. The function f' makes a quarter as many recursive calls, and will have a quarter as many frames. The frames of f' will be slightly larger, because f' has more local variables than f. But the *overhead* associated with each frame—return address, frame pointer, and the (surprisingly large) "other stuff"—will occur one fourth as many times.

For a concrete example, consider the **map** function such that **map**(f, l) applies function f to each element of list l, returning a list of the results. In Standard ML this is:

```
fun map(f,nil) = nil
  | map(f,a::b) = f a :: map(f,b)
```

In Scheme:

```
(define (map proc lis)
   (if (null? lis) '()
      (cons (proc (car lis))
            (map proc (cdr lis)))))
```

And in C, for a list of integers:

```
typedef struct list
    {int v; struct list *p;} *List;

List map(int (*f)(int), List p) {
 if (p) {int a=f(p->v);
         List b = map(f,p->p);
         return cons(a,b);}
 else return leaf();
}
```

One can unroll this recursion quite simply (I'll just show the ML version):

```
fun map'(f,nil) = nil
  | map'(f,[a]) = [f a]
  | map'(f,[a,b]) = [f a, f b]
  | map'(f,[a,b,c]) = [f a, f b, f c]
  | map'(f,a::b::c::d::rest) =
    f a :: f b :: f c :: f d :: map'(f,rest)
```

To measure the effect of the optimization, I mapped the integer increment function over a list of 100,000 integers, and repeated it ten times to overcome any garbage-collection transients; once with a simple **map** function and once with a four-times unrolled **map** function. I used an ML compiler, two Scheme compilers, and three C compilers, on two different platforms. I ran all compilers at their highest optimization level. I bound the increment function late, so compilers could not in-line expand it. Figure 1 shows the results.

Standard ML of New Jersey heap-allocates all frames [1], so the height of the runtime stack is not measurable directly; on the other hand, larger or more numerous frames will increase the load on the garbage collector. Chez Scheme heap-allocates stack segments [4], so in a deep one-way recursion the effect will be quite similar. C stack-allocates and does not have a garbage collector (times shown include the cost of `malloc` and `free` on each copied cell). So for ML and Scheme I have shown execution time for compiled code and garbage collection separately, and for C I have shown execution time and stack height.

The results are, for the most part, gratifying. With almost all compilers, run time went down, and stack size (or garbage-collection time) went down by approximately a factor of two. (Comparisons between different compilers on such a small program should not be taken too seriously, however.)

Using the MIT Scheme *interpreter* (results not shown in the table), the unrolled loop is slightly slower than the original; I speculate that the interpreter represents frames as linked lists of values, so that the overhead of a frame is proportional to the number of variables in it.

It is instructive to compare the sizes of frames used by the different C compilers on different machines. In the C program I used for measurement, the original **map** function has 4 local variables, of which 1 is live after the call (and therefore needs to be saved in the stack frame); the unrolled function has 10 local variables, of which 4 are live. To calculate the size of each frame, simply divide stack height by recursion depth; the results are shown in figure 2. Also shown are the sizes of Chez Scheme frames, and (heap-allocated) frames in Standard ML of New Jersey.

Compiler	Machine	Original		Unrolled	
		Non-G.C. (sec.)	G.C. (sec.)	Non-G.C. (sec.)	G.C. (sec.)
Standard ML of NJ 0.76	DEC 5000/200	3.84	11.18	2.72	6.87
MIT Scheme[1] 7.1.1	DEC 5000/200	11.42	1.35	9.97	0.92
Chez Scheme[2] 4.0a	DEC 5840	14.07	11.23	11.99	0.44
Standard ML of NJ 0.76	SparcStation 2	4.39	7.15	4.05	3.61
Chez Scheme 4.0a	SparcStation 2	6.55	4.63	5.31	0.24
		Time (sec.)	Stack (bytes)	Time (sec.)	Stack (bytes)
gcc 1.37.1	DEC 5000/200	8.87	4,800,304	7.28	1,400,312
lcc[3] 1.8	DEC 5000/200	9.14	3,200,240	7.33	1,600,272
/bin/cc[4] 2/28/91	DEC 5000/200	7.91	4,800,312	6.48	2,800,376
lcc 1.8	SparcStation 2	21.97	9,600,432	15.78	2,600,440

Figure 1: Performance of original and unrolled **map** functions.

1 I couldn't get MIT Scheme to expand its heap big enough for a recursion 100,000 deep, so I used 100 iterations of a 10,000 deep recursion instead.

2 Thanks to Kent Dybvig for providing the Chez Scheme results.

3 A research C compiler developed by Chris Fraser and David Hanson[2]

4 The Mips C compiler, as sold by Digital Equipment Corporation

	Original 4 locals (1 live)	Unrolled 10 locals (4 live)
SML/NJ	16	28
Chez Scheme	12	44
gcc, DEC	48	56
lcc, DEC	32	64
/bin/cc, DEC	40	104
lcc, SPARC	96	104

Figure 2: Frame sizes, in bytes

The C compilers suffer for two reasons: the calling conventions prescribe a space for holding arguments, even if they are passed in registers; and some compilers allocate space for variables even if dead across the call. Clearly, if deep recursions are to be used in C the compiler writers and calling-sequence designers should pay some attention to frame size.

In any case, it seems that unrolling single-branch recursions can be worthwhile. Programmers can use this transformation "by hand" when efficiency is required; and perhaps compilers can even be made to do it automatically.

In the presence of automatic in-line expansion of functions, this transformation should be used with care. If, in SML/NJ, the expression `map(fn x=>x+1, biglist)` is considered for in-line expansion, expanding the original function (with a smaller body) will look more attractive than expanding the unrolled function. Unrolling should probably be done *after* in-line expansion.

References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] Christopher W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(8), August 1991.

[3] Williams Ludwell Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*, 2(3):179–396, October 1989.

[4] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proc. ACM SIGPLAN '90 Conf. on Prog. Lang. Design and Implementation*, pages 66–77, New York, 1990. ACM Press.