Limbo profilers in Inferno

J R Firth

Vita Nuova

13 June 2002

1. Introduction

Currently there are three application level profiling tools in the Inferno package. *Prof* is a time profiler which, by sampling, can provide statistics on the percentage of time spent on each line of limbo source. *Cprof* is a coverage profiler which provides the execution profile for limbo source code. It can accumulate results over a series of runs to allow full coverage testing. Finally, *mprof* is a memory profiler which provides statistics on the amount of memory used by each line of limbo source.

Two gui versions of these tools currently exist. *Wm/cprof* shows the coverage per module and highlights those lines which have not been executed or have been only partially executed. *Wm/mprof* shows the memory usage per module and highlights lines with high memory allocation in darker shades of red. *Prof* itself does not have a gui equivalent as it was originally written to determine why acme was so slow when using it's global editing command. A gui for it was not a requirement at that stage.

All these tools use a common library module `/module/profile.m` and `/appl/lib/profile.b` that acts as the direct interface with the kernel profiling device.

Note that none of these tools give kernel profile statistics. For that, the devmem driver should be used.

Although the use of these tools is very similar, there are a few differences when it comes to interactive testing as the profilers were written to answer different questions. Thus *prof* tries to determine 'why is blah so slow ?', *cprof* tries to accumulate coverage records over time and *mprof* tries to give a series of memory statistics at intervals during the execution of a program or series of programs.

2. Prof

The time profiler works by sampling. A kernel procedure sleeps for the given sample time and then notes the particular dis instruction currently being executed before repeating the process. After many such samples, an accurate profile can be obtained.

At it's simplest we can profile a particular command by giving the command to execute followed by any arguments to the command eg

```
prof wm/polyhedra
```

profiles the polyhedra displayer. After letting the latter run for a reasonable amount of time, we exit and then get the following statistics.

```
Module: Wmlib(/dis/lib/wmlib.dis)

34    0.06         str = load String String->PATH;

**** module sampling points 1 ****
```

```
Module: Bufio(/dis/lib/bufio.dis)

340  0.06            n := 0;
341  0.12            while(b.index < b.size){
342  0.99                (ch, i, nil) = sys->byte2char(b.buffer[0:b.size], b.index);

**** module sampling points 19 ****

Module: Polyfill(/dis/math/polyfill.dis)

37   10.80       for(i := 0; i < n; i++)
38   19.86           b0[i] = b1[i] = ∞;
57   0.06        p.y += y;
58   1.18        dst.line((left, y), (right, y), Endsquare, Endsquare, 0, src, p);
63   0.12        prevx := ∞;
64   9.93        for(x := left; x <= right; x++){
65   20.61           if(z+e < zbuf0[k] || (z- e<= zbuf1[k] && x != right && prevx != ∞)){
66   6.46                zbuf0[k] = z- e;
67   5.71                zbuf1[k] = z+e;
68   0.74                if(prevx == ∞)
69   0.74                    prevx = x;
71   0.12            else if(prevx != ∞){
72   0.25                fillline(dst, prevx, x- 1,y, src, p);
73   2.61                prevx = ∞;
75   4.35            z += dx;
76   3.17            k++;
78   0.06        if(prevx != ∞)
79   0.87            fillline(dst, prevx, right, y, src, p);
80   0.06    }
152  0.06            return (vx/z, mod);
186  0.06            sp.dzrem = mod(sp.num, sp.den) << fixshift;
187  0.06            sp.dz += sdiv(sp.dzrem, sp.den);
217  0.62            for(q = p = 0; p < ep; p++) {
218  0.37                sp = seg[p];
220  0.12                    continue;
221  0.12                sp.z += sp.dz;
222  0.19                sp.zerr += sp.dzrem;
223  0.12                if(sp.zerr >= sp.den) {
224  0.19                    sp.z++;
225  0.25                    sp.zerr - =sp.den;
226  0.25                    if(sp.zerr < 0 || sp.zerr >= sp.den)
227  0.25                        sys->print("bad ratzerr1: %d den %d dzrem %d0, sp.zerr, sp.den, sp.dzrem);
229  0.31                seg[q] = sp;
230  0.31                q++;
233  0.25            for(p = next; seg[p] != nil; p++) {
234  0.06                sp = seg[p];
247  0.12            ep = q;
248  0.06            next = p;
257  0.06                continue;
260  0.62            zsort(seg, ep);
262  0.25            for(p = 0; p < ep; p++) {
263  0.19                sp = seg[p];
264  0.06                cnt = 0;
265  0.06                x = sp.z;
266  0.25                ix = (x + onehalf) >> fixshift;
267  0.06                if(ix >= maxx)
271  0.06                cnt += sp.d;
```

```
272  0.12                   p++;
273  0.25                   sp = seg[p];
275  0.06                           if(p == ep) {
277  0.06                                   return;
279  0.06                           cnt += sp.d;
280  0.12                           if((cnt&wind) == 0)
283  0.19                           sp = seg[p];
286  0.25                   ix2 = (x2 + onehalf) >> fixshift;
291  1.92                   filllinez(dst, ix, ix2, iy, zv+ix*dx, er, dx, k+ix- zr.min.x,zbuf0, zbuf1, src, spt);
293  0.06           y += (1<<fixshift);
294  0.31           iy++;
295  0.06           k += xlen;
296  0.06           zv += dy;
298  0.06   }
310  0.06                   done = 1;
311  0.12                   q- - ;
312  0.25                   for(p = 0; p < q; p++) {
313  0.87                           if(seg[p].z > seg[p+1].z) {
367  0.06           t = a[0]; a[0] = a[i]; a[i] = t;
373  0.06                   while(i < n && ycompare(a[i], a[0]) < 0);
379  0.12                   t = a[i]; a[i] = a[j]; a[j] = t;
384  0.06                   qsortycompare(a, j);

**** module sampling points 1584 ****

Module: Polyhedra(/dis/wm/polyhedra.dis)

327  0.12       return (int (geo.sx*v.x)+geo.tx, int (geo.sy*v.y)+geo.ty);
471  0.06                           if(allf || dot(geo.view, newn[j]) < 0.0)
472  0.06                                   polyfilla(fv[j], new, newn[j], dot(geo.light, newn[j]), geo, concave, inc);
496  0.06           ap[j] = map(vtx, geo);
512  0.06           if(a <= - LIMIT|| a >= LIMIT)
531  0.06           fillpoly(RDisp, ap, ~0, face, (0, 0), geo.zstate, dc, dx, dy);

**** module sampling points 7 ****


**** total sampling points 1611 ****
```

The output lists all lines in all modules with a sampling point. Each line shows the line number in the corresponding source file, the percentage of time spent on that line and the source code. We can see that about 60% of the sampling points occur on lines 37, 38, 64 and 65 of the Polyfill module. With this information we might then try to speed up this part of the code by altering the algorithm or making the limbo code more efficient (for instance by moving constant calculations or addressing out of loops).

The number of sampling points is also shown. The sampling rate can be increased with the - soption to give better granularity. This will cause a decrease in apparent performance but increases the accuracy of the results. The above example showed the results for all modules sampled. We might have restricted attention to the two main polyhedra modules instead by executing

```
prof - mPolyhedra - mPolyfill wm/polyhedra
```

See the manual page for other options to *prof* and further examples.

3. Cprof

Coverage of instructions is achieved by running a special dis instruction execute routine in place of the usual one (just as the debugger does). This routine notes down each instruction as it is executed. The profile device then passes this information to *cprof* via the io system.

The coverage profiler is used in a similar way to the time profiler.

```
cprof - mZeros zeros 1024 2880
```

gives

```
Module: Zeros(zeros.dis)     56% coverage

1           implement Zeros;
2
3           include "sys.m";
4                   sys: Sys;
5           include "arg.m";
6                   arg: Arg;
7           include "string.m";
8                   str: String;
9           include "keyring.m";
10          include "security.m";
11                  random: Random;
12
13          include "draw.m";
14
15          Zeros: module
16          {
17                  init: fn(nil: ref Draw->Context, argv: list of string);
18          };
19
20          init(nil: ref Draw->Context, argv: list of string)
21          {
22                  z: array of byte;
23                  i: int;
24    +             sys = load Sys Sys->PATH;
25    +             arg = load Arg Arg->PATH;
26    +             str = load String String->PATH;
27
28    +             if(sys == nil || arg == nil)
29    -                     return;
30
31    +             bs := 0;
32    +             n := 0;
33    +             val := 0;
34    +             rflag := 0;
35    +             arg->init(argv);
36    +             while ((c := arg->opt()) != 0)
37    -                     case c {
38    -                     'r' => rflag = 1;
39    -                     'v' => (val, nil) = str->toint(arg->arg(), 16);
40    -                     * => sys->raise(sys->sprint("fail: unknown option (%c)0, c));
41                          }
42   +          argv = arg->argv();
43   +          if(len argv >= 1)
44   +                  bs = int hd argv;
45              else
46   -                  bs = 1;
47   +          if (len argv >= 2)
48   +                  n = int hd tl argv;
49              else
50   -                  n = 1;
51   +          if(bs == 0 || n == 0) {
52   -                  sys->fprint(sys->fildes(2), "usage: zeros [- r][- vvalue] blocksize [number]0);
53   -                  sys->raise("fail: usage");
54              }
55   +          if (rflag) {
56   -                  random = load Random Random->PATH;
57   -                  if (random == nil)
58   -                          sys->raise("fail: no security module0);
59   -                  z = random->randombuf(random->NotQuiteRandom, bs);
60              }
61              else {
62   +                  z = array[bs] of byte;
63   +                  for(i=0;i<bs;i++)
64   +                          z[i] = byte val;
65              }
66   +          for(i=0;i<n;i++)
67   +                  sys->write(sys->fildes(1), z, bs);
68   +      }

**** module dis instructions 39725 ****
```

Here the - moption has restricted attention to the Zeros module itself. The output shows the source line number, an indicator of coverage and the source. The indicator is + if the line has been executed, - if it hasn't and ? if only part of it has (for instance a loop statement that has never had it's iteration part executed). Lines with no indicator have no corresponding dis instructions associated with them. Another option (- f) shows coverage frequencies instead.

An alternative to *cprof* is *wm/cprof* which shows the statistics graphically. It's options are pretty much the same as those to *cprof*. Unexecuted and partially executed lines of code are shown in colour. See the man page for exact details of the colouring convention

Results may be accumulated with the - roption so that multiple runs of code can be made. The resulting statistics go into a file <xxx>.prf when <xxx>.dis is the corresponding dis file. See the manual page for further details on how to use this option and then review the accumulated results.

4. Mprof

When memory profiling, the kernel profile device associates each heap allocation with a line of limbo source and each heap deallocation with the line of limbo source that allocated it. In this way, current memory usage and high- waterusage can be determined on a line by line basis.

Here it seems that memory usage at a particular point in the execution of a program is more appropriate than the post- mortemapproach of *prof* and *cprof* , so an interactive example is described (though *mprof* can be used non- interactivelyand *prof* interactively if so desired). See the manual pages for complete details and further examples.

To do this we execute

```
mprof - b- mPolyhedra
```

which kicks off profiling and restricts attention to the Polyhedra module whenever it runs. The - bsimply says begin profiling. Note that no command to execute is given to *mprof* at this stage. Then run the command

```
wm/polyhedra &
```

and interact with it. Now show memory statistics

```
mprof
```

This gives

```
Module: Polyhedra(/dis/wm/polyhedra.dis)

44   100  100    sys = load Sys Sys->PATH;
45   132  132    draw = load Draw Draw->PATH;
46   68   68     tk = load Tk Tk->PATH;
47   1788 1788   wmlib = load Wmlib Wmlib->PATH;
48   232  232    bufio = load Bufio Bufio->PATH;
49   68   68     math = load Math Math->PATH;
50   204  204    rand = load Rand Rand->PATH;
51   0    3504   daytime = load Daytime Daytime->PATH;
52   544  544    polyfill = load Polyfill Polyfill->PATH;
53   1824 1824   smenu = load Smenu Smenu->PATH;
86   36   36     cmdch := chan of string;
95   36   36     sync := chan of int;
96   36   36     chanθ := chan of real;
103  68   68     shade = array[NSHADES] of ref Image;
116  36   36     yieldc := chan of int;
120  36   36     sm := array[2] of ref Scrollmenu;
378  68   176        s += " (" + string p.indx + ")";
403  36   36     vec := array[2] of array of Vector;
404  740  740    vec[0] = array[V] of Vector;
405  740  740    vec[1] = array[V] of Vector;
407  36   36         norm = array[2] of array of Vector;
408  612  612        norm[0] = array[F] of Vector;
409  612  612        norm[1] = array[F] of Vector;
492  68   68     ap := array[n+1] of Point;
609  164  164    geo := ref Geom;
610  36   36     TM := array[4] of array of real;
612  272  272        TM[i] = array[4] of real;
663  8000 8000       p := ref Polyhedron;
707  740  740    p.v = array[p.V] of Vector;
710  612  612    p.f = array[p.F] of Vector;
713  132  132    p.fv = array[p.F] of array of int;
716  164  164    p.vf = array[p.V] of array of int;
729  9504 9640       return s[0: len s - 1];
750  3672 3672   a := array[n+2] of int;
768  0    136            return (n, s[i+1:]);
779  0    104            return (r, s[i+1:]);
802  0    68         s = s[1:];
806  0    72         s = s[0: ln- 1];


866  0    200    cmd(mainwin, ".f1.txt configure - text{" + s + "}");
874  0    356    labs := array[n] of string;
881  0    5128       labs[i++] = string q.indx + " " + name;
884  0    68     cmd(top, mname + " configure - borderwidth3");
920  0    104        cmd(win, ". configure - height" + string (scrsize.y - bd * 2));
934  0    244    cmd(win, ". configure - x" + string actr.min.x + " - y" + string actr.min.y);

Module totals    31416    33984
```

We get the source line number, the amount of memory in bytes currently allocated on that line, the high-water mark in bytes and then the source. Thus loading the Sys module on line 44 used 100 bytes and this memory is still allocated. Loading Daytime on line 51 used 3504 bytes but this is now released (because the module pointer is set to nil in the source and the memory has been reclaimed). The string concatenation on line 378 currently uses 68 bytes but at it's worst it was 176 bytes.

Further interaction with wm/polyhedra can now be done and memory statistics reviewed before the profiling session is ended, throwing away the accumulated memory statistics inside the kernel with

```
mprof - c
```

The - coption simply says cease profiling.

An alternative to *mprof* is *wm/mprof* which shows the statistics graphically. It's options are pretty much the same as those to *mprof*. Lines of code which have allocated more of the memory are shown in darker shades of red.

5. Manual pages

Further information and other examples are given in the following manual pages :-

cprof(1)
mprof(1)
prof(1)
wm-cprof(1)
wm-mprof(1)

For the lower level library module interface to these profilers

prof(2)

For the kernel profile device which gathers timing, coverage and memory statistics

prof(3)

6. Sources

The relevant sources are

```
/module/profile.m
/appl/lib/profile.b
/appl/cmd/cprof.b
/appl/cmd/mprof.b
/appl/cmd/prof.b
/appl/wm/cprof.b
/appl/wm/mprof.b
/emu/devprof.c
/os/port/devprof.c
```

7. Addendum

A gui version of *prof* has been added for completeness. See the manual page *wm-prof(1)* and the source `/appl/wm/prof.b`.