Need for Speed

Alexander Burger
abu@software-lab.de

2011-11-19

Need for Speed

One of the greatest mysteries in the history of computer language comparisons is to me the question why most people are more interested in the relative speed of a language implementation, rather than in features like expressiveness, flexibility and orthogonality.

Several years ago I wrote an article[1] where - among other things - I compared PicoLisp with a "compiled" Lisp (CLisp). After that, postings appeared who claimed that picking CLisp was an unfortunate choice, because it compiles only to bytecode, and that SBCL would have been better.

Contrary to the intention of those postings, I see this quite as an assertion of my argument. After all, why go through the troubles and disadvantages of supporting a compiler, when the resulting speed is *lower* than without?

And while I still believe that raw execution speed is a relatively unimportant issue, I feel I should supply an update.

I did a local install of SBCL on a Linux x86-64 System with two Dual-Core 1-GHz Opterons, and compared that with the current 64-bit version of PicoLisp.

Fibonacci

In the above article, the fibonacci function was called with an argument of 30. As today's machines are faster, I took 40 instead, and got:

```
(fibo 40)
   PicoLisp    34.8 sec
   sbcl         5.1 sec
   sbcl(i)    33:45 min
```

(sbcl(i) means "SBCL interpreted" – More than half an hour is beyond good and evil, of course)

The relation of SBCL (5.1 sec) to PicoLisp (34.8 sec) looks reasonable. Fibonacci on compiled SBCL runs about 6.8 times faster than on (interpreted) PicoLisp. And can probably be even improved with some declaration magic. PicoLisp, on the other hand, is not designed for arithmetic speed, it is always handicapped by its bignum-only number type.

But, as I also wrote in the above article, integer primitive operations can be easily optimized by a compiler. They are, however, not typical for a Lisp program, where direct list mappings are used instead of array index calculations.

BTW, out of interest I also tried an equivalent Python program. It took 1:45 minutes. In general I can say that on most occasions where I compared PicoLisp to Python I observed such a factor of 1 to 3.

[1] http://software-lab.de/radical.pdf

List Operations

So I tried the second example from that article, the `tst` function doing extensive list operations.

```
(load "tst.l")
   PicoLisp     2.0 sec
   sbcl         1.8 sec
   sbcl(i)     72.8 sec
```

The difference is negligible. Not much to say here.

Binary Trees

Some people claimed the above examples are not "real" benchmarks. Let's move to the Alioth Benchmarks Game platform, where the Binary Trees[2] benchmark does things quite similar to the above `tst` (though also a certain amount of arithmetics). The SBCL version is

```
(defun build-btree (item depth)
  (declare (fixnum item depth))
  (if (zerop depth) (list item)
      (let ((item2 (+ item item))
            (depth-1 (1- depth)))
        (declare (fixnum item2 depth-1))
        (cons item
            (cons (build-btree (the fixnum (1- item2)) depth-1)
                 (build-btree item2 depth-1))))))

(defun check-node (node)
  (declare (values fixnum))
  (let ((data (car node))
        (kids (cdr node)))
    (declare (fixnum data))
    (if kids
        (- (+ data (check-node (car kids)))
           (check-node (cdr kids)))
        data)))

(defun loop-depths (max-depth &key (min-depth 4))
  (declare (type fixnum max-depth min-depth))
  (loop for d of-type fixnum from min-depth by 2 upto max-depth do
       (loop with iterations of-type fixnum = (ash 1 (+ max-depth min-depth (- d)))
          for i of-type fixnum from 1 upto iterations
          sum (+ (the fixnum (check-node (build-btree i d)))
                 (the fixnum (check-node (build-btree (- i) d))))
          into result of-type fixnum
          finally
            (format t "~D   trees of depth ~D    check: ~D~%"
                    (the fixnum (+ iterations iterations )) d result))))

(defun main (&optional (n (parse-integer
                           (or (car (last #+sbcl sb-ext:*posix-argv*
                                          #+cmu  extensions:*command-line-strings*
```

[2] http://shootout.alioth.debian.org/u64q/performance.php?test=binarytrees

```
                                          #+gcl  si::*command-args*))
                                  "1"))))
  (declare (type (integer 0 255) n))
  (format t "stretch tree of depth ~D   check: ~D~%" (1+ n) (check-node (build-btree 0 (1+ n))))
  (let ((*print-pretty* nil) (long-lived-tree (build-btree 0 n)))
    (purify)
    (loop-depths n)
    (format t "long lived tree of depth ~D    check: ~D~%" n (check-node long-lived-tree))))
```

The corresponding PicoLisp program is

```
(de buildTree (Item Depth)
   (cons Item
      (and
         (n0 Depth)
         (cons
            (buildTree
               (dec (setq Item (>> -1 Item)))
               (dec 'Depth) )
            (buildTree Item Depth) ) ) ) )

(de checkNode (Node)
   (if2 (cadr Node) (cddr Node)
      (- (+ (car Node) (checkNode (cadr Node))) (checkNode @))
      (+ (car Node) (checkNode @))
      (- (car Node) (checkNode @))
      (car Node) ) )

(let (N (format (opt))  Min 4)
   (prinl
      "stretch tree of depth "
      (inc N)
      "^I check: "
      (checkNode (buildTree 0 (inc N))) )
   (let LongLivedTree (buildTree 0 N)
      (for (D Min (>= N D) (+ 2 D))
         (let (Sum 0  Iterations (>> (- D Min N) 1))
            (for I Iterations
               (inc 'Sum
                  (+
                     (checkNode (buildTree I D))
                     (checkNode (buildTree (- I) D)) ) ) )
            (prinl
               (* 2 Iterations)
               "^I trees of depth "
               D
               "^I check: "
               Sum ) ) )
      (prinl
         "long lived tree of depth "
         N
         "^I check: "
         (checkNode LongLivedTree) ) ) )
```

When called with an argument of 20, we get

```
PicoLisp    4:03 min
sbcl        1:02 min
```

If we optimize the PicoLisp version, by calling (gc 100) at the beginning, the time is reduced to three and a half minutes, but this seems be forbidden by the benchmark rule.

In any case, here a factor of 4 is also not really overwhelming.

Fannkuch

Finally, I looked at the Alioth Fannkuch[3] benchmark. The SBCL version is

```
(defun write-permutation (perm)
  (loop for i across perm do
   (princ (1+ i)))
  (terpri))

(defun fannkuch (n)
  (declare (optimize (speed 3) (safety 0) (debug 0)) (fixnum n))
  (assert (< 1 n 128))
  (let ((perm (make-array n :element-type 'fixnum))
        (perm1 (make-array n :element-type 'fixnum))
        (count (make-array n :element-type 'fixnum))
        (flips 0) (flipsmax 0) (r n) (check 0) (k 0)
   (i 0) (perm0 0))

    (declare ((simple-array fixnum (*)) perm perm1 count)
             (fixnum flips flipsmax check k r i perm0))

    (dotimes (i n) (setf (aref perm1 i) i))

    (loop

     (when (< check 30)
       (write-permutation perm1)
       (incf check))

     (loop while (> r 1) do
           (setf (aref count (1- r)) r)
           (decf r))

     (unless (or (= (aref perm1 0) 0)
       (= (aref perm1 (1- n)) (1- n)))
       (setf flips 0)
       (dotimes (i n) (setf (aref perm i) (aref perm1 i)))
       (setf k (aref perm1 0))
       (loop while (/= k 0) do
             (loop for j fixnum downfrom (1- k)
                   for i fixnum from 1
                   while (< i j) do (rotatef (aref perm i) (aref perm j)))
             (incf flips)
             (rotatef k (aref perm k)))
       (setf flipsmax (max flipsmax flips)))
```

[3] http://shootout.alioth.debian.org/u64q/performance.php?test=fannkuch

```
      (loop do
       (when (= r n)
         (return-from fannkuch flipsmax))
       (setf i 0 perm0 (aref perm1 0))
       (loop while (< i r) do
        (setf k (1+ i)
              (aref perm1 i) (aref perm1 k)
              i k))
       (setf (aref perm1 r) perm0)
       (when (> (decf (aref count r)) 0) (loop-finish))
       (incf r)))))

(defun main ()
  (let ((n (parse-integer (second *posix-argv*))))
    (format t "Pfannkuchen(~D) = ~D~%" n (fannkuch n))))
```

Wow, what a piece! Compare that to the equivalent PicoLisp program:

```
(let (N (format (opt))  Lst (range N 1)  L Lst  M)
   (recur (L)  # Permute
      (if (cdr L)
         (do (length L)
            (recurse (cdr L))
            (rot L) )
         (let I 0  # For each permutation
            (and (ge0 (dec (30))) (prinl (reverse Lst)))
            (for (P (copy Lst)  (> (car P) 1)  (flip P (car P)))
               (inc 'I) )
            (setq M (max I M)) ) ) )
   (prinl "Pfannkuchen(" N ") = " M) )
```

But at last we can find some significance:

```
(fannkuch 10)
   PicoLisp     6.4 sec
   sbcl         1.0 sec
   sbcl(i)     > 30 min   (aborted)

(fannkuch 11)
   PicoLisp    71.1 sec
   sbcl         5.0 sec
```

We see a factor of 14.2.

But at what a price! I'm not only talking about the discussed disadvantages of the compiler per se, but of that mess of code. I would not want to write my production code in such a style, and always prefer simplicity and succinctness over a bureaucratic monster.

If we remove the (declare (optimize ..)) statement, the execution time of SBCL doubles - from 5.0 to 10.0 seconds - and the factor goes down to 7.1.

BTW, the speed advantage is melting down if we use this parallized PicoLisp version (using the later[4] function):

```
(let (N (format (opt))  Lst (range N 1)  L Lst)
```

[4] http://software-lab.de/doc/refL.html#later

```
   (let (Res (need N)  M)
      (for (R Res R (cdr R))
         (later R
            (let L (cdr Lst)
               (recur (L)  # Permute
                  (if (cdr L)
                     (do (length L)
                        (recurse (cdr L))
                        (rot L) )
                     (let I 0  # For each permutation
                        (for (P (copy Lst)  (> (car P) 1)  (flip P (car P)))
                           (inc 'I) )
                        (setq M (max I M)) ) ) )
               M ) )
         (rot Lst) )
      (wait NIL (full Res))
      (prinl "Pfannkuchen(" N ") = " (apply max Res)) ) )
```

Then we get on the above 4-core machine

```
(fannkuch 10)
   PicoLisp     1.9 sec
   sbcl         1.0 sec

(fannkuch 11)
   PicoLisp    18.4 sec
   sbcl         5.0 sec
```

Up to `N` this scales almost linearly with the number of cores. With an 8-core machine it would well outperform SBCL.

Note: The printing of the first 30 results - as required by the Alioth benchmark - was omitted here, because their order is unpredictable for parallel execution and thus would not match the Alioth byte-by-byte comparison. A conformant solution (it shows no measurable timing difference) would be:

```
(let (N (format (opt))  Lst (range N 1)  L Lst)
   (catch NIL
      (recur (L)  # Print the first 30 permutations
         (cond
            ((cdr L)
               (do (length L)
                  (recurse (cdr L))
                  (rot L) ) )
            ((ge0 (dec (30)))
               (prinl (reverse Lst)) )
            (T (throw)) ) ) )
   (let (Res (need N)  M)
      (for (R Res R (cdr R))
         (later R
            (let L (cdr Lst)
               (recur (L)  # Permute
                  (if (cdr L)
                     (do (length L)
                        (recurse (cdr L))
                        (rot L) )
                     (let I 0  # For each permutation
```

```
                  (for (P (copy Lst)  (> (car P) 1)  (flip P (car P)))
                     (inc 'I) )
                  (setq M (max I M)) ) ) )
         M ) )
   (rot Lst) )
(wait NIL (full Res))
(prinl "Pfannkuchen(" N ") = " (apply max Res)) ) )
```