Guile RTL compiler, very preliminary performance test of completely alpha-code

Arne Babenhauserheide

August 19, 2013

Contents

1 Simple test harness

This test uses a simple recursive for-loop as performance measurement. Naturally that is not representative (insert all kinds of disclaimers), but it is interesting to me nontheless :)

1.1 Benchmark file

```
; #!/usr/bin/env guile
; #lang racket ; for testing with racket
; !#

; with the new guile RTL compiler, let-recursion, explicit compile
; (use-modules
;  (system vm assembler) (system base compile)
;  (language cps))
; (define* (my-compile x #:key (env (current-module)))
;   ((assemble-program (compile x #:env env #:to 'rtl))))
; (my-compile
;  '(define (forlooptest stop)
;     (let loop ((i 1) (x 42))(unless (>= i stop)
;                                     (loop (+ i 1) (/ i 2))))))

; direct recursion
; (define (forlooptest i stop x)
;   (unless (>= i stop)
;           (forlooptest (+ i 1) stop (/ i 2))))

; let-recursion
(define (forlooptest stop)
  (let loop ((i 1) (x 42))(unless (>= i stop)
                                  (loop (+ i 1) (/ i 2)))))

; while loop
; (define (forlooptest i stop x)
;   (while (< i stop)
;          (set! i (+ i 1)) (set! x (/ i 2))))

; test recursive
;  (forlooptest 1 100000000 42)

; test while loop or let-recursion
  (forlooptest 100000000)

; be nice to the shell :)
  (newline)
```

Listing 1: Benchmark function

1.2 Shell calls

```
hg clone git://git.sv.gnu.org/guile.git
cd guile
hg up -C wip-cps-bis
autoreconf -i && ./configure && make
for i in {1..10}; do time meta/guile ../fortran-lernen/forlooptest.scm; done
```

Listing 2: shell call, RTL guile

```
for i in {1..10}; do time guile ../fortran-lernen/forlooptest.scm; done
```

Listing 3: shell call, standard-guile

Then simply reprocess the output with ‘C-u M-| grep user‘ and ‘C-x r k‘ (rectangle kill).

2 RTL-compiler explicit compile

```
(let ((usertime '(
                  15.640
                  15.790
                  15.670
                  15.720
                  15.770
                  15.700
                  15.640
                  15.990
                  15.780
                  15.630
                  )))
  <<eval>>
   )
```

15\.732999999999999+-0\.09767343548785411

3 RTL-compiler no compile

```
(let ((usertime '(
                  16.810
                  16.480
                  16.470
                  16.710
                  16.290
                  16.430
                  16.740
                  16.170
                  16.560
                  16.540
                  )))
    <<eval>>
)
```

16\.52+-0\.18915602025840947

4 Guile 2.0

```
(let ((usertime '(
                  18.040
                  17.500
                  17.450
                  17.950
                  17.510
                  17.690
                  17.330
                  17.920
                  17.330
                  17.520
                  )))
<<eval>>
)
```

17\.624+-0\.24552474417052186

5 Summary

The compile step gives a significant performance boost for this let-recursive function. The runtime decreases by about 6.6% when compared to the same codebase without compile step. The test takes 15.7+-0.1s instead of 16.5+-0.2s.

A similar increase can already be seen when going from Guile 2.0.9 to the new code (16.5+-0.2s vs. 17.6+-0.2s).

Remember that this is an **ALPHA**-grade, work-in-progress feature.

6 Appendix: eval

```
(let* ((N (length usertime))
       (mean (/ (apply + usertime) N))
       (sigma (let ((numbers usertime))
                (let loop ((devsum 0)
                           (i (list-ref numbers 0))
                           (processed '())
                           (nums (list-tail numbers 1)))
                  (if (not (equal? nums '()))
                      (loop (+ devsum (* (- i mean) (- i mean)))
                            (list-ref nums 0)
                            (append processed (list i))
                            (list-tail nums 1))
                                        ; else
                      (sqrt (/ devsum N)))))))
  (display (format "~a~a~a"
                   (number->string mean)
                   "+-"
                   (number->string sigma))))
```

Listing 4: Calculate and output mean and standard deviation from a list of numbers (usertime)