Faster Programs with Guile 3

FOSDEM 2019, Brussels
Andy Wingo | wingo@igalia.com
wingolog.org | @andywingo
What?
› Your programs are faster with Guile 3!

How?
› The path to Guile 3

Where?
› The road onward
results

Guile 3 – it’s Guile, but faster!

Sum 10 million element f32vector

- 2.7x as fast

Expand (sxml ssax)

- 1.5x as fast

Guix graft

- ... as fast

And it will only get faster!
In 2006, I had Guile programs that ran too slowly.

I did everything

- C hot-paths
- Extensive caching/memoizing
- Built a profiler...

In the end, problem was: Guile ran Scheme code too slowly.

Solution: make Guile faster.
Guile in 2006
Guile in 2010
At run-time: interpret instructions from bytecode

Bytecode interpreter: `vm.c`

Like turing machine: bytecode is the tape

Interpreter sometimes called “virtual” machine

 Defined on top of “native” machine (e.g. x86, C, ...)
but then

A faster Guile means more kinds of programs can be written in Guile.

Also, I got hooked – making compilers is fun.

This is my job now.
Guile in 2017
current Guile needs

Language needs to evolve

- Approach Racket (frontend work)

Guile itself could be faster

- Enlarge set of Guile-appropriate problems

- Speed inception: speed up Guile, speed up compiler

- Maintain low-latency programming

- I am a junkie
Guile in 2019

(fib 42)

#<call #<ref fib> #<const 42>>

"CPS soup"

really low-level bytecode

machine code (e.g. x86-64)
Guile in 2019

(This is the Guile 3 work)

Next step in incremental, compatible improvement

2.9.1 released October 2018

“Done”-ish
Guile

3 goal

Generate good native code

› Avoid code bloat

› Limit complexity of implementation

› Keep support for all platforms

Two steps:

› Lower-level bytecode

› Generate native code
lower-level bytecode

Guile 2.2:

```
scheme@(guile-user)> ,x (lambda (x) (vector-ref x 0))
  0    (assert-nargs-ee/locals 2 0)
  1    (vector-ref/immediate 0 0 0)
  2    (handle-interrupts)
  3    (return-values 2)
```
lower-level bytecode

Guile 3.0:

```
scheme@(guile-user)> ,x (lambda (x) (vector-ref x 0))
  0  (instrument-entry 229)
  2  (assert-nargs-ee/locals 2 0) ;; 2 slots (1 arg)
  3  (immediate-tag=? 0 7 0) ;; heap-object?
  5  (jne 15) ;; -> L2
  6  (heap-tag=? 0 127 13) ;; vector?
  8  (jne 12) ;; -> L2
  9  (word-ref/immediate 1 0 0)
 10  (ursh/immediate 1 1 8)
 11  (imm-s64<? 1 0)
 12  (jnl 5) ;; -> L1
 13  (scm-ref/immediate 1 0 1)
 14  (reset-frame 1) ;; 1 slot
 15  (handle-interrupts)
 16  (return-values)
L1:
 17  (make-short-immediate 1 2) ;; 0
 18  (throw/value+data 1 177) ;; #(out-of-range ...)
L2:
 20  (throw/value+data 0 201) ;; #(wrong-type-arg ...)
compared to Guile 2.2

Instructions closer to machine code
More instructions
More control flow
More optimization opportunities (e.g. elide type checks)
More work for optimizer
compared to Guile 2.2

Compile time *could* be longer

- More instructions means more work for compiler

Run time *could* be longer

- More instructions means more work at run-time for instruction dispatch

*But...*
Interpreter:

/* make-short-immediate dst:8 low-bits:16 */
* Make an immediate whose low bits are
* LOW-BITS, and whose top bits are 0.
*/
{
  uint8_t dst;
  scm_t_bits val;

  UNPACK_8_16 (op, dst, val);
  SP_SET (dst, SCM_PACK (val));
  NEXT (1);
}

Compiler:

jit_movi (T0, SCM_UNPACK (val));
jit_stxi (8 * dst, SP, T0);
code generation

GNU Lightning: implementations of jit_mov_i, etc for all common architectures

Native code performs same operations on Guile stack that VM interpreter would

❖  No register allocation yet
❖  Tier-up possible anywhere
❖  Tier-down anywhere to debug

Complete JIT support in 5 kLOC

Only 1 reserved reg (current thread)
AOT?

Ahead-of-time (AOT) code generation perfectly possible

Native code currently a pure function of bytecode, not specialized on run-time values

Store result in ELF

Not yet implemented
when: JIT?

Just-in-time (JIT): generate native code at run-time

But when, specifically?

need to avoid codegen for bytecode that doesn’t matter

Guile: per-function counter incremented at call and loop iteration

Configurable tier-up threshold
status

GNU Lightning impedance probs :

Lightning 1: Close! But limited platforms

Lightning 2: API good, but...

- Crashes in optimizer sometimes :
- Do not want optimizer
- Regalloc useless for Guile
- Custom calling conventions hard

Need solution before 3.0
next?

Register allocation

Consistently comparable perf to Chez WASM backend! (Depends on "GC" proposal)

Racketification

(Figure out how I can play well with others!)
questions? https://gnu.org/s/guile
https://wingolog.org/
#guile on freenode
@andywingo
Happy hacking!
oh no
it’s the
bonus slides
JIT environment variables

**GUILE_JIT_THRESHOLD=50000**: When to JIT; -1 for never, 0 for always
- Call increments by 2, loop by 30
- High default == JIT slow currently

**GUILE_JIT_LOG=0**: Log level; up to 4.

**GUILE_JIT_STOP_AFTER=0**: Stop JIT compilation after this many functions. Useful for debug.

**GUILE_JIT_PAUSE_WHEN_STOPPING=0**: Pause for GDB to attach after stopping JIT.