NASH
an experimental
TRACING JIT VM
for
GNU GUILE
TRACING JIT
ASSUMPTIONS

- Programs spend most of their runtime in loops.
- Several iterations of the same loop are likely to take similar code paths.
TERMINOLOGY

*Trace*: Recorded instruction sequence.

*Fragment*: Artifact made from a trace.

*Guard*: A test inserted to native code.
GNU GUILE
Guile compiles Scheme source code to bytecode

Bytecode has flat structure.

Bytecode contains labels for jump destination.

Disassembly of mandelbrot at #x2a8:

```
0  (assert-nargs-ee/locals 3 9)
1  (static-ref 11 3087)
3  (sub 11 9 11)
4  (make-short-immediate 9 2)
5  (static-ref 8 3093)
7  (toplevel-box 7 3093 2925 2923 #t)
12 (box-ref 7 7)
13 (static-ref 6 3097)
15 (mov 5 8)
16 (mov 4 8)
17 (mov 8 9)
L1:
18  (br-if-< 7 8 #f 26)
21  (mul 3 4 4)
22  (mul 2 5 5)
23  (toplevel-box 1 3089 2909 2897 #t)
28  (box-ref 1 1)
29  (add 0 3 2)
30  (br-if-< 1 0 #f 12)
33  (add/immediate 8 8 1)
34  (sub 3 2 3)
35  (add 3 3 11)
36  (mul 5 6 5)
37  (mul 5 5 4)
38  (add 5 5 10)
39  (mov 4 5)
40  (mov 5 3)
41  (br -23)
L2:
42  (mov 10 8)
43  (return-values 2)
L3:
44  (mov 10 9)
45  (return-values 2)
```
Compiled bytecode instructions are interpreted by a C function.

... which is called VM-engine.

VM-engine could be viewed as a huge switch ... case statement.
NASH OVERVIEW
Key components:
- Interpreter
- Compiler
- Native code
User programs start from the interpreter. Interpreter observes each bytecode instruction, seeks for hot loops. When a hot loop was detected, the interpreter looks up accompanying native code. If no native code were found, the interpreter starts recording the instructions in the loop.
IP 41 (br -23) is a backward jump instruction. The interpreter jumps back to IP 18 (br-if-7 8 #f 20), which is marked as L1, the label one.

Then the interpreter records the instructions between IP 18 and 41.

Recorded trace:
The recorded instructions are then passed to compiler.

Compiler is written in Scheme.

Uses *A-normal form* IR.

Uses *GNU Lightning* as assembler backend.
IR contains a prologue section and a loop body section.

Prologue section loads locals from the stack, the stack is shared with the interpreter.

IR to native code compilation is done in almost straightforward manner.
After compilation, the control flow goes back to the interpreter.

The interpreter will find the native code of the loop from the next iteration.

Native code runs until guard fails. The guard failure triggers a bailout code.

Bailout code recovers VM state and passes back the control of user program to the interpreter.
IR of recorded trace

```scheme
;; trace 1: anf
(lambda ()
  (let* ((
    (v0 (%snap 0 2))
    (v2 (%sref 2 2))
    (v3 (%sref 3 2))
    (v4 (%sref 4 2))
    (v5 (%sref 5 2))
    (v6 (%sref 6 2))
    (v7 (%sref 7 1))
    (v8 (%sref 8 1))
    (v10 (%sref 10 2))
    (v11 (%sref 11 2)))
  (loop v0 v2 v3 v4 v5 v6 v7 v8 v10 v11))))
```

Native code of loop body (x86-64)

```assembly
;; trace 1: ncode 624
...
loop:
0x01ee61c8 cmp r14,r15
0x01ee61cb jl 0x01efe028
0x01ee61d1 movsd xmm13,xmm14
0x01ee61d6 mulsd xmm13,xmm14
0x01ee61db movsd xmm12,xmm15
0x01ee61e0 mulsd xmm12,xmm15
0x01ee61e5 mov r9,QUWORD PTR ds:0x1cac5a8
0x01ee61ed movsd xmm11,xmm13
0x01ee61f2 addsd xmm11,xmm12
0x01ee61f7 test r9,0x6
0x01ee61fe jne 0x01ee6200
0x01ee6204 mov rax,QUWORD PTR [r9]
0x01ee6207 and rax,0xffff
0x01ee620d cmp rax,0x217
0x01ee6213 jne 0x01ee6200
0x01ee6219 movsd xmm10,QUWORD PTR [r9+0x10]
0x01ee621f ucomisd xmm11,xmm10
0x01ee6224 ja 0x01ef030
0x01ee622a mov r11,r15
0x01ee623d add r11,0x4
0x01ee623j jo 0x01ee6280
0x01ee6237 mov r15,r11
0x01ee623a movsd xmm8,xmm13
0x01ee623f movsd xmm13,xmm12
0x01ee6244 subsd xmm13,xmm8
0x01ee6249 addsd xmm13,xmm5
0x01ee624e mulsd xmm15,xmm7
0x01ee6253 mulsd xmm15,xmm14
0x01ee6258 addsd xmm15,xmm6
0x01ee625d movsd xmm14,xmm15
0x01ee6262 movsd xmm15,xmm13
0x01ee6267 jmp 0x01ee61c8
...
BENCHMARKS
Total time normalized to Guile bytecode interpreter

<table>
<thead>
<tr>
<th></th>
<th>sumfp</th>
<th>mbrot</th>
<th>sum</th>
<th>ctak</th>
<th>string</th>
<th>parsing</th>
<th>fibc</th>
<th>dynamic</th>
<th>GM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nash</td>
<td>0.024</td>
<td>0.034</td>
<td>0.119</td>
<td>1.073</td>
<td>1.115</td>
<td>1.654</td>
<td>1.678</td>
<td>2.506</td>
<td>0.400</td>
</tr>
</tbody>
</table>

"sumfp" and "mbrot" contain loops with floating point number arithmetic.

"string" and "fibc" mostly use procedures implemented in C.

"parsing" and "dynamic" contain large amount of data-driven conditional branches.
Geometric standard scores of the benchmark suite from 10 Scheme native code compilers

Nash showed the best score in "string", but Guile bytecode interpreter was faster. "parsing" was the slowest benchmark for Nash, but the score of Pycket was not so bad.
SIGNIFICANT IMPROVEMENT IN TIGHT LOOP WITH FLOATING POINT ARITHMETIC.

No need to use "fl+, "fx+", ... etc.

Not much differences in procedures implemented in C.

Not much suited for programs with large amount of conditional branches, e.g.: parser, interpreter, and compiler.

JIT warming up time.

PROS AND CONS OF NASH
QUESTIONS?
THANK YOU

http://github.com/8c6794b6/guile-tjit