Nash: A Tracing JIT For Guile

Atsuro Hoshino

hoshinoatsuro@gmail.com

Abstract

This paper introduces *Nash*, an experimental *virtual machine* (VM) for GNU Guile with tracing *just-in-time* (JIT) compiler. Nash is designed as a drop-in replacement for Guile's existing VM. Nash could be used for running scripts, used at a REPL, and embedded in other programs. Design of Nash internals is discussed, including its VM interpreter which records frequently executed instructions found in Guile's bytecode, and its JIT compiler which emits native code from recorded instructions. Nash coexists with Guile's existing VM. Lots of Guile's features, such as bytecode interpreter, are reused in Nash. When conditions are met, Nash runs more than $40\times$ *faster* than Guile's existing VM without modifying the input program. Benchmark results of Nash are shown, including comparisons with other Scheme implementations.

Keywords Just-In-Time Compilation, Virtual Machine, Implementation, Scheme Programming Language

1. Introduction

From its simple design, Scheme is used for various purposes, has many implementations. One of the uses is as an extension language embedded in other program. Implementations such as Chibi-Scheme and TinyScheme are designed with use as an extension language in mind. On the other hand, there are Scheme implementations used for more expensive computations. This kind of Scheme implementations typically compiles to native code before executing. The compilation could be done *ahead-of-time* (AOT), such as in Bigloo (Serrano and Weis 1995) and Gambit (Feeley 1998), or incrementally, such as in Chez (Dybvig 2006) and Larceny (Hansen 1992), or in a mixture of AOT and JIT compilation, such as in Pycket (Bauman et al. 2015), and Racket (Flatt et al. 2013).

[Copyright notice will appear here once 'preprint' option is removed.]

```
(define (sumv-positive vec)
  (let lp ((i 0) (acc 0))
    (if (<= (vector-length vec) i)
        acc
        (lp (+ i 1)
            (if (< 0 (vector-ref vec i))
                (+ acc (vector-ref vec i))
                acc)))))
```

Figure 1. Scheme source code of sample procedure.

There exists a performance gap between Scheme implementations which does native code compilation and which doesn't. Tracing JIT compilation is a technique used in VM to improve performance by compiling the frequently executed instruction code paths. Dynamo (Bala et al. 2000) has pioneered the use of tracing JIT by tracing native code. Later the technique was used in various VMs for dynamic programming language to achieve performance improvement. Languages such as Lua (Pall 2016), JavaScript (Gal et al. 2009), and Python (Bolz et al. 2009) have made success with VMs which implement tracing JIT.

Nash is a new experimental tracing JIT VM for GNU Guile. Guile is a general-purpose Scheme implementation which could be used as an extension language, as a scripting engine, and for application development. Guile offers *libguile* to allow itself to be embedded in other program. GnuCash, gEDA, GNU Make, and GDB uses Guile as an extension language. Guile implements standard R5RS (Abelson et al. 1998), most of R6RS (Sperber et al. 2010), several SRFIs, and many extensions of its own, including delimited continuations and native POSIX thread support (Galassi et al. 2002). Nash is designed to be a drop-in replacement for Guile's existing VM, which is called *VM-regular* in this paper, to achieve performance improvement.

Figure 1 shows Scheme source code of a sample procedure `sumv-positive` which contains a loop. Details of Nash internal are explained with using `sumv-positive`. The `sumv-positive` procedure takes a single argument `vec`, a vector containing numbers. The loop inside the procedure checks whether the `i`-th `vector-ref` of `vec` is greater than `0`, adds up the element if true. The loop repeat the comparison and addition with incremented `i` until `i` is greater than the `vector-length` of `vec`.

Section 2 briefly mentions some background of Guile. In Guile, Scheme source codes are compiled to bytecode before the execution.[1] When Nash executes `sumv-positive`, the computation starts with bytecode interpretation. After executing the bytecode for a while, the bytecode interpreter detects a hot loop in the body of `sumv-positive`. Then the bytecode interpreter switches its state, starts recording the bytecode instructions and corresponding stack values of the loop. Recording of instructions are described in Section 3. When the bytecode interpreter reached to the beginning of the observed loop, recorded data are passed to JIT compiler. The JIT compiler is written in Scheme, executed with VM-regular. The compiler uses recorded bytecode and stack values to emit optimized native code. The variables from the stack are used to specify types, possibly emitting *guards* to exit from the compiled native code. More detail of the JIT compiler internals are covered in Section 4.

The rest of the sections are organized as follows. Section 5 shows results from benchmark, including comparisons between Nash and other Scheme implementations. Section 6 discusses current limitations and possibilities for future work. Finally, Section 7 mentions related works, and Section 8 concludes this paper. The source code of Nash is available at:

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

2. Background

This section describes background information and brief history of tracing JIT and GNU Guile. Some of the Guile internals affecting the design of Nash are mentioned.

2.1 Tracing JIT

Tracing JIT is one of JIT compilation styles (Bolz et al. 2009) which assumes that:

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

After Dynamo (Bala et al. 2000) used the technique to trace native code, various development has been done in the area. LuaJIT is one of the successful implementation of tracing JIT VM for the Lua (Ierusalimschy et al. 1996) programming language. Pypy (Bolz et al. 2009) is a tracing JIT VM for the Python programming language. Pypy is implemented with RPython framework, which is a *meta-tracing* infrastructure to develop a tracing JIT VM by defining an interpreter of the target language. The framework was adapted to other language than Python, including Pycket (Bauman et al. 2015), a tracing JIT VM for Racket, and Pixie, a tracing JIT VM for the Pixie language, which is a dialect of Lisp.

[1] By default, source codes are byte-compiled before executed. Guile can run Scheme source code without compilation so that trivial computations could be done quickly.

Tag	Type	Scheme value
`xxxxxxxxxxxxx000`	heap object	'foo, #(1 2 3) ...
`xxxxxxxxxxxxxx10`	small integer	1, 2, 3, 4, 5 ...
`xxxxxxxxxxxx100`	boolean false	#f
`xxxxxxxx00001100`	character	#\ a, #\ b, ...
`xxxxxx1100000100`	empty list	'()
`xxxxx10000000100`	boolean true	#t

Table 1. Tag value with corresponding Scheme type and Scheme value. The "x" in the tag column indicates any value.

Typical settings for tracing JIT of dynamic programming language contains an interpreter and a JIT compiler. The interpreter observes the execution of instructions, detects hot loops, and records frequently executed instructions. The recorded instructions are often called *trace*. JIT compiler then compiles the trace to get optimized native code of the hot loop. The interpreter typically has a functionality to switch between a phase for observing the loop, a phase for recording the instructions, and a phase for executing the compiled native code. Compiled native code contains *guards* to terminate the execution of native code, and bring the control of program back to the interpreter. Guards are inserted when a trace contained conditions which might not satisfied in later iteration of the loop. For instance, the loop in Figure 1 will emit a guard which compares the value of `i` with the length of the vector. In dynamic programming languages such as Scheme, guards for type check may be inserted as well, since compiler could generate more optimized native code when the types of the values are known at compilation time.

2.2 GNU Guile

GNU Guile was born to be an official extension language for GNU projects (Galassi et al. 2002). Since then, various developers have made changes to the implementation. As of version 2.1.2, Guile contains a bytecode compiler and a VM which interprets the compiled bytecode. Guile uses conservative Boehm-Demers-Weiser garbage collector (Boehm and Weiser 1988).

2.2.1 SCM Data Type

Guile's internal data type for Scheme object is defined as typedef `SCM` in C (Galassi et al. 2002). `SCM` value contains a type tag to identify its type in Scheme, which could be categorized as *immediates* or *heap objects*. Immediates consists of a combination of type tag and the value to identify itself. Immediates includes booleans, characters, small integers, the empty list, the end of file object, the *unspecified* object, *nil* object used in the Emacs-Lisp compatibility mode, and other special objects used internally. Heap objects are all the other types which could not fit itself in the bit size of `SCM`, such as symbols, lists, vectors, strings, procedures,

```
#<sumv-positive (vec)> at #x7f4ddb7fc51c:

   0    (assert-nargs-ee/locals 2 5)
        ...
L2:
  22    (uadd/immediate 0 6 1)
  23    (vector-ref 6 5 6)
  24    (br-if-u64-<-scm 3 6 #t 4)      ;; -> L3
  27    (add 1 1 6)
L3:
  28    (br-if-u64-<= 4 0 #f 9)         ;; -> L6
  31    (mov 6 0)
  32    (br -10)                        ;; -> L2
        ...
L6:
  37    (mov 5 1)
  38    (return-values 2)
```

Figure 2. Byte compiled code of `sumv-positive`. The contents is slightly modified from output of disassembler for displaying purpose and simplicity.

multi-precision integer numbers, and so on. When Guile decide the type of `SCM` value, firstly the three least significant bits (called *tc3* tag in Guile) of `SCM` value is checked to see whether the value belongs to immediates or heap objects. Table 1 shows the type tags for immediates and heap objects. When tc3 tag was `000`, the value belongs to heap objects. All the other values of tc3 tag are immediates, though some of the tag values are unused. For instance, tc3 tag `010` and `110` are used for small integers, the first two bits are used for a tag and the rest of the bits are used for an integer value. Scheme value 0 is SCM 00000010, Scheme value 1 is SCM 00000110, Scheme value 2 is SCM 00001010, and so on. Types of various heap objects are decided by non-tc3 tag part of `SCM` value. In such case, the non-tc3 tag part contains extra tag values and an address pointing to the contents of the heap object.

2.2.2 Bytecode Compiler

Guile's bytecode compiler is designed as *compiler tower*. The compiler consists of several compilers defining tower of languages. Each step of the compilation sequence knows how to compile down to the step below, until the compiled output turns into bytecode instructions executed by the VM.

In Guile version 2.1.2, Scheme input programs are first translated to a program in *tree-il* language, an internal representation used by Guile. Then resulting tree-il program is compiled to *cps*, which is another internal representation. Then the resulting cps code is translated to bytecode instructions. Guile contains compilers for Emacs-Lisp to tree-il, and Ecmascript to tree-il. The resulting tree-il language made out from Emacs-Lisp and Ecmascript reuse the rest of compilation path to bytecode instructions. The definition of the compilation steps could be modified, which helps the user to add a new high-level language compiled to `tree-il`, or directly compiling to bytecode, or compiling to different new target. Guile's bytecode compiler applies various optimizations, with all of them turned on by default. The optimizations could be turned off to save compilation time by sacrificing run time performances.

Figure 2 shows compiled bytecode of `sumv-positive`. Each bytecode instruction takes arguments and its use varies, some arguments are used as constant, some are used as an index value to read or write a value in current stack. The first line of the figure shows the procedure name and memory address of the byte-compiled data. The numbers shown in the left of each line are bytecode *instruction pointer* (IP) offset. For IP offsets specified as a jump destination, a numbered label starting from `L` is shown (IP offset 22, 28, and 37 in the figure). The bytecode which may cause a jump contains a comment with the destination label (IP offset 24, 28, and 32 in the figure).

When `sumv-positive` was called, VM-regular executes the bytecode from IP offset 0. Later the execution reaches to IP offset 32, `(br -10)`. The `br` bytecode instruction performs an unconditional jump by adding its argument to current IP. Negative offset means a backward jump, which is the case shown in the figure. After the jump, current IP offset is 22 (labeled as `L2` in the figure), which indicates a loop. In the bytecode instructions inside the loop, IP offset 24 and 28 are branching instructions. IP offset 24 may skip `(add 1 1 6)` instruction at IP offset 27. IP offset 28 may jump to `L6` to exit from the loop and return the value to the caller of `sumv-positive`.

3. Nash Interpreter

3.1 Nash Overview

Nash is designed as a drop-in replacement for VM-regular, could be used to run scripts, has REPL, and could be embedded in a C program as an extension language. Figure 3 shows flowchart of program execution in Nash, clustered by the software component in subject.

Typical control flow works as follows. **1.** Nash starts executing bytecode of Scheme program from interpreter. The interpreter evaluates bytecode instructions and keeps track of loops. **2.** The interpreter detects a hot loop. Then the interpreter looks up for native code. If native code was not found, the interpreter continues the evaluation of instructions with recording the bytecode and the values from current stack. **3.** The bytecode IP reaches to the end of the loop. The recording ends and traced information are passed to Nash compiler. **4.** Compilation finishes with native code generation. Control flow gets back to the interpreter. Bytecode interpretation continues from the IP where the recording has ended. **5.** The interpreter encounters the bytecode IP of the hot loop again. This time compiled native code exists for the IP. The interpreter executes the native code. **6.** A guard in the native code fails. Native code goes through a bailout code to recover the state of the interpreter and control flow gets back to the in-

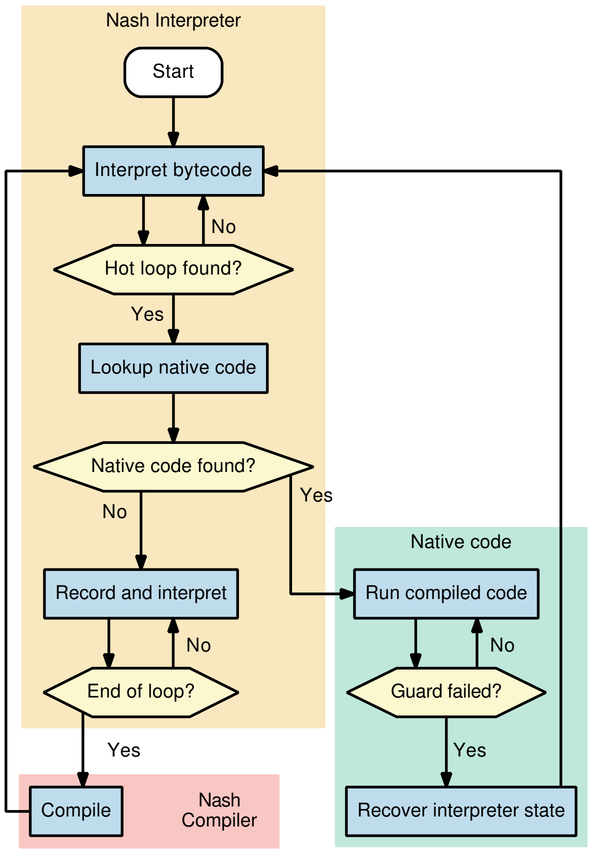

Figure 3. A flowchart diagram showing program execution in Nash.

terpreter. **7.** Interpreter reaches to the end of input bytecode, and program terminates. Transition **2** and **5** are repeated as necessary.

3.2 VM Engine For Nash

Guile uses C functions to interpret compiled bytecode. This C function is called *VM engine*. VM engine is defined in a dedicated file named `vm-engine.c`, which is included multiple times from other source code. This design enables defining multiple VM engine in few lines of C codes. As of Guile 2.1.2, there are two VM engines, one named `vm_regular_engine` which is used by VM-regular to interpret bytecode, and another named `vm_debug_engine` which is used for debugging. A single function named `VM_NAME` is defined in `vm-engine.c`. When including `vm-engine.c`, `VM_NAME` is defined with unique literal. The following C codes are written in other file than `vm-engine.c` to define `vm_regular_engine` and `vm_debug_engine`:

```
#define VM_NAME vm_regular_engine
#include "vm-engine.c"
#undef VM_NAME
...
#define VM_NAME vm_debug_engine
#include "vm-engine.c"
#undef VM_NAME
```

Inside the `VM_NAME` function, each bytecode instructions is defined by `VM_DEFINE_OP` macro with unique instruction number, with C macro `NEXT` at the last of definition body to perform next instruction. `VM_DEFINE_OP` macro fills in the jump table used by `VM_NAME` to define jump destinations.[2] Some of the instructions continue the interpretation with `NEXT` using constant offset value. For instance, `uadd_immediate` instruction always progress to the bytecode stored at 1 byte next from the current IP, thus the `NEXT` macro takes constant value 1. Some instructions use offset value specified from argument, such as `(br -10)` shown in Figure 2. The bytecode interpreter in Nash uses the same `vm-engine.c` file. The file is included once more with following codes:

```
#define VM_NAME vm_nash_engine
#define VM_NASH 1
#include "vm-engine.c"
#undef VM_NAME
```

Few small modifications were made to `vm-engine.c`. Nash uses C macros to add tracing functionality to `VM_NAME` function. C macros `VM_NASH_CALL`, `VM_NASH_TAILCALL`, and `VM_NASH_JUMP` were added to detect hot loops and enter compiled native code, and `VM_NASH_MERGE` to record instructions.

3.2.1 Finding Loops

Figure 4 shows a snippet containing modifications made to `VM_NAME`. `VM_NASH_JUMP` is in the definition body of `br`, `VM_NASH_CALL` in `call`, and `VM_NASH_TAIL_CALL` in `tail-call`. Bytecode definitions `br`, `call`, and `tail-call` were marked since these bytecode perform jumps to start loops.[3] Bytecode definitions which do not start a loop, such as `uadd_immediate`, are unmodified. The definition of `br` contains `VM_NASH_JUMP` with a parameter `offset`. When the `offset` was negative, the backward jump will be detected as a loop by Nash. When `br` with negative offset was found in interpreted bytecode, `VM_NASH_JUMP` looks for a native code with the next IP. If a native code was found, the native code is executed. Otherwise, `VM_NASH_JUMP` increment the counter value for the IP, and if the counter value exceeds a threshold parameter, `vm_nash_engine` starts recording the bytecode instruction in current loop. Similarly, `VM_NASH_CALL` and `VM_NASH_TAIL_CALL` use its argument `old_ip` to detect loops from consequent calls and tail-calls, respectively.

[2] Strictly speaking, Guile chooses jump table or `switch ... case` expression at build time for dispatching bytecode, by deciding whether the platform supports label as values (computed `goto`).

[3] Guile has more bytecode instructions for branching, such as `br-if-u64-<=`. These branching instructions are marked with `VM_NASH_JUMP` since they may perform a backward jump, though not shown in the figure.

```
 static SCM
 VM_NAME (scm_i_thread *thread, struct scm_vm *vp,
          scm_i_jmp_buf *registers, int resume) {
   ...
   VM_DEFINE_OP (1, call, ...) {
       ...
-      NEXT(offset);
+      VM_NASH_CALL (old_ip);
   }
   VM_DEFINE_OP (3, tail_call ...) {
       ...
-      NEXT(offset);
+      VM_NASH_TAIL_CALL (old_ip);
   }
   VM_DEFINE_OP (33, br, ...) {
       ...
-      NEXT(offset);
+      VM_NASH_JUMP (offset);
   }
   VM_DEFINE_OP (152, uadd_immediate, ...) {
       ...
       NEXT (1);
   }
   ...
 }
```

Figure 4. Modified VM_NAME function. Lines starting with - were deleted, + were added for Nash.

The internal C codes of VM_NASH_JUMP, VM_NASH_CALL, and VM_NASH_TCALL are mostly shared. One of the differences between the three is to use different strategy to decide hot loops by using different values to increment the loop counter. For instance, VM_NASH_JUMP may add 2 to the loop counter, while VM_NASH_CALL may add 1, which will result in a setting that backward jumps get hot sooner than consequent calls. VM_NASH_JUMP, VM_NASH_CALL, and VM_NASH_TAIL_CALL are defined as NEXT when including the file vm-engine.c to define other VM engines.

3.2.2 Recording Instructions

Figure 5 shows the modified contents of NEXT in VM_NAME function. When vm_nash_engine found a loop, observed bytecode and stack values are recorded by VM_NASH_MERGE. Figure 5 shows how the interpretation continues with updating the value of ip, which is a variable in VM_NAME used for bytecode IP. When the value of ip matched the beginning of the loop, VM_NASH_MERGE will stop recording, and pass the recorded data to JIT compiler. Figure 6 shows dumped sample data of recorded bytecode and stack values made by running sumv-positive with a length 1000 vector argument containing random small integer numbers from -10 to 10. The hexadecimal numbers in left are the absolute bytecode IP of each bytecode instruction, and the commented out vector in each line contains SCM representation of the values in the stack at the time of recording. The first bytecode (uadd/immediate 0 6 1) in Figure 6 is shown at

```
 # define NEXT(n)                              \
   do {                                        \
       ip += n;                                \
+      VM_NASH_MERGE ();                       \
       ...
       op = *ip;                               \
       goto *jump_table[op & 0xff];            \
   } while (0)
```

Figure 5. Modified definition of NEXT. Line starting with + was added for Nash.

IP offset 22 in Figure 2. vm_nash_engine continued the recording with IP offset 23, 24, 27, 28, and 31. Then at IP offset 32, bytecode (br -10) will be recorded and the interpreter jumps back to IP offset 22, which was labeled as L2 in Figure 2. The bytecode IP matches the IP where the recording has started, vm_nash_engine stops the recording. VM_NASH_MERGE is defined with empty body when including vm-engine.c file for other VM engines.

4. Nash Compiler

This section describes the details of JIT compiler in Nash, which is written in Scheme. Compiled bytecode of the JIT compiler is executed by vm_regular_engine.

4.1 Trace To IR

Nash compiles traces to relaxed A-normal form (Flanagan et al. 1993) internal representation (IR) before assigning registers and assembling to native code. Figure 7 shows IR of primitive operations compiled from the recorded trace in Figure 6. The IR primitives contain two lambda terms, the first block is for prologue, and the second block is for loop body. The IR uses let* instead of let to express the sequence of computations. Each primitive operation takes two arguments, except for %snap operation. Primitive operations updating a variable, such as %add, have the variable to be updated on the left-hand side of the expression. Primitive operations without variable update contain a symbol _ at the left-hand side. The variables starting with the letter v indicates that the variable was loaded from current stack. The variables starting with the letter r indicates that the variable is for temporal use only.

4.1.1 Snapshot

Between some recorded bytecode instructions, %snap expressions are inserted to make *snapshot* data. Snapshots contain various information to recover the state of vm_nash_engine when native code pass the control back. Snapshot data contains local indices to store variables, and a bytecode IP to tell vm_nash_engine where the interpretation should continue. The expression %snap takes variable number of arguments: the first argument is a *snapshot ID*, which is a unique integer number to identify the snapshot in single trace. The rest of the arguments are local variables to be stored in current

```
7f4ddb7fc574  (uadd/immediate 0 6 1)           ; #(#x3e #x2a2 #x1 #x0 #x3e8 #x7f4ddb808660 #x3e)
7f4ddb7fc578  (vector-ref 6 5 6)               ; #(#x3f #x2a2 #x1 #x0 #x3e8 #x7f4ddb808660 #x3e)
7f4ddb7fc57c  (br-if-u64-<-scm 3 6 #t 4)       ; #(#x3f #x2a2 #x1 #x0 #x3e8 #x7f4ddb808660 #x1a)
7f4ddb7fc588  (add 1 1 6)                      ; #(#x3f #x2a2 #x1 #x0 #x3e8 #x7f4ddb808660 #x1a)
7f4ddb7fc58c  (br-if-u64-<= 4 0 #f 9)          ; #(#x3f #x2ba #x1 #x0 #x3e8 #x7f4ddb808660 #x1a)
7f4ddb7fc598  (mov 6 0)                        ; #(#x3f #x2ba #x1 #x0 #x3e8 #x7f4ddb808660 #x1a)
7f4ddb7fc59c  (br -10)                         ; #(#x3f #x2ba #x1 #x0 #x3e8 #x7f4ddb808660 #x3f)
```

Figure 6. Bytecode instructions and stack values recorded with running `sumv-positive`.

```
 1 (lambda ()
 2    (let* ((_    (%snap 0))
 3           (v0   (%sref 0 #f))
 4           (v1   (%sref 1 1))
 5           (v3   (%sref 3 67108864))
 6           (v4   (%sref 4 67108864))
 7           (v5   (%sref 5 131072))
 8           (v6   (%sref 6 67108864)))
 9      (loop v0 v1 v3 v4 v5 v6)))
10 (lambda (v0 v1 v3 v4 v5 v6)
11    (let* ((v0   (%add v6 1))
12           (_    (%snap 1 v0 v1 v6))
13           (r2   (%cref v5 0))
14           (r2   (%rsh r2 8))
15           (_    (%lt v6 r2))
16           (r2   (%add v6 1))
17           (v6   (%cref v5 r2))
18           (_    (%snap 2 v0 v1 v6))
19           (_    (%typeq v6 1))
20           (_    _)
21           (r2   (%rsh v6 2))
22           (_    (%lt v3 r2))
23           (_    (%snap 3 v0 v1 v6))
24           (v1   (%addov v1 v6))
25           (v1   (%sub v1 2))
26           (_    (%snap 4 v0 v1 v6))
27           (_    (%gt v4 v0))
28           (v6   v0))
29      (loop v0 v1 v3 v4 v5 v6)))
```

Figure 7. IR of recorded trace in relaxed A-normal form.

stack. In Figure 7, Nash inserted `%snap` expression at the beginning, and before the primitive operations `%lt`, `%typeq`, `%addov`, and `%gt`, which act as guards. The primitives `%lt` and `%gt` does arithmetic less-than and greater-than comparisons, respectively. The primitive `%typeq` does type check with given variable and type, and the primitive `%addov` does addition with overflow check. When the result of guard differed from the result observed at the time of JIT compilation, native code executes the recovering steps to setup the state in `vm_nash_engine`, and the program continues with interpreting bytecode instructions.

4.1.2 Prologue section

The prologue section, the first `lambda` block shown in Figure 7, loads initial values from the stack with `%sref` primitive. The first argument passed to `%sref` is a local index offset, the second argument is an integer representation of the type of expected local in the stack. For instance, the value 1 is for `fixnum`, which means small integer value in Scheme, `131072` is vector object, `67108864` is `u64`, which is an unsigned 64 bit integer value to alias `scm_t_uint64` type defined in C, and so on.

The value `#f` in `%sref` primitive means that there is no need for type check, for instance the local is overwritten without referencing. The variable `v0`, which hold local `0` in above example is immediately overwritten by result of `%add` primitive in line 11 of Figure 7. There is not need to load this local from current stack, though such dead-code eliminations are not yet implemented.

4.1.3 Loop body section

The loop body section, the second `lambda` block in Figure 7, is compiled by translating each recorded bytecode instruction sequentially.

uadd/immediate The first primitive operation contains `%add`, which does arithmetic addition with variable `v6` and constant value 1. The result of addition overwrites variable `v0`. No overflow check is done with `uadd/immediate` in bytecode interpreters, and result will wrap around. Native code followed this behavior.

vector-ref Then a snapshot 1 is inserted, and the primitive operations for `vector-ref` follows. The primitive operations contain vector index range check, by comparing the length of vector with the index value passed to `vector-ref` instruction. For Scheme `vector` object, Guile uses the first one word to store a *tc7* tag and the length of the vector. A tc7 tag is, like tc3 tag, a 7 bits long tag value used to distinguish types. The length is left shifted for 8 bits so that the tc7 tag value and the length could fit in single word. Actual vector elements are stored from the memory address of the SCM object plus one word. The primitive operation `%cref` in line 13 loads a value from Scheme heap object with offset 0 and stores the loaded value to temporary register `r2`. Then `r2` is passed to `%rsh` in line 14, which does arithmetic right shift for 8 bits and overwrite the value of `r2`. Now `r2` contains the reproduced vector length, and compared with `v6`, which is the variable holding local 6, which is the index value used in recorded `vector-ref` instruction. Line 16 adds 1 to `v6` to get the offset of vector element. Line 17 does another `%cref` to load the vector element, and overwrites the value of `v6`.

br-if-u64-<-scm Line 18 contains a snapshot used by next primitive operation `%typeq`, which does a type check of `v6` with `fixnum`. The variable `v6` is an element in the argument `vector`, which was observed as `fixnum` at the time of JIT compilation. Line 20 shows empty value assigned to empty value. This line used to contain a `%snap` expression, though the JIT compiler has optimized away the snapshot, since the bytecode IP destination in snapshot data was identical with the previous snapshot, and no variables were updated by `%typeq`. Nash does few on-the-fly optimizations, such as this cached snapshot reuse, duplicated guard elimination, and constant folding. Variable `v6` is right shifted for 2 bits to move away the tc3 tag of `fixnum`, and the result is stored to variable `r2` in line 21, to compare with variable `v3` in line 22. Bytecode instruction `br-if-u64-<-scm` takes `u64` type as its first argument, SCM type as its second argument, and compares the two. The type of variable `v3` is determined as `u64` at the time of Scheme to bytecode compilation, no type checks are done in native code.

add Line 23 contains a snapshot, which will be used when arithmetic overflow occurred. Line 24 adds two `fixnum` values in `v1` and `v6` with `%addov` primitive, and overwrites the contents of `v1`. Type checks for `v1` and `v6` are not done, since the `fixnum` type check done by `%typeq` for `v6` is still valid, and `fixnum` type check for `v1` is already done in prologue section. Resulting type from addition of two `fixnum` is again a `fixnum` unless it overflows, thus type check for `v1` inside loop body is eliminated. The `%sub` primitive in line 25 subtracts the extra tc2 bits added by `%addov`.

br-if-u64-<= Line 26 inserts another snapshot. Then in line 27, two 64 bit unsigned values in `v4` and `v0` are compared. Types of the variables are determined by the bytecode.

mov Line 28 simply does a move, and overwrites the contents of `v6` with `v0`.

br Then the IR shows a call to `loop`, which tells that the computation jumps to the beginning of the loop body section. A loop body of native code exits when any of the guards failed. For instance, when the result returned by `(%gt v4 v0)` differed from the result observed at the time of JIT compilation, native code will pass the control back to `vm_nash_engine`, recover the interpreter state by using snapshot data from `(%snap 3 v0 v1 v6)` and the bytecode interpretation of input program will continue from bytecode IP `7f4ddb7fc5b0`, which is the jump destination of `(br-if-u64-<= 4 0 #f 9)` in recorded trace. [4]

[4] `7f4ddb7fc5b0 = 7f4ddb7fc58c + (9 * 4)`. `7f4ddb7fc58c` is the absolute bytecode IP of `(br-if-u64-<= 4 0 #f 9)` shown in Figure 6, 9 is the offset argument passed to `br-if-u64-<=`, and 4 is the size of single byte used for bytecode.

```
----      [snap  0] ()
0001      (%sref    r14 +0 ---)
0002      (%sref    r15 +1 fixn)
0003      (%sref    r9 +3 u64)
0004      (%sref    r8 +4 u64)
0005      (%sref    rcx +5 vect)
0006      (%sref    rdx +6 u64)
==== loop:
0007      (%add     r14 rdx +1)
----      [snap  1] ((0 u64) (1 fixn) (6 u64))
0009      (%cref    r11 rcx +0)
0010      (%rsh     r11 r11 +8)
0011    > (%lt      rdx r11)
0012      (%add     r11 rdx +1)
0013      (%cref    rdx rcx r11)
----      [snap  2] ((0 u64) (1 fixn) (6 scm))
0015    > (%typeq   rdx fixn)
0016      (%rsh     r11 rdx +2)
0017    > (%lt      r9 r11)
----      [snap  3] ((0 u64) (1 fixn) (6 scm))
0019    > (%addov   r15 r15 rdx)
0020      (%sub     r15 r15 +2)
----      [snap  4] ((0 u64) (1 fixn) (6 scm))
0022    > (%gt      r8 r14)
0023      (%move    rdx r14)
```

Figure 8. Primitive operation of recorded trace under x86-64 architecture. Slightly modified from dumped output for displaying purpose.

4.2 IR To Native Code

Figure 8 shows the IR in Figure 7 after register assignment. The numbers in left of each line, which are omitted for snapshot data, tells primitive operation numbers. The variables in Figure 8 are using register names for x86-64 architecture. Nash has a module with architecture specific register definitions. At the time of writing, supported architecture is x86-64 only.

Nash uses GNU lightning as an assembler backend. GNU lightning is a JIT compilation library, which runs under various architectures including aarch64, alpha, arm, ia64, mips, powerpc, s390, sparc, and x86. Nash contains a thin C wrapper which binds SCM type to the types understood by GNU lightning. The resulting bindings are called from Scheme code under `vm_regular_engine` just like other Scheme procedures defined in C.

Figure 9 shows dumped x86-64 native code of primitive operations in Figure 8. The native code contains a mark to show the beginning of the loop body with `loop:`, and snapshot ID numbers in lines with jump instruction.

Native code compiled in Nash contains debug symbols to interact with JIT compilation interface of GDB (Stallman et al. 2002), a well-known open-source debugger. This debugging support is turned off by default, but can be turned on

```
0x02cc005b mov    r14,QWORD PTR [rbx]
0x02cc005e mov    r15,QWORD PTR [rbx+0x8]
0x02cc0062 mov    r9,QWORD PTR [rbx+0x18]
0x02cc0066 mov    r8,QWORD PTR [rbx+0x20]
0x02cc006a mov    rcx,QWORD PTR [rbx+0x28]
0x02cc006e mov    rdx,QWORD PTR [rbx+0x30]
0x02cc0072 nop    WORD PTR [rax+rax*1+0x0]
loop:
0x02cc0078 lea    r14,[rdx+0x1]
0x02cc007c mov    r11,QWORD PTR [rcx]
0x02cc007f sar    r11,0x8
0x02cc0083 cmp    rdx,r11
0x02cc0086 jge    0x02ccc028    ->1
0x02cc008c lea    r11,[rdx+0x1]
0x02cc0090 lea    rax,[r11*8+0x0]
0x02cc0098 mov    rdx,QWORD PTR [rax+rcx*1]
0x02cc009c test   rdx,0x2
0x02cc00a3 je     0x02ccc030    ->2
0x02cc00a9 mov    r11,rdx
0x02cc00ac sar    r11,0x2
0x02cc00b0 cmp    r9,r11
0x02cc00b3 jge    0x02ccc030    ->2
0x02cc00b9 mov    r11,r15
0x02cc00bc add    r11,rdx
0x02cc00bf jo     0x02ccc038    ->3
0x02cc00c5 mov    r15,r11
0x02cc00c8 sub    r15,0x2
0x02cc00cc cmp    r8,r14
0x02cc00cf jle    0x02ccc040    ->4
0x02cc00d5 mov    rdx,r14
0x02cc00d8 jmp    0x02cc0078    ->loop
0x02cc00dd nop    DWORD PTR [rax]
```

Figure 9. Native code compiled from trace, under x86-64 architecture.

with command line option when invoking `guile` executable. [5]

5. Evaluation

5.1 Settings

Performance of Nash is evaluated with cross platform benchmark suite from Pycket project. The source code of the benchmarks originate from Larceny and Gambit project. Some modifications were made to the Pycket version. Benchmarks `bv2string`, `ntakl` and `quicksort` were added, which exist in the original Larceny and Gambit benchmark suite. Iteration counts for `ctak`, `fft`, `pnpoly`, `fibc`, and `ray` are decreased, which were taking long execution time in Guile's VM-regular. The modified benchmark suite contains 57 programs. Total elapsed time of each program including JIT warm up time was measured. The benchmark results

[5] Other than debug symbol, Nash contains a command line option to dump intermediate data during compilation. Contents of Figure 6, 7, 8, and 9 are obtained from the dumped output.

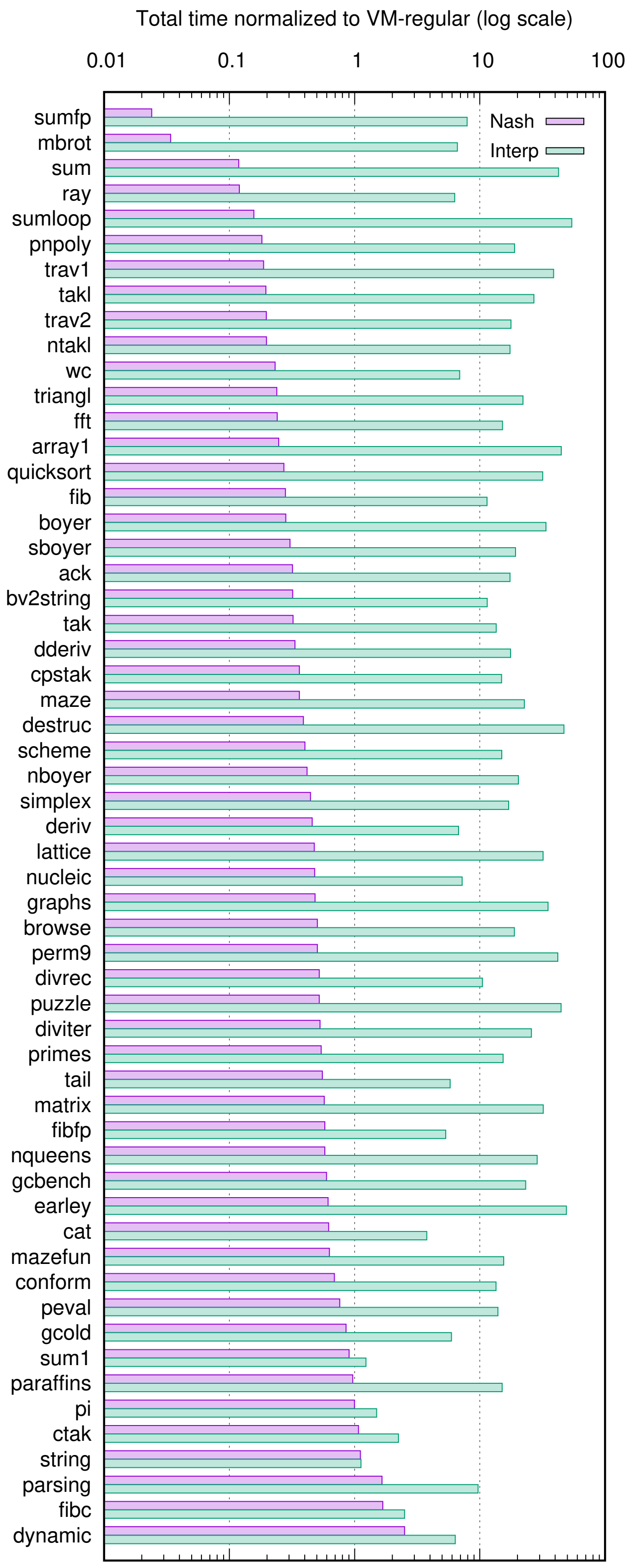

Figure 10. Benchmark results of VM-Nash and Guile interpreter, total time normalized to VM-regular. Lower is better.

	Nash	Interpreter
sumfp	0.024	7.926
mbrot	0.034	6.626
sum	0.119	42.579
sumloop	0.157	54.323
sum1	0.905	1.232
pi	0.999	1.503
string	1.115	1.125
ctak	1.073	2.243
fibc	1.678	2.507
parsing	1.654	9.685
dynamic	2.506	6.359
GM	0.400	13.770

Table 2. Selected benchmark results from Figure 10 and geometric mean (GM) of the benchmarks, shown in numbers.

were taken under a machine with Intel Core-i5 3427-U and 8GB of memory, running Arch Linux, Linux kernel 4.5.4.

5.2 Results

5.2.1 Comparison With VM-regular And Interpreter

Figure 10 shows total times of benchmark results of Nash and Guile's plain interpreter, normalized to Guile's VM-regular, in logarithmic scale. The Guile interpreter runs Scheme source code without bytecode compilation. Table 2 contains selected benchmark results and geometric means of the benchmark results in numbers.

Nash achieved significant performance improvements in benchmark programs containing tight loop with Scheme flonum values, such as `sumfp` and `mbrot`. Nash also achieved relatively large performance improvements with programs containing tight loop without Scheme flonum values, such as `sum` and `sumloop`.

Some of the benchmarks, such as `sum1`, `pi`, and `string`, showed similar performance in Nash, VM-regular, and Guile interpreter. In these benchmarks, large amount of the time were spent in procedures written in C functions, such as `read`, `string-append`, and arithmetic procedures for large integer numbers. All three implementations are calling the same underlying C functions, thus the performance differences were relatively small.

In benchmarks with many continuation object creations, such as `ctak` and `fibc`, Nash runs slower than VM regular. Nash implements `call/cc` with C function to keep compatibility with C interface of libguile, which did not lead to performance improvement. Performance overhead caused by JIT compilation and stack state recovery have slowed down the overall performance. Benchmarks `parsing` and `dynamic` did not contain creation of continuation objects, though these benchmarks had large amount of JIT compilation overhead caused by conditional branches.

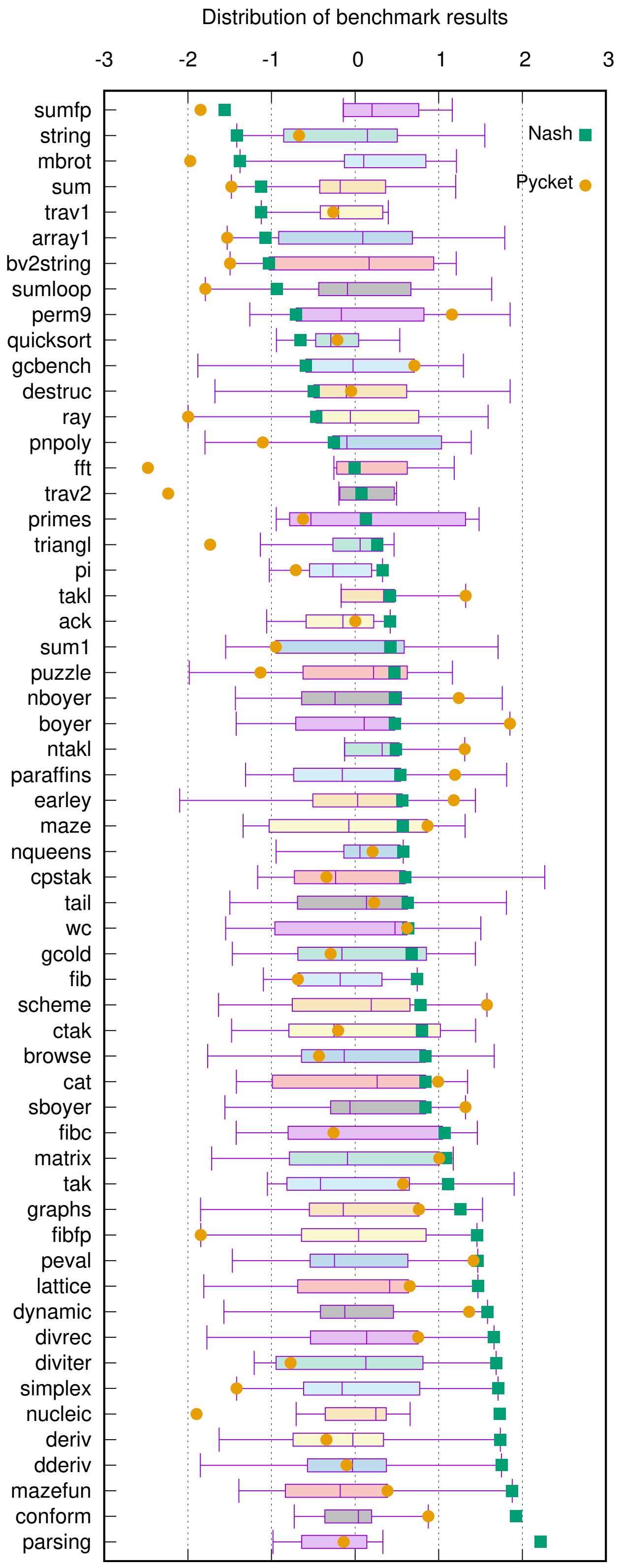

Figure 11. Distribution of benchmark results from native code compilers, and geometric standard scores of Nash and Pycket. Outliers are not shown. Lower is better.

Name (version)	GM	N	Failed	Strategy
Chez (9.4.1)	0.148	57		Incremental
Bigloo (4.2c)	0.236	54	pi, trav1, trav2	AOT via C
Ikarus (0.0.4rc1)	0.244	56	parsing	Incremental
Pycket (5f98bf3)	0.252	57		Tracing JIT
Gambit (4.8.5)	0.274	56	gcold	AOT via C
Larceny (0.99)	0.301	57		Incremental
Racket (6.6)	0.324	57		Method JIT
Nash	0.400	57		Tracing JIT
Chicken (4.11.0)	0.448	54	gcold, maze, pi	AOT via C
MIT (9.2)	0.486	55	parsing, nucleic	AOT

Table 3. Benchmark results of Scheme native code compilers. Geometric means of benchmark results are normalized to VM-regular, lower is better. Geometric means are calculated from valid results only, column `N` shows the number of valid results. Version for Pycket is git revision.

5.2.2 Comparison With Native Code Compilers

The benchmark results were taken with various Scheme native code compilers. Table 3 summarizes benchmark results from native code compilers with geometric means computed from valid benchmark results, number of succeeded benchmarks, and name of the failed benchmarks. Scheme to C compilers used GCC version 6.1.1 to compile the resulting C codes. Figure 11 shows boxplot of benchmark results and geometric standard scores from Nash and Pycket. Table 4 shows selected benchmark results in numbers.

Implementations using tracing JIT had some characteristics in common. Both Nash and Pycket performed well with benchmarks containing tight loops. In `sumfp`, `mbrot`, `sum`, `array1` and `sumloop`, Pycket and Nash are the fastest and second to the fastest. In `matrix`, `peval` and `dynamic`, Nash and Pycket performed badly. These benchmarks contained lots of conditional branches caused from data dependent control flows, which is a known weakness of tracing JIT (Bauman et al. 2015).

Pycket performed well with benchmark containing extensive use of `call/cc`, such as `fibc` and `ctak`. The geometric standard deviation of `fibc` and `ctak` are relatively larger than other benchmarks. The implementation of continuation differed across native code compilers, which lead to wider range of benchmark results.

In `fibfp`, `simplex`, and `nucleic`, Pycket was the fastest implementation and Nash was the slowest implementation.

	GSD	Nash	Pycket
sumfp	3.40	-1.56	-1.85
mbrot	2.73	-1.38	-1.97
sum	2.33	-1.12	-1.48
array1	1.62	-1.07	-1.53
sumloop	2.26	-0.94	-1.79
matrix	1.80	+1.09	+1.01
peval	2.18	+1.47	+1.42
dynamic	2.34	+1.58	+1.36
ctak	7.21	+0.80	-0.20
fibc	6.29	+1.07	-0.26
fibfp	1.89	+1.46	-1.84
simplex	1.61	+1.71	-1.42
nucleic	1.58	+1.73	-1.90
parsing	2.55	+2.22	-0.14

Table 4. Selected benchmark results from Figure 11. Showing geometric standard deviation, geometric standard score of Nash, and geometric standard score of Pycket. Lower standard score is better.

All three of the benchmarks contained Scheme flonum values, which were efficiently handled in Pycket but not in Nash. Nash performed badly in `parsing` benchmark also, but the performance result from Pycket was the median value. Performance improvement for `parsing` benchmark is a known problem for Nash which requires further efforts.

6. Limitations and Future Work

Nash is still an experimental implementation. Some of the possibilities for future works follows.

Support more bytecode instructions Nash still has bytecode instructions which are not implemented at all, such as `prompt` and `abort` used for delimited continuation, or partially implemented, such as arithmetic operations. Arithmetic operations for multi-precision numbers and complex numbers are not yet implemented. When JIT compiler encountered unsupported bytecode, it aborts the work and falls back to `vm_nash_engine`.

Detect more loops Nash detects loops with backward jump, tail-calls, and consequent calls for non-tail-call recursion (*down-recursion*[6]), though not with consequent returns from non-tail-call recursions (*up-recursion*) yet.

Optimize IR Nash does only a few IR level optimizations. In prologue section of IR shown in Section 4.1.2, unnecessary load from stack exist, which could be omitted by dead-code elimination. More optimizations could be done, such as loop invariant code motion, escape analysis, allocation removals, and so on. Also, Nash currently uses naive

[6] Consequent calls to procedure are called as down-recursion in Nash, because Guile's stack grows down

method to assign registers. More sophisticated method such as Linear-Scan register allocation (Poletto and Sarkar 1999) could be used.

Blacklist Benchmarks such as `parsing` and `dynamic` were slower than Guile VM-regular. Programs containing large amount of branching conditions need some treatments to perform well under tracing JIT. One approach is to limit the JIT compilation when branching exceed certain threshold. These thresholds mechanism are sometime called *blacklisting* of trace. Nash could take more sophisticated approach to blacklist unwanted traces.

7. Related Work

Implementation of Nash was inspired from pioneers in tracing JIT field. Nash does type checks in VM interpreter before running compiled native code, which was done in TraceMonkey (Gal et al. 2009) with similar design.

The mechanisms in interpreter to record instructions and to detect hot loops were inspired from Pypy and RPython (Bolz et al. 2009). The way how Nash marks `NEXT` and loop starting bytecode definitions are influenced by the approach used in interpreters written with RPython. RPython provides a framework to define a new language by writing an interpreter, while Guile provides a framework to define a new language by writing a compiler. Use of A-normal form (Flanagan et al. 1993) was inspired from Pycket (Bauman et al. 2015). Though Pycket expands the source code and generates JSON data, and parses it to AST for execution. Both Nash and Pycket use tracing JIT in its implementation. Results shown in Section 5 proved that Pycket is generally faster than Nash. However, Pycket is designed as a batch file compiler in single threaded computation. As of the version used in the benchmarks, Pycket lacks REPL, threads, and FFI support.

Design of snapshot was inspired from LuaJIT (Pall 2009). LuaJIT uses more sophisticated approach and compressed snapshot data. LuaJIT used NaN-tagging, which enables efficient handling of unboxed floating point numbers. Guile once had an attempt to use NaN-tagging (Wingo 2011), though the attempt wasn't merged to Guile source code, due to supporting conservative garbage collector under 32 bit architectures.

8. Conclusion

This paper has shown how Nash is designed. Nash reused existing bytecode interpreter in Guile, turned it to a trace recording interpreter with few small modifications. Nash coexist with existing VM, which is used for JIT compiler written in Scheme. Existing bytecode compiled by Guile are traced, recorded and compiled to native code. Performance comparison between Nash, Guile's existing VM, and various Scheme compilers were done with 57 benchmarks. With keeping itself as an extension language, Nash showed significant speed ups in programs with tight loops, and achieved competitive speed with Scheme implementations with native code compiler in overall performance.

Acknowledgments

Nash exists because of the hard works done for GNU Guile by many open source developers.

References

H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr, D. H. Bartley, R. Halstead, et al. Revised5 report on the algorithmic language scheme. *Higher-order and symbolic computation*, 11 (1):7–105, 1998.

V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

S. Bauman, C. F. Bolz, R. Hirschfeld, V. Kirilichev, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. Pycket: A tracing jit for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pages 22–34. ACM, 2015.

H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9): 807–820, 1988.

C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

R. K. Dybvig. The development of chez scheme. In *ACM SIGPLAN Notices*, volume 41, pages 1–12. ACM, 2006.

M. Feeley. Gambit-c version 3.0. *An implementation of Scheme available via http://www. iro. umontreal. ca/~ gambit*, 6, 1998.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM Sigplan Notices*, volume 28, pages 237–247. ACM, 1993.

M. Flatt et al. The racket reference, 2013.

A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. *ACM Sigplan Notices*, 44(6):465–478, 2009.

M. Galassi, J. Blandy, G. Houston, T. Pierce, N. Jerram, and M. GrabmÃijller. Guile reference manual, 2002.

L. T. Hansen. *The impact of programming style on the performance of Scheme programs*. PhD thesis, Citeseer, 1992.

R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho. Lua-an extensible extension language. *Softw., Pract. Exper.*, 26(6): 635–652, 1996.

M. Pall. Luajit 2.0 intellectual property disclosure and research opportunities. `http://lua-users.org/lists/lua-l/2009-11/msg00089.html`, 2009. [Online, accessed June 14, 2016].

M. Pall. Luajit project. `http://www.luajit.org`, 2016. [Online, accessed June 14, 2016].

M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.

M. Serrano and P. Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *Static Analysis*, pages 366–381. Springer, 1995.

M. Sperber, R. K. Dybvig, and M. Flatt. *Revised [6] report on the algorithmic language Scheme*, volume 19. Cambridge University Press, 2010.

R. Stallman, R. Pesch, S. Shebs, et al. Debugging with gdb. *Free Software Foundation*, 51:02110–1301, 2002.

A. Wingo. value representation in javascript implementations. `https://wingolog.org/archives/2011/05/18/value-representation-in-javascript-implementations`, 2011. [Online, accessed June 14, 2016].