Challenges of ‘pairs’ and ‘next’ JIT compilation

Maxim Bolshov, IPONWEB

Lua in Moscow 2019

IPONWEB

- Building platforms for real-time advertising
- Lua in production for more than 11 years
- Forked LuaJIT 2.0 in Q2 2015

Not Yet Implemented

Builtins and bytecodes that are not supported by **JIT compiler**

- Cannot be compiled, interpreter overhead
- Prevents other code from being compiled and optimized

Optimization case study

```lua
function keys(t, mapper)
  local res = {}
  for k, _ in pairs(t)
    table.insert(res, mapper(k))
  end
end
```

Optimization case study

```lua
function keys(t, mapper)
  local res = {}
  for k, _ in pairs(t)
    res[#res + 1] = mapper(k)
  end
end
```

Optimization case study

```lua
function keys(t, mapper)
  local res = {}
  for k, _ in pairs(t)
    table.insert(res, mapper(k))
  end
end
```

Optimization case study

```lua
-- pairs NYI
-- table.insert slow
function keys(t, mapper)
  local res = {}
  for k, _ in pairs(t)
    table.insert(res, mapper(k))
  end
end
```

```lua
-- pairs still NYI
-- faster but less idiomatic
function keys(t, mapper)
  local res, len = {}, 0
  for k, _ in pairs(t)
    local nk = mapper(k)
    if nk ~= nil then
      len = len + 1
      res[len] = nk
    end
  end
end
```

Another approach: builtins

- shallow copy

```
function shallowcopy(t)
  assert(type(t) == "table", "invalid input")
  local copy = {}
  for key, value in pairs(t) do
    copy[key] = value
  end
end
```

Another approach: builtins

- Implement popular ‘pairs’ patterns in a platform

- JIT support included

Do we need ‘next’ compiled?

Abort reasons:

● C function	74.2%
● FastFunc pairs + FastFunc next	13.1%
● unsupported variant of FastFunc string.find	3.5%
● register coalescing too complex	1.4%
● bytecode TSETM	1.2%

Tracing JIT in action

```lua
local sum = 0
local t = {1, 2, 3, ...}
for _, v in ipairs(t) do
  if v < 100 then
    sum = sum + v
  else
    sum = sum - v
  end
end
```

Tracing JIT in action

```lua
local sum = 0
local t = {1, 2, 3, ...}
for _, v in ipairs(t) do
  if v < 100 then
    sum = sum + v
  else
    sum = sum - v
  end
end
```

```
...
0013    KSHORT   7 100
0014    ISGE     5   7
0015    JMP      7 => 0018
0016    ADD      1   1   6
0017    JMP      7 => 0019
0018    SUB      1   1   6
0019    ITERC    5   3   3
0020    ITERL    5 => 0013
0021    RET0     0   1
```

Tracing JIT in action

```lua
local sum = 0
local t = {1, 2, 3, ...}
for _, v in ipairs(t) do
  if v < 100 then
    sum = sum + v
  else
    sum = sum - v
  end
end
```

```
...
0013    KSHORT   7 100
0014    ISGE     5   7
0015    JMP      7 => 0018
0016    ADD      1   1   6
0017    JMP      7 => 0019
0018    SUB      1   1   6
0019    ITERC    5   3   3
0020    ITERL    5 => 0013
0021    RET0     0   1
```

Tracing JIT: Intermediate Representation (IR)

```lua
local sum = 0
local t = {1, 2, 3, ...}
for _, v in ipairs(t) do
  if v < 100 then
    sum = sum + v
  else
    sum = sum - v
  end
end
```

```
0017 ------------- LOOP -------------
0018 xmm7      flt CONV   0011  flt.int
0019         > flt LT     0018  100
0020 xmm6    + flt ADD    0016  0005
0021 rbp     + int ADD    0011  +1
0022         > int ABC    0012  0021
0023 r15       p64 AREF   0014  0021
0024 xmm7   >+ flt ALOAD  0023
```

Tracing JIT: types

```lua
local sum = 0
local t = {1, 2, 3, ...}
for _, v in ipairs(t) do
  if v < 100 then
    sum = sum + v
  else
    sum = sum - v
  end
end
```

```
0017 ------------- LOOP ------------
0018 xmm7      flt CONV   0011  flt.int
0019        >  flt LT     0018  100
0020 xmm6    + flt ADD    0016  0005
0021 rbp     + int ADD    0011  +1
0022        >  int ABC    0012  0021
0023 r15       p64 AREF   0014  0021
0024 xmm7   >+ flt ALOAD  0023
```

Tracing JIT: hot path

```lua
local sum = 0
local t = {1, 2, 3, ...}
for _, v in ipairs(t) do
  if v < 100 then
    sum = sum + v
  else
    sum = sum - v
  end
end
```

```
0017 ------------- LOOP -------------
0018 xmm7      flt CONV   0011  flt.int
0019         > flt LT     0018  100
0020 xmm6    + flt ADD    0016  0005
0021 rbp     + int ADD    0011  +1
0022         > int ABC    0012  0021
0023 r15       p64 AREF   0014  0021
0024 xmm7   >+ flt ALOAD  0023
```

Tracing JIT: guards

```
local sum = 0
local t = {1, 2, 3, ...}
for _, v in ipairs(t) do
  if v < 100 then
    sum = sum + v
  else
    sum = sum - v
  end
end
```

```
0017 ------------- LOOP ------------
0018 xmm7      flt CONV   0011  flt.int
0019         > flt LT     0018  100
0020 xmm6    + flt ADD    0016  0005
0021 rbp     + int ADD    0011  +1
0022         > int ABC    0012  0021
0023 r15       p64 AREF   0014  0021
0024 xmm7   >+ flt ALOAD  0023
```

Tracing JIT: control flow

The only elements:

- Exit to interpreter or another trace
- Single backward branch to a ‘LOOP’ label

Tracing JIT

	Bytecode (VM)	**IR (JIT)**
Coverage	Entire code	Hot path only
Control flow	Explicit	Implicit
Type information	Implicit	Explicit

pairs / next

From Lua Reference Manual:

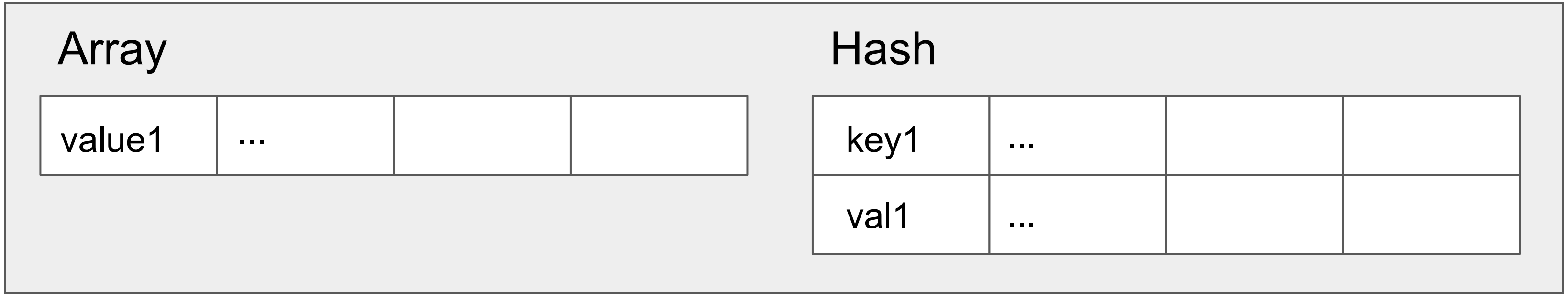
- pairs(t):
  - returns next, t, nil
- next(table[, index]): allows a program to traverse all fields of a table.
  - returns initial key-value pair when called with nil
  - returns ‘next’ to index key and associated value
  - returns ‘nil’ when all elements were traversed

Table traversal

```lua
for k, v in pairs(t) do
  -- body
end
```

Table traversal: implementation

```lua
for k, v in pairs(t) do
  -- body
end
```



- ‘array’ and ‘hash’ traversals are different

Table traversal: implementation

```
for k, v in pairs(t) do
  -- body
end
```

Array

value1	...	nil	

Hash

key1	...	nil	
val1	...		

- ‘array’ and ‘hash’ traversals are different
- need to skip ‘nil’ elements

Table traversal: tracing JIT

- ‘array’ and ‘hash’ traversals are different
- need to skip ‘nil’ elements

Table traversal: tracing JIT

- ‘array’ and ‘hash’ traversals are different
  - generate separate traces for array and hash parts

- need to skip ‘nil’ elements

Table traversal: tracing JIT

- ‘array’ and ‘hash’ traversals are different
  - generate separate traces for array and hash parts

- need to skip ‘nil’ elements
  - use CALL IR to skip nils

‘next’ compilation: CALL IR

Special IR that allows to call functions compiled by C compiler

‘next’ compilation: CALL IR

Special IR that allows to call functions compiled by C compiler

- Good: don’t need to implement machine code generation
- Bad: black box for JIT compiler
  - Prevents JIT IR optimizations
  - Suboptimal register allocation

‘next’ compilation: example of traces

```
0013 ------------ LOOP ------------
0014           flt CONV   0007  flt.int
0016 rax     + int CALLL  lj_tab_nexta  (0002 0007)
0017        >  int ABC    0008  0016
0018 rbx       p64 AREF   0010  0016
0019 xmm7   >+ flt ALOAD  0018
```

‘next’ compilation: example of traces

```
0010 ------------- LOOP -------------
0011 rdx      p64 HREF   0002  0008
0013 rax      p64 CALLL  lj_tab_nexth  (0002 0011)
0014 xmm6  >+ flt HLOAD  0013
0015 rcx      p64 HREF   0002  0014
0016 xmm7  >+ flt HLOAD  0015
```

‘next’ compilation: HREF and HLOAD

- HREF
  - Hashes key and return anchor position in hash-table
  - Resolves hash collisions
  - Returns pointer to desired hash node
- HLOAD
  - Loads value from pointer to hash node

‘next’ compilation: example of traces

```
0010 ------------- LOOP ------------
0011 rdx      p64 HREF   0002  0008
0013 rax      p64 CALLL  lj_tab_nexth  (0002 0011)
0014 xmm6   >+ flt HLOAD  0013
0015 rcx      p64 HREF   0002  0014
0016 xmm7   >+ flt HLOAD  0015
```

- Two hash-table lookups per iteration
- CALL is a black box for JIT

Adding new IR instruction

- Implement machine code generation
- Optimize: default behavior may be unexpected

Adding new IR instruction

```
...
0003 flt ADD    0001  0002
...
???? flt ADD    0001  0002
```

Adding new IR instruction

```
...
0003 flt ADD    0001  0002
...
???? flt ADD    0001  0002  -- do not emit, use 0003rd IR
```

Adding new IR instruction

```
...
0003 flt ADD    0001  0002
...
???? flt ADD    0001  0002  -- do not emit, use 0003rd IR



...
0002 flt HLOAD  0001
...
???? flt HLOAD  0001
```

Adding new IR instruction

```
...
0003 flt ADD    0001  0002
...
???? flt ADD    0001  0002  -- do not emit, use 0003rd IR


...
0002 flt HLOAD  0001
...
???? flt HLOAD  0001        -- analysis is needed
```

‘next’ compilation: going further

- HKLOAD
- NEXTA / NEXTH

```
0010 ------------ LOOP ------------
0011 rdx        p64 HREF   0002  0008
0013 rax        p64 NEXTH  0002  0011
0014 rbp     >+ flt HKLOAD 0013
0015 xmm7    >+ flt HLOAD  0013
```

Iterators: translating to bytecode

```
local t = {}
for k, v in ipairs(t) do
  -- body
end
```

```
GGET      1   0        ; "ipairs"
MOV       2   0
CALL      1   4   2
JMP       4 => 0007
-- body
ITERC     4   3   3
ITERL     4 => 0006
RET0      0   1
```

Iterators: translating to bytecode

```lua
local t = {}
for k, v in pairs(t) do
  -- body
end
```

```
GGET      1   0        ; "pairs"
MOV       2   0
CALL      1   4   2
ISNEXT    4 => 0007
-- body
ITERN     4   3   3
ITERL     4 => 0006
RET0      0   1
```

Iterators: translating to bytecode

```
GGET      1   0         ; "ipairs"
MOV       2   0
CALL      1   4   2
JMP       4 => 0007
-- body
ITERC     4   3   3
ITERL     4 => 0006
RET0      0   1
```

```
GGET      1   0         ; "pairs"
MOV       2   0
CALL      1   4   2
ISNEXT    4 => 0007
-- body
ITERN     4   3   3
ITERL     4 => 0006
RET0      0   1
```

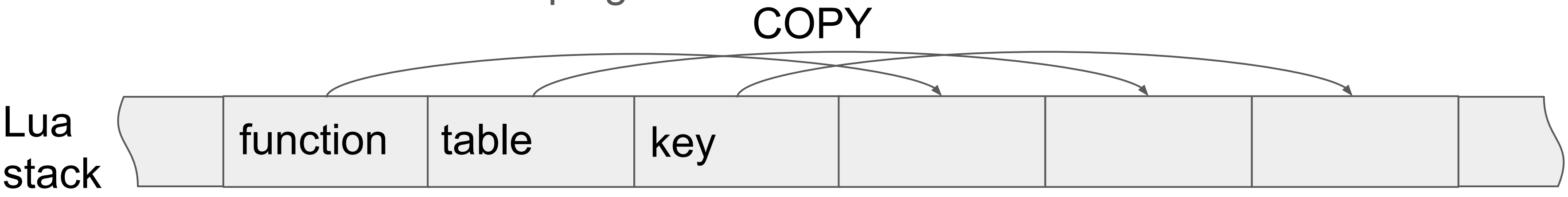
```
isnext = nvars <= 5 && predict_next(ls, fs, exprpc);
```

ISNEXT vs JMP

- Check that
  - Iterator is built-in ‘next’
  - First iterator argument is a table
  - Second iterator argument is nil
- Initializes ‘hidden’ control variable
- If any of checks failed, despecializes to JMP and ITERC

ITERN vs ITERC

- ITERC
  - Copy arguments
  - Call iterator
  - ITERL handles looping



ITERN vs ITERC

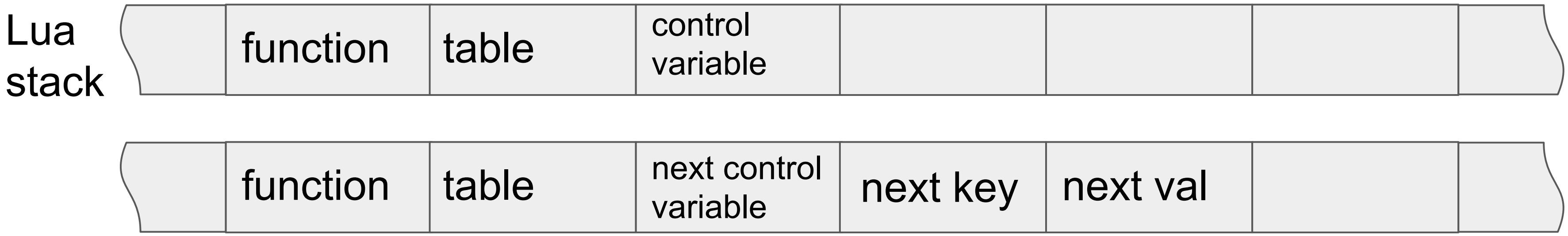
- ITERC
  - Copy arguments
  - Call iterator
  - ITERL handles looping

ITERN vs ITERC

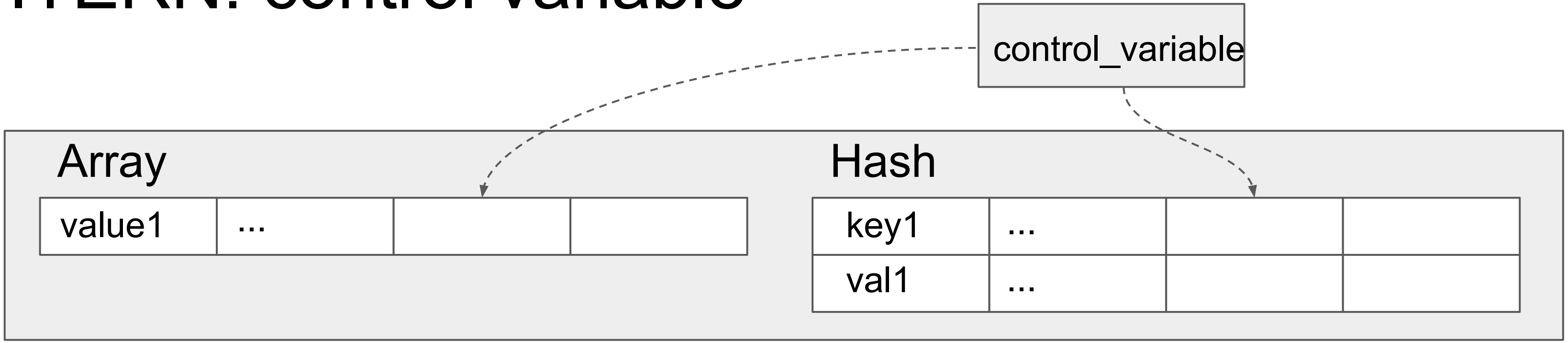
- ITERC
  - Copy arguments
  - Call iterator
  - ITERL handles looping

ITERN vs ITERC

- ITERN
  - Fetches arguments in-place
  - Computes key-value pair by control variable
  - Handles iteration itself, ITERL is never executed



ITERN: control variable




```
if (control_variable < t.arraysize)
  key = control_variable
  value = array[control_variable]
else if (i < (t.arraysize + t->hashsize))
  key = node[control_variable - t.arraysize].key
  value = node[control_variable - t.arraysize].value
else
  key = nil # end of iteration
end
```

ITERN: control variable

0xfffe7fff	control var
32 bits	32 bits

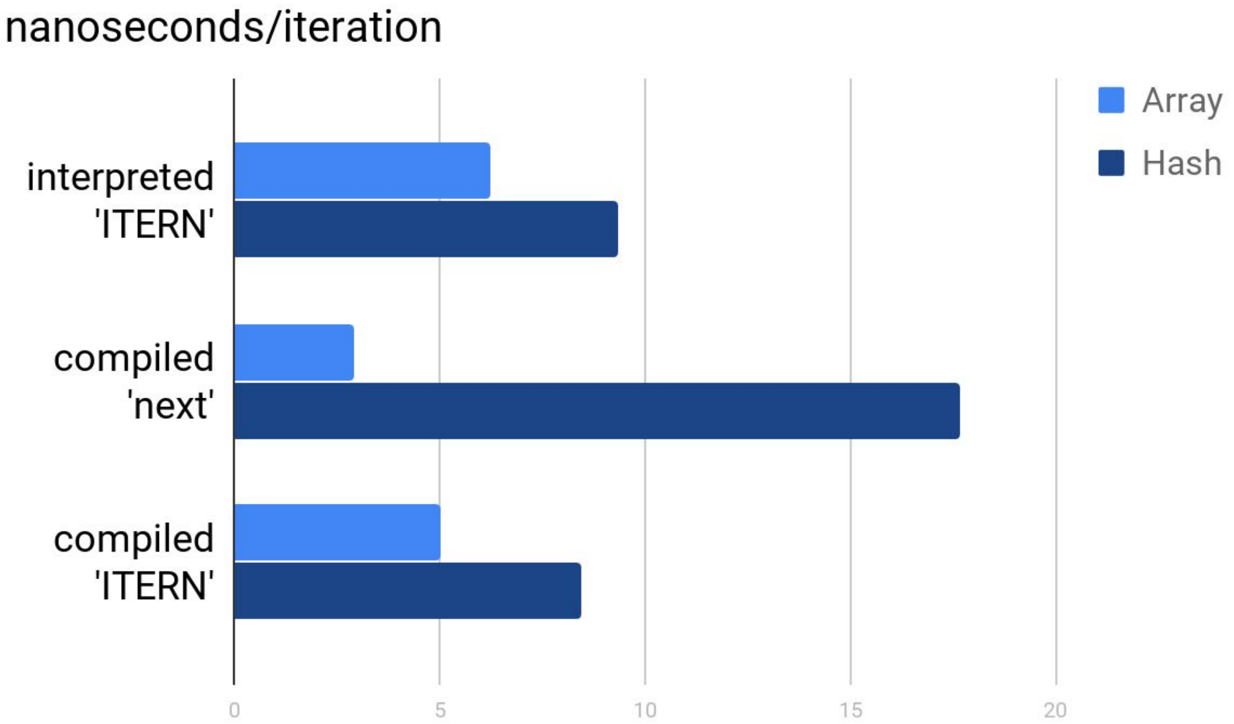
- Control variable needs watermarking
- Implemented as a special type of NaN-tagging

ITERN compilation

- HKLOAD is useful again
  - now 0 hash lookups per iteration
- ITERN — ITERL decoupling

```
0015 ------------- LOOP ------------
0017 rax       u32 NEXT   0001 0014
0018        >  int NE     0017  +0
0019 rbp       int ADD    0017  -1
0020 rbp       int SUB    0019  0007
0021 rbp       int MUL    0020  +40
0022 rbp       u64 ADD    0021  0006
0023 xmm7   >+ flt HKLOAD 0022
0024        >  flt HLOAD  0022
0025 [8]     + flt TRIDX  0017
```

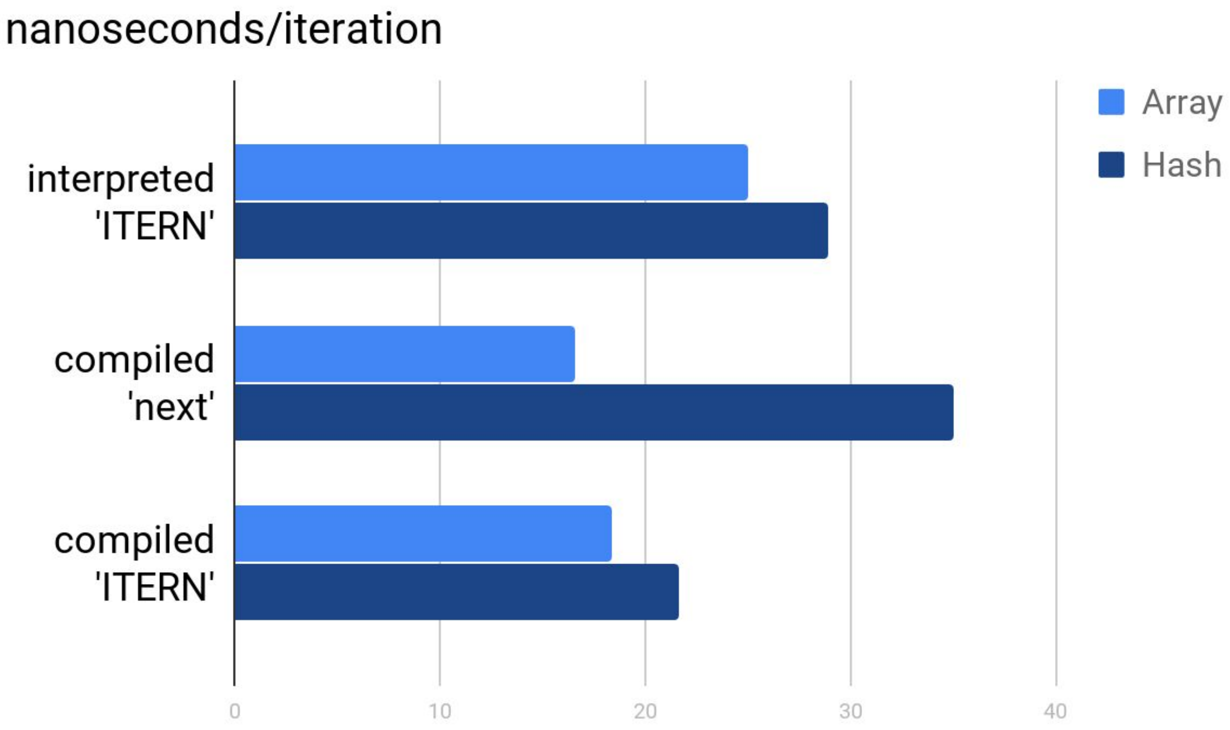
Benchmarks: empty loop



Benchmarks: small loop body

```lua
local r, sum, len = {}, 0, 0
for k, v in pairs(t) do
  len = len + 1
  r[len] = k
  sum = sum + v
end
```

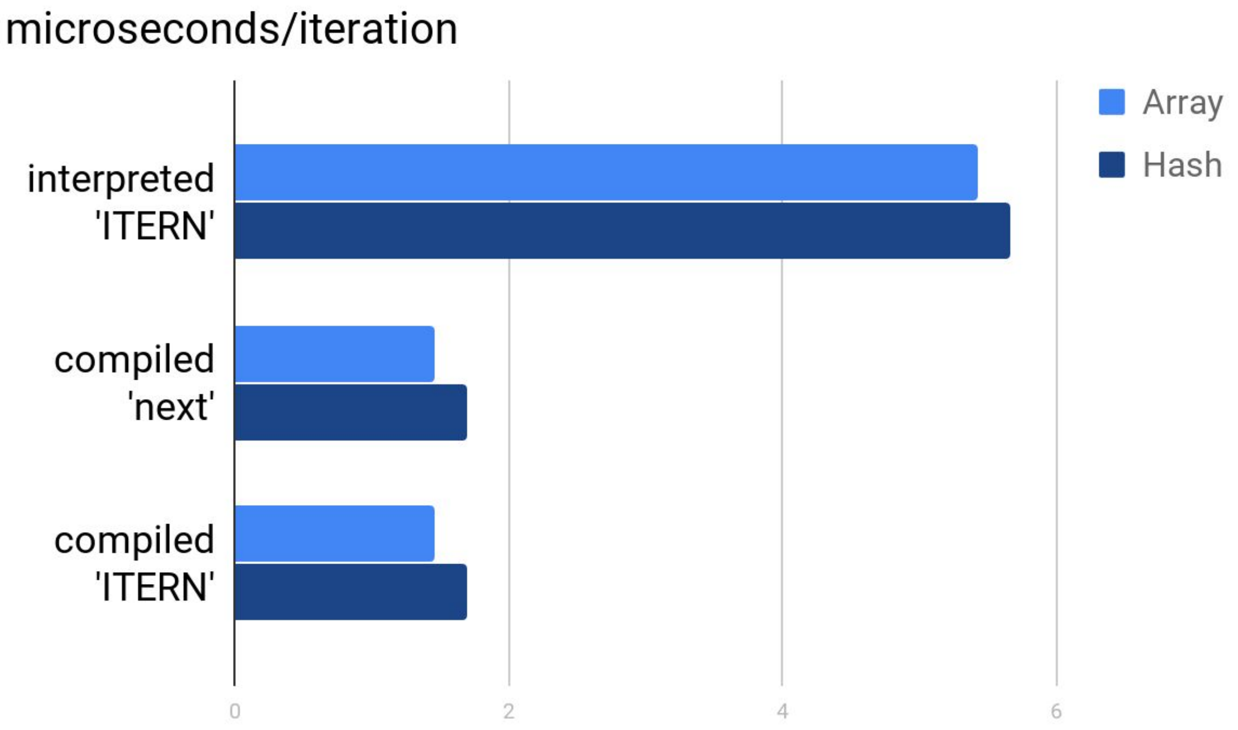
Benchmarks: small loop body



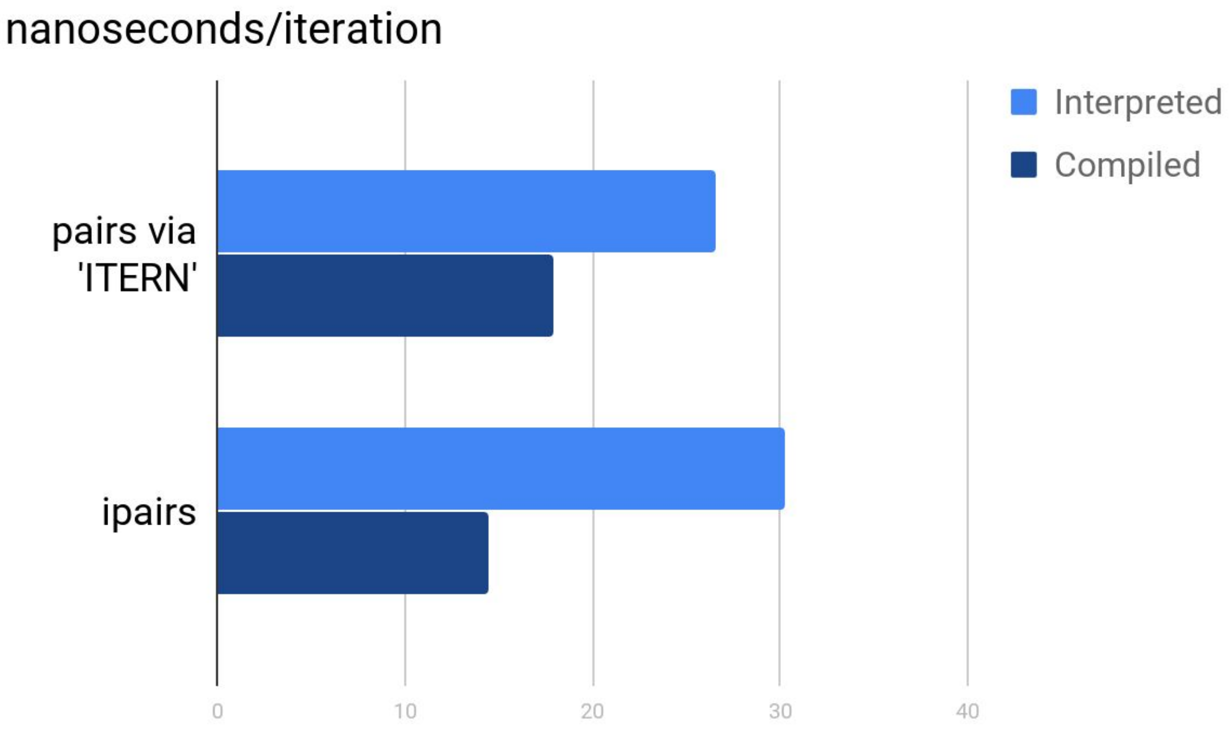
Benchmarks: big loop body (nested ‘pairs’)

- Checksum computation over 50 table elements

Benchmarks: big loop body (nested ‘pairs’)



Benchmarks: ‘pairs’ vs ‘ipairs’ for array



Conclusion

- Tuning JIT is a complex task

Conclusion

- Tuning JIT is a complex task
- But it makes LuaJIT / PUC-Lua split less dramatic and allows to have unified Lua programming best practices

Questions?

- bit.ly/hacking-luajit
- bit.ly/dumpanalyze
- wiki.luajit.org/Bytecode-2.0
- wiki.luajit.org/SSA-IR-2.0