Towards Ruby3x3 Performance

Introducing RTL and MJIT

Vladimir Makarov

Red Hat

September 21, 2017

About Myself

- Red Hat, Toronto office, Canada
- Tools group (GCC, Glibc, LLVM, Rust, Go, OpenMP)
  - part of a bigger platform enablement team (porting Linux kernel to new hardware)
- 20 years of work on GCC
- 2 years of work on MRI

Ruby 3 performance goal

- Matz set a very ambitious goal: MRI 3 should be 3x faster than MRI 2
  - Koichi Sasada improved MRI performance by about 3x
  - It is symbolic to expect MRI 3 should be 3x faster than MRI 2
- Doable for CPU intensive programs
- Hardly possible for memory or IO bound programs
- I treat Matz's performance goal as: MRI needs another cardinal performance improvement

RTL insns

- **IR for Ruby code analysis, optimizations, and JIT**
  - Importance of easy **data dependence** discovery
  - **Stack based insns** are an inconvenient IR for such goals
- Stack insns vs RTL insns for Ruby code `a = b + c`:

```
getlocal_OP__WC__0 <b index>
getlocal_OP__WC__0 <c index>
opt_plus
setlocal_OP__WC__0 <a index>
```

```
plus <a index>, <b index>, <c index>
```

Using RTL insns for interpretation

- RTL for analysis and JIT code generation
- RTL or stack insns for interpretation?

Feature	Stack insns	RTL insns
Insn length	**shorter**	longer
Insn number	more	**less**
Code length	**less**	more
Insn decoding	**less**	more
Code data locality	**more**	less
Insn dispatching	more	**less**
Memory traffic	more	**less**

Instructions: **Pros** & Cons for interpretation

- Decision: Use RTL for the interpreter too
  - Allows sharing code between the interpreter and JIT

How to generate RTL

- A **simpler** way is to generate RTL insns from the stack insns
- A **faster** approach is to generate directly from MRI parse tree nodes
- Decision: generate RTL **directly** from MRI nodes

RTL insn operands

- What could be an operand:
  - only temporaries
  - temporaries and locals
  - temporaries and locals even from higher levels (outside Ruby block)
  - the above + instance variables
  - the above + class variables, globals
- Decoding overhead of numerous type operands will not be compensated by processing smaller number of insns
- Complicated operands also complicate optimizations and JIT
- Currently **we use only temporaries and locals**. This gives best performance results according to my experiments

RTL complications

- Practically any RTL insn might be an ISEQ call. A call always puts a result on the stack top. We need to move this result to a destination operand:
  - If an RTL insn is actually a call, change the return PC so the next insn executed after the call will be an insn moving the result from the stack top to the insn destination
  - To decrease memory overhead, the move insn is a **part** of the original insn
  - For example, if the following insn

    `plus <move opcode>, <call data>, dst, op1, op2`

    is a method call, the next executed insn will be

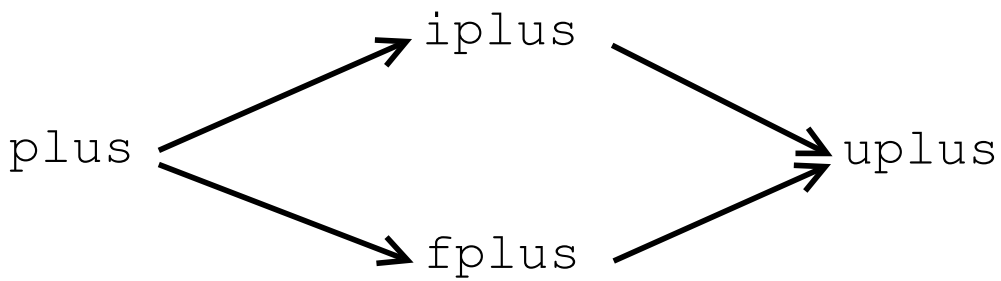
    `<move opcode> <call data>, dst, op1, op2`

RTL insn combining and specialization

- Immediate value specialization
  - e.g. `plus` $->$ `plusi` - addition with immediate fixnum as an operand
- Frequent insn sequence combining
  - e.g. `eq` + `bt` $->$ `bteq` - comparison and branch if the operands are equal

Speculative insn generation

- Some initially generated insns can be transformed into **speculative** ones **during their execution**
  - Speculation is based on **operand types** (e.g. plus can be transformed into an integer plus) and on the **operand values** (e.g. no multi-precision integers)
- Speculative insns can be transformed into **unchanging regular insns** if the speculation is wrong
  - Speculation insns include code checking the speculation correctness



- Speculation will be more important for JITted code performance
  - It creates a lot of big extended basic blocks which a C compiler optimizes well

RTL insn status and future work

- It mostly works (`make check` reports no regressions)
- Slightly better performance than stack based insns
  - 27% GeoMean improvement on 23 small benchmarks (+110% to -7%)
  - Code Change (Optcarrot):

	Stack insns → RTL insns
Executed insns number	**-23%**
Executed insn length	+19%

- Still some work to do for RTL improvement:
  - Reducing code size
  - Reducing overhead in operand decoding

Possible JIT approaches

1. Writing own JIT from scratch
   - LuaJIT, JavaScript V8, etc
2. Using widely used optimizing compilers
   - GCC, LLVM
3. Using existing JITs
   - JVM, OMR, RPython, Graal/Truffle, etc.

Option 1: Writing own JIT from scratch

- Full control, small size, fast compilation
- Fast compilation is mostly a result of fewer optimizations than in industrial optimizing compilers
- Still a huge effort to implement decent optimizations
- Ongoing burden in maintenance and porting

Option 2: Using widely used optimizing compilers

- Highly optimized code (GCC has > 300 optimization passes), easier implementation and porting and extremely well maintained ($> 2K$ contributors since GCC 2.95)
- Portable (currently supports 49 targets)
- Reliable and well tested ($> 16K$ reporters since GCC 2.95)
- No new dependencies
- **But** slower compilation
  - Slower mostly because it does much more than a typical JIT
  - Compilation can be made faster by disabling less valuable optimizations

Option 3: Using existing JITs

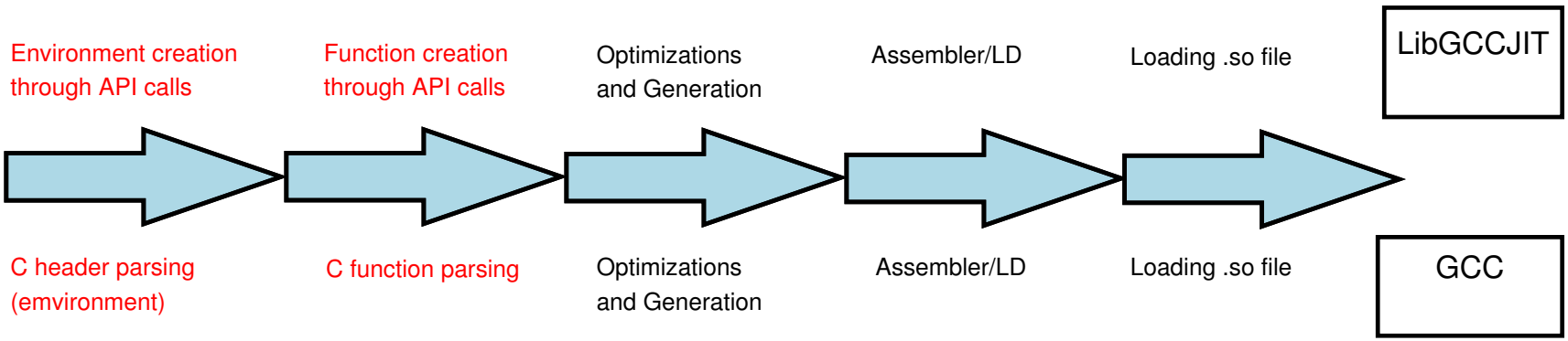
- Duplication: already used for JRuby, Topaz (Rpython), Opal (JS), OMR Ruby, Graal/Truffle Ruby
- JVM is stable, reliable, optimizing, and ubiquitous
- But still worse code performance than GCC/JIT
  - Azul Falcon (LLVM based JIT) up to 8x better performance than JVM C2 (source: http://stuff-gil-says.blogspot.ca/2017)
- License issues and patent minefield

Own or existing JITs vs GCC/LLVM based JITs

- Webkit moved from LLVM JIT to own JIT (source: https://webkit.org/blog/5852/introducing-the-b3-jit-compiler)
  - Implemented about 20 optimizations
  - 4-5 speedup in compilation time
  - Final results: Jetstream, Kraken, Octane (-9% to +8%)
- ISP RAS research: JS V8 ported to LLVM (source http://llvm.org/devmtg/2016-09/slides/Melnik-LLV8.pdf)
  - GeoMean speedup 8-16% on Sunspider
- Resulting situation: is the glass half full or half empty?
  - In my opinion, considering implementation and maintenance efforts, GCC/LLVM JIT is a winner, especially for long running server programs

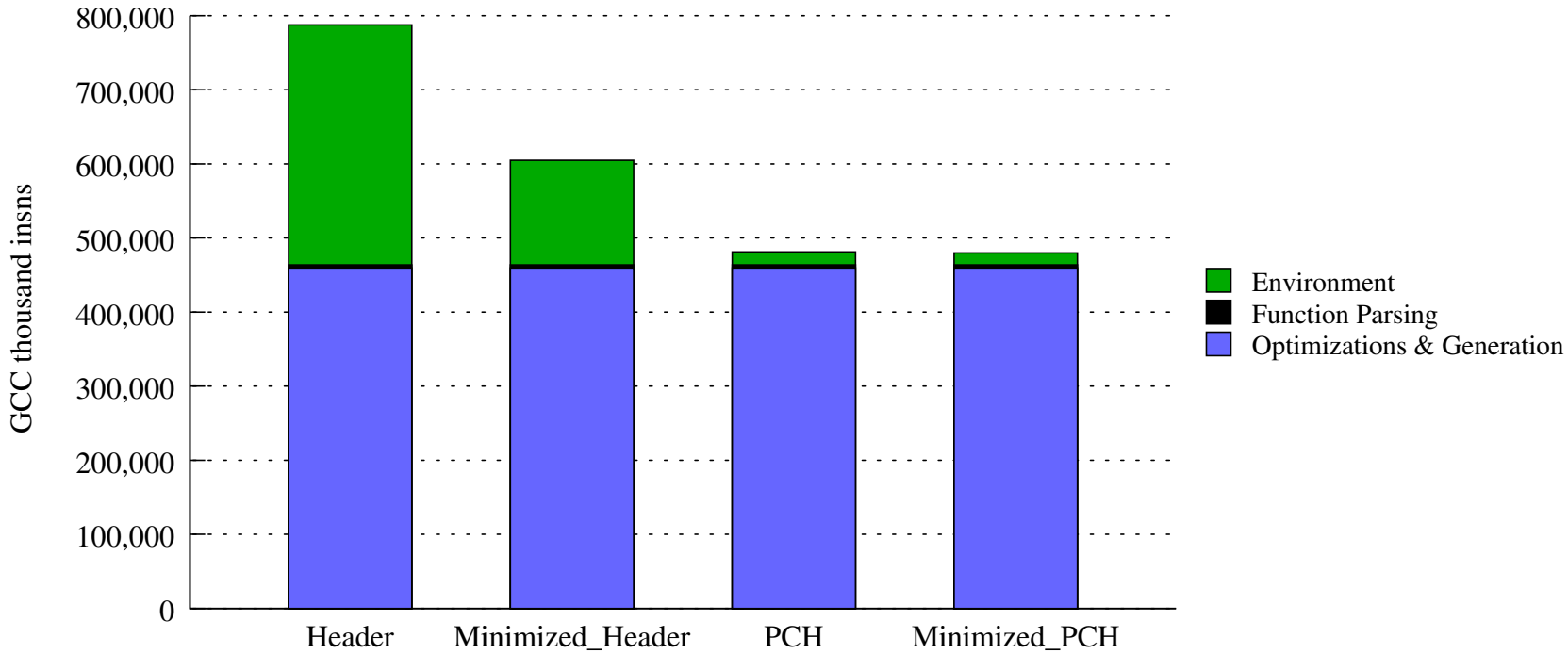
How to use GCC/LLVM for implementing JITs

- Using LibGCCJIT/MCJIT/ORC:
  - New, unstable interfaces
  - A lot of tedious calls to create the environment (see GNU Octave and PyPy port to libgccjit)
- Generating C code:
  - No dependency on a particular compiler, easier debugging
  - But some people call it a heavy, "junky" approach
    - ★ **Wrong!** if we implement it carefully
- LibGCCJIT vs GCC data flow (red parts are different):



How to use GCC/LLVM for implementing JITs – cont'd

- Generating C code
  - Environment takes from 21% to 41% of all compilation time
  - Using a precompiled header (PCH) decreases this to less than 3.5%
  - Function parsing takes less than 1%



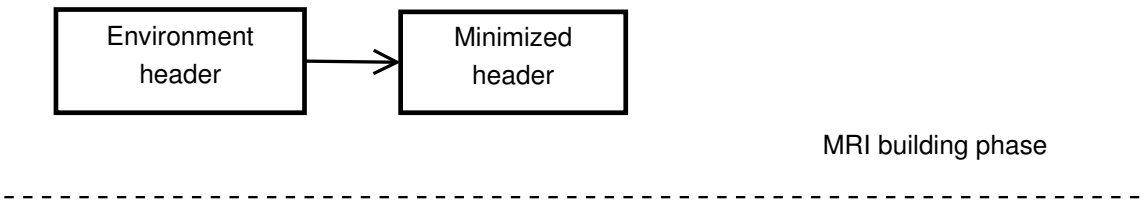
GCC –O2 processing a function with 44 RTL insns

  - GCC with C executable size: 25.1 MB for cc1 vs. 22.6MB for libgccjit (only 10% difference)

MJIT

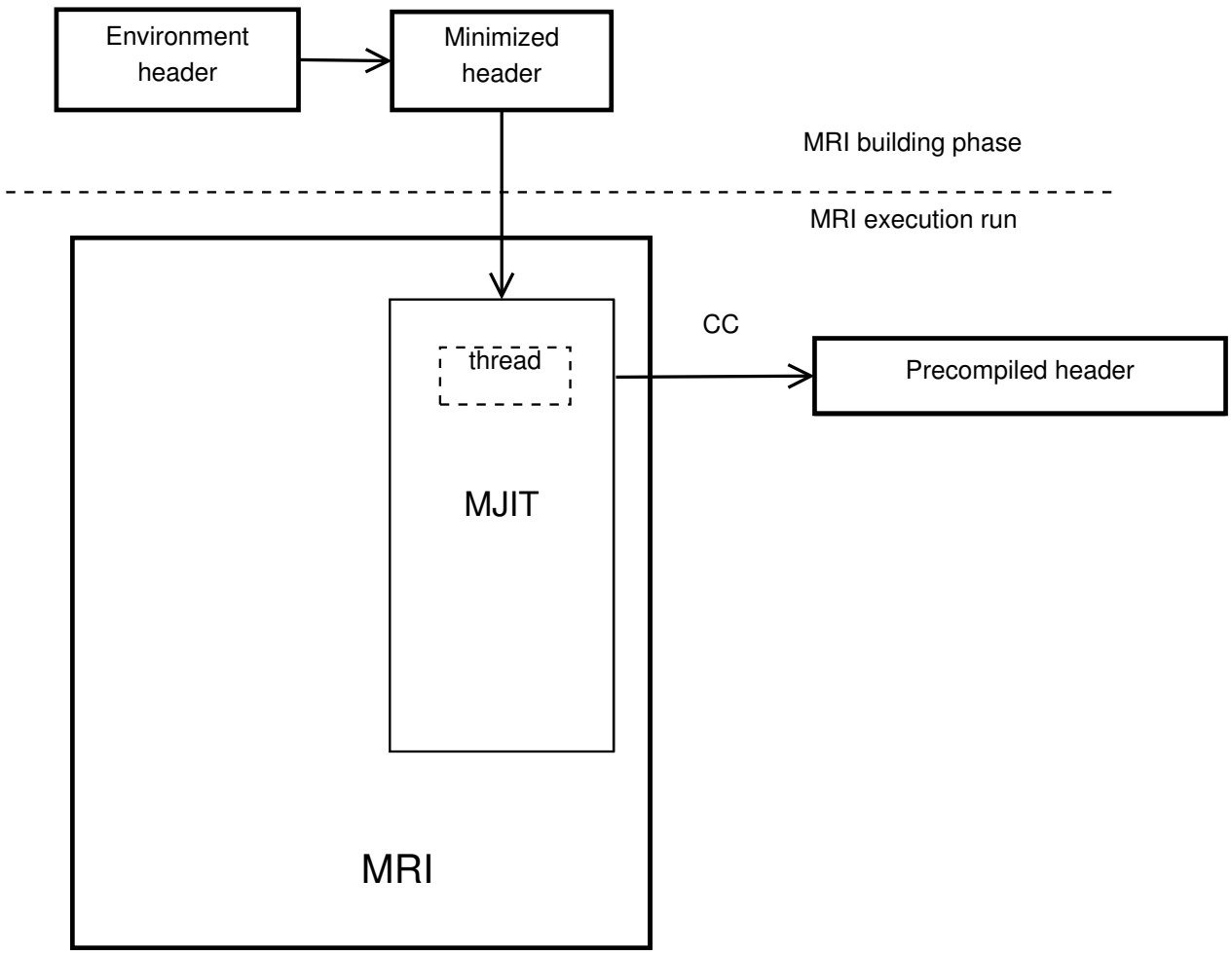
- MJIT is **M**RI JIT
- MJIT is **M**ethod JIT
- MJIT is a JIT based on C code generation and PCH
- MJIT can use GCC or LLVM, in the future other C compilers

MJIT architecture



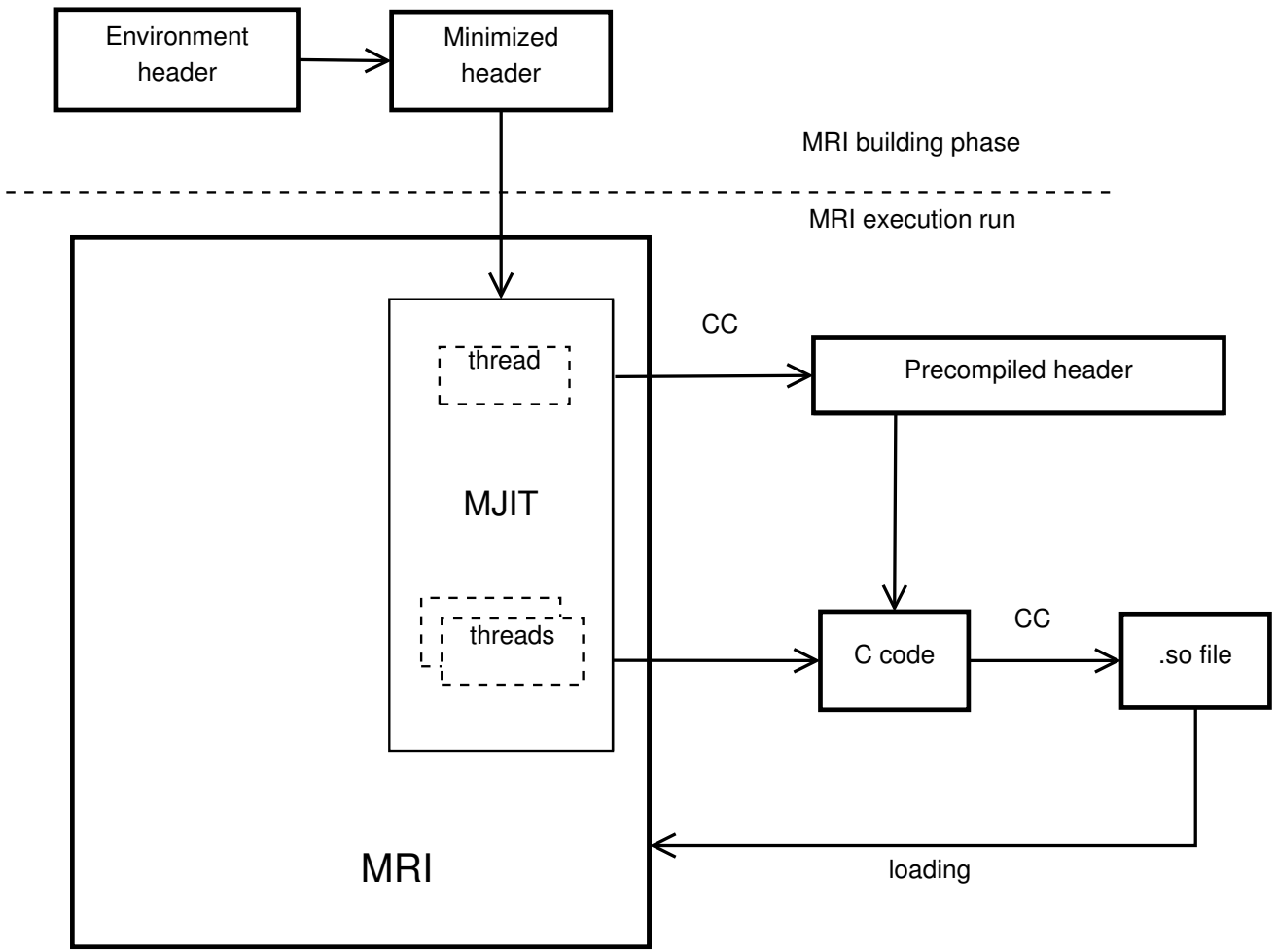
New MRI MJIT environment building step

MJIT architecture



MJIT initialized in parallel with Ruby program execution

MJIT architecture



MJIT works in parallel with Ruby program execution

Example

- Ruby code:

```
def loop
  i = 0; while i < 100_000; i += 1; end
  i
end
```

Example

- Ruby code:

```
def loop
  i = 0; while i < 100_000; i += 1; end
  i
end
```

- RTL code right after compilation:

```
...
0004 val2loc          3, 0
0007 goto             15
0009 plusi            cont_op2, <calldata...>, 3, 3, 1
0015 btlti            cont_btcmp, 9, <calldata...>, -1, 3, 100000
0022 loc_ret          3, 16
...
```

Example

- Ruby code:

```
def loop
  i = 0; while i < 100_000; i += 1; end
  i
end
```

- Speculative RTL code after some execution:

```
...
0004 val2loc          3, 0
0007 goto             15
0009 iplusi           _, _, 3, 3, 1
0015 ibtlti           _, 9, _, -1, 3, 100000
0022 loc_ret          3, 16
...
```

Example

- Ruby code:

```
def loop
  i = 0; while i < 100_000; i += 1; end
  i
end
```

- MJIT generated C code:

```
     ...
l4:  cfp->pc = (void *) 0x5576729ccd88; val2loc_f(cfp, &v0, 3, 0x1);
l7:  cfp->pc = (void *) 0x5576729ccd98; ruby_vm_check_ints(th); goto l15;
l9:  if (iplusi_f(cfp, &v0, 3, &v0, 3, &new_insn)) {
       vm_change_insn(cfp->iseq, (void *) 0x5576729ccda6, new_insn);
       goto stop_spec;
     }
l15: flag = ibtlti_f(cfp, &t0, -1, &v0, 200001, &val, &new_insn);
     if (val == RUBY_Qundef) {
       vm_change_insn(cfp->iseq, (void *) 0x5576729ccdd6, new_insn);
       goto stop_spec;
     }
     if (flag) goto l9;
l22: cfp->pc = (void *) 0x5576729cce26;
     loc_ret_f(th, cfp, &v0, 16, &val);
     return val;
     ...
```

Example

- Ruby code:

```
def loop
  i = 0; while i < 100_000; i += 1; end
  i
end
```

- GCC optimized x86-64 code:

```
...
movl  $200001, %eax
...
ret
```

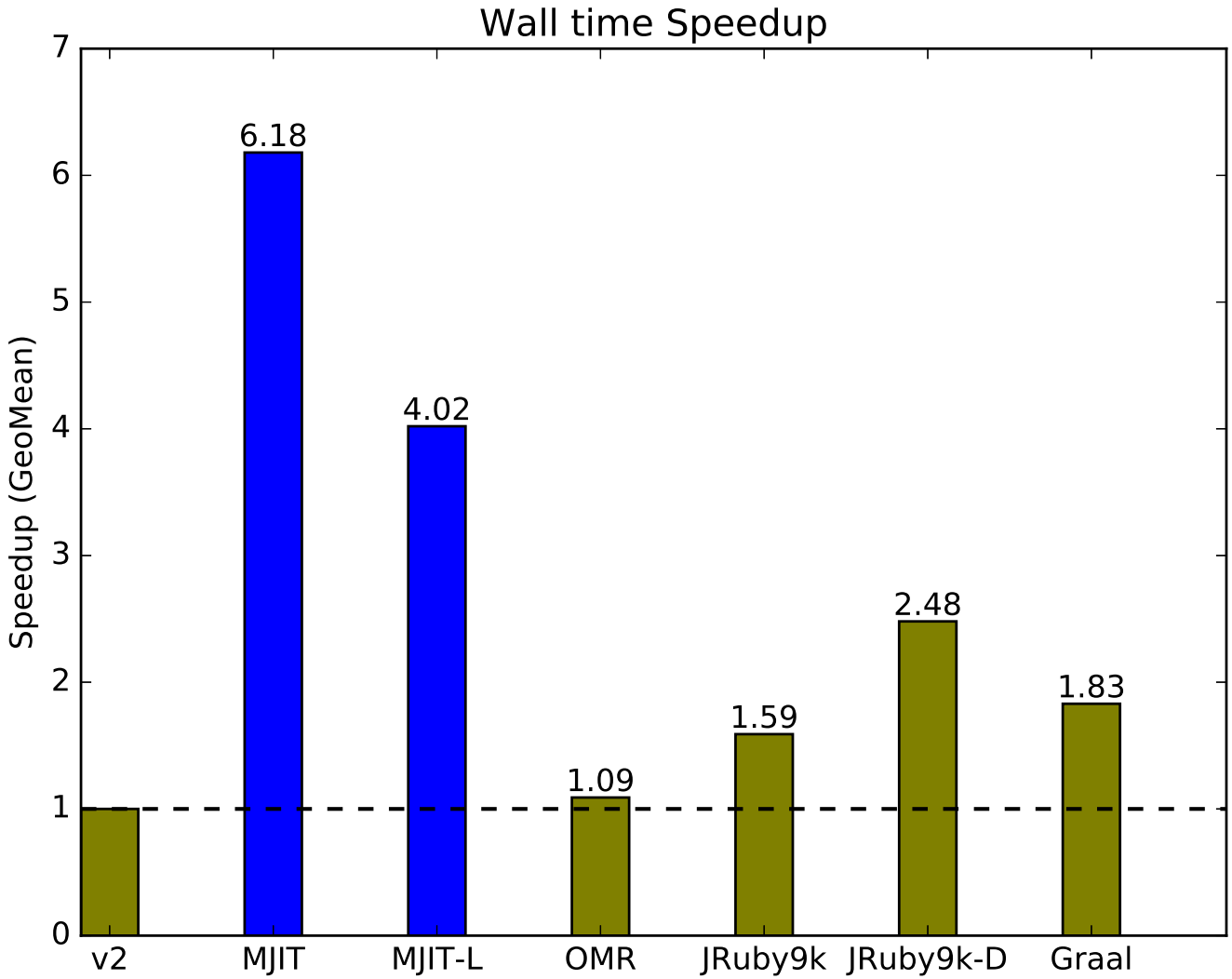
- There is no loop
- JVM can not do this

MJIT performance results

- Benchmarking MRI v2 (v2), MRI GCC MJIT (MJIT), MRI LLVM MJIT (MJIT-L), OMR Ruby rev. 57163 using JIT (OMR), JRuby9k 9.1.8 (JRuby9K), JRuby9k -Xdynamic (JRuby9k-D), Graal Ruby 0.22 (Graal)
- Mainstream CPU (i3-7100) under Fedora 25 with GCC-6.3 and Clang-3.9
- Microbenchmarks and small benchmarks (dir MJIT-benchmarks)
  - Each benchmark runs at least 20-30sec on MRI v2
- Optcarrot (https://github.com/mame/optcarrot)

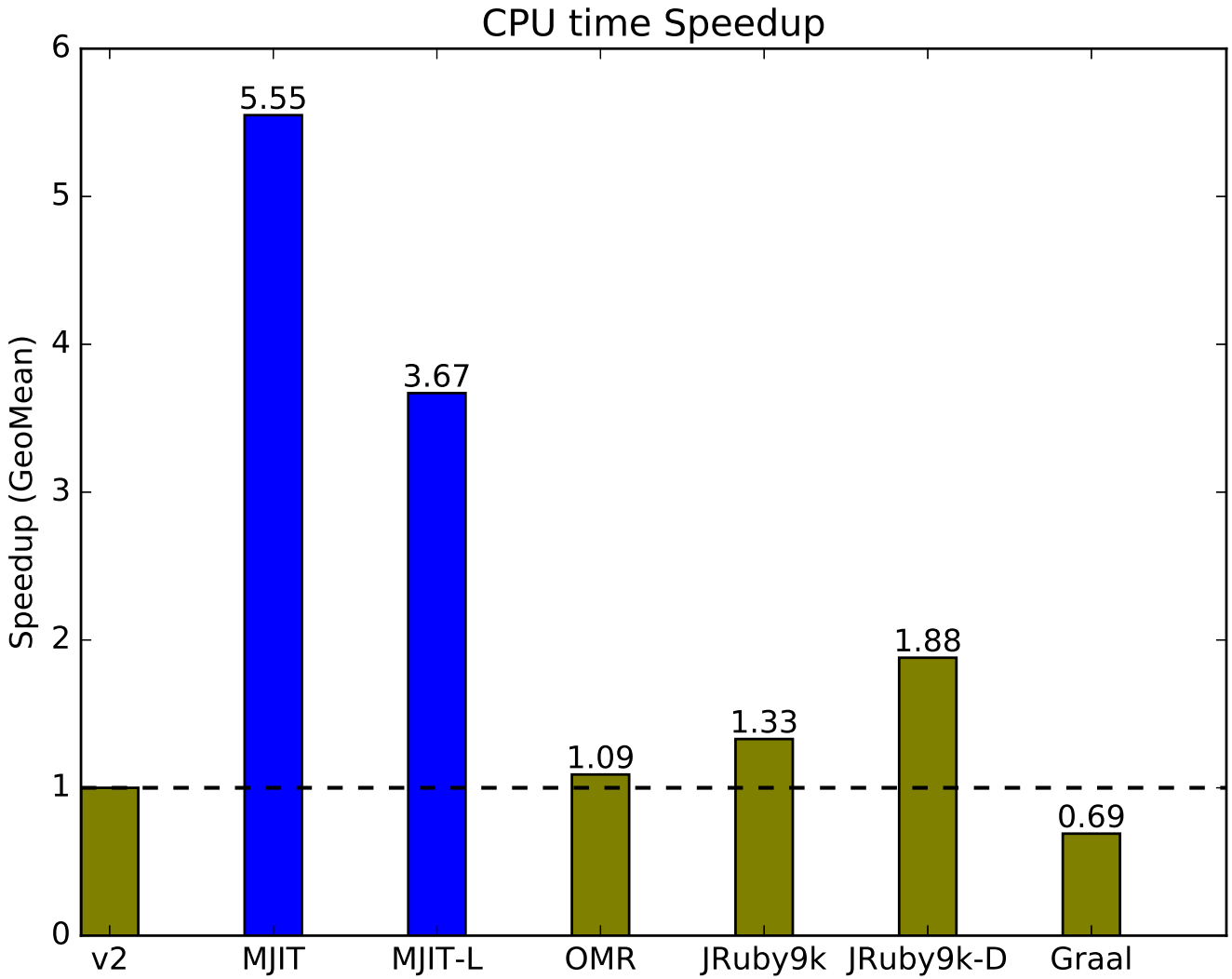
MJIT performance results

- **Microbenchmarks**: Geomean **Wall** time improvement relative to MRI v2



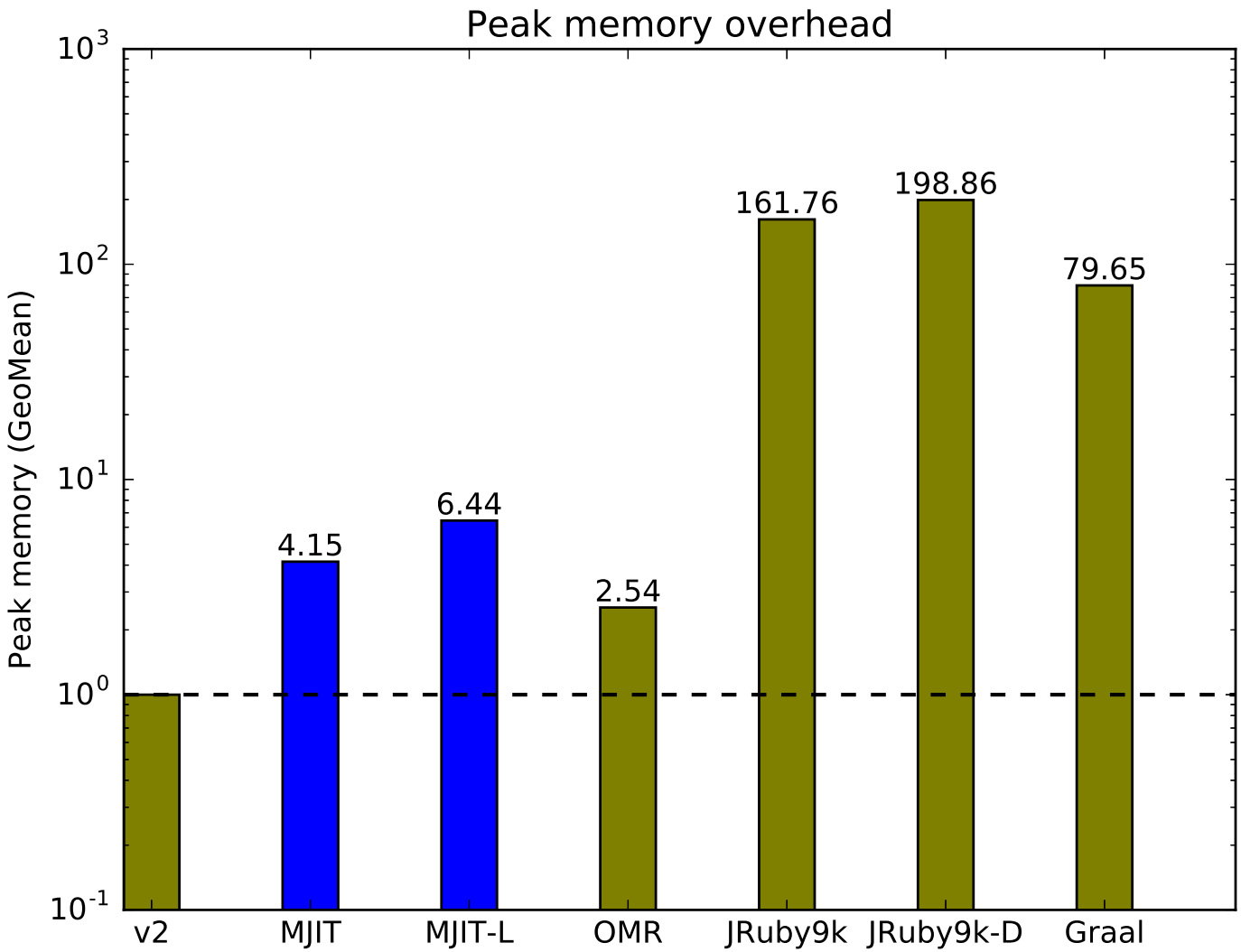
MJIT performance results

- **Microbenchmarks**: Geomean **CPU** time improvement relative to MRI v2



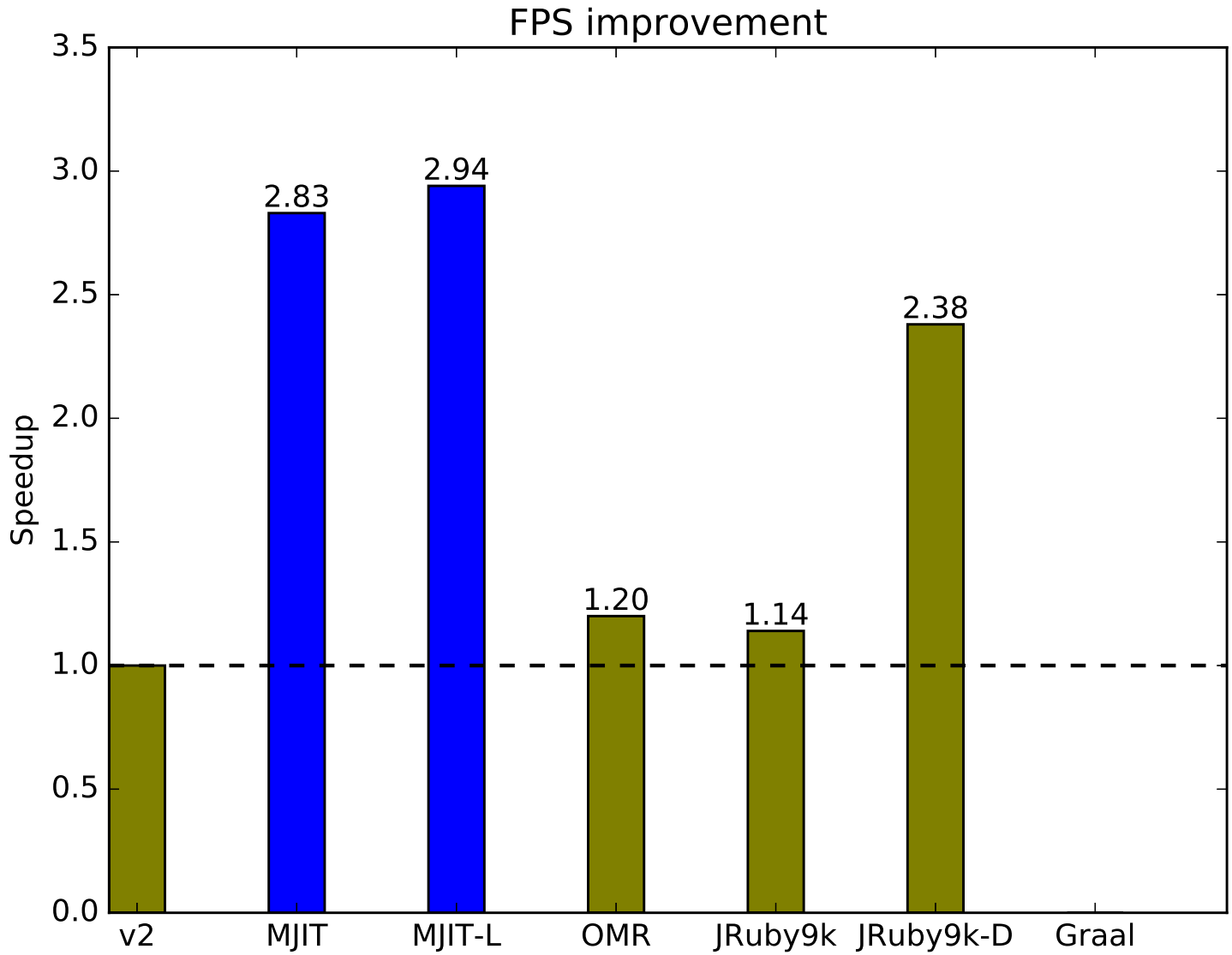
MJIT performance results

- **Microbenchmarks**: Geomean **Peak memory** overhead relative to MRI v2



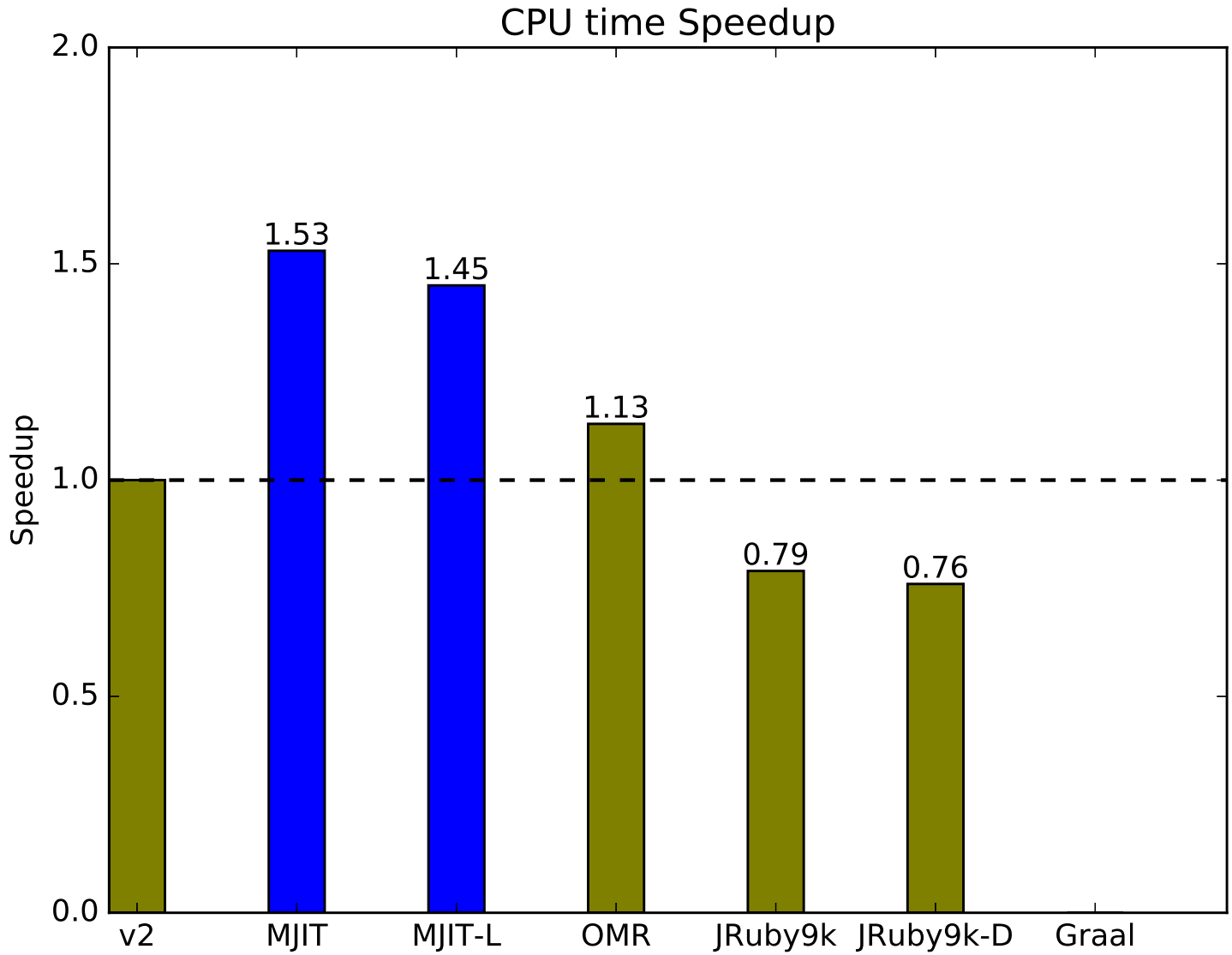
MJIT performance results

- **Optcarrot**: **FPS** speedup relative to MRI v2



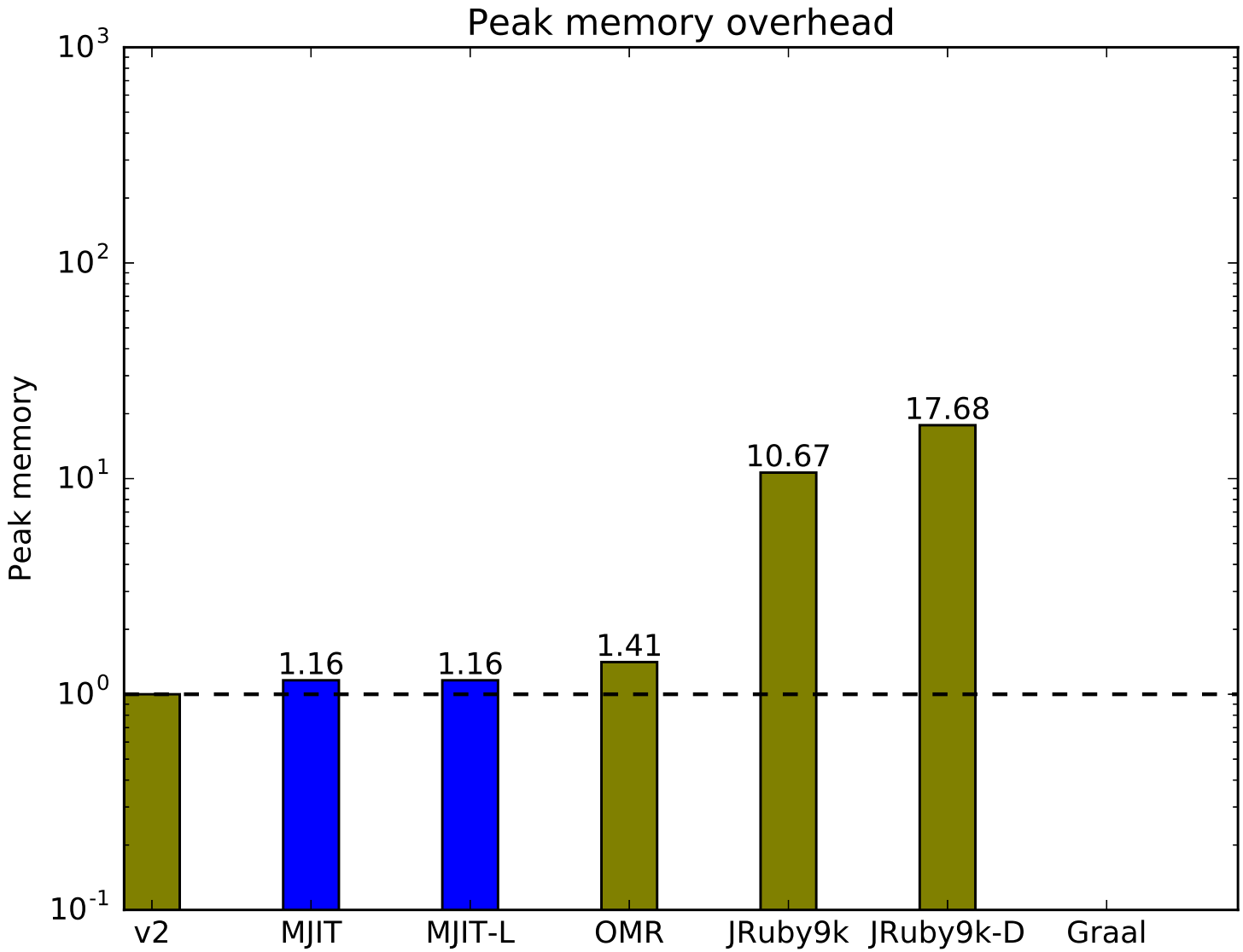
MJIT performance results

- **Optcarrot**: **CPU** time improvement relative to MRI v2



MJIT performance results

- **Optcarrot**: **Peak memory** overhead relative to MRI v2



Recommendations to use GCC/LLVM for a JIT

- My recommendations in order of importance:
  - Don't use MCJIT, ORC, or LIBGCCJIT
  - Use a pre-compiled header (JIT code environment) in a memory FS
  - Compile code in parallel with program interpretation
  - Use a good strategy to choose byte code for JITting
  - Minimize the environment if you don't use PCH

MJIT status and future directions

- The project is at an early development stage:
  - Unstable, passes '`make test`', can not pass '`make check`' yet
  - Doesn't work on Windows
  - At least one more year to mature
- Need more optimizations:
  - No inlining yet. The most important optimization!
  - Different approaches to implement inlining:
    - ★ Node or RTL level
    - ★ Use C inlining (I'll pursue this one)
    - ★ New GCC/LLVM extension (a new inline attribute) would be useful
- Will RTL and MJIT be a part of MRI?
  - It does not depend on me
  - I am going to work in this direction
  - Will be happy if even some project ideas will be used in future MRI