The Light Weight JIT Compiler Project

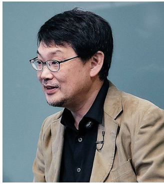
Vladimir Makarov

RedHat

Linux Plumbers Conference, Aug 24, 2020

Some context

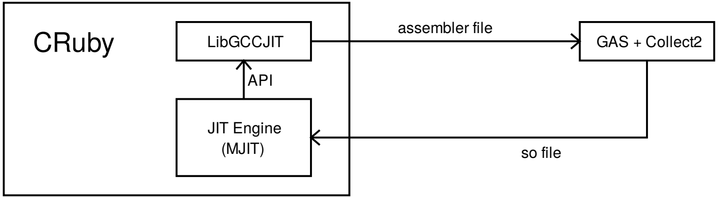
- CRuby is a major Ruby implementation written on C
- Goals for CRuby 3.0 set up by Yukihiro Matsumoto (Matz) in 2015
  - 3 times faster in comparison with CRuby 2.0
  - Parallelism support
  - Type checking
- IMHO, successful fulfilling these goals could prevent GO eating Ruby market share
- CRuby VM since version 2.0 has a very fine tuned interpreter written by Koichi Sasada
  - 3 times faster Ruby code execution can be achieved only by JIT



Ruby JITs

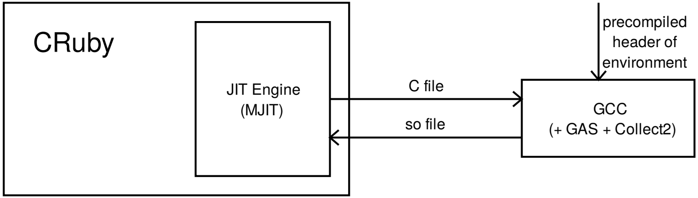
- A lot of Ruby implementations with JIT
- Serious candidates for CRuby JIT were
  - Graal Ruby (Oracle)
  - OMR Ruby (IBM)
  - JRuby (major developers are now at RedHat)
- I've decided to try GCC for CRuby JIT which I called MJIT
  - MJIT simply means a **M**ethod **JIT**

Possible Ruby JIT with LibGCCJIT



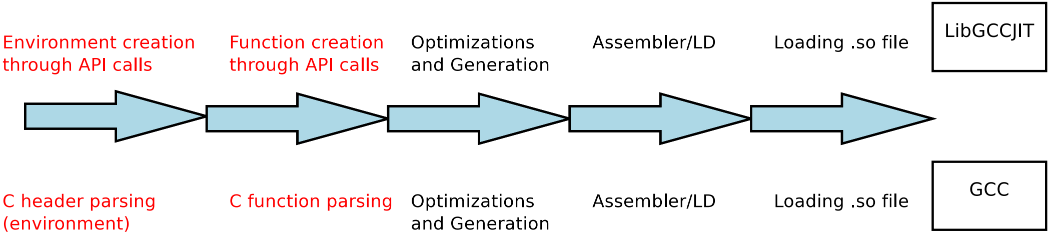
- David Malcolm's LibGCCJIT is a big step forward to use GCC for JIT compilers
- But using LibGCCJIT for CRuby JIT would
  - Prevent inlining
    - Inlining is important for effective using **environment** (couple thousand lines of inlined C functions used for CRuby bytecode implementation)
  - Make creation of the environment through LibGCCJIT API is a tedious work and a nightmare for maintenance

Actual CRuby JIT approach with GCC



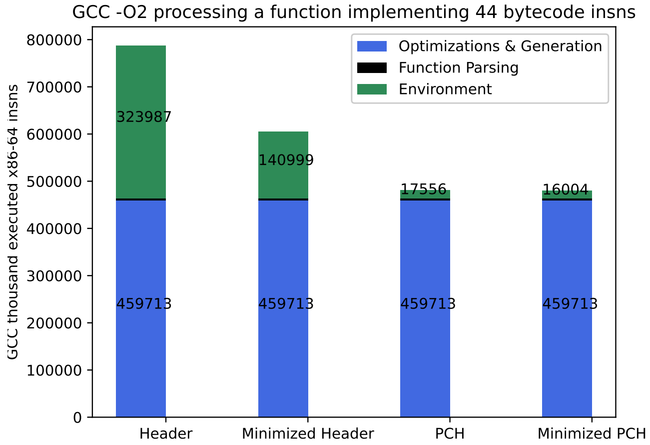
- C as an interface language
  - Stable interface
  - Simpler implementation, maintenance and debugging
  - Possibility to use Clang instead of GCC
- Faster compilation speed achieved by
  - Precompiled header usage
  - Memory FS (/tmp is usually a memory FS)
  - Ruby methods are compiled in parallel with their execution

LibGCCJIT vs GCC data flow



- Red parts are different in LIBGCCJIT and GCC data flow
- How to make GCC red part run time minimal?

Header processing time

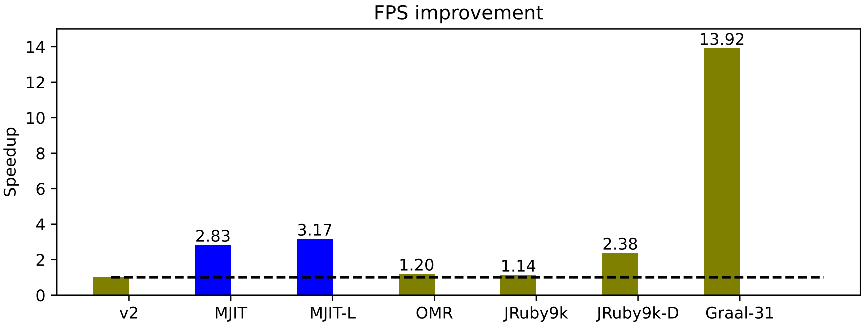


- Processing C code for 44 bytecode insns and the environment

Performance Results – Test

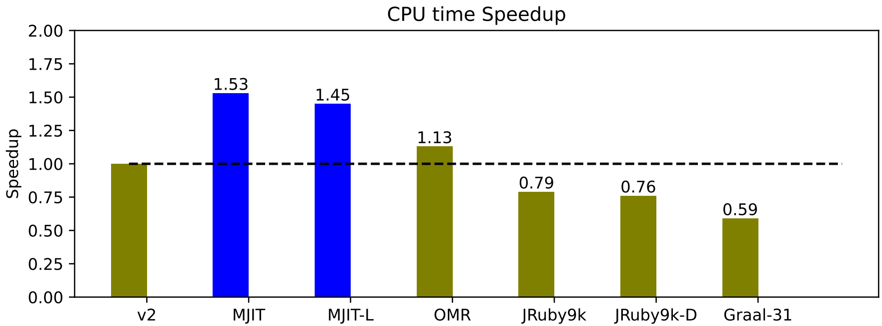
- Intel 3.9GHz i3-7100 with 32GB memory under x86-64 FC25
- CPU-bound test OptCarrot v2.0 (NES emulator), first 2000 frames
- Tested Ruby implementations:
  - CRuby v2.0 (v2)
  - CRuby v2.5 + GCC JIT (mjit)
  - CRuby v2.5 + Clang/LLVM JIT (mjit-l)
  - OMR Ruby rev. 57163 (omr) in JIT mode
  - JRuby v9.1.8 (jruby9k)
  - jruby9k with invokedynamic=true (jruby9k-d)
  - Graal Ruby v0.31 (graal31)

Performance Results – OptCarrot (Frames per Sec)



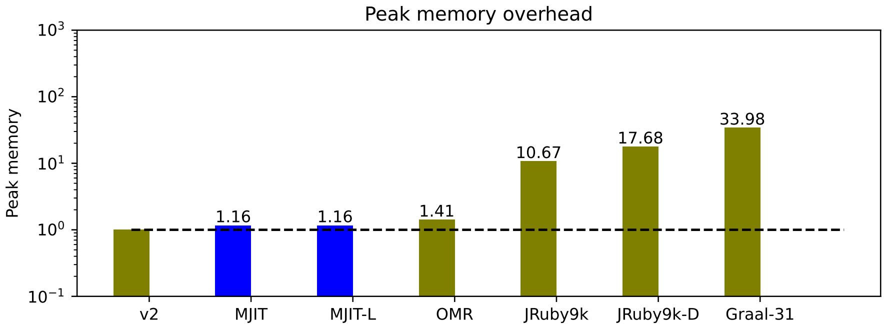
- Graal performance is the best because of very aggressive **speculation/deoptimization** and inlining Ruby **standard** methods
- Performance of CRuby with GCC or Clang JIT is 3 times better than CRuby v2.0 one and second the best

Performance Results – CPU time



- CPU time is important too for cloud (money) or mobile (battery)
- Only CRuby with GCC/Clang JIT and OMR Ruby spend less CPU resources (and energy) than CRuby v2.0
- Graal Ruby is the worst because of numerous compilations of speculated/deoptimized code on other CPU cores

Performance Results – Memory Usage



- GCC/Clang compiler peak memory is also taken into account for CRuby with GCC/Clang JIT

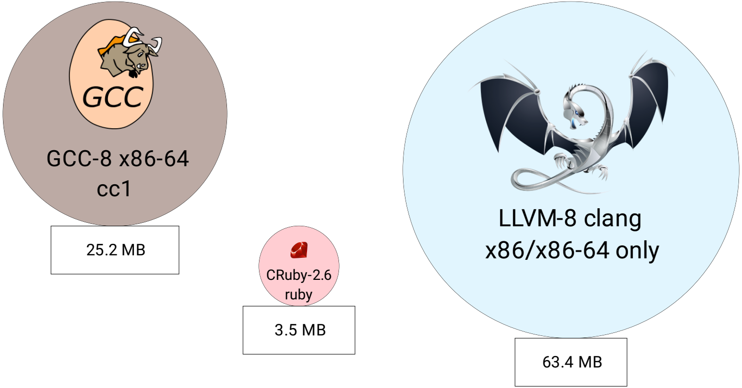
Official CRuby MJIT

- The MJIT was adopted and modified by Takashi Kokubun and became official CRuby JIT since version 2.6
- Major differences:
  - Using existing stack based VM insns instead of new RTL ones
  - No speculation/deoptimization
  - Much less aggressive JIT compilation thresholds
  - JITted code compaction into one shared object
    - Solving under-utilization of page space (usually 4KB) for one method generated code (typically 100-400 bytes) and decreasing TLB misses
  - Optcarrot performance is worse for official MJIT

GCC/LLVM based JIT disadvantages

- Big comparing to CRuby
- Slow compilation speed for some cases
- Difficult for optimizing on borders of code written on different programming languages
- Some people are uncomfortable to have GAS (for LibGCCJIT) or GCC in their production environment
- TLB misses for a lot of small objects generated with LibGCCJIT or GCC
    - Under-utilization of page space by dynamic loader for typical shared object

CRuby/GCC/LLVM Binary Size

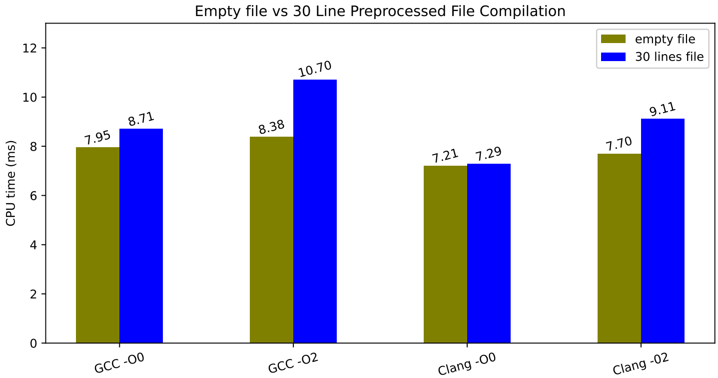


- Scaled to the corresponding binary sizes
- GCC and LLVM binaries are ~7-18 times bigger

GCC/LLVM Compilation Speed

- ~20ms for a small method compilation by GCC/LLVM (and MJIT) on modern Intel CPUs
- ~0.5s for Raspberry PI 3 B+ on ARM64 Linux
  - SPEC2000 Est 176.gcc: 320 (PI 3 B+) vs 8520 (i7-9700K)
- Slow environments for GCC/LibGCCJIT based JITs
  - MingW, CygWin, environments w/o memory FS
- Example of JIT compilation speed difference: Java implementation by Azul Systems (LLVM 2017 conference keynote)
  - **100ms** for a typical Java method compiled with aggressive inlining by Falcon, a tier 2 JIT compiler implemented with LLVM
  - **1ms** for the method compiled by a tier 1 JIT compiler

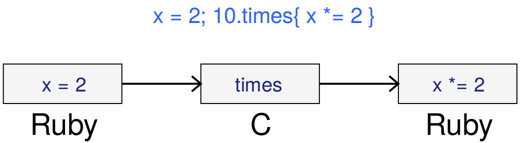
GCC/LLVM startup

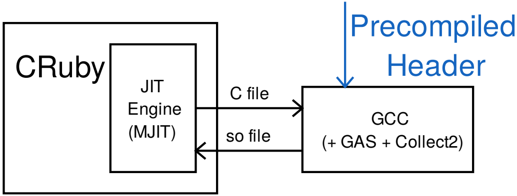


- x86_64 GCC-8/LLVM-8, Intel i7-9700K, FC29
- Most time is spent in compiler (and assembler!) data initialization
  - Builtins descriptions, different optimization data, etc

Inlining C and Ruby code in MJIT

- Inlining is the most important JIT optimization
- Many Ruby standard methods are written on C
- Adding C code of Ruby standard methods to the precompiled header
  - Slower startup, slower compilation





Some conclusions about GCC and LLVM JITs

- GCC/LLVM based JITs **can not be a good tier 1 JIT compiler**
- GCC/LLVM based JITs **can be an excellent tier 2 JIT compiler**
- LibGCCJIT needs **embedded assembler and loader** analogous what LLVM (MCJIT) has
- LibGCCJIT needs **readable streamable input language**, not only API
- GCC/LLVM based JITs need **higher input language**
- GCC/LLVM based JITs need **speculation support**

Light-Weight JIT Compiler

- One possible solution is a light-weight JIT compiler in addition to existing MJIT one:
  - The light-weight JIT compiler as a tier 1 JIT compiler
  - Existing MJIT generating C as a tier 2 JIT compiler for more frequently running code
- Or only the light-weight JIT compiler for environments where the current MJIT compiler does not work
- It could be a good solution for **MRuby JIT**
  - It could help to expand Ruby usage from mostly server market to mobile and IOT market

MIR for Light-Weight JIT compiler

- My initially spare-time project:
  - **Universal** light-weight JIT compiler based on MIR
- **MIR** is **M**edium **I**nternal **R**epresentation
  - MIR means peace and world in Russian
  - MIR is strongly typed
  - MIR can represent machine insns of different architectures
- Plans to try the light-weight JIT compiler first for CRuby or/and MRuby

Example: C Prime Sieve

```
#define Size 819000
int sieve (int iter) {
  int i, k, prime, count, n; char flags[Size];
  for (n = 0; n < iter; n++) {
    count = 0;
    for (i = 0; i < Size; i++)
      flags[i] = 1;
    for (i = 2; i < Size; i++)
      if (flags[i]) {
        prime = i + 1;
        for (k = i + prime; k < Size; k += prime)
          flags[k] = 0;
        count++;
      }
  }
  return count;
}
```

Example: MIR Prime Sieve

```
m_sieve: module
         export sieve
sieve:   func i32, i32:iter
         local i64:flags, i64:count, i64:prime, i64:n, i64:i, i64:k
         alloca flags, 819000
         mov flags, fp;   mov n, 0
loop:    bge fin, n, iter
         mov count, 0;   mov i, 0
loop2:   mov ui8:(flags, i), 1;   add i, i, 1;    blt loop2, i, 819000
         mov i, 2
loop3:   beq cont3, ui8:(flags,i), 0
         add prime, i, 1;   add k, i, prime
loop4:   bgt fin4, k, 819000
         mov ui8:(flags, k), 0;   add k, k, prime;   jmp loop4
fin4:    add count, count, 1
cont3:   add i, i, 1;   blt loop3, i, 819000
         add n, n, 1;   jmp loop
fin:     ret count
         endfunc
         endmodule
```

The Light-Weight JIT Compiler Goals

- Comparing to GCC -O2
  - 70% of generated code speed
  - 100 times faster compilation speed
  - 100 times faster start-up
  - 100 times smaller code size
- Less 10K C LOC
- No external dependencies – only standard C (no LIBFFI, YACC, LEX, etc)

How to achieve the performance goals?

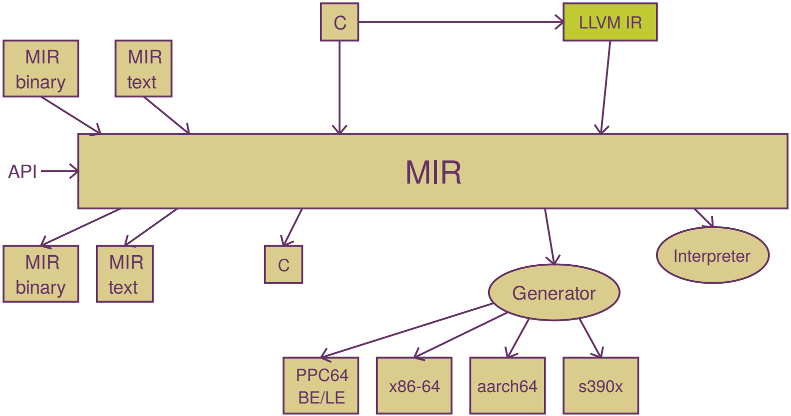
- Use few most valuable optimizations
- Optimize only frequent cases
- Use algorithms with the best combination of simplicity (code size) and performance

How to achieve the performance goals?

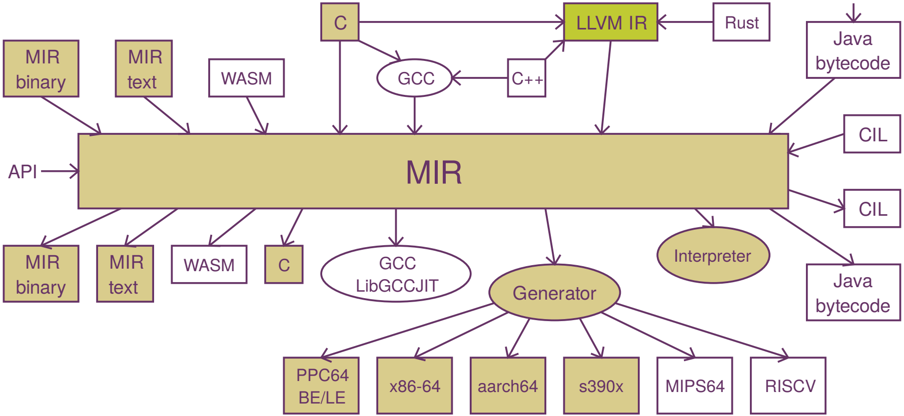
- What are the most valuable GCC optimizations for x86-64?
  - A decent RA
  - Code selection
- GCC-9.0, i7-9700K under FC29

SPECInt2000 Est.	GCC -O2	GCC -O0 + simple RA + combiner
-fno-inline	5458	4342 **(80%)**
-finline	6141	4339 **(71%)**

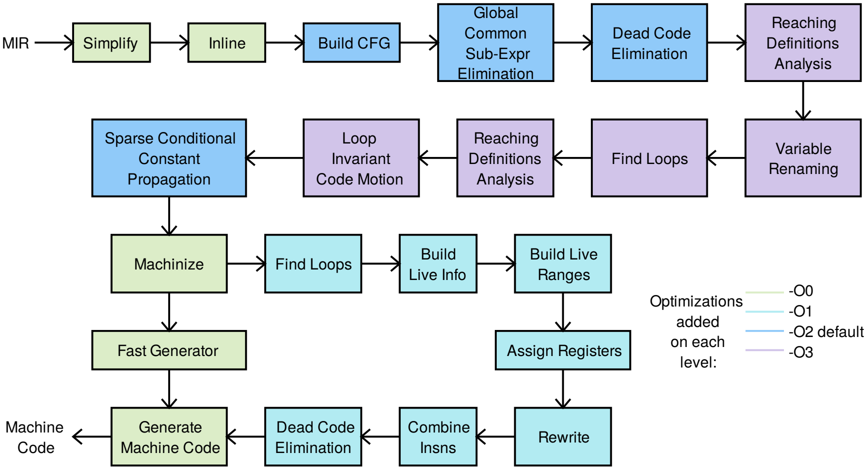
The current state of MIR project



Possible future directions of MIR project



MIR Generator



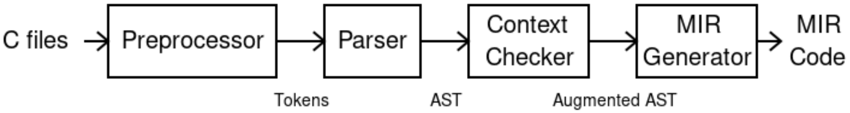
Some MIR Generator Features

- No Static Single Assignment Form
  - In and Out SSA passes are expensive, especially for short initial MIR-generator pass pipeline
  - SSA absence complicates conditional constant propagation and global common sub-expression elimination
  - Plans to use conventional SSA for optimizations before register allocator
- No Position Independent Code
  - It speeds up the generated code a bit
  - It simplifies the code generation

Possible ways to compile C to MIR

- LLVM IR to MIR or GCC Port
  - Dependence to a particular external project
  - Big efforts to implement
  - Maintenance burden
- Own C compiler
  - Practically the same efforts to implement
    - Examples: tiny CC, 8cc, 9cc
  - No dependency to any external project
- Considering GCC MIR port and MIR as input to LIBGCCJIT

C to MIR compiler



- C11 standard w/o standard optional variable arrays, complex, and atomics
- No any tools, like YACC or LEX
  - PEG (parsing expression grammar) parser
- Can be used as a library and from a command line
- Passing about 1K C tests and successfully bootstrapped
- Not call ABI compatible yet

Current MIR Performance Results

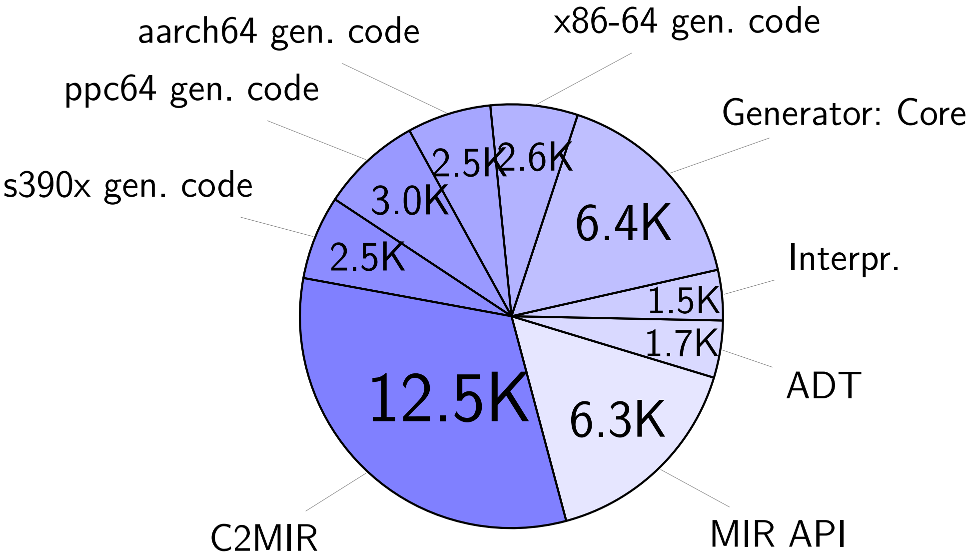
- Intel i7-9700K under FC32 with GCC-8.2.1:

	MIR-gen	MIR-interp	gcc -O2	gcc -O0
compilation[1]	**1.0** (51us)	0.35 (18us)	**393** (20ms)	294 (15ms)
execution[1]	**1.0** (2.78s)	6.7 (18.6s)	**0.95** (2.64s)	2.18 (6.05s)
code size[2]	**1.0** (320KB)	0.54 (173KB)	**80** (25.6MB)	80 (25.6MB)
startup[3]	**1.0** (10us)	0.5 (5us)	**1200** (12ms)	1000 (10ms)
LOC[4]	**1.0** (17K)	0.70 (12K)	**87** (1480K)	87 (1480K)

Table: Sieve[5]: MIR vs GCC

[1] Best wall time of 10 runs (MIR-generator with -O1)
[2] Stripped size of cc1 and minimal program running MIR code
[3] Wall time to generate code for empty C file or empty MIR function
[4] Size of minimal files to create and run MIR code or build x86-64 GCC compiler
[5] 28 lines of preprocessed C code, MIR is created through API

Current MIR SLOC distribution



MIR Project Competitors

- LibJIT started as a part of DotGNU Project
  - 80K SLOC, GPL/LGPL License
  - Only register allocation and primitive copy propagation
- RyuJIT, a part of runtime for .NET Core
  - 360K SLOC, MIT License
  - MIR-generator optimizations plus loop invariant motion minus SCCP
  - SSA
- Other candidates:
  - QBE: standalone+, small+ (10K LOC), SSA, ASM generation-, MIT License
  - LIBFirm: less standalone-, big- (140K LOC), SSA, ASM generation-, LGPL2
  - CraneLift: less standalone-, big- (70K LOC of Rust-), SSA, Apache License

MIR Project Plans

- First release at the end of this year
- Short term plans:
  - Prototype of MIR based JIT compiler in MRuby
  - Make C to MIR compiler call ABI compatible
  - Speculation support on MIR and C level
  - Porting MIR to MIPS64 and RISCV
- https://github.com/vnmakarov/mir