Maxime Chevalier-Boisvert

maximecb · she/her

YJIT

Compiler	Years active	Base VM	Stage	General approach	Frontend	Interpreter	Intermediate representations	Key authors
Hokstad	2008-present	Custom Ruby	AOT	Template compilation of an AST	Custom recursive descent and operator precedence parser	None	Enhanced AST	Hokstad
Hyperdrive	2019-2020	MRI	JIT	Tracing of YARV instructions then template compilation to Cranelift	Tracing YARV interpreter	Instrumented base interpreter	None	Matthews
IronRuby	2007-2011	Custom C#	JIT	Generation of CIL				Lam
JRuby	2006-present	Custom Java	JIT[1]	Generation of JVM bytecode	Parser to AST, to internal IR	Internal IR interpreter	CFG of linear RTL instructions	Nutter, Enebo, Sastry
LLRB	2017	MRI	JIT	Generation of LLVM IR				Kokubun
Ludicrous	2008-2009	MRI	JIT	Template compilation through DotGNU LibJIT				Brannan
MacRuby	2008-2013	MRI	AOT/ JIT	Generation of LLVM IR				Sansonetti
MagLev	2008-2016	Custom Gemstone Smalltalk	JIT					McLain, Felgentreff
MRuby JIT								Hideki
Natalie	2019-present	Custom C++	AOT	AST incrementally lowered to C++			Enhanced AST	Morgan
Ruby+OMR	2016-2017	MRI	JIT	Generation of J9 IR				Gaudet, Stoodley
RTL MJIT	2017	MRI	JIT	Generation of C				Makarov
Rubinius	2008-2016	Custom C++ and Ruby	JIT	Generation of LLVM IR	Parser to AST, to custom stack bytecode	Stack bytecode	None	Phoenix, Bussink, Shirai
RuJIT	2015	MRI	JIT	Tracing				Ide
Rhizome	2017	MRI, JRuby, Rubinius	JIT	Conventional speculative compiler with in-process assembler	Base bytecode or IR to custom bytecode	Stack bytecode	Graphical sea-of-nodes	Seaton
RubyComp	2004		AOT					Alexandersson
RubyX	2014-2020		AOT	Conventional compiler with in-process assembler	Parser to AST	None	Multiple IRs gradually removing abstraction and lowering from AST to linear	Rüger
Ruby.NET	2008	Custom C#		Generation of CIL				Kelly
Rucy	2021		AOT	Template compilation to eBPF				Uchio
Sorbet	2019-present	MRI	AOT	Generation of MRI LLVM IR 'C' extension	Parser to AST	None	Sorbet's typechecking IR	Tarjan, Petrashko, Froyd
TenderJIT	2021	MRI	JIT	Lazy Basic Block Versioning with in-process assembler	Template compiler of YARV bytecode	Base interpreter	None	Patterson
Topaz	2012-2014	Custom RPython and Ruby	JIT	Metatracing of a stack bytecode interpreter	Parser to AST	Stack bytecode interpreter		Gaynor, Felgentreff
TruffleRuby	2013-present	Custom Java and Ruby	JIT	Partial evaluation of self-specialising AST	Parser to AST	Self-specialising AST interpreter	Graphical sea-of-nodes	Seaton, Daloze, Menard, Chalupa, MacGregor
XRuby	2006-2008	Custom Java	AOT	Template compilation to Java bytecode	Parser to AST	None	None	Zhi
yarv2llvm	2008-2010	MRI	JIT	Generation of LLVM IR				Hideki
YARV MJIT	2018-present	MRI	JIT	Generation of C		Base interpreter		Kokubun
YJIT	2020-present	MRI	JIT	Lazy Basic-Block Versioning with in-process assembler	Template compiler of YARV bytecode	Base interpreter	None	Chevalier-Boisvert
YTL	2009-2014							Hideki

https://ruby-compilers.com/

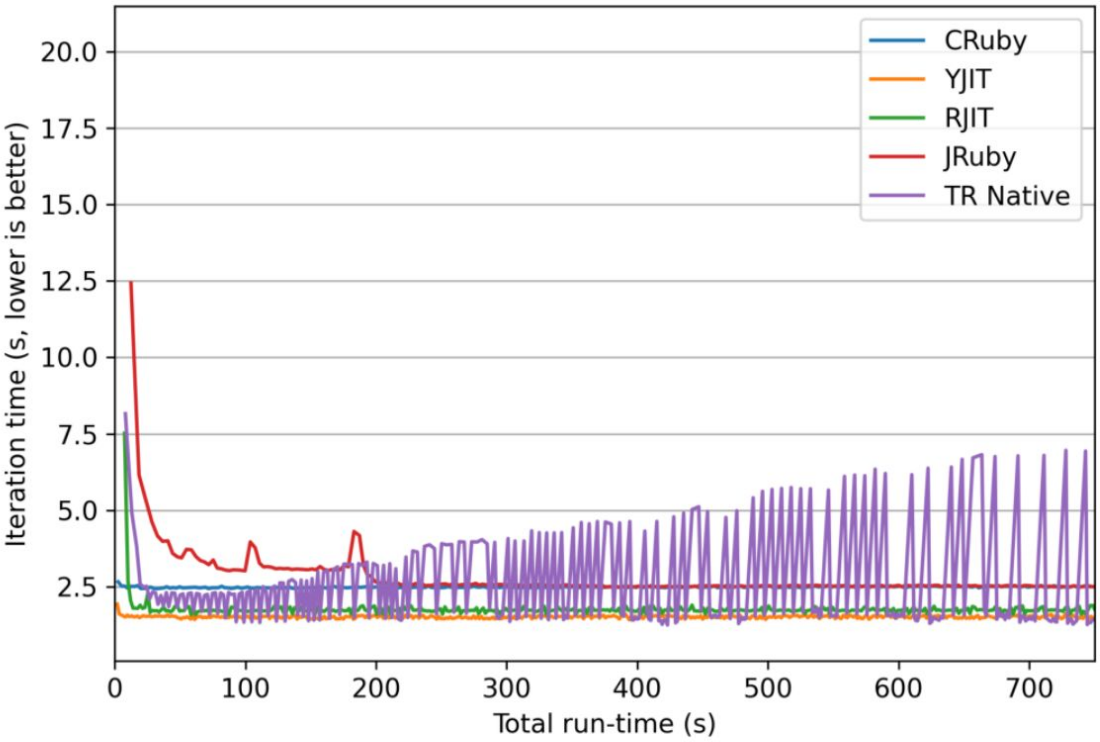
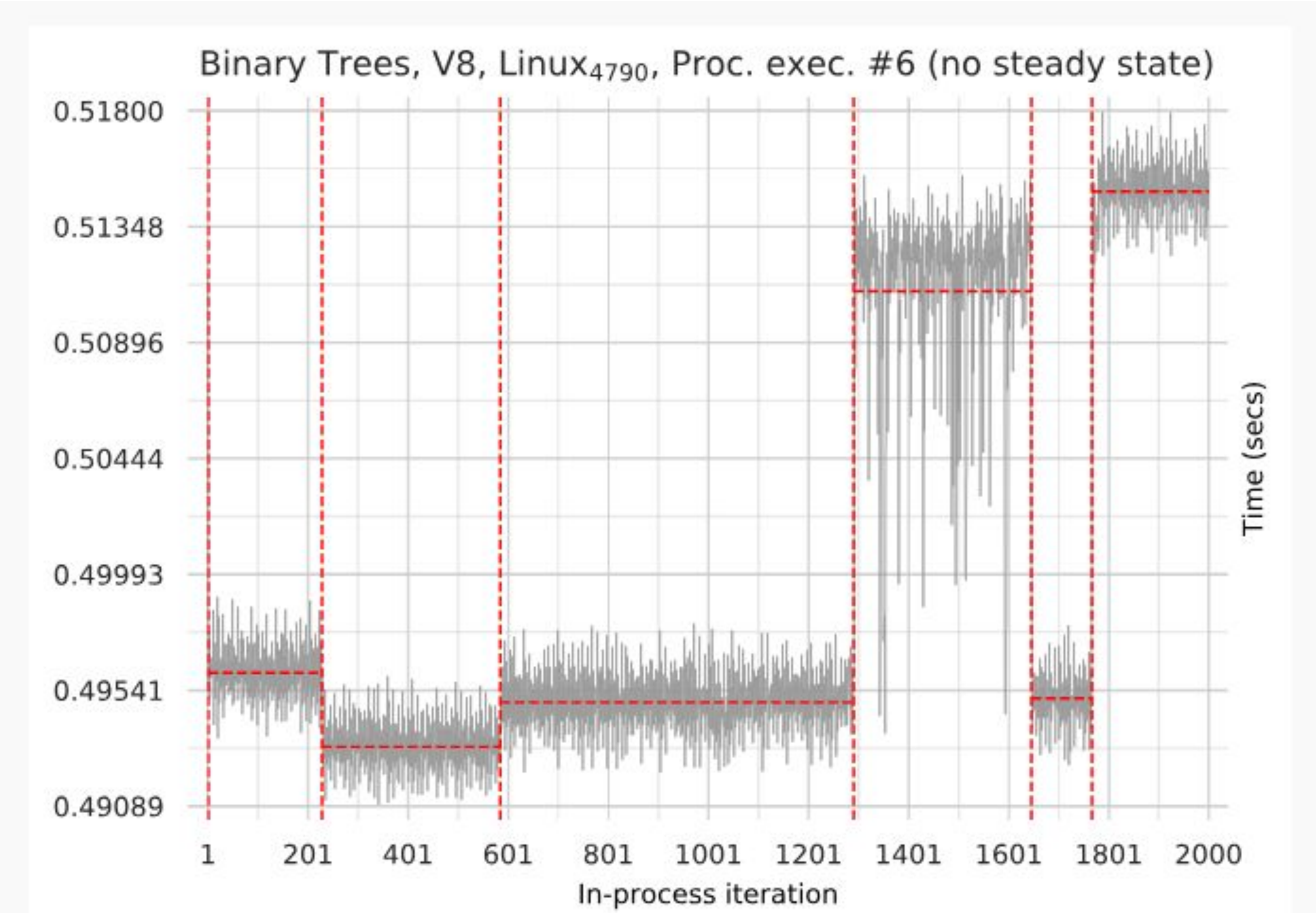


Figure 12. Time per iteration over total run-time during the first 750s for the hexapdf benchmark. YJIT has fast and predictable warm-up.

https://dl.acm.org/doi/pdf/10.1145/3617651.3622982



https://tratt.net/laurie/blog/2022/more_evidence_for_problems_in_vm_warmup.html

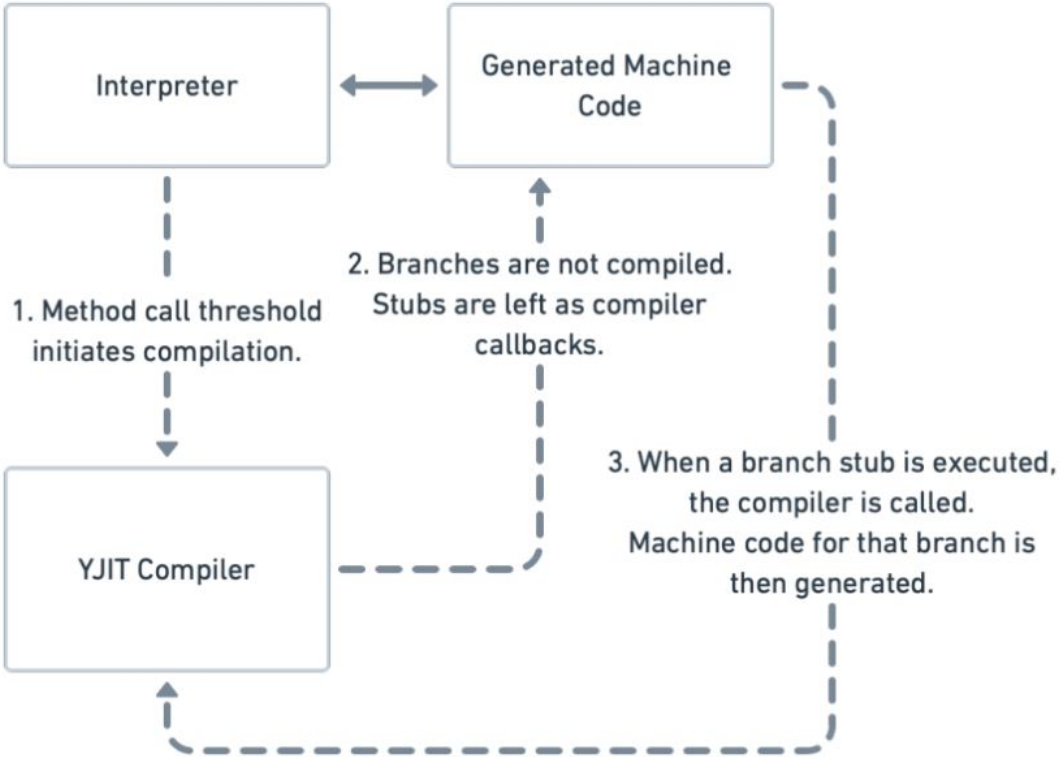


Figure 1. YJIT Compilation Pipeline.

https://dl.acm.org/doi/pdf/10.1145/3486606.3486781

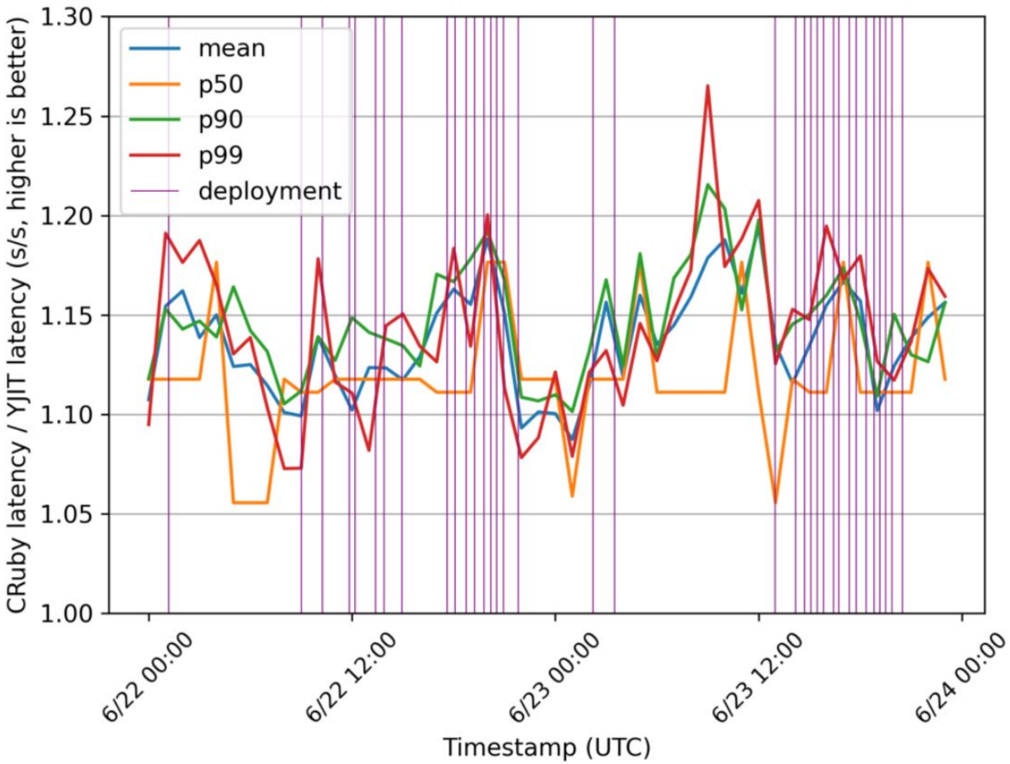


Figure 1. YJIT speedup ratio relative to the interpreter on SFR. YJIT maintains a positive speedup throughout the period examined, even on the slowest p99 requests.

https://dl.acm.org/doi/pdf/10.1145/3617651.3622982

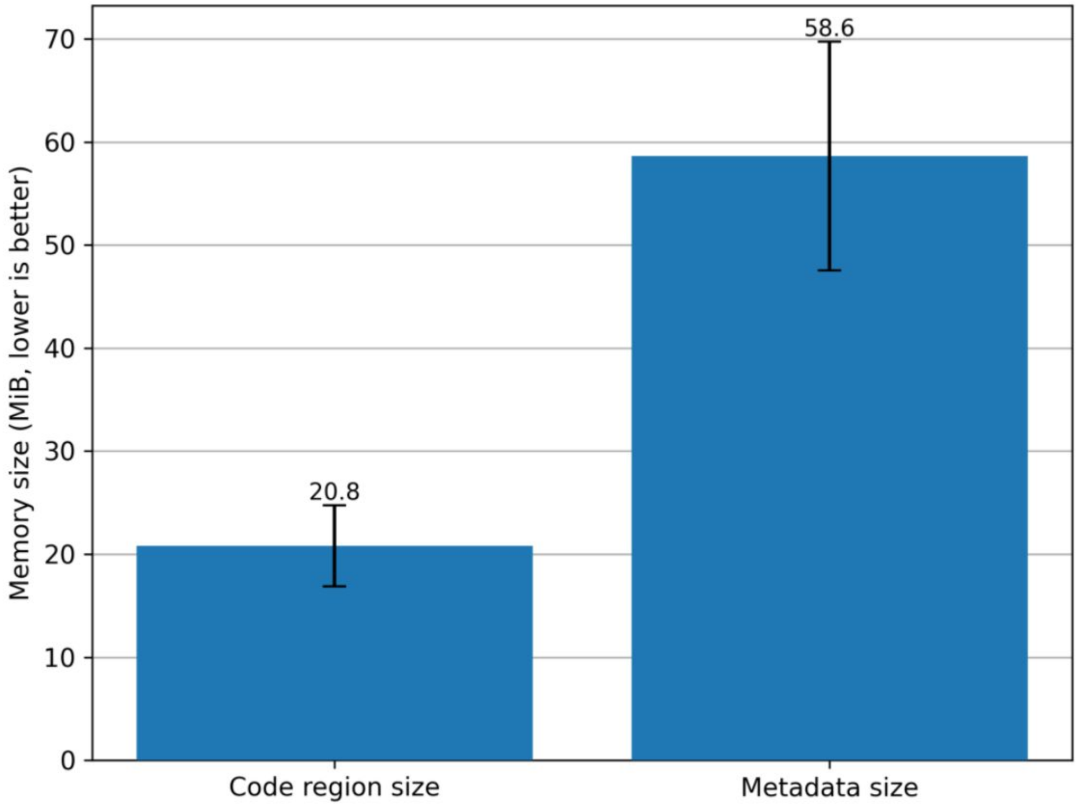


Figure 5. The mean size of JIT code region and metadata on SFR. YJIT's memory overhead largely comes from metadata.

https://dl.acm.org/doi/pdf/10.1145/3617651.3622982

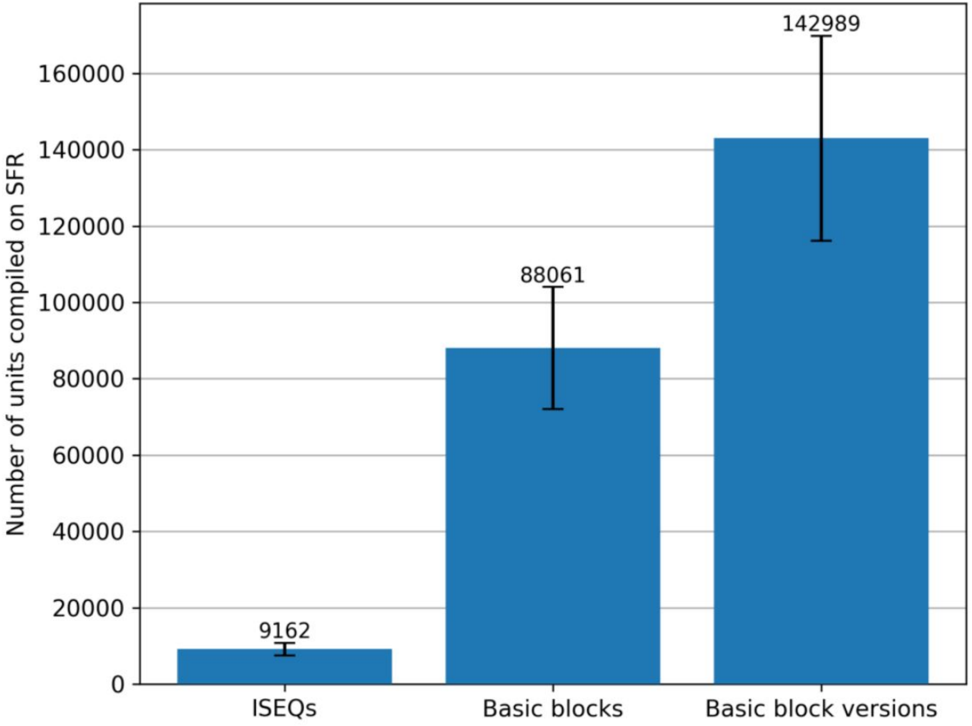


Figure 6. The mean number of compiled ISEQs, basic blocks, and basic block versions on SFR. YJIT generates many basic block versions, each of which requires metadata. On average, YJIT generated 1.62 versions per block.

https://dl.acm.org/doi/pdf/10.1145/3617651.3622982

```rust
pub struct Context {
    // Number of values currently on the temporary stack
    stack_size: u8,

    // Offset of the JIT SP relative to the interpreter SP
    // This represents how far the JIT's SP is from the "real" SP
    sp_offset: i8,

    /// Which stack temps or locals are in a register
    reg_mapping: RegMapping,

    // Depth of this block in the sidechain (eg: inline-cache chain)
    // 6 bits, max 63
    chain_depth: u8,

    // Whether this code is the target of a JIT-to-JIT Ruby return ([Self::is_return_landing])
    is_return_landing: bool,

    // Whether the compilation of this code has been deferred ([Self::is_deferred])
    is_deferred: bool,

    // Type we track for self
    self_type: Type,

    // Local variable types we keep track of
    local_types: [Type; MAX_CTX_LOCALS],

    // Temp mapping type/local_idx we track
    temp_mapping: [TempMapping; MAX_CTX_TEMPS],

    /// A pointer to a block ISEQ supplied by the caller. 0 if not inlined.
    inline_block: Option<IseqPtr>,
}
```

```rust
// Encode into a compressed context representation in a bit vector
fn encode_into(&self, bits: &mut BitVector) -> usize {
    let start_idx: usize = bits.num_bits();

    // Most of the time, the stack size is small and sp offset has the same value
    if (self.stack_size as i64) == (self.sp_offset as i64) && self.stack_size < 4 {
        // One single bit to signify a compact stack_size/sp_offset encoding
        debug_assert!(self.sp_offset >= 0);
        bits.push_u1(val: 1);
        bits.push_u2(val: self.stack_size);
    } else {
        // Full stack size encoding
        bits.push_u1(val: 0);

        // Number of values currently on the temporary stack
        bits.push_u8(val: self.stack_size);

        // sp_offset: i8,
        bits.push_u8(val: self.sp_offset as u8);
    }

    // Which stack temps or locals are in a register
    for &temp: Option<RegOpnd> in self.reg_mapping.0.iter() {
        if let Some(temp: RegOpnd) = temp {
            bits.push_u1(val: 1); // Some
            match temp {
                RegOpnd::Stack(stack_idx: u8) => {
                    bits.push_u1(val: 0); // Stack
                    bits.push_u3(val: stack_idx);
                }
                RegOpnd::Local(local_idx: u8) => {
                    bits.push_u1(val: 1); // Local
                    bits.push_u3(val: local_idx);
                }
            }
        } else {
            bits.push_u1(val: 0); // None
        }
    }
```

```rust
    bits.push_bool(val: self.is_deferred);
    bits.push_bool(val: self.is_return_landing);

    // The chain depth is most often 0 or 1
    if self.chain_depth < 2 {
        bits.push_u1(val: 0);
        bits.push_u1(val: self.chain_depth);

    } else {
        bits.push_u1(val: 1);
        bits.push_u5(val: self.chain_depth);
    }

    // Encode the self type if known
    if self.self_type != Type::Unknown {
        bits.push_op(CtxOp::SetSelfType);
        bits.push_u4(val: self.self_type as u8);
    }

    // Encode the local types if known
    for local_idx: usize in 0..MAX_CTX_LOCALS {
        let t: Type = self.get_local_type(local_idx);
        if t != Type::Unknown {
            bits.push_op(CtxOp::SetLocalType);
            bits.push_u3(val: local_idx as u8);
            bits.push_u4(val: t as u8);
        }
    }
```

```rust
    // Encode stack temps
    for stack_idx: usize in 0..MAX_CTX_TEMPS {
        let mapping: TempMapping = self.get_temp_mapping(temp_idx: stack_idx);

        match mapping {
            MapToStack(temp_type: Type) => {
                if temp_type != Type::Unknown {
                    // Temp idx (3 bits), known type (4 bits)
                    bits.push_op(CtxOp::SetTempType);
                    bits.push_u3(val: stack_idx as u8);
                    bits.push_u4(val: temp_type as u8);
                }
            }

            MapToLocal(local_idx: u8) => {
                bits.push_op(CtxOp::MapTempLocal);
                bits.push_u3(val: stack_idx as u8);
                bits.push_u3(val: local_idx as u8);
            }

            MapToSelf => {
                // Temp idx (3 bits)
                bits.push_op(CtxOp::MapTempSelf);
                bits.push_u3(val: stack_idx as u8);
            }
        }
    }

    // Inline block pointer
    if let Some(iseq: *const rb_iseq_t) = self.inline_block {
        bits.push_op(CtxOp::SetInlineBlock);
        bits.push_uint(val: iseq as u64, num_bits: 64);
    }

    // TODO: should we add an op for end-of-encoding,
    // or store num ops at the beginning?
    bits.push_op(CtxOp::EndOfCode);

    start_idx
} fn encode_into
```

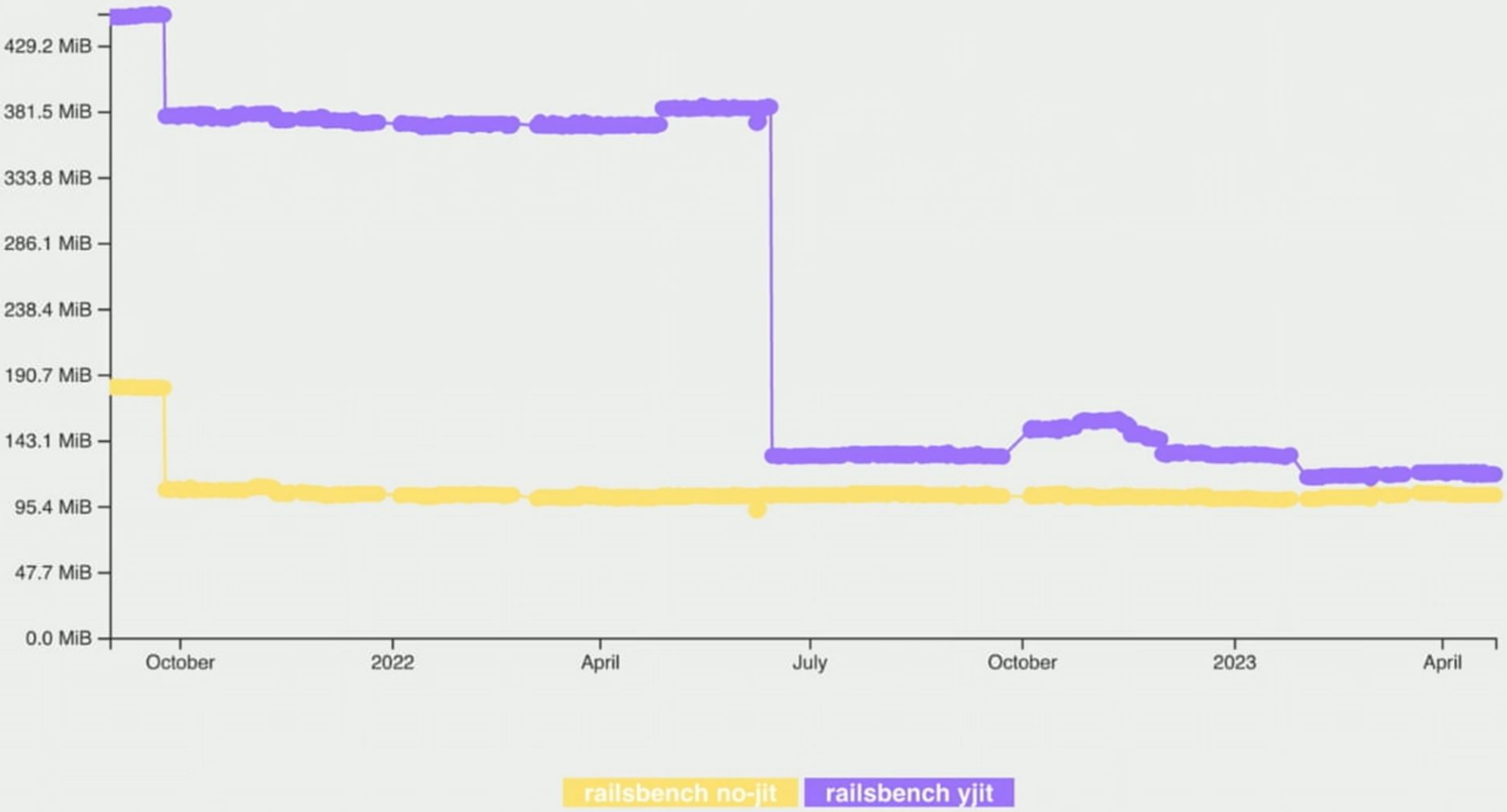
RubyVM::YJIT.runtime_stats

All YJIT metrics are available in a Hash returned by `RubyVM::YJIT.runtime_stats`. By default, the Hash looks like this:

```
$ RUBYOPT=--yjit irb
irb(main)[01:0]> RubyVM::YJIT.runtime_stats
=>
{:inline_code_size=>338600,
 :outlined_code_size=>338428,
 :freed_page_count=>0,
 :freed_code_size=>0,
 :live_page_count=>42,
 :code_gc_count=>0,
 :code_region_size=>688128,
 :object_shape_count=>635}
```

You can read a field like `RubyVM::YJIT.runtime_stats[:code_region_size]` and send the metric to whatever monitoring service you use.

https://railsatscale.com/2023-06-05-monitoring-yjit-in-production/



Y axis values are the maximum memory usage while running the benchmark - lower is better.

https://youtu.be/X0JRhh8w_4I?t=2434

```
fn gen_opt_plus(
    jit: &mut JITState,
    asm: &mut Assembler,
) -> Option<CodegenStatus> {
    let two_fixnums: bool = match asm.ctx.two_fixnums_on_stack(jit) {
        Some(two_fixnums: bool) => two_fixnums,
        None => {
            defer_compilation(jit, asm);
            return Some(EndBlock);
        }
    };

    if two_fixnums {
        if !assume_bop_not_redefined(jit, asm, klass: INTEGER_REDEFINED_OP_FLAG, BOP_PLUS) {
            return None;
        }

        // Check that both operands are fixnums
        guard_two_fixnums(jit, asm);

        // Get the operands from the stack
        let arg1: Opnd = asm.stack_pop(1);
        let arg0: Opnd = asm.stack_pop(1);

        // Add arg0 + arg1 and test for overflow
        let arg0_untag: Opnd = asm.sub(left: arg0, right: Opnd::Imm(1));
        let out_val: Opnd = asm.add(left: arg0_untag, right: arg1);
        asm.jo(Target::side_exit(Counter::opt_plus_overflow));

        // Push the output on the stack
        let dst: Opnd = asm.stack_push(val_type: Type::Fixnum);
        asm.mov(dest: dst, src: out_val);

        Some(KeepCompiling)
    } else {
        gen_opt_send_without_block(jit, asm)
    }
} fn gen_opt_plus
```

```rust
// Invalidate one specific block version
pub fn invalidate_block_version(blockref: &BlockRef) {
    //ASSERT_vm_locking();

    // TODO: want to assert that all other ractors are stopped here. Can't patch
    // machine code that some other thread is running.

    let block: &Block = unsafe { (*blockref).as_ref() };
    let id_being_invalidated: BlockId = block.get_blockid();
    let mut cb: &mut CodeBlock = CodegenGlobals::get_inline_cb();
    let ocb: &mut OutlinedCb = CodegenGlobals::get_outlined_cb();

    verify_blockid(id_being_invalidated);

    #[cfg(feature = "disasm")]
    {
        // If dump_iseq_disasm is specified, print to console that blocks for matching ISEQ names were invalidated.
        if let Some(substr) = get_option_ref!(dump_iseq_disasm).as_ref() {
            let iseq_range = &block.iseq_range;
            let iseq_location = iseq_get_location(block.iseq.get(), iseq_range.start);
            if iseq_location.contains(substr) {
                println!("Invalidating block from {}, ISEQ offsets [{}, {})", iseq_location, iseq_range.start, iseq_range.end);
            }
        }
    }

    // Remove this block from the version array
    remove_block_version(blockref);

    // Get a pointer to the generated code for this block
    let block_start: CodePtr = block.start_addr;

    // Make the start of the block do an exit. This handles OOM situations
    // and some cases where we can't efficiently patch incoming branches.
    // Do this first, since in case there is a fallthrough branch into this
    // block, the patching loop below can overwrite the start of the block.
    // In those situations, there is hopefully no jumps to the start of the block
    // after patching as the start of the block would be in the middle of something
    // generated by branch_t::gen_fn.
    let block_entry_exit: CodePtr = block &Block
        .entry_exit Option<CodePtr>
        .expect(msg: "invalidation needs the entry_exit field");
    {
        let block_end: CodePtr = block.get_end_addr();

        if block_start == block_entry_exit {
            // Some blocks exit on entry. Patching a jump to the entry at the
            // entry makes an infinite loop.
        } else {
            // Patch in a jump to block.entry_exit.

            let cur_pos: CodePtr = cb.get_write_ptr();
            let cur_dropped_bytes: bool = cb.has_dropped_bytes();
            cb.set_write_ptr(code_ptr: block_start);

            let mut asm: Assembler = Assembler::new_without_iseq();
            asm.jmp(target: block_entry_exit.as_side_exit());
            cb.set_dropped_bytes(false);
            asm.compile(&mut cb, ocb: Some(ocb)).expect(msg: "can rewrite existing code");

            assert!(
                cb.get_write_ptr() <= block_end,
                "invalidation wrote past end of block (code_size: {:?}, new_size: {}, start_addr: {:?})",
                block.code_size(),
                cb.get_write_ptr().as_offset() - block_start.as_offset(),
                block.start_addr.raw_ptr(cb),
            );
            cb.set_write_ptr(code_ptr: cur_pos);
            cb.set_dropped_bytes(cur_dropped_bytes);
        }
    }

    // For each incoming branch
    for branchref: &NonNull<Branch> in block.incoming.0.take().iter() {
        let branch: &Branch = unsafe { branchref.as_ref() };
        let target_idx: usize = if branch.get_target_address(target_idx: 0) == Some(block_start) {
            0
        } else {
            1
        };

        // Assert that the incoming branch indeed points to the block being invalidated
        // SAFETY: no mutation.
        unsafe {
            let incoming_target: &Box<BranchTarget> = branch.targets[target_idx].ref_unchecked().as_ref().unwrap();
            assert_eq!(Some(block_start), incoming_target.get_address());
            if let Some(incoming_block: &NonNull<Block>) = &incoming_target.get_block() {
                assert_eq!(blockref, incoming_block);
            }
        }
```

```rust
        // Create a stub for this branch target
        let stub_addr: Option<CodePtr> = gen_branch_stub(block.ctx, iseq: block.iseq.get(), ocb, branch_struct_address… branchref.as_p

        // In case we were unable to generate a stub (e.g. OOM). Use the block's
        // exit instead of a stub for the block. It's important that we
        // still patch the branch in this situation so stubs are unique
        // to branches. Think about what could go wrong if we run out of
        // memory in the middle of this loop.
        let stub_addr: CodePtr = stub_addr.unwrap_or(default: block_entry_exit);

        // Fill the branch target with a stub
        branch.targets[target_idx].set(val: Some(Box::new(BranchTarget::Stub(Box::new(BranchStub {
            address: Some(stub_addr),
            iseq: block.iseq.clone(),
            iseq_idx: block.iseq_range.start,
            ctx: block.ctx,
        })))));

        // Check if the invalidated block immediately follows
        let target_next: bool = block.start_addr == branch.end_addr.get();

        if target_next {
            // The new block will no longer be adjacent.
            // Note that we could be enlarging the branch and writing into the
            // start of the block being invalidated.
            branch.gen_fn.set_shape(new_shape: BranchShape::Default);
        }

        // Rewrite the branch with the new jump target address
        let old_branch_size: usize = branch.code_size();
        regenerate_branch(cb, branch);

        if target_next && branch.end_addr > block.end_addr {
            panic!("yjit invalidate rewrote branch past end of invalidated block: {:?} (code_size: {})", branch, block.code_size());
        }
        if !target_next && branch.code_size() > old_branch_size {
            panic!(
                "invalidated branch grew in size (start_addr: {:?}, old_size: {}, new_size: {})",
                branch.start_addr.raw_ptr(cb), old_branch_size, branch.code_size()
            );
        }
    }

    // Clear out the JIT func so that we can recompile later and so the
    // interpreter will run the iseq.
    //
    // Only clear the jit_func when we're invalidating the JIT entry block.
    // We only support compiling iseqs from index 0 right now.  So entry
    // points will always have an instruction index of 0.  We'll need to
    // change this in the future when we support optional parameters because
    // they enter the function with a non-zero PC
    if block.iseq_range.start == 0 {
        // TODO:
        // We could reset the exec counter to zero in rb_iseq_reset_jit_func()
        // so that we eventually compile a new entry point when useful
        unsafe { rb_iseq_reset_jit_func(iseq: block.iseq.get()) };
    }

    // FIXME:
    // Call continuation addresses on the stack can also be atomically replaced by jumps going to the stub.

    // SAFETY: This block was in a version_map earlier
    // in this function before we removed it, so it's well connected.
    unsafe { remove_from_graph(*blockref) };

    delayed_deall yjit::asm::CodeBlock
                  pub fn mark_all_executable(&mut self)

    ocb.unwrap().mark_all_executable();
    cb.mark_all_executable();

    incr_counter!(invalidation_count);
} fn invalidate_block_version
```

```rust
// Invariants to track:
// assume_bop_not_redefined(jit, INTEGER_REDEFINED_OP_FLAG, BOP_PLUS)
// assume_method_lookup_stable(comptime_recv_klass, cme, jit);
// assume_single_ractor_mode()
// track_stable_constant_names_assumption()

/// Used to track all of the various block references that contain assumptions
/// about the state of the virtual machine.
1 implementation
pub struct Invariants {
    /// Tracks block assumptions about callable method entry validity.
    cme_validity: HashMap<*const rb_callable_method_entry_t, HashSet<BlockRef>>,

    /// A map from a class and its associated basic operator to a set of blocks
    /// that are assuming that that operator is not redefined. This is used for
    /// quick access to all of the blocks that are making this assumption when
    /// the operator is redefined.
    basic_operator_blocks: HashMap<(RedefinitionFlag, ruby_basic_operators), HashSet<BlockRef>>,
    /// A map from a block to a set of classes and their associated basic
    /// operators that the block is assuming are not redefined. This is used for
    /// quick access to all of the assumptions that a block is making when it
    /// needs to be invalidated.
    block_basic_operators: HashMap<BlockRef, HashSet<(RedefinitionFlag, ruby_basic_operators)>>,

    /// Tracks the set of blocks that are assuming the interpreter is running
    /// with only one ractor. This is important for things like accessing
    /// constants which can have different semantics when multiple ractors are
    /// running.
    single_ractor: HashSet<BlockRef>,

    /// A map from an ID to the set of blocks that are assuming a constant with
    /// that ID as part of its name has not been redefined. For example, if
    /// a constant `A::B` is redefined, then all blocks that are assuming that
    /// `A` and `B` have not be redefined must be invalidated.
    constant_state_blocks: HashMap<ID, HashSet<BlockRef>>,
    /// A map from a block to a set of IDs that it is assuming have not been
    /// redefined.
    block_constant_states: HashMap<BlockRef, HashSet<ID>>,

    /// A map from a class to a set of blocks that assume objects of the class
    /// will have no singleton class. When the set is empty, it means that
    /// there has been a singleton class for the class after boot, so you cannot
    /// assume no singleton class going forward.
    /// For now, the key can be only Array, Hash, or String. Consider making
    /// an inverted HashMap if we start using this for user-defined classes
    /// to maintain the performance of block_assumptions_free().
    no_singleton_classes: HashMap<VALUE, HashSet<BlockRef>>,

    /// A map from an ISEQ to a set of blocks that assume base pointer is equal
    /// to environment pointer. When the set is empty, it means that EP has been
    /// escaped in the ISEQ.
    no_ep_escape_iseqs: HashMap<IseqPtr, HashSet<BlockRef>>,
}
```

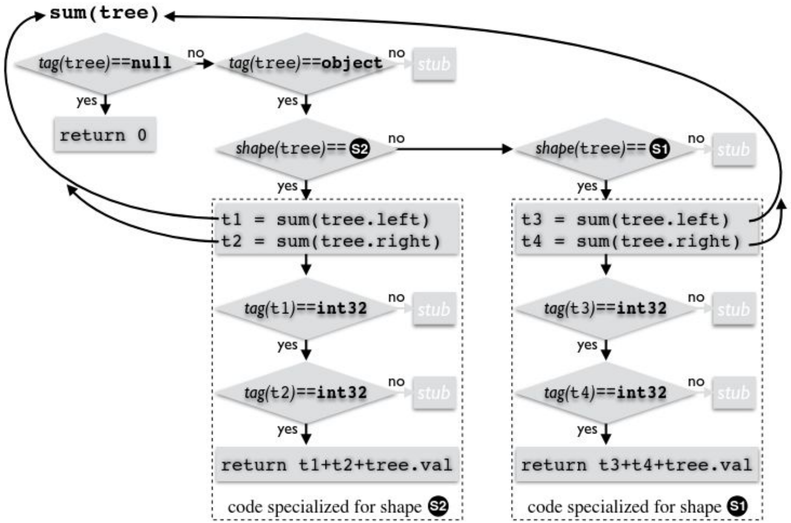


Figure 6. Generated code for the sum function with intraprocedural BBV

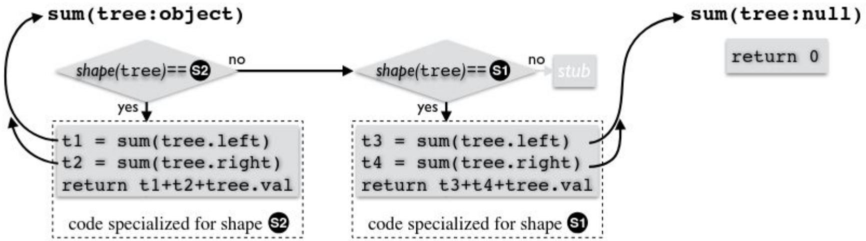


Figure 7. Generated code for the sum function with interprocedural BBV

https://arxiv.org/pdf/1511.02956

```rust
/// YJIT IR instruction
2 implementations
pub enum Insn {
    /// Add two operands together, and return the result as a new operand.
    Add { left: Opnd, right: Opnd, out: Opnd },

    /// This is the same as the OP_ADD instruction, except that it performs the
    /// binary AND operation.
    And { left: Opnd, right: Opnd, out: Opnd },

    /// Bake a string directly into the instruction stream.
    BakeString(String),

    // Trigger a debugger breakpoint
    #[allow(dead_code)]
    Breakpoint,

    /// Add a comment into the IR at the point that this instruction is added.
    /// It won't have any impact on that actual compiled code.
    Comment(String),

    /// Compare two operands
    Cmp { left: Opnd, right: Opnd },

    /// Pop a register from the C stack
    CPop { out: Opnd },

    /// Pop all of the caller-save registers and the flags from the C stack
    CPopAll,

    /// Pop a register from the C stack and store it into another register
    CPopInto(Opnd),

    /// Push a register onto the C stack
    CPush(Opnd),

    /// Push all of the caller-save registers and the flags to the C stack
    CPushAll,

    // C function call with N arguments (variadic)
    CCall { opnds: Vec<Opnd>, fptr: *const u8, out: Opnd },

    // C function return
    CRet(Opnd),

    /// Conditionally select if equal
    CSelE { truthy: Opnd, falsy: Opnd, out: Opnd },

    /// Conditionally select if greater
    CSelG { truthy: Opnd, falsy: Opnd, out: Opnd },

    /// Conditionally select if greater or equal
    CSelGE { truthy: Opnd, falsy: Opnd, out: Opnd },

    /// Conditionally select if less
    CSelL { truthy: Opnd, falsy: Opnd, out: Opnd },

    /// Conditionally select if less or equal
    CSelLE { truthy: Opnd, falsy: Opnd, out: Opnd },

    /// Conditionally select if not equal
    CSelNE { truthy: Opnd, falsy: Opnd, out: Opnd },

    /// Conditionally select if not zero
    CSelNZ { truthy: Opnd, falsy: Opnd, out: Opnd },

    /// Conditionally select if zero
    CSelZ { truthy: Opnd, falsy: Opnd, out: Opnd },

    /// Set up the frame stack as necessary per the architecture.
    FrameSetup,

    /// Tear down the frame stack as necessary per the architecture.
    FrameTeardown,

    // Atomically increment a counter
    // Input: memory operand, increment value
    // Produces no output
    IncrCounter { mem: Opnd, value: Opnd },

    /// Jump if below or equal (unsigned)
    Jbe(Target),

    /// Jump if below (unsigned)
    Jb(Target),

    /// Jump if equal
    Je(Target),

    /// Jump if lower
    Jl(Target),

    /// Jump if greater
    Jg(Target),

    /// Jump if greater or equal
    Jge(Target),

    // Unconditional jump to a branch target
    Jmp(Target),

    // Unconditional jump which takes a reg/mem address operand
    JmpOpnd(Opnd),

    /// Jump if not equal
    Jne(Target),

    /// Jump if not zero
    Jnz(Target),

    /// Jump if overflow
    Jo(Target),

    /// Jump if overflow in multiplication
    JoMul(Target),

    /// Jump if zero
    Jz(Target),

    /// Jump if operand is zero (only used during lowering at the moment)
    Joz(Opnd, Target),

    /// Jump if operand is non-zero (only used during lowering at the moment)
    Jonz(Opnd, Target),

    // Add a label into the IR at the point that this instruction is added.
    Label(Target),

    /// Get the code address of a jump target
    LeaJumpTarget { target: Target, out: Opnd },

    // Load effective address
    Lea { opnd: Opnd, out: Opnd },

    /// Take a specific register. Signal the register allocator to not use it.
    LiveReg { opnd: Opnd, out: Opnd },

    // A low-level instruction that loads a value into a register.
    Load { opnd: Opnd, out: Opnd },

    // A low-level instruction that loads a value into a specified register.
    LoadInto { dest: Opnd, opnd: Opnd },

    // A low-level instruction that loads a value into a register and
    // sign-extends it to a 64-bit value.
    LoadSExt { opnd: Opnd, out: Opnd },

    /// Shift a value left by a certain amount.
    LShift { opnd: Opnd, shift: Opnd, out: Opnd },

    // A low-level mov instruction. It accepts two operands.
    Mov { dest: Opnd, src: Opnd },

    // Perform the NOT operation on an individual operand, and return the result
    // as a new operand. This operand can then be used as the operand on another
    // instruction.
    Not { opnd: Opnd, out: Opnd },

    // This is the same as the OP_ADD instruction, except that it performs the
    // binary OR operation.
    Or { left: Opnd, right: Opnd, out: Opnd },

    /// Pad nop instructions to accommodate Op::Jmp in case the block or the insn
    /// is invalidated.
    PadInvalPatch,

    // Mark a position in the generated code
    PosMarker(PosMarkerFn),

    /// Shift a value right by a certain amount (signed).
    RShift { opnd: Opnd, shift: Opnd, out: Opnd },

    // Low-level instruction to store a value to memory.
    Store { dest: Opnd, src: Opnd },

    // This is the same as the add instruction, except for subtraction.
    Sub { left: Opnd, right: Opnd, out: Opnd },

    // Integer multiplication
    Mul { left: Opnd, right: Opnd, out: Opnd },

    // Bitwise AND test instruction
    Test { left: Opnd, right: Opnd },

    /// Shift a value right by a certain amount (unsigned).
    URShift { opnd: Opnd, shift: Opnd, out: Opnd },

    // This is the same as the OP_ADD instruction, except that it performs the
    // binary XOR operation.
    Xor { left: Opnd, right: Opnd, out: Opnd }
}
```

Binding

Objects of class `Binding` encapsulate the execution context at some particular place in the code and retain this context for future use. The variables, methods, value of `self`, and possibly an iterator block that can be accessed in this context are all retained. Binding objects can be created using `Kernel#binding`, and are made available to the callback of `Kernel#set_trace_func`.

These binding objects can be passed as the second argument of the `Kernel#eval` method, establishing an environment for the evaluation.

```
class Demo
  def initialize(n)
    @secret = n
  end
  def get_binding
    binding
  end
end

k1 = Demo.new(99)
b1 = k1.get_binding
k2 = Demo.new(-3)
b2 = k2.get_binding

eval("@secret", b1)   #=> 99
eval("@secret", b2)   #=> -3
eval("@secret")       #=> nil
```

Binding objects have no class-specific methods.

https://ruby-doc.org/core-2.5.4/Binding.html

```c
RBIMPL_ATTR_NONNULL(())
/**
 * Creates a binding object of the point where the trace is at.
 *
 * @param[in]  trace_arg  A trace instance.
 * @retval     RUBY_Qnil  The point has no binding.
 * @retval     otherwise  Its binding.
 *
 * @internal
 *
 * @shyouhei has no idea on which situation shall this function return
 * ::RUBY_Qnil.
 */
VALUE rb_tracearg_binding(rb_trace_arg_t *trace_arg);
```

```
/* check `target' matches `pattern'.
    `flag & VM_CHECKMATCH_TYPE_MASK' describe how to check pattern.
     VM_CHECKMATCH_TYPE_WHEN: ignore target and check pattern is truthy.
     VM_CHECKMATCH_TYPE_CASE: check `patten === target'.
     VM_CHECKMATCH_TYPE_RESCUE: check `pattern.kind_of?(Module) && pattern === target'.
    if `flag & VM_CHECKMATCH_ARRAY' is not 0, then `patten' is array of patterns.
 */
DEFINE_INSN
checkmatch
(rb_num_t flag)
(VALUE target, VALUE pattern)
(VALUE result)
// attr bool leaf = leafness_of_checkmatch(flag);
{
    result = vm_check_match(ec, target, pattern, flag);
}
```

I fail to see the difference to trace compilation (and the predecessor of trace compilation, dynamic binary translation) […] Constant propagation and conditional elimination in a trace compiler lead to the same type check elimination that you present.

The one big problem I have with the paper is that it does not motivate and put into context lazy block versioning properly. The paper needs to do a better job at explaining which precise problems of current Javascript JIT approaches that are used in production are solved by lazy basic block versioning.

https://pointersgonewild.com/2014/07/17/my-paper-was-rejected-again/

ABSTRACT

The Smalltalk-80* programming language includes dynamic storage allocation, full upward funargs, and universally polymorphic procedures; the Smalltalk-80 programming system features interactive execution with incremental compilation, and implementation portability. These features of modern programming systems are among the most difficult to implement efficiently, even individually. A new implementation of the Smalltalk-80 system, hosted on a small microprocessor-based computer, achieves high performance while retaining complete (object code) compatibility with existing implementations. This paper discusses the most significant optimization techniques developed over the course of the project, many of which are applicable to other languages. The key idea is to represent certain runtime state (both code and data) in more than one form, and to convert between forms when needed.

https://dl.acm.org/doi/pdf/10.1145/800017.800542

The papers we have submitted with truly new ideas and techniques, and years of work behind them, get reviews asking you to do 2-4 years more work. For example, they ask you to create a completely different system by another team with no knowledge of your ideas and run an A vs. B test (because that commercial system you compared to had different goals in mind). Oh, and 8-10 participants doing 3-4 hour sessions/participant isn't enough for an evaluation. You need lots more... They go on and on like this. Essentially setting you up for a level of rigor that is almost impossible to meet in the career of a graduate student.

This attitude is a joke and it offers researchers **no** incentive to do systems work. Why should they? Why should we put 3-4 person years into every CHI publication? Instead we can do 8 weeks of work on an idea piece or create a new interaction technique and test it tightly in 8-12 weeks and get a full CHI paper. I know it is not about counting publications, but until hiring and tenure policies change, this is essentially what happens in the real world. The HCI systems student with 3 papers over their career won't even get an interview. Nor will any systems papers win best paper awards (yes, it happens occasionally but I know for a fact that they are *usually* the ones written by big teams doing 3-4 person-years of work).

https://dubfuture.blogspot.com/2009/11/i-give-up-on-chiuist.html

Go back to thinking about and *building systems*. Narrowness is irrelevant; breadth is relevant: it's the essence of *system*.

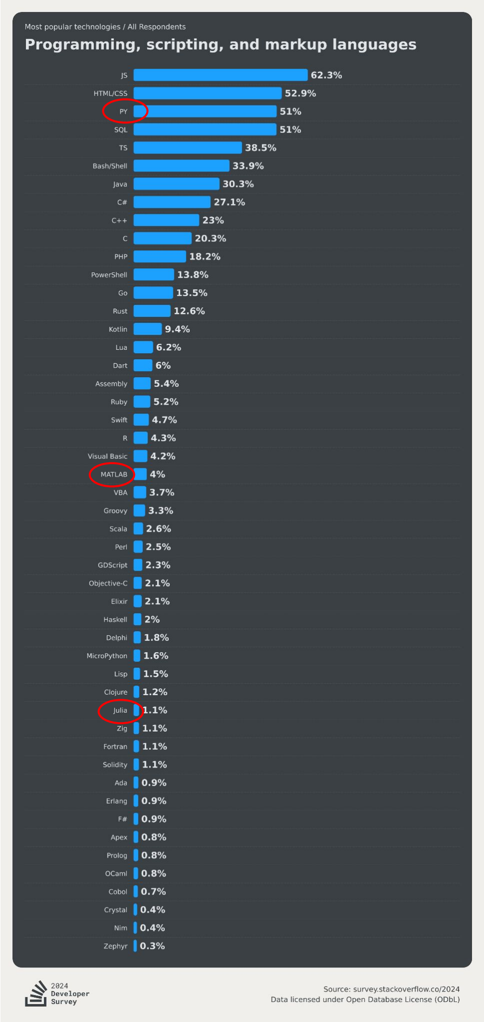
Work on how systems behave and work, not just how they compare. Concentrate on interfaces and architecture, not just engineering.

Be courageous. Try different things; experiment. *Try to give a cool demo.*

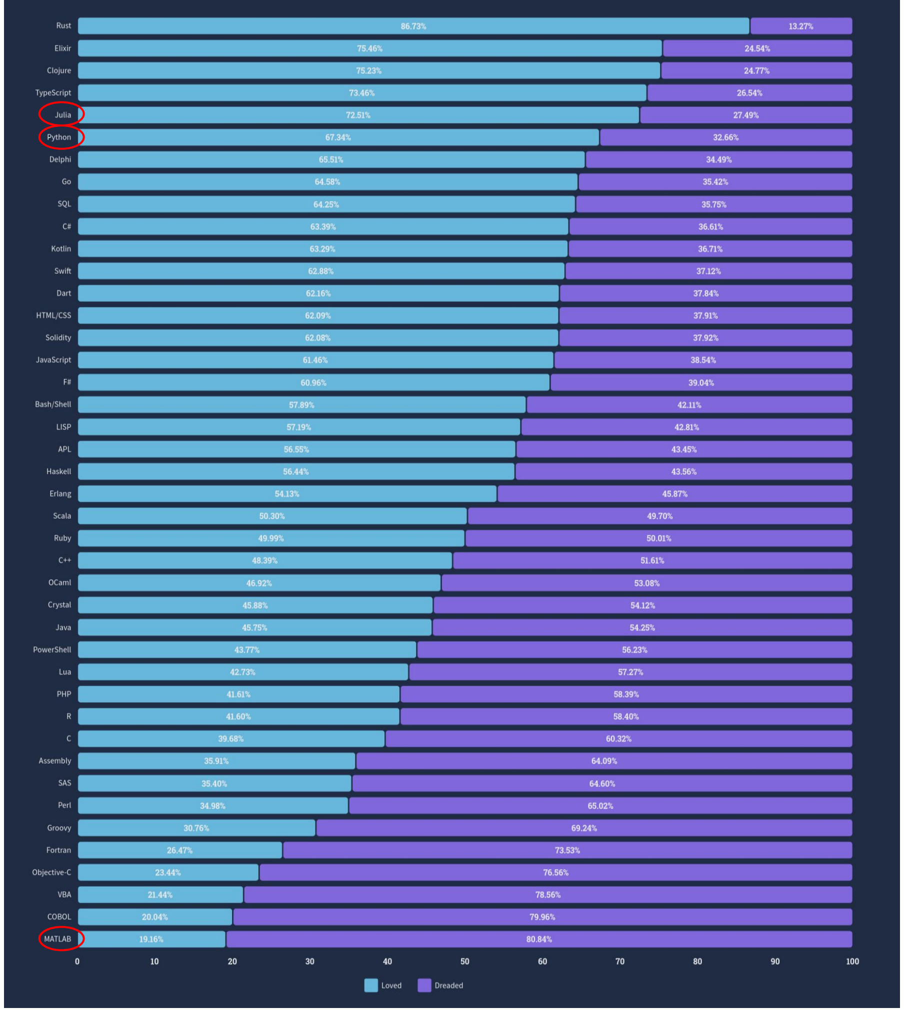
Funding bodies: fund more courageously, particularly long-term projects. Universities, in turn, should explore ways to let students contribute to long-term projects.

Measure success by ideas, not just papers and money. Make the industry *want* your work.

http://herpolhode.com/rob/utah2000.pdf



https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language



https://survey.stackoverflow.co/2022/#technology-most-loved-dreaded-and-wanted