# IR JIT Framework a base for the next generation JIT for PHP

**Presentation** · October 2023

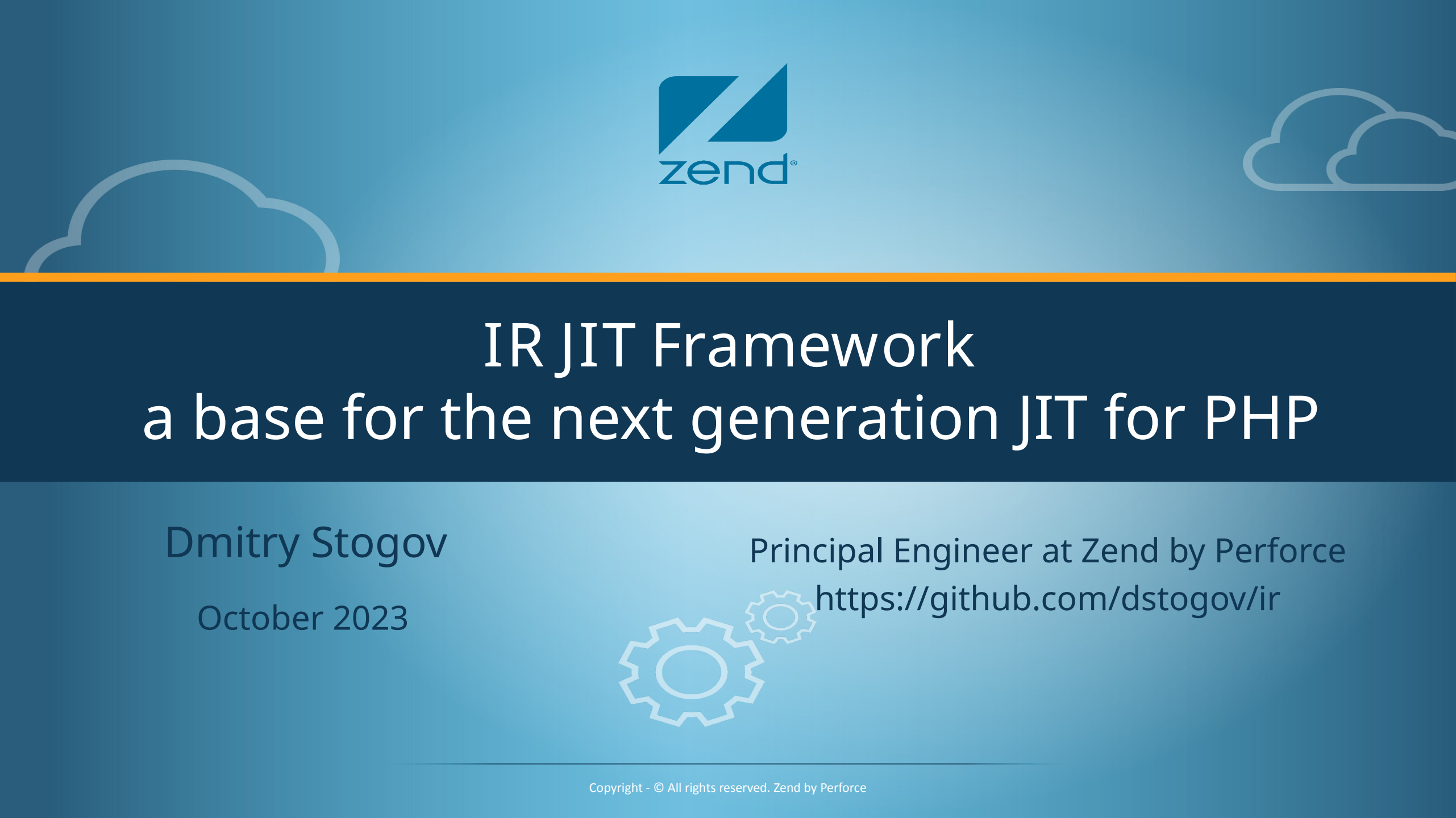
DOI: 10.13140/RG.2.2.20626.43201



**1 author:**

Dmitry Stogov
Zend by Perforce
**2** PUBLICATIONS **3** CITATIONS

SEE PROFILE




# IR JIT Framework
# a base for the next generation JIT for PHP

Dmitry Stogov

Principal Engineer at Zend by Perforce

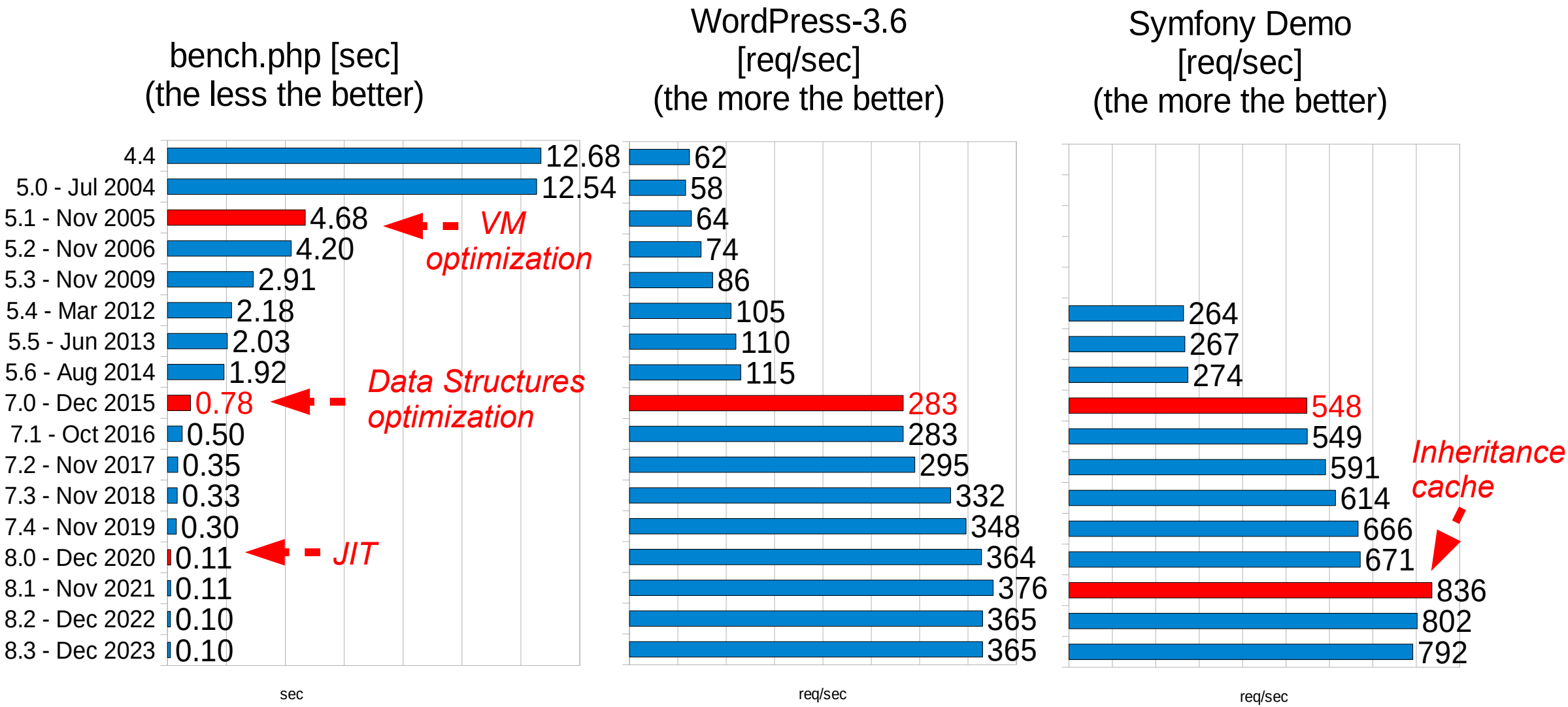
October 2023

https://github.com/dstogov/ir

# Who am I?

- First programming experience in 1984
- Work in IT since 1991
- First PHP experience in 2002
- Author of Turck MMCache for PHP (eAccelerator)
- Work at Zend since 2004
- Principal engineer at Zend by Perforce
- Author of few PHP extemsions ext/soap, ext/ffi, pecl/perl
- One of the lead Open Source PHP contributor and maintainer
- Zend OPcache maintainer
- PHPNG project leader
- The lead developer of JIT for PHP

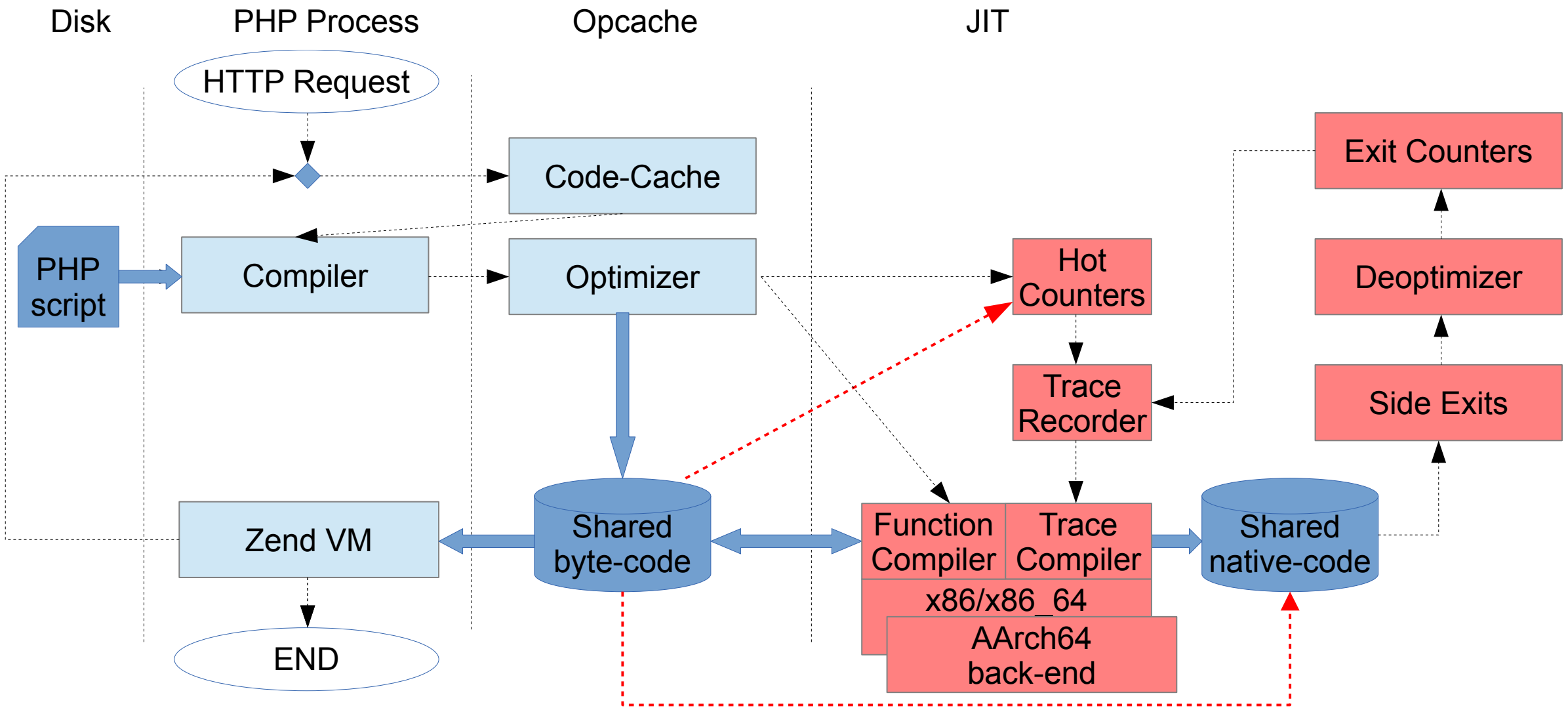
# PHP Performance Evaluation (opcache.jit=tracing)



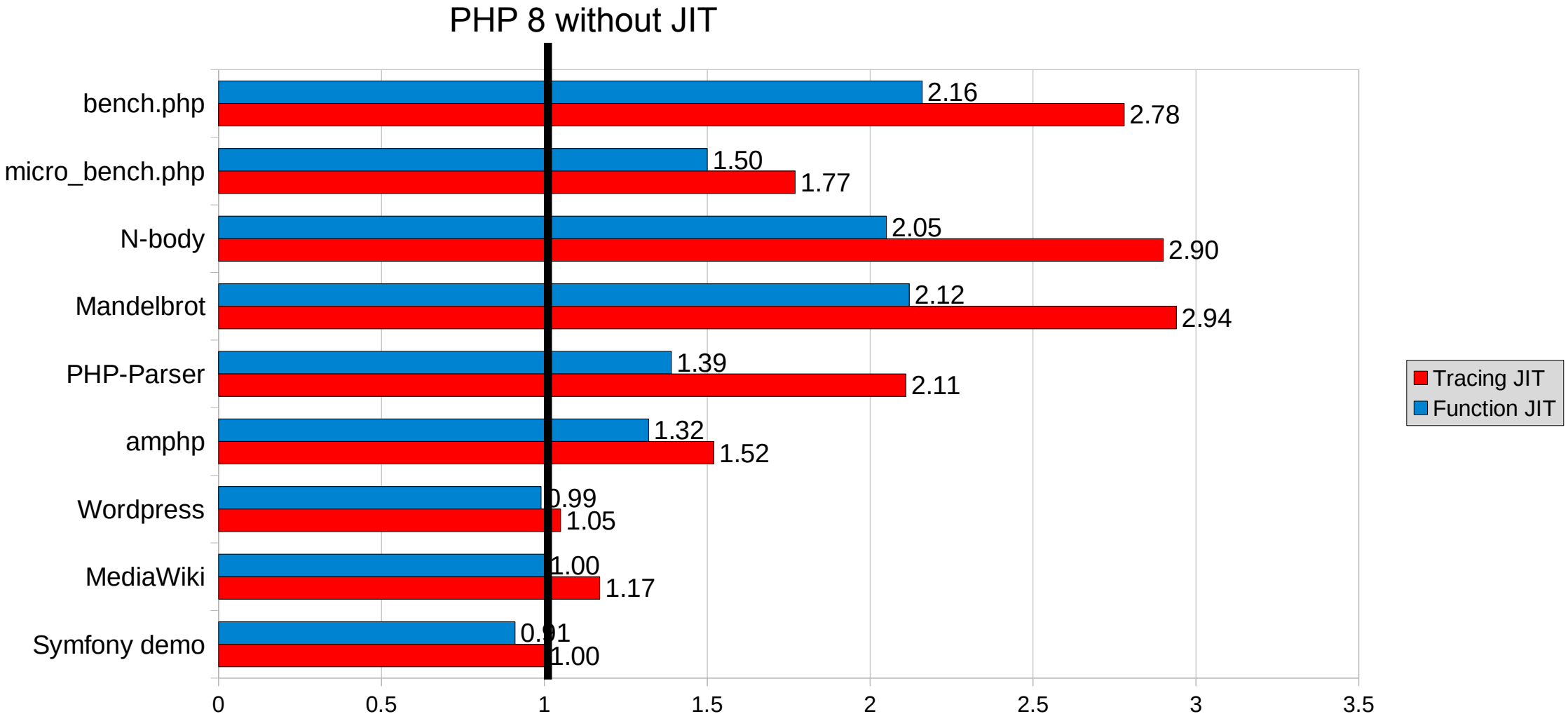
# PHP JIT History

- Zend started JIT related research in 2011
- The 1-st generation JIT was developed in 2011-2012 and supported only x86 code
- The 2-nd LLVM based generation was developed in 2012-2013
- The early results made a base for the VM and byte-code Optimizer improvements
  - Most the improvements were released as parts of PHP-7.0 and PHP-7.1
- The first officially released JIT came in PHP-8.0 (developed in 2018-2020)
  - DynAsm based function and tracing JIT for x86 and x86_64
- In PHP-8.1 we added DynAsm based AArch64 back-end (2021)
- In 2022 we started developing PHP independent IR JIT framework

# PHP + OPCache + Function and Tracing JIT



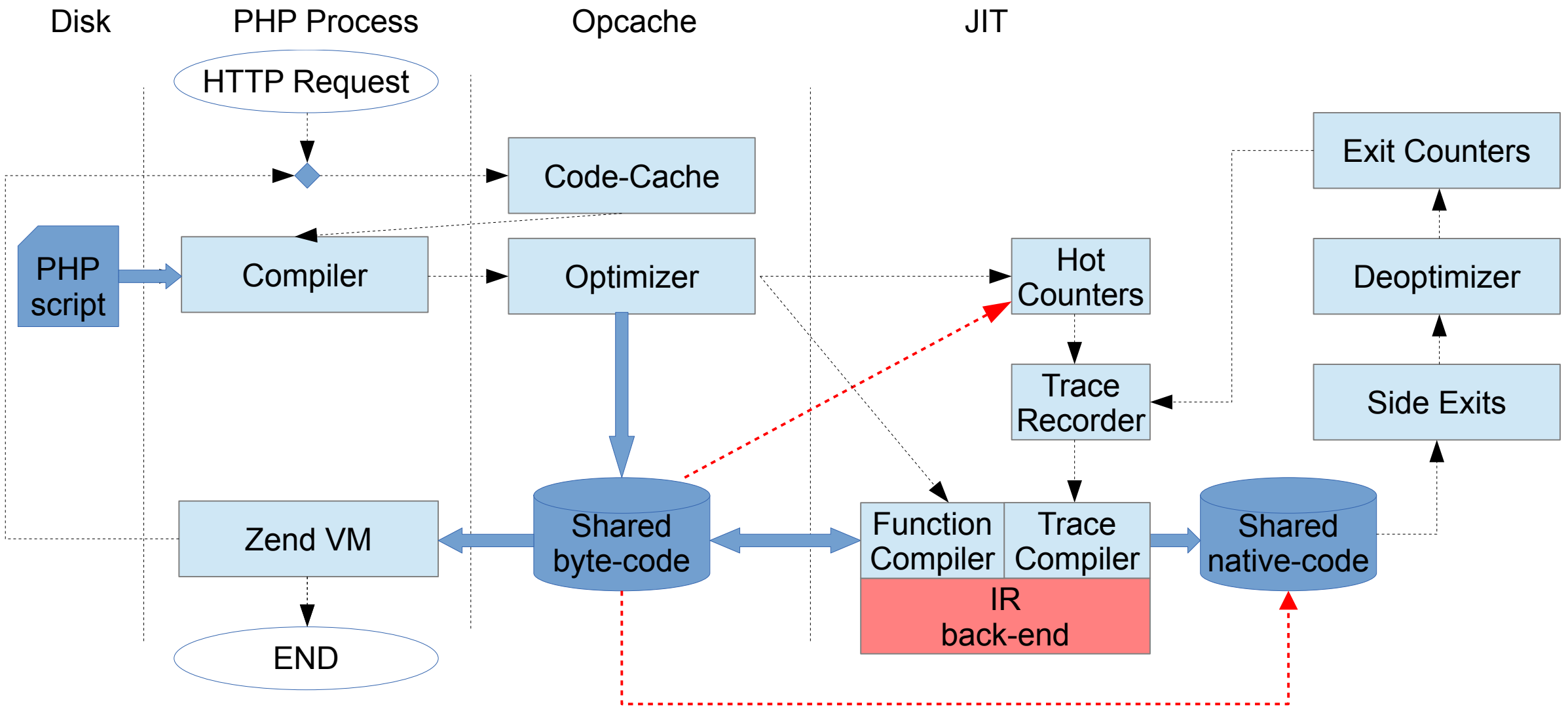
# Relative JIT Contribution to PHP 8.0 Performance



# The State of PHP JIT in 8.3

- PHP JIT generates code directly from PHP VM byte-code
  - No any IR (Intermediate Representation)
  - All static analyzes are done on byte-code level
  - JIT performs SSA based type-inference
  - CPU registers are allocated directly for PHP variables
- PHP JIT “manually” generates machine code directly for target CPU
  - We have to support each CPU (x86[_64] and AArch64) separately
  - We are limited in optimization and register allocation scope
- PHP JIT produces intermediate quality code in a very short time
- PHP JIT related fixes may require changes in assembler code for all supported platforms
  - Hard to support

# Idea of a New IR Back-End (Common, Target Independent)



# IR JIT Framework

# Why a new JIT framework?

- There is no good universal library suitable for JIT of mid size project
  - LLVM and LIBGCCJIT - very big and too slow
  - Eclipse OMR - big (800K lines of code)
  - LIBJIT - misses AArch64 support, development is stopped
  - LIBFIRM – x86_64 support is limited, no AArch64, development is stopped
  - QBE – small and simple, generates assembler code as text
  - NanoJIT – no optimizations, just a portable code generator with many limitations, development is stopped
  - MIR – interesting project that may be suitable for JIT, but misses disassembler, support for debugger and x86/32-bit. It’s actually not faster than LLVM on big functions and not actively developed
- There are a plenty of successful specialized JITs (HotSpot, V8, LuaJIT)

# Ideas Behind the IR JIT Framework

- IR Framework should be kept simple, compact and fast
- It should produce good quality code in short time
- Single SSA based IR during all the phases of JIT compilation (no HIR, MIR, LIR, no lowering)
- Sea of Nodes IR
- Compact and Efficient physical in-memory IR representation
- On-the-fly optimizations during IR construction (Folding)
- Sparse Conditional Constants Propagation (SCCP)
- Global Code Motion (GCM)
- Efficient CPU instruction selection (Maximal-Munch matcher, should be replaced by BURS)
- Linear Scan Register Allocation with second chance bin-packing (LSRA)
- DynASM based code generator (should be replaced by something more efficient)
- IR provides extensions useful for JIT (fixed stack frame, OSR entries, binding to VM stack)
- Implemented in C and developed under permissive license independently from PHP

# IR: QUADS

```
int foo(int a) {
    i = 0;
    do {
        b = a + 1;
        i = i + b;
        c = i * 2;
    } while (i < 10);
    return c;
}
```

START
a=PARAM(1)
i=COPY(0)

L1:

b=ADD(a, 1)
i=ADD(i, b)
c=MUL(i, 2)
t=LT(i, 10)

IF(t, L1, L2)

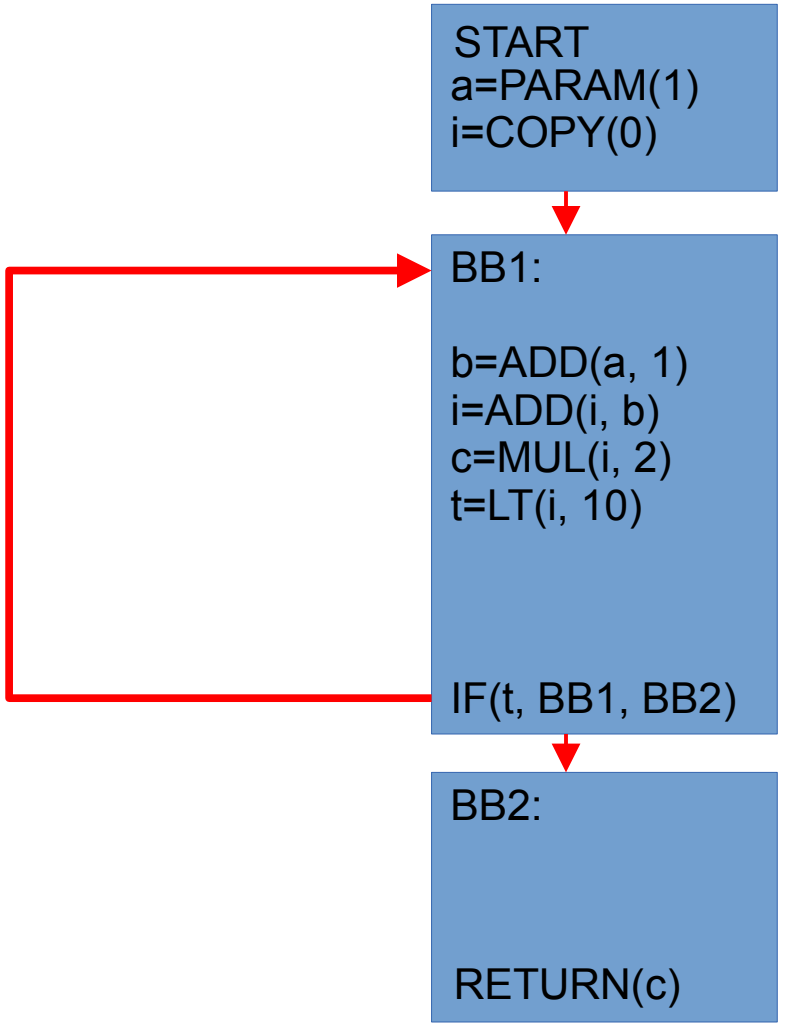
L2:

RETURN(c)

- old classic
- quad – (res, op, op1, op2)
- Dependencies by name
- simple to generate
- labels and “goto”
- hard to optimize

# IR: QUADS + CFG (Control Flow Graph)

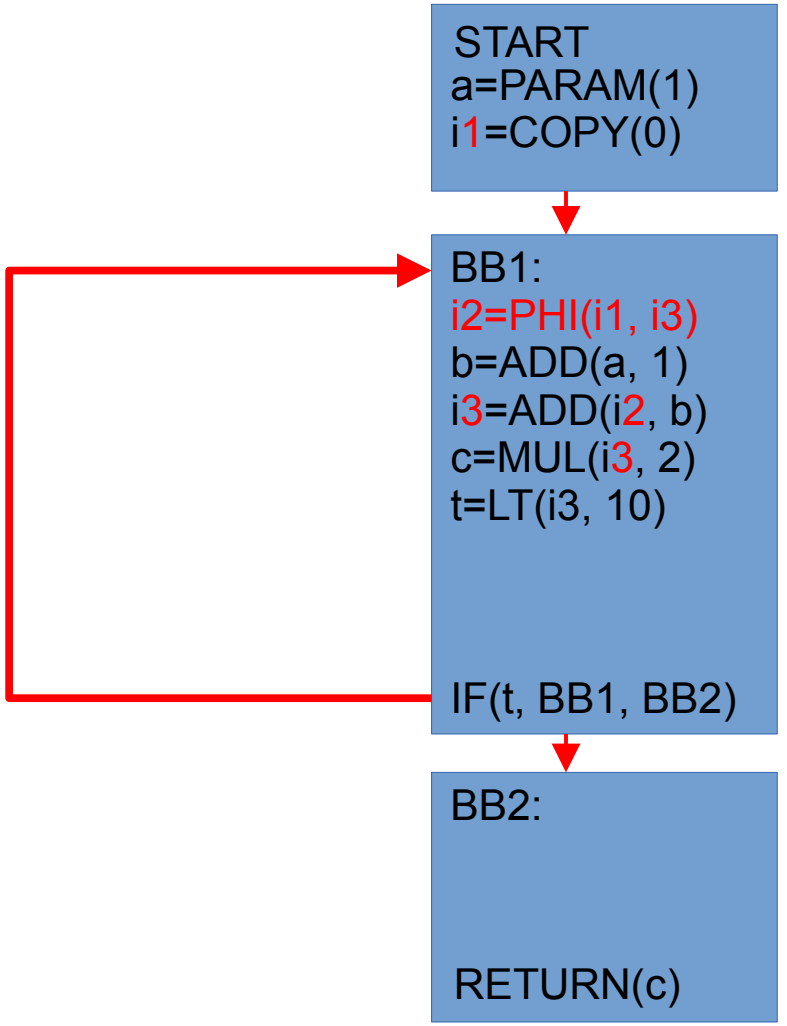
```
int foo(int a) {
    i = 0;
    do {
        b = a + 1;
        i = i + b;
        c = i * 2;
    } while (i < 10);
    return c;
}
```




- old classic
- Function split into basic block
- no control split or merge inside BB
- Control represented by edges
- No labels, no “goto”
- Data dependencies are the same

# IR: CFG + SSA (Static Single Assignment Form)

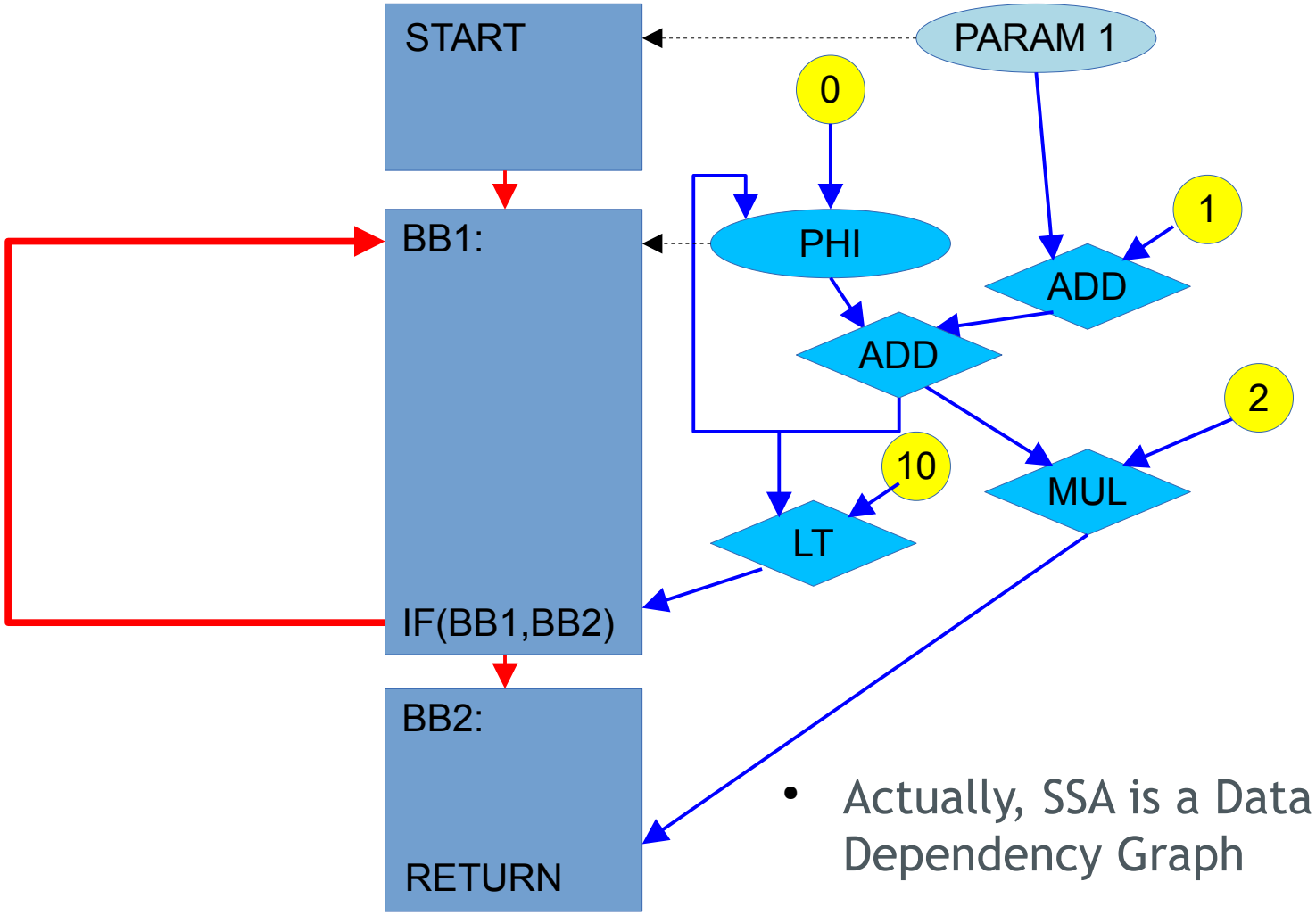
```
int foo(int a) {
    i = 0;
    do {
        b = a + 1;
        i = i + b;
        c = i * 2;
    } while (i < 10);
    return c;
}
```




- SSA - new classic
- Each variable may be assigned only once
- Assignment to the same variable creates a new version of variable (or a new name)
- Phi() functions may be added at start of BB to create a new version of variable
- Only direct data dependencies
- No anti-dependencies

# IR: CFG + SSA (Data Dependency Graph)

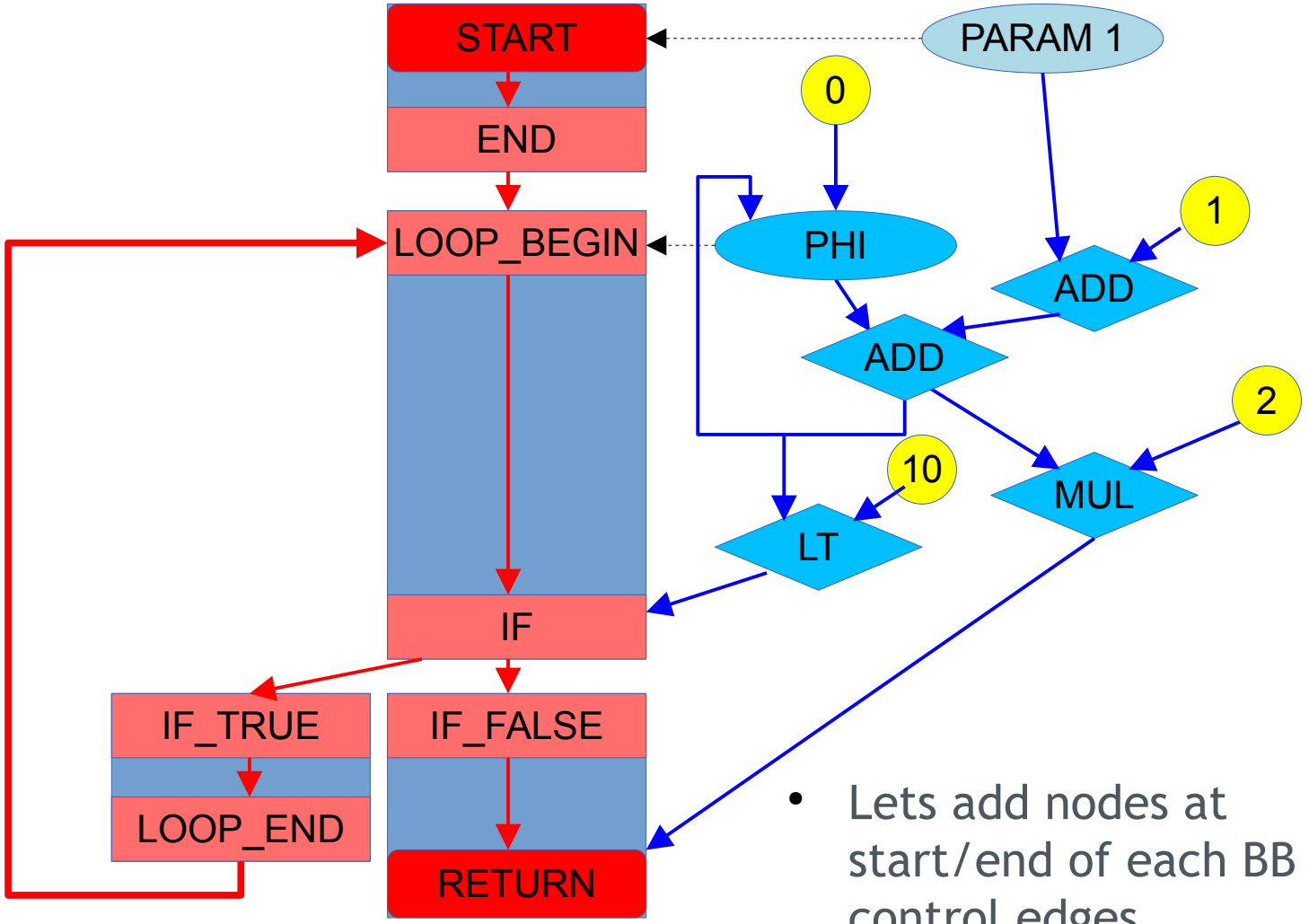
```
int foo(int a) {
    i = 0;
    do {
        b = a + 1;
        i = i + b;
        c = i * 2;
    } while (i < 10);
    return c;
}
```




- Actually, SSA is a Data Dependency Graph

# IR: Almost Sea of Nodes

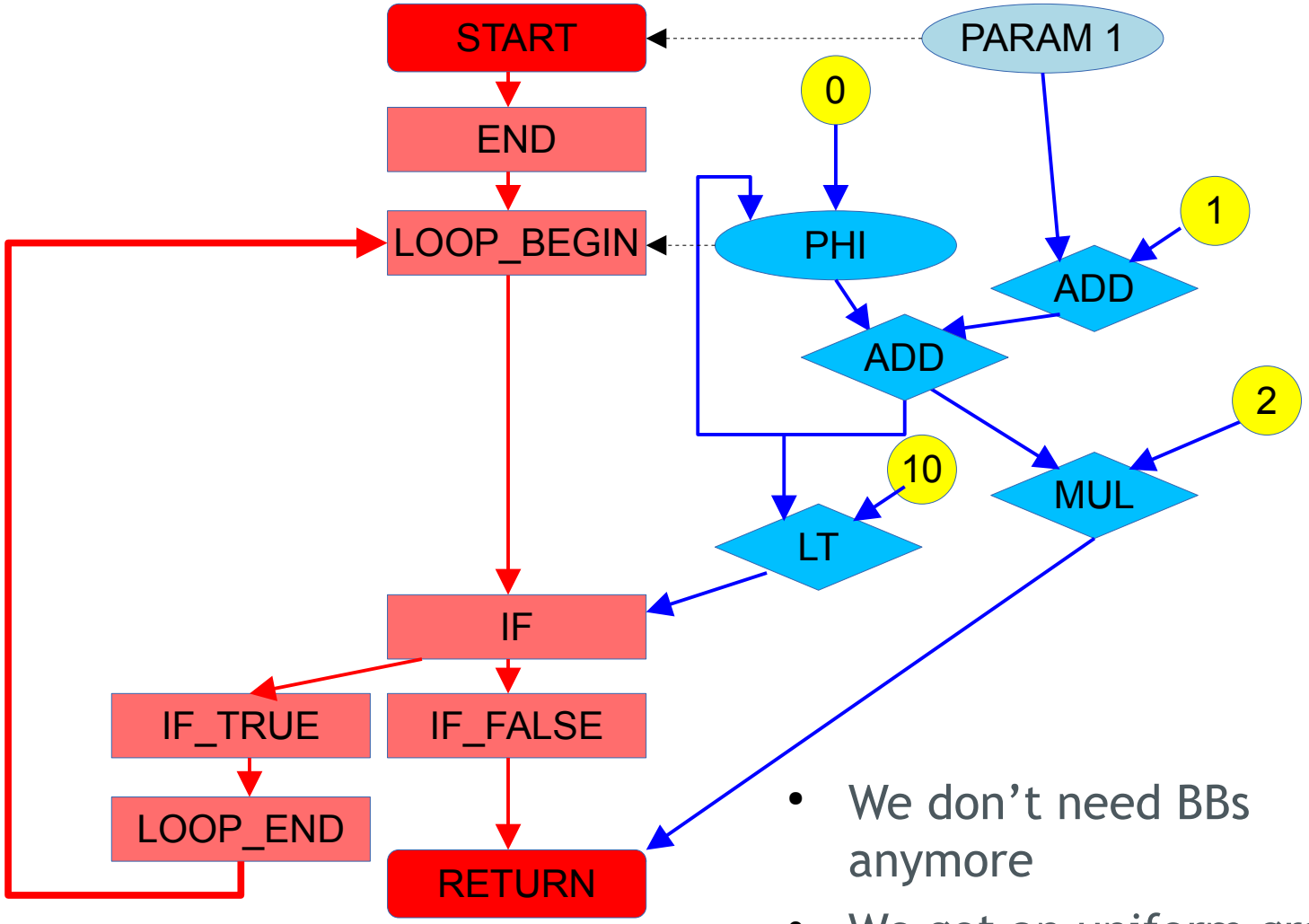
```
int foo(int a) {
    i = 0;
    do {
        b = a + 1;
        i = i + b;
        c = i * 2;
    } while (i < 10);
    return c;
}
```




- Lets add nodes at start/end of each BB and control edges

# IR: Sea of Nodes

```
int foo(int a) {
    i = 0;
    do {
        b = a + 1;
        i = i + b;
        c = i * 2;
    } while (i < 10);
    return c;
}
```

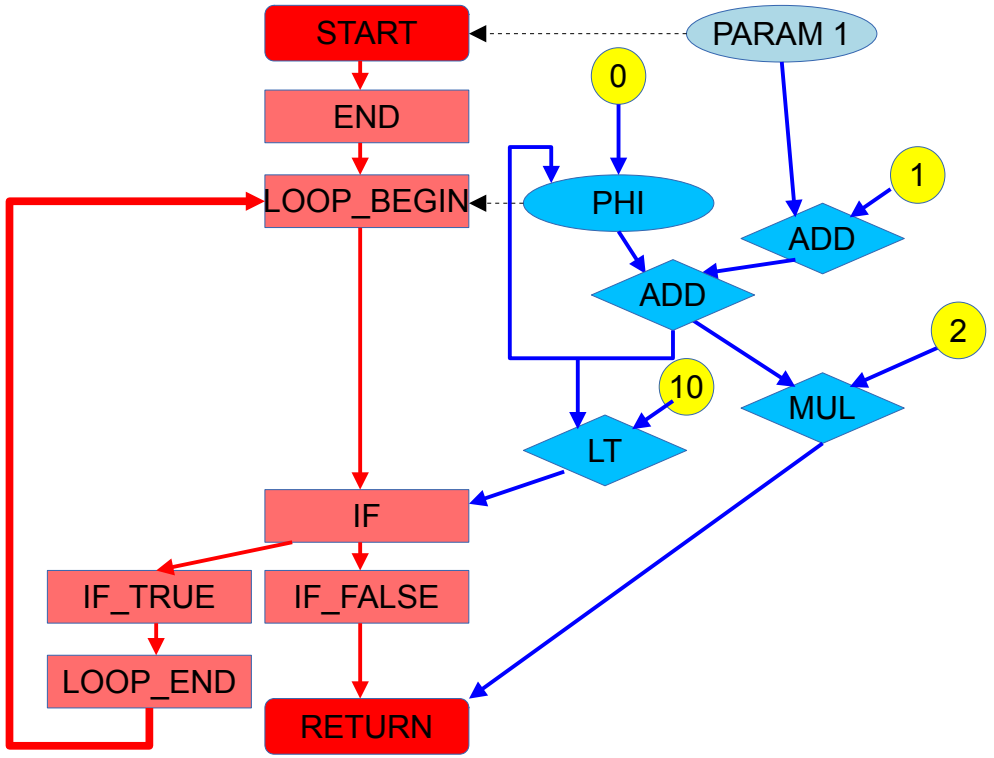



- We don’t need BBs anymore
- We got an uniform graph

# IR: Sea of Nodes

- A single uniform graph representation (only nodes and edges)
- Control dependency edges (red) define the control flow
- Data dependency edges (blue) define the data flow
- Data dependencies are represented in SSA form
- Most data nodes are floating (no specific “place”)
- Some data nodes are “pinned” to control (dashed)
- Nodes may depend on both control and data
- Nodes may produce multiple controls and data
- No basic blocks, no critical edges
- Graph may be traversed in both directions

- Sea of Nodes was first proposed by Clifford Click
- Used in Java HotSpot Server Compiler, V8 TurboFan

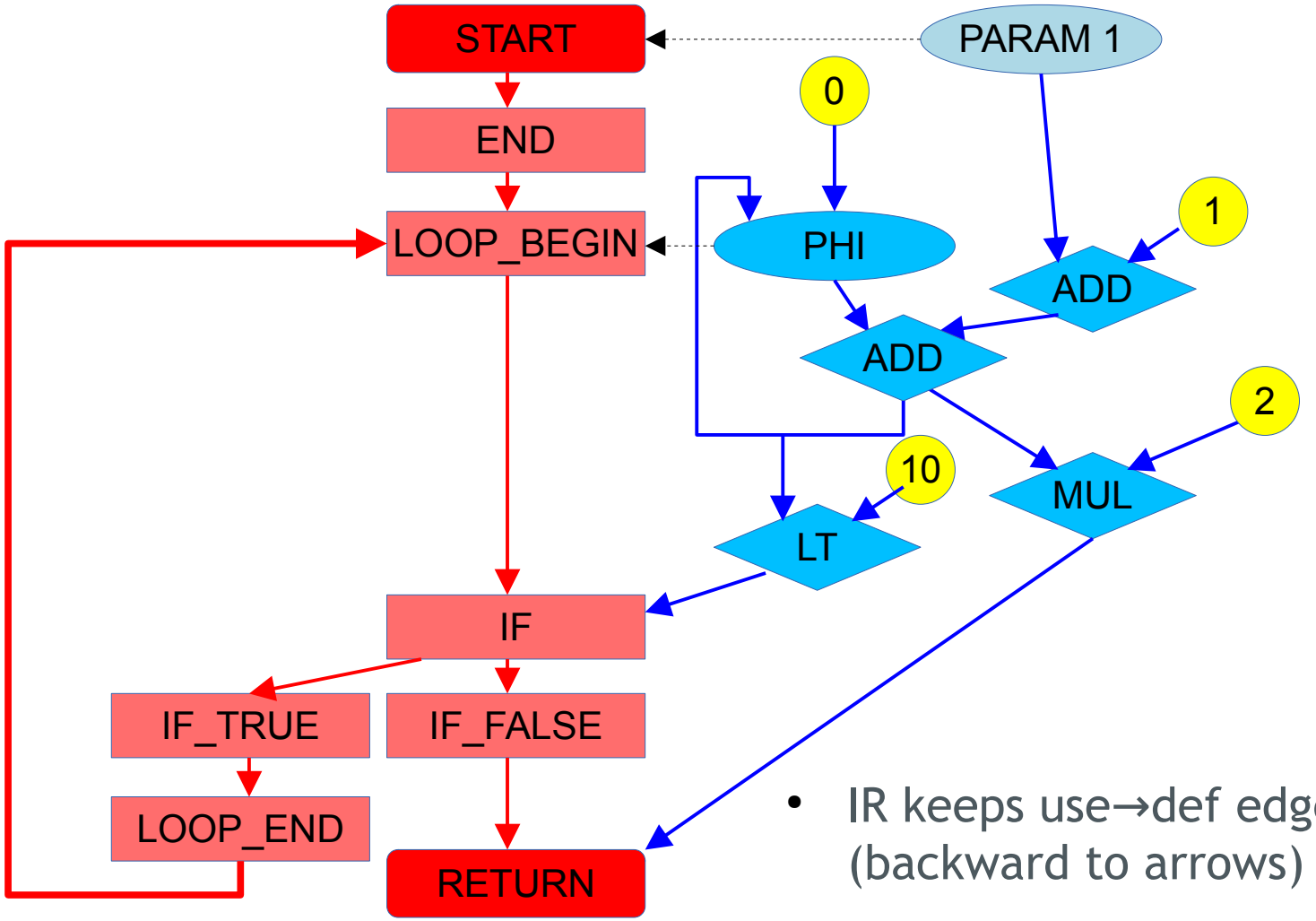


# IR Nodes

- Data Nodes
  - Comparison EQ, NE, LT, LE, GE, GT, ULT, ULE, UGE, UGT (U - unsigned)
  - Math ADD, SUB, MUL, DIV, MOD, NEG, ABS
  - Type conversion SEXT, ZEXT, TRUNC, BITCAST, INT2FP, FP2INT
  - Overflow checking math ADD_OV, SUB_OV, MUL_OV, OVERFLOW
  - Bit-wise and shift ops NOT, OR, AND, XOR, SHL, SHR, SAR, ROL, ROR, BSWAP
  - Branch less conditions MIN(src1, src2), MAX(src1, src2), COND(cond, src1, src2)
  - Miscellaneous PHI, COPY, PI, PARAM, VAR
- Control Nodes
  - Control start START, BEGIN, IF_TRUE/FALSE, CASE_VAL/DEFALT, MERGE, LOOP_BEGIN
  - Control end END, LOOP_END, IF, SWITCH, RETURN, IJMP, UNREACHABLE
- Memory Nodes (Control and Data)
  - Call CALL(ctrl, func, arg1, … argN), TAILCALL
  - Memory access LOAD, STORE, RLOAD, RSTORE, VLOAD, VSTORE, VADDR, ALLOCA, AFREE, TLS
- JIT specific GUARD, GUARD_NOT, SNAPSHOT, EXITCALL, ENTRY

# IR: Physical Representation

| | | | | | |
|---|---|---|---|---|---|
| -7 | | I32 | | | 10 |
| -6 | | I32 | | | 2 |
| -5 | | I32 | | | 1 |
| -4 | | I32 | | | 0 |
| -3 | | BOOL | | | TRUE |
| -2 | | BOOL | | | FALSE |
| -1 | | ADDR | | | NULL |
| | OP | TYPE | op1 | op2 | op3 |
| 1 | START | | 14 | | |
| 2 | PARAM | I32 | 1 | «a» | 1 |
| 3 | END | | 1 | | |
| 4 | LOOP_BEGIN | | 3 | 12 | |
| 5 | PHI | I32 | 4 | -4 | 7 |
| 6 | ADD | i32 | 2 | -5 | |
| 7 | ADD | I32 | 5 | 6 | |
| 8 | MUL | I32 | 7 | -6 | |
| 9 | LT | BOOL | 7 | -7 | |
| 10 | IF | | 4 | 9 | |
| 11 | IF_TRUE | | 10 | | |
| 12 | LOOP_END | | 11 | | |
| 13 | IF_FALSE | | 10 | | |
| 14 | RETURN | | 13 | 8 | |




- IR keeps use→def edges (backward to arrows)

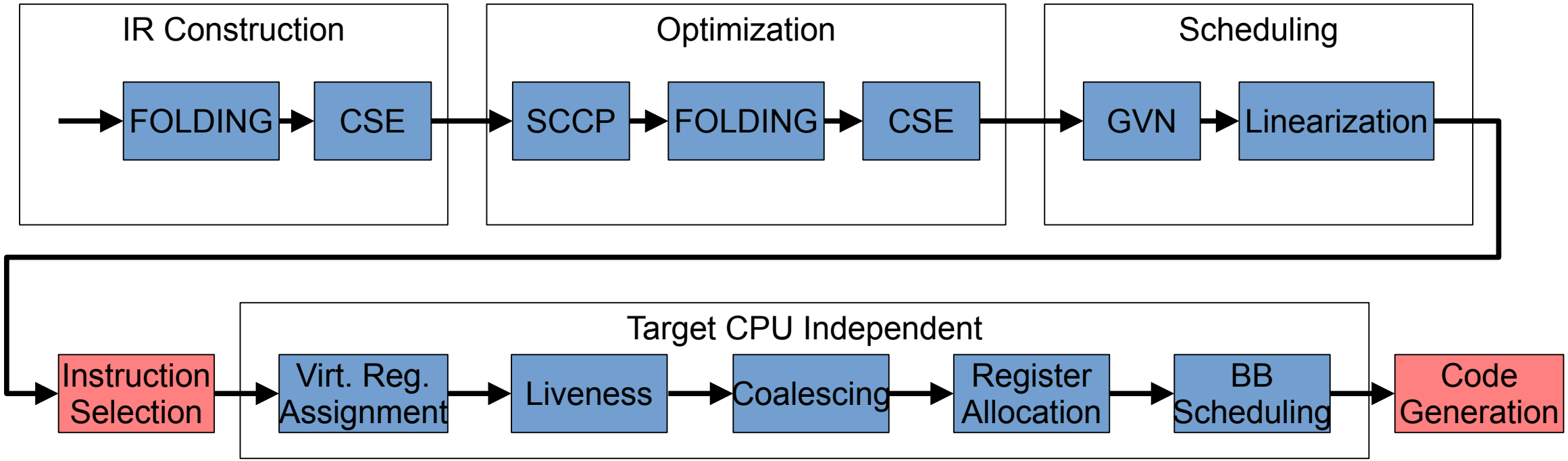
# IR: Physical Representation

| | | | | | |
|---|---|---|---|---|---|
| -7 | | I32 | | | 10 |
| -6 | | I32 | | | 2 |
| -5 | | I32 | | | 1 |
| -4 | | I32 | | | 0 |
| -3 | | BOOL | | | TRUE |
| -2 | | BOOL | | | FALSE |
| -1 | | ADDR | | | NULL |
| | OP | TYPE | op1 | op2 | op3 |
| 1 | START | | 14 | | |
| 2 | PARAM | I32 | 1 | «a» | 1 |
| 3 | END | | 1 | | |
| 4 | LOOP_BEGIN | | 3 | 12 | |
| 5 | PHI | I32 | 4 | -4 | 7 |
| 6 | ADD | i32 | 2 | -5 | |
| 7 | ADD | I32 | 5 | 6 | |
| 8 | MUL | I32 | 7 | -6 | |
| 9 | LT | BOOL | 7 | -7 | |
| 10 | IF | | 4 | 9 | |
| 11 | IF_TRUE | | 10 | | |
| 12 | LOOP_END | | 11 | | |
| 13 | IF_FALSE | | 10 | | |
| 14 | RETURN | | 13 | 8 | |

- A single two directional growing table
- Each row represents a node
- Constants are added at the top, others at the bottom
- Each row takes 16-bytes
- Data nodes are typed
- Use→def edges represented by indexes (4-bytes int)
- Constants have negative indexes, instructions - positive
- Order of rows doesn’t matter
- Some nodes may have multiple inputs MERGE, CALL, PHI (and use few sequential rows)
- Some edges are special (14 in START - ref to terminator)
- Def→use lists are maintained separately

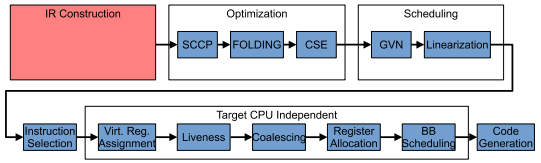
- Influenced by LuaJIT implementation by Mike Pall

# IR Compilation Pipeline




- The same IR is used during all phases of compilation
- No lowering, no SSA deconstruction
- Most passes are platform independent

# IR Builder API (ir_builder.h)

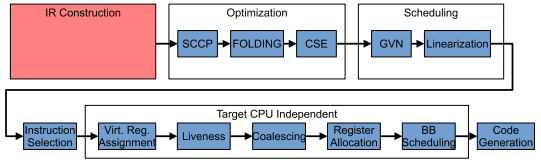



```
int foo(int a) {
    i = 0;
    do {
        b = a + 1;
        i = i + b;
        c = i * 2;
    } while (i < 10);
    return c;
}
```

```
int gen_foo(ir_ctx *ctx) {
    ir_START();
    ir_ref a = ir_PARAM_I32("a", 1);
    ir_ref i_1 = ir_COPY_I32(ir_CONST_i32(0));
    ir_ref end = ir_END();
    ir_ref loop = ir_LOOP_BEGIN(end);
        ir_ref i_2 = ir_PHI_2(IR_I32, i_1, IR_UNUSED);
        ir_ref b = ir_ADD_I32(a, ir_CONST_i32(1));
        ir_ref i_3 = ir_ADD_I32(i_2, b);
        ir_ref c = ir_MUL_I32(i_3, ir_CONST_i32(2));
        ir_ref cond = ir_IF(ir_LT(i_3, ir_CONST_i32(10));
            ir_IF_TRUE(cond);
                end = ir_LOOP_END();
                /* close loop */
                ir_MERGE_SET_OP(loop, 2, end);
                ir_PHI_SET_OP(i_2, 2, i_3);
            ir_IF_FALSE(cond);
                ir_RETURN(c);
}
```

- The code on the right creates IR for the function on the left

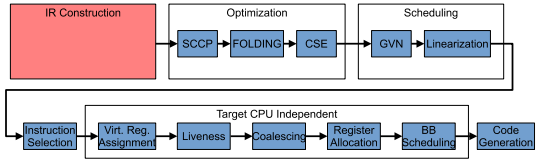
# IR Low-Level Construction API (ir.h)




```
int gen_foo(ir_ctx *ctx) {
    ir_ref start = ir_emit0(ctx, IR_START);
    ir_ref a = ir_param(ctx, IR_I32, start, "a", 1);
    ir_ref i_1 = ir_fold1(ctx, IR_OPT(IR_COPY, IR_I32), ir_const_i32(ctx, 0));
    ir_ref end = ir_emit1(ctx, IR_END, start);
    ir_ref loop = ir_emit2(ctx, IR_LOOP_BEGIN, end, IR_UNUSED);
      ir_ref i_2 = ir_fold3(ctx, IR_OPT(IR_PHI, IR_I32), loop, i_1, IR_UNUSED);
      ir_ref b = ir_fold2(ctx, IR_OPT(IR_ADD, IR_I32), a, ir_const_i32(ctx, 1));
      ir_ref i_3 = ir_fold2(ctx, IR_OPT(IR_ADD, IR_I32), i_2, b);
      ir_ref c = ir_fold2(ctx, IR_OPT(IR_MUL, IR_I32), i_3, ir_const_i32(ctx, 2));
      ir_ref cond = ir_emit2(ctx, IR_IF, loop,
          ir_fold2(ctx, IR_OPT(IR_LT, IR_BOOL), i_3, ir_const_i32(ctx, 10));
        ir_ref t = ir_emit1(ctx, IR_IF_TRUE, cond);
          end = ir_emit1(ctx, t);
          /* close loop */
          ir_set_op(ctx, loop, 2, end);
          ir_set_op(ctx, i_2, 3, i_3);
        ir_ref f = ir_emit1(ctx, IR_IF_FALSE, cond);
          ir_ref ret = ir_emit2(ctx, IR_RETURN, f, c);
          ir_set_op(ctx, start, 1, ret); /* link RETURN to START */
}
```

- The code from the previous slide with expanded macro

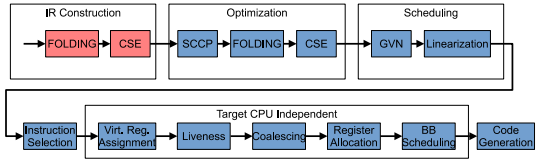
# IR Low-Level Construction API (ir.h)




- **ir_const_...()** - create new constant nodes or reuse the existing ones
- **ir_emit?()** - add new data or control nodes
- **ir_fold?()** - pass new nodes to the folding engine
  - Constants folding
  - Copy propagation
  - Algebraic simplifications
  - Re-association
  - Common sub-expression elimination
  - Fallback to ir_emit?()
- **ir_set_op()** - back-patching
- **ir_param()**, **ir_var()** - helpers on top of **ir_emit()** that accept string argument

- The constructed IR graph may be verified by **ir_check()**

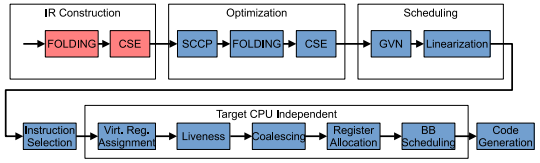
# IR Folding Engine



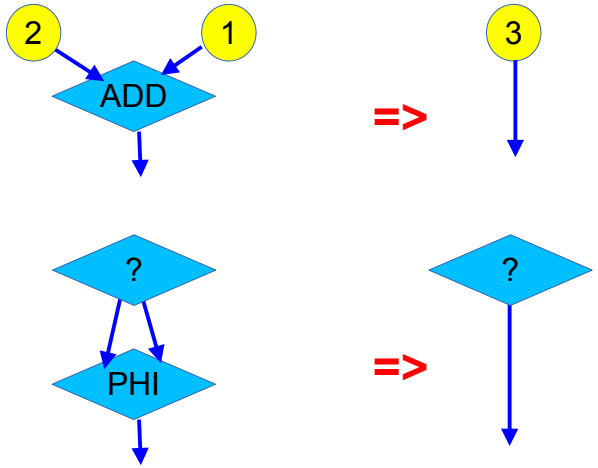

- Performs local optimizations on the fly, during IR construction – **ir_fold?()**
- Optimization Rules are defined in declarative style in **ir_fold.h**
- CSE is performed by searching an identical node above (through per-op skip lists)
- The scope of “local” optimizations is not limited to a single basic-block
- All nodes must be defined before their usage (except PHI inputs), or scope must be limited through **ir_ctx.fold_cse_limit**

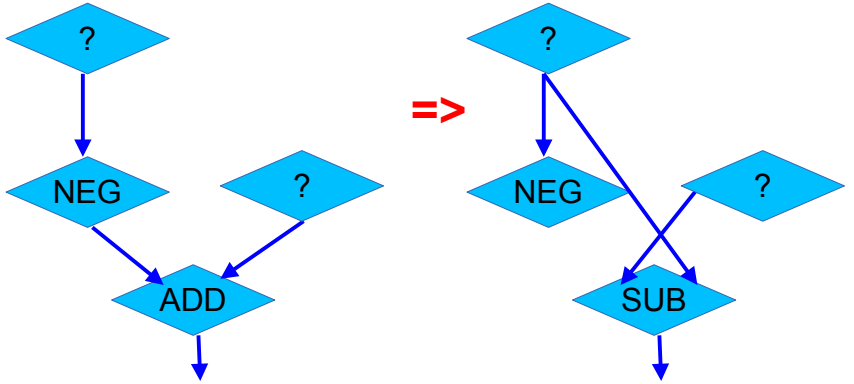
- Repeats LuaJIT implementation by Mike Pall

# IR Folding Rules (example rules from ir_fold.h)

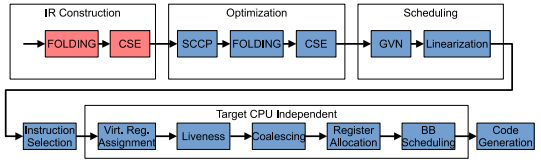



```
/* Constant Folding */
IR_FOLD(ADD(C_DOUBLE, C_DOUBLE)) {
    IR_FOLD_CONST_D(op1_insn->val.d + op2_insn->val.d);
}

/* Copy Propagation */
IR_FOLD(PHI(_, _, _)) {
    if (op2 == op3) {
        IR_FOLD_COPY(op2);
    }
    IR_FOLD_EMIT; /* skip CSE */
}

/* Algebraic simplifications */
IR_FOLD(ADD(NEG, _)) { /* (-a) + b => b - a */
    opt++; /* ADD -> SUB */
    op1 = op2;
    op2 = op1_insn->op1;
    IR_FOLD_RESTART;
}
```



# IR Folding Engine in Action




- `IR_FOLD(ADD(NEG, _))` rule

```
ir_START();
ir_ref a = ir_PARAM_I32("a", 1);
ir_ref b = ir_PARAM_I32("b", 2);
ir_ref c = ir_NEG_I32(a);
ir_ref d = ir_ADD_I32(c, b);
```

| | OP | TYPE | op1 | op2 | op3 |
|---|---|---|---|---|---|
| 1 | START | | 14 | | |
| 2 | PARAM | I32 | 1 | «a» | 1 |
| 3 | PARAM | I32 | 1 | «b» | 2 |
| 4 | NEG | I32 | 2 | | |
| 5 | SUB | I32 | 3 | 2 | |

Dead code

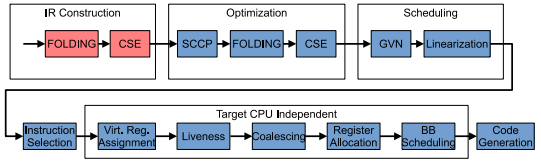
- CSE (common Sub-Expression Elimination)

```
ir_START();
ir_ref a = ir_PARAM_I32("a", 1);
ir_ref b = ir_PARAM_I32("b", 2);
ir_ref c = ir_ADD_I32(a, b);
ir_ref d = ir_ADD_I32(a, b);
ir_ref e = ir_ADD_I32(c, d);
```

| | OP | TYPE | op1 | op2 | op3 |
|---|---|---|---|---|---|
| 1 | START | | 14 | | |
| 2 | PARAM | I32 | 1 | «a» | 1 |
| 3 | PARAM | I32 | 1 | «b» | 2 |
| 4 | ADD | I32 | 2 | 3 | |
| 5 | ADD | I32 | 4 | 4 | |

| Skip Lists |
|---|
| ... |
| ADD |
| ... |

# IR Folding (“local” optimizations across BBs)




```
if (…) {
    c = a + b;
} else {
    d = a + b + 42;
}
```

*Folding CSE*

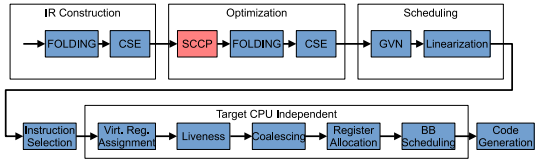
```
if (…) {
    c = a + b;
} else {
    d = c + 42;
}
```

*Global Code Motion*

```
c = a + b;
if (…) {

} else {
    d = c + 42;
}
```

- This code is wrong, but all data nodes are floating and they are properly placed by GCM later

# SCCP - Sparse Conditional Constants Propagation



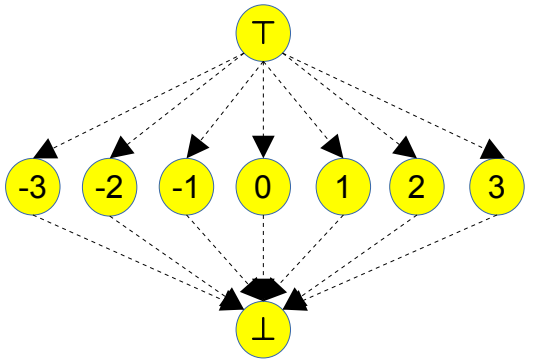

- M. N. Wegman and F. K. Zadeck. "Constant propagation with conditional branches"
- Values are defined as a 3-level Lattice

  T – TOP, any constant

  ⊥ - BOTTOM, not a constant

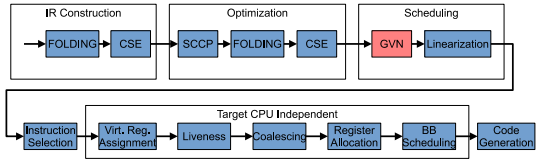
  Rules:

  $$ANY \bigcap T => ANY$$
  $$ANY \bigcap \bot => \bot$$
  $$A \bigcap B => A \text{ if } A = B$$
  $$A \bigcap B => \bot \text{ if } A \neq B$$



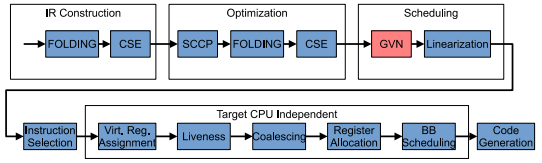

- Original algorithms work on the SSA graph
- It operates on two work-lists: executable control edges and SSA dependency edges
- Adapting the algorithm for Sea of Nodes made it even simple

# GCM – Global Code Motion

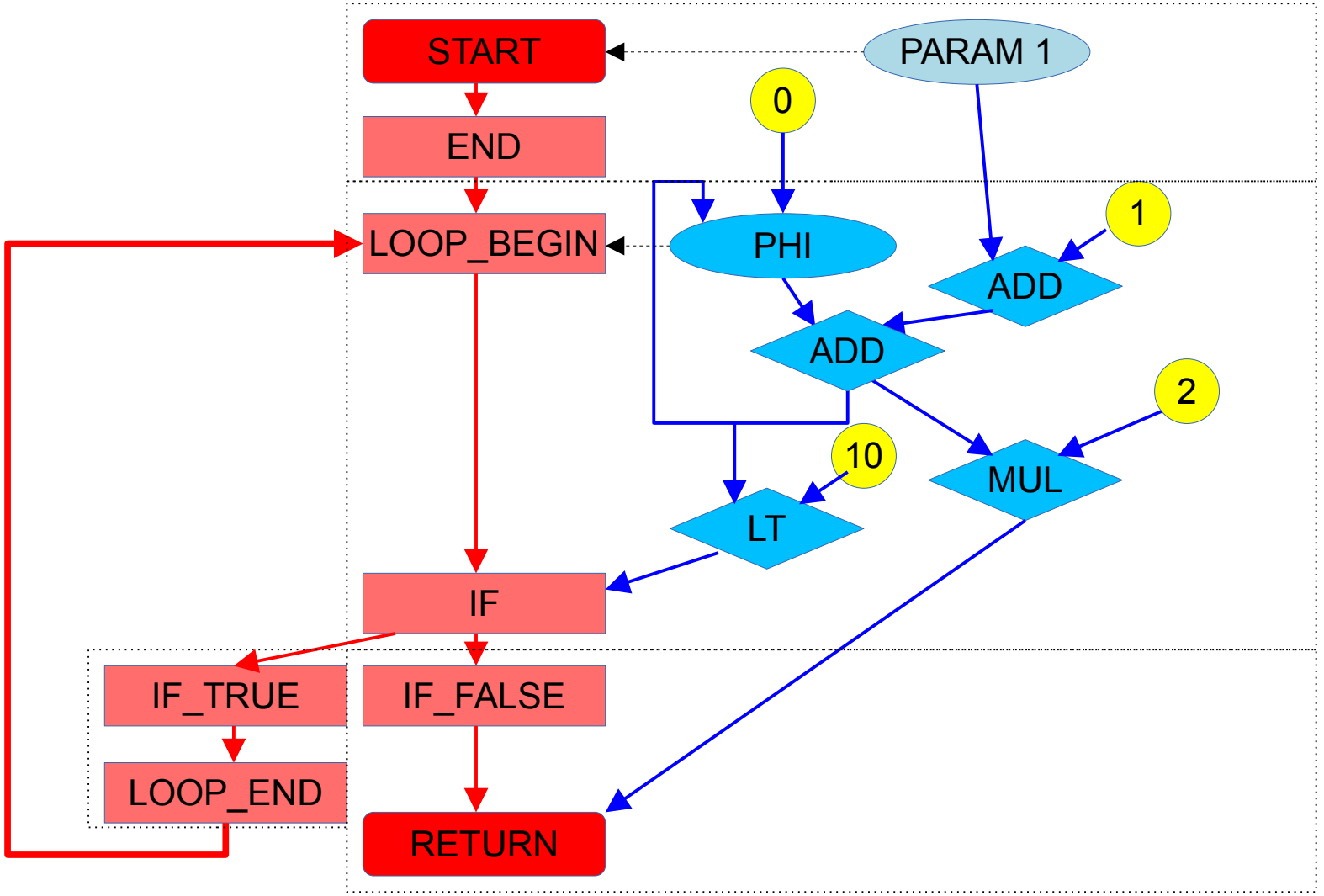



- C. Click. "Global code motion, global value numbering"
- The algorithm was designed especially for Sea-of-Nodes
- It aims to find the best “place” for all floating nodes
- Requires reconstruction of CFG, Dominator Tree, Loop Nesting Forest
- First we traverse the floating nodes and place them as early as possible
  - All the inputs must be defined before
- Then we traverse them once again and move them down
  - into the last common ancestor block
  - into the less nested loop
- LICM is done out of the box
- DCE is not needed. Dead nodes aren’t included into the final schedule.

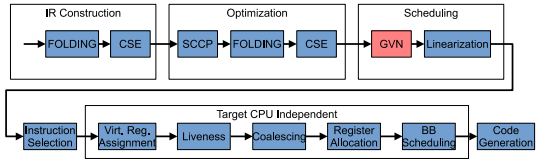
# GCM in Action



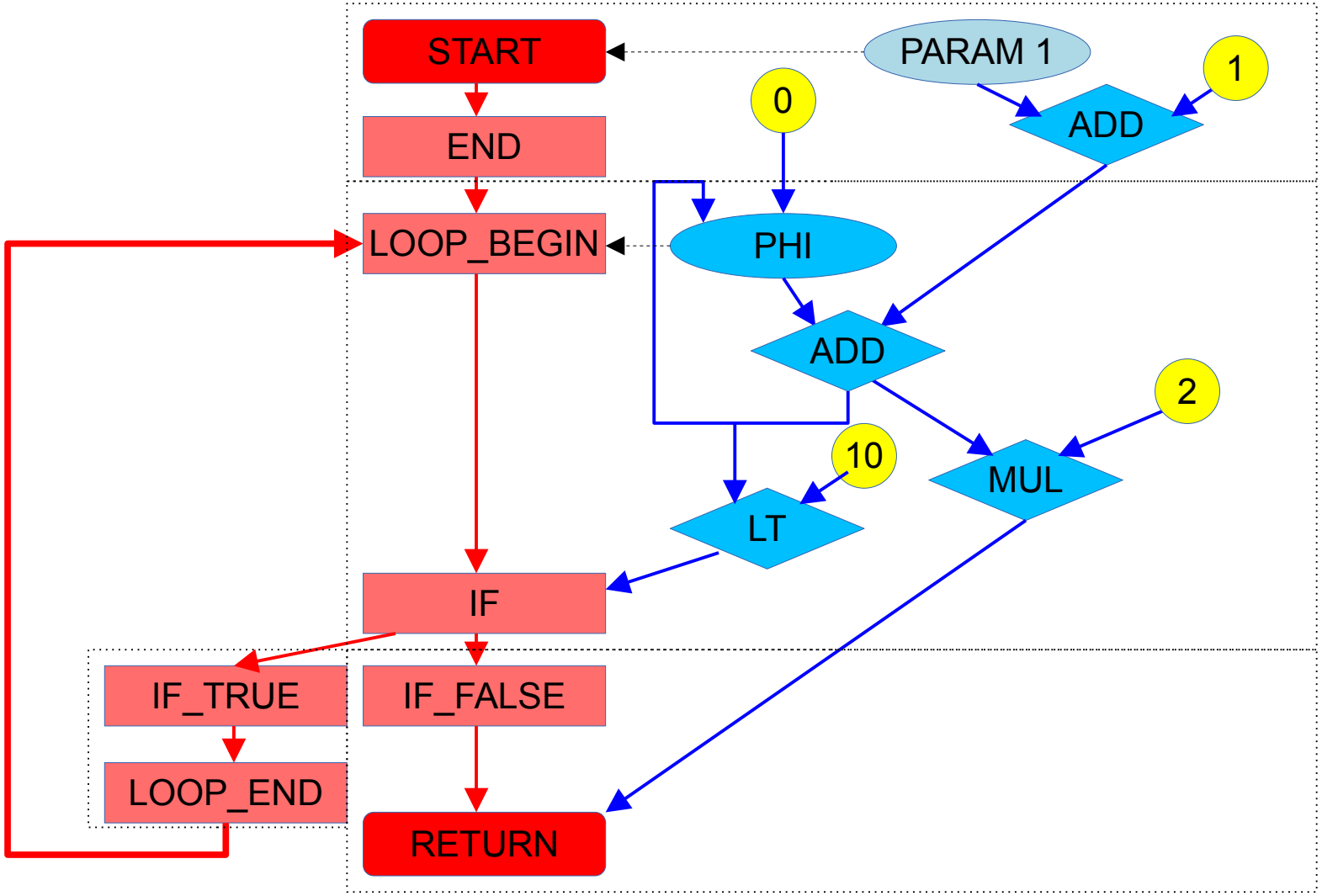

```
int foo(int a) {
    i = 0;
    do {
        b = a + 1;
        i = i + b;
        c = i * 2;
    } while (i < 10);
    return c;
}
```



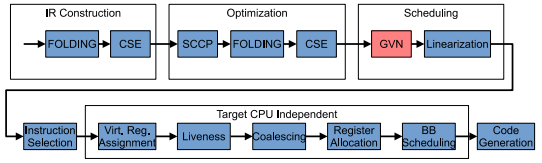
# GCM in Action (Schedule Early - ADD moved up)



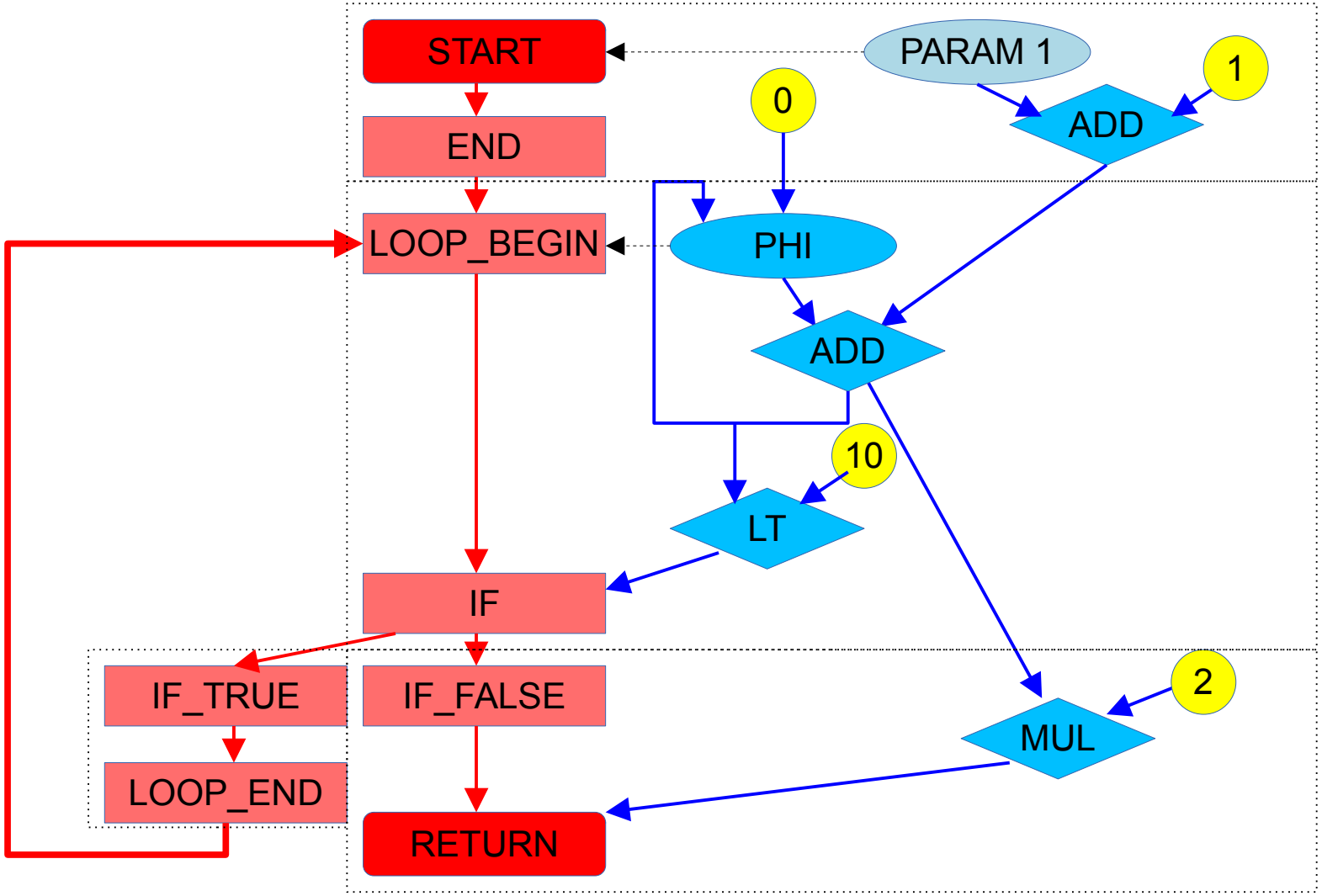

```
int foo(int a) {
    i = 0;
    b = a + 1;
    do {
        i = i + b;
        c = i * 2;
    } while (i < 10);
    return c;
}
```



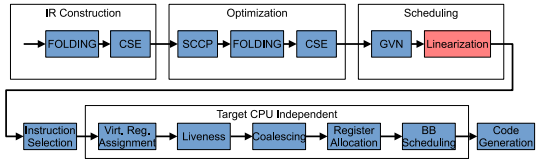
# GCM in Action (Schedule Later - MUL moved down)




```
int foo(int a) {
    i = 0;
    b = a + 1;
    do {
        i = i + b;
    } while (i < 10);
    c = i * 2;
    return c;
}
```



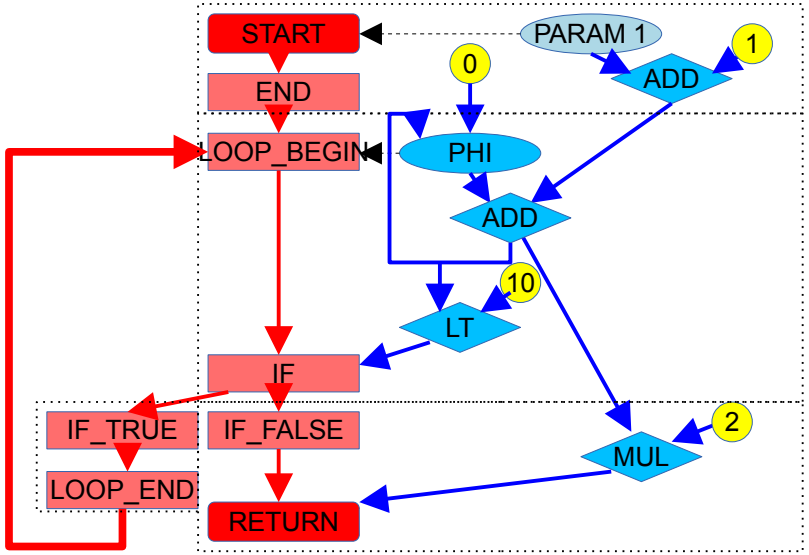
# Linearization (rewriting table in the new order)



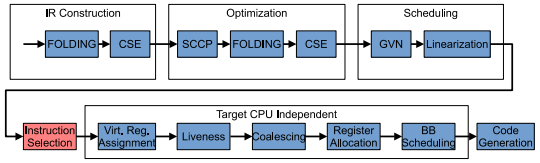

| | | | | | | |
|---|---|---|---|---|---|---|
| -7 | | I32 | | | 10 | |
| -6 | | I32 | | | 2 | |
| -5 | | I32 | | | 1 | |
| -4 | | I32 | | | 0 | |
| -3 | | BOOL | | | TRUE | |
| -2 | | BOOL | | | FALSE | |
| -1 | | ADDR | | | NULL | |
| | OP | TYPE | op1 | op2 | op3 | |
| 1 | START | | 14 | | | |
| 2 | PARAM | I32 | 1 | «a» | 1 | |
| 3 | ADD | I32 | 2 | -5 | | |
| 4 | END | | 1 | | | |
| 5 | LOOP_BEGIN | | 4 | 11 | | |
| 6 | PHI | i32 | 5 | -4 | 7 | |
| 7 | ADD | I32 | 6 | 3 | | |
| 8 | LT | BOOL | 7 | -7 | | |
| 9 | IF | | 5 | 8 | | |
| 10 | IF_TRUE | | 9 | | | |
| 11 | LOOP_END | | 10 | | | |
| 12 | IF_FALSE | | 9 | | | |
| 13 | MUL | I32 | 7 | -6 | | |
| 14 | RETURN | | 12 | 13 | | |

| | | | | | |
|---|---|---|---|---|---|
| | I32 | | | 10 | |
| | I32 | | | 2 | |
| | I32 | | | 1 | |
| | I32 | | | 0 | |
| | BOOL | | | TRUE | |
| | BOOL | | | FALSE | |
| | ADDR | | | NULL | |
| OP | TYPE | op1 | op2 | op3 | BB |
| START | | 14 | | | 1 |
| PARAM | I32 | 1 | «a» | 1 | 1 |
| END | | 1 | | | 1 |
| LOOP_BEGIN | | 3 | 12 | | 2 |
| PHI | I32 | 4 | -4 | 7 | 2 |
| ADD | i32 | 2 | -5 | | 1 |
| ADD | I32 | 5 | 6 | | 2 |
| MUL | I32 | 7 | -6 | | 4 |
| LT | BOOL | 7 | -7 | | 2 |
| IF | | 4 | 9 | | 2 |
| IF_TRUE | | 10 | | | 3 |
| LOOP_END | | 11 | | | 3 |
| IF_FALSE | | 10 | | | 4 |
| RETURN | | 13 | 8 | | 4 |

- GCM only assign nodes to blocks
- Now we rewrite IR table, doing topological sort inside each BB

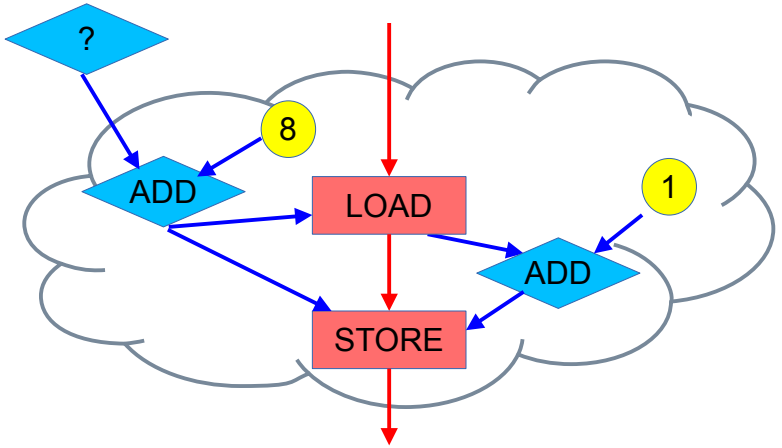


# Instruction Selection



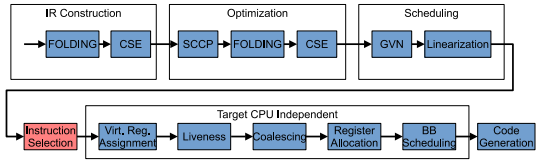

| | | | | | |
|---|---|---|---|---|---|
| -7 | | I32 | | | 10 |
| -6 | | I32 | | | 2 |
| -5 | | I32 | | | 1 |
| -4 | | I32 | | | 0 |
| -3 | | BOOL | | | TRUE |
| -2 | | BOOL | | | FALSE |
| -1 | | ADDR | | | NULL |
| | OP | TYPE | op1 | op2 | op3 |
| 1 | START | | 14 | | |
| 2 | PARAM | I32 | 1 | «a» | 1 |
| 3 | ADD | I32 | 2 | -5 | |
| 4 | END | | 1 | | |
| 5 | LOOP_BEGIN | | 4 | 11 | |
| 6 | PHI | i32 | 5 | -4 | 7 |
| 7 | ADD | I32 | 6 | 3 | |
| 8 | LT | BOOL | 7 | -7 | |
| 9 | IF | | 5 | 8 | |
| 10 | IF_TRUE | | 9 | | |
| 11 | LOOP_END | | 10 | | |
| 12 | IF_FALSE | | 9 | | |
| 13 | MUL | I32 | 7 | -6 | |
| 14 | RETURN | | 12 | 13 | |

- Translate CPU independent instructions into CPU specific
- Cover IR Graph with non-overlapping DAG patterns (tiles)
- We use simple maximal munch approach (non optimal)
- It’s possible to achieve an optimal solution using BURS
- We don’t do any lowering or graph rewriting




=> **incl 8(%reg)**

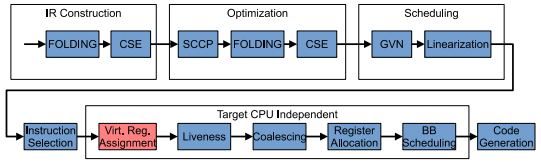
# Instruction Selection




| | | | | | | |
|---|---|---|---|---|---|---|
| -7 | | I32 | | | 10 | |
| -6 | | I32 | | | 2 | |
| -5 | | I32 | | | 1 | |
| -4 | | I32 | | | 0 | |
| -3 | | BOOL | | | TRUE | |
| -2 | | BOOL | | | FALSE | |
| -1 | | ADDR | | | NULL | |
| | OP | TYPE | op1 | op2 | op3 | RULES |
| 1 | START | | 14 | | | |
| 2 | PARAM | I32 | 1 | «a» | 1 | PARAM |
| 3 | ADD | I32 | 2 | -5 | | LEA 1(%src), %dst |
| 4 | END | | 1 | | | |
| 5 | LOOP_BEGIN | | 4 | 11 | | |
| 6 | PHI | i32 | 5 | -4 | 7 | PHI |
| 7 | ADD | I32 | 6 | 3 | | BINOP |
| 8 | LT | BOOL | 7 | -7 | | FUSED \| CMP |
| 9 | IF | | 5 | 8 | | CMP_AND_BRANCH |
| 10 | IF_TRUE | | 9 | | | |
| 11 | LOOP_END | | 10 | | | |
| 12 | IF_FALSE | | 9 | | | |
| 13 | MUL | I32 | 7 | -6 | | LEA (%src, 2), %dst |
| 14 | RETURN | | 12 | 13 | | RETURN_INT |

- We remember an id of the selected pattern for each instruction
- Code Generation **Rules** (4 bytes)
- Some instructions may be **SKIPPED** or **FUSED**
- Code generation for each rule is simple
- We only need to know where the data came from, and where to store the result.

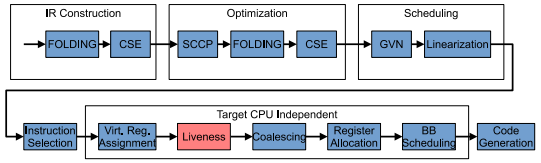
# Virtual Register Assignment




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -7 | | I32 | | | 10 | | |
| -6 | | I32 | | | 2 | | |
| -5 | | I32 | | | 1 | | |
| -4 | | I32 | | | 0 | | |
| -3 | | BOOL | | | TRUE | | |
| -2 | | BOOL | | | FALSE | | |
| -1 | | ADDR | | | NULL | | |
| | OP | TYPE | op1 | op2 | op3 | RULES | VREGS |
| 1 | START | | 14 | | | | |
| 2 | PARAM | I32 | 1 | «a» | 1 | PARAM | R1 |
| 3 | ADD | I32 | 2 | -5 | | LEA 1(%src), %dst | R2 |
| 4 | END | | 1 | | | | |
| 5 | LOOP_BEGIN | | 4 | 11 | | | |
| 6 | PHI | i32 | 5 | -4 | 7 | PHI | R3 |
| 7 | ADD | I32 | 6 | 3 | | BINOP | R4 |
| 8 | LT | BOOL | 7 | -7 | | FUSED \| CMP | |
| 9 | IF | | 5 | 8 | | CMP_AND_BRANCH | |
| 10 | IF_TRUE | | 9 | | | | |
| 11 | LOOP_END | | 10 | | | | |
| 12 | IF_FALSE | | 9 | | | | |
| 13 | MUL | I32 | 7 | -6 | | LEA (%src, 2), %dst | R5 |
| 14 | RETURN | | 12 | 13 | | RETURN_INT | |

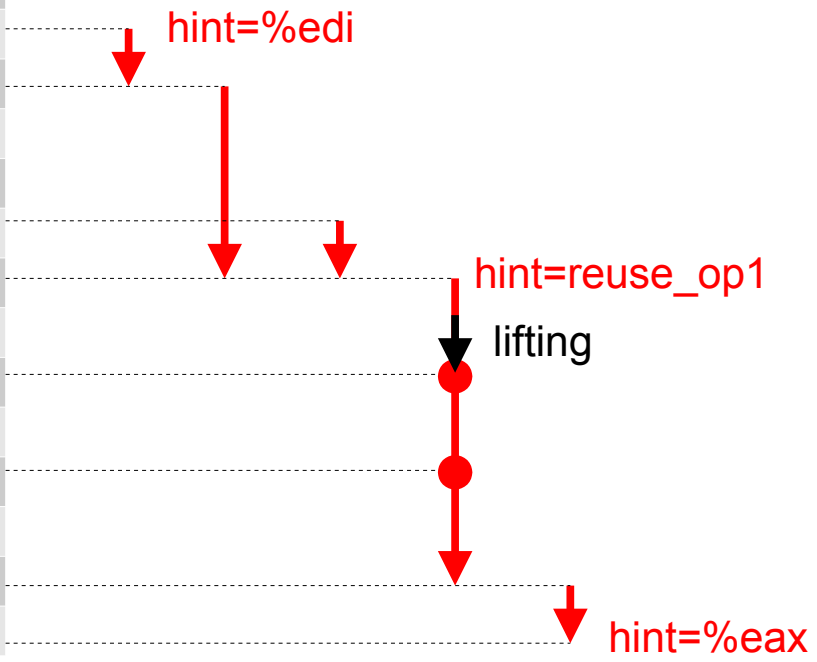
- Allocate an unique virtual register for each **rule** that produces data
- The number of virtual registers is unlimited

# Build Live Intervals

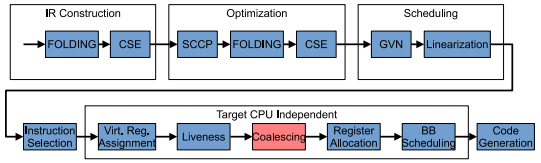



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -7 | | I32 | | | 10 | | |
| -6 | | I32 | | | 2 | | |
| -5 | | I32 | | | 1 | | |
| -4 | | I32 | | | 0 | | |
| -3 | | BOOL | | | TRUE | | |
| -2 | | BOOL | | | FALSE | | |
| -1 | | ADDR | | | NULL | | |
| | OP | TYPE | op1 | op2 | op3 | RULES | VREGS |
| 1 | START | | 14 | | | | |
| 2 | PARAM | I32 | 1 | «a» | 1 | PARAM | R1 |
| 3 | ADD | I32 | 2 | -5 | | LEA 1(%src), %dst | R2 |
| 4 | END | | 1 | | | | |
| 5 | LOOP_BEGIN | | 4 | 11 | | | |
| 6 | PHI | i32 | 5 | -4 | 7 | PHI | R3 |
| 7 | ADD | I32 | 6 | 3 | | BINOP | R4 |
| 8 | LT | BOOL | 7 | -7 | | FUSED \| CMP | |
| 9 | IF | | 5 | 8 | | CMP_AND_BRANCH | |
| 10 | IF_TRUE | | 9 | | | | |
| 11 | LOOP_END | | 10 | | | | |
| 12 | IF_FALSE | | 9 | | | | |
| 13 | MUL | I32 | 7 | -6 | | LEA (%src, 2), %dst | R5 |
| 14 | RETURN | | 12 | 13 | | RETURN_INT | |

- Build live intervals, ranges, lists of use positions
- Use target constraints for rules to add hints, restrictions, etc

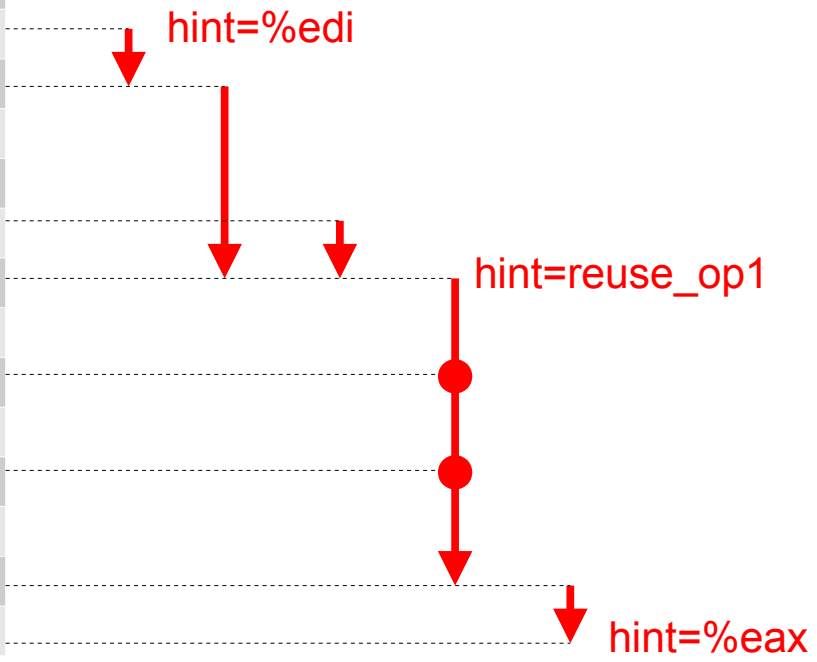


# Coalescing

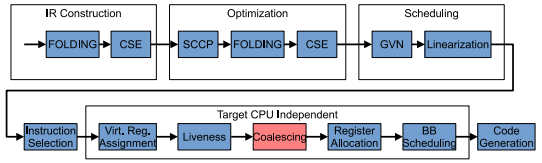



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -7 | | I32 | | | 10 | | |
| -6 | | I32 | | | 2 | | |
| -5 | | I32 | | | 1 | | |
| -4 | | I32 | | | 0 | | |
| -3 | | BOOL | | | TRUE | | |
| -2 | | BOOL | | | FALSE | | |
| -1 | | ADDR | | | NULL | | |
| | OP | TYPE | op1 | op2 | op3 | RULES | VREGS |
| 1 | START | | 14 | | | | |
| 2 | PARAM | I32 | 1 | «a» | 1 | PARAM | R1 |
| 3 | ADD | I32 | 2 | -5 | | LEA 1(%src), %dst | R2 |
| 4 | END | | 1 | | | | |
| 5 | LOOP_BEGIN | | 4 | 11 | | | |
| 6 | PHI | i32 | 5 | -4 | 7 | PHI | R3 |
| 7 | ADD | I32 | 6 | 3 | | BINOP | R4 |
| 8 | LT | BOOL | 7 | -7 | | FUSED \| CMP | |
| 9 | IF | | 5 | 8 | | CMP_AND_BRANCH | |
| 10 | IF_TRUE | | 9 | | | | |
| 11 | LOOP_END | | 10 | | | | |
| 12 | IF_FALSE | | 9 | | | | |
| 13 | MUL | I32 | 7 | -6 | | LEA (%src, 2), %dst | R5 |
| 14 | RETURN | | 12 | 13 | | RETURN_INT | |

- We check virtual registers that are involved as inputs and outputs of **PHI** instructions.
- For each pair of virtual registers we check if their live ranges overlap

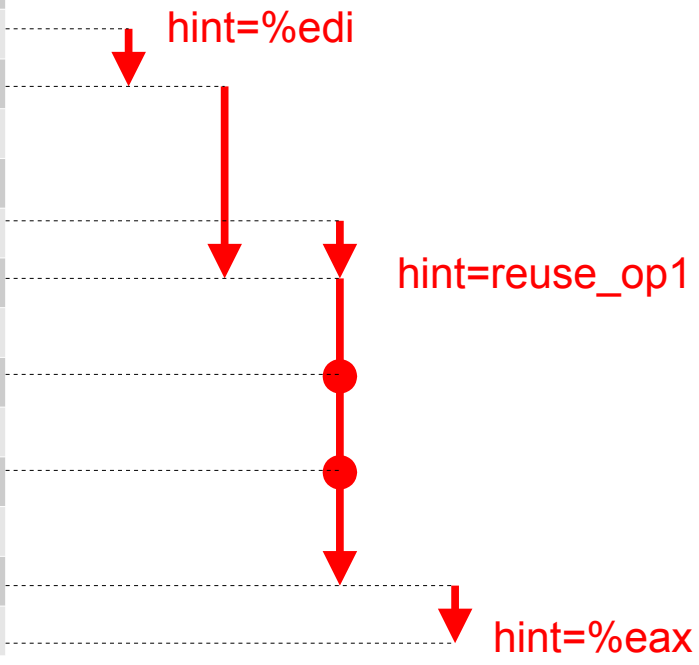


# Coalescing

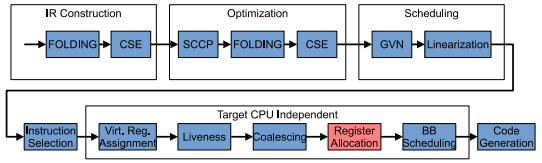



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -7 | | I32 | | | 10 | | |
| -6 | | I32 | | | 2 | | |
| -5 | | I32 | | | 1 | | |
| -4 | | I32 | | | 0 | | |
| -3 | | BOOL | | | TRUE | | |
| -2 | | BOOL | | | FALSE | | |
| -1 | | ADDR | | | NULL | | |
| | OP | TYPE | op1 | op2 | op3 | RULES | VREGS |
| 1 | START | | 14 | | | | |
| 2 | PARAM | I32 | 1 | «a» | 1 | PARAM | R1 |
| 3 | ADD | I32 | 2 | -5 | | LEA 1(%src), %dst | R2 |
| 4 | END | | 1 | | | | |
| 5 | LOOP_BEGIN | | 4 | 11 | | | |
| 6 | PHI | i32 | 5 | -4 | 7 | PHI | R3 |
| 7 | ADD | I32 | 6 | 3 | | BINOP | |
| 8 | LT | BOOL | 7 | -7 | | FUSED \| CMP | |
| 9 | IF | | 5 | 8 | | CMP_AND_BRANCH | |
| 10 | IF_TRUE | | 9 | | | | |
| 11 | LOOP_END | | 10 | | | | |
| 12 | IF_FALSE | | 9 | | | | |
| 13 | MUL | I32 | 7 | -6 | | LEA (%src, 2), %dst | R4 |
| 14 | RETURN | | 12 | 13 | | RETURN_INT | |

- Non-overlapping intervals may be combined to simplify LSRA and eliminate SSA deconstruction moves

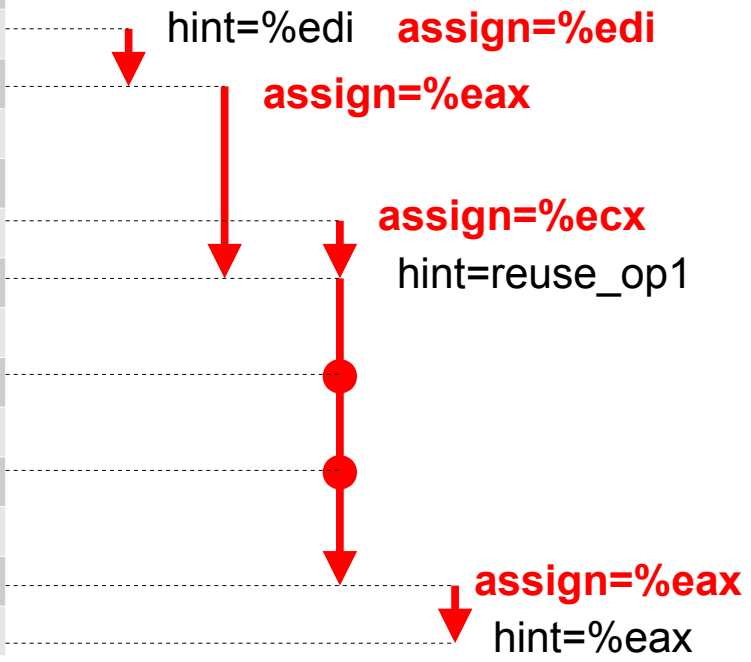


# Linear Scan Register Allocation

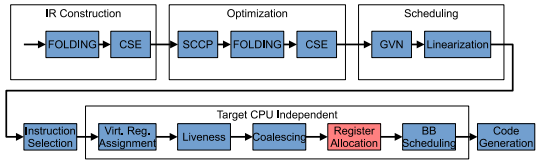



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -7 | | I32 | | | 10 | | |
| -6 | | I32 | | | 2 | | |
| -5 | | I32 | | | 1 | | |
| -4 | | I32 | | | 0 | | |
| -3 | | BOOL | | | TRUE | | |
| -2 | | BOOL | | | FALSE | | |
| -1 | | ADDR | | | NULL | | |
| | OP | TYPE | op1 | op2 | op3 | RULES | VREGS |
| 1 | START | | 14 | | | | |
| 2 | PARAM | I32 | 1 | «a» | 1 | PARAM | R1 |
| 3 | ADD | I32 | 2 | -5 | | LEA 1(%src), %dst | R2 |
| 4 | END | | 1 | | | | |
| 5 | LOOP_BEGIN | | 4 | 11 | | | |
| 6 | PHI | i32 | 5 | -4 | 7 | PHI | R3 |
| 7 | ADD | I32 | 6 | 3 | | BINOP | |
| 8 | LT | BOOL | 7 | -7 | | FUSED \| CMP | |
| 9 | IF | | 5 | 8 | | CMP_AND_BRANCH | |
| 10 | IF_TRUE | | 9 | | | | |
| 11 | LOOP_END | | 10 | | | | |
| 12 | IF_FALSE | | 9 | | | | |
| 13 | MUL | I32 | 7 | -6 | | LEA (%src, 2), %dst | R4 |
| 14 | RETURN | | 12 | 13 | | RETURN_INT | |

- Now intervals are processed one by one and registers are allocated

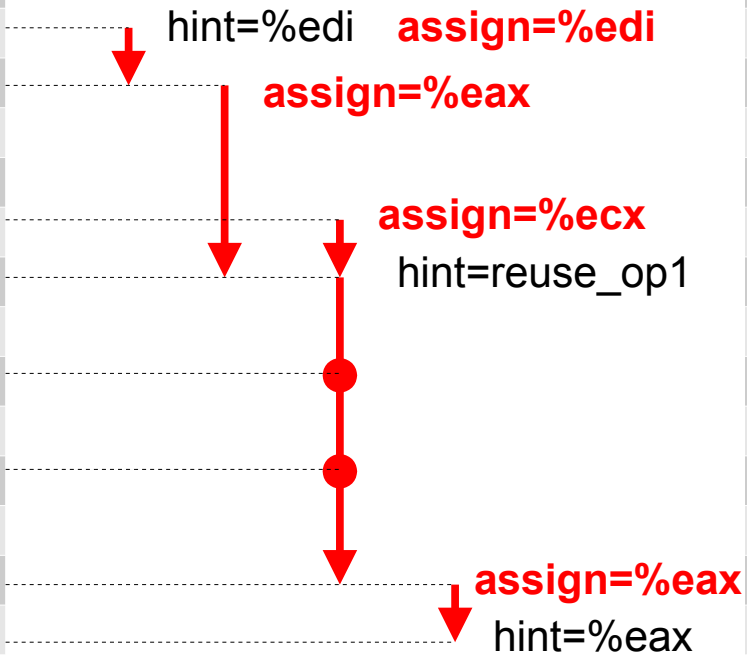


# Register Assignment



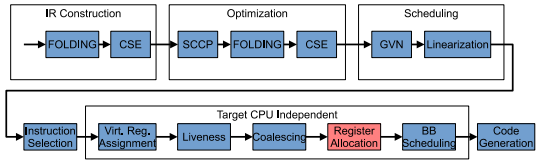

| | | | | | |
|---|---|---|---|---|---|
| -7 | | I32 | | | 10 |
| -6 | | I32 | | | 2 |
| -5 | | I32 | | | 1 |
| -4 | | I32 | | | 0 |
| -3 | | BOOL | | | TRUE |
| -2 | | BOOL | | | FALSE |
| -1 | | ADDR | | | NULL |

| | OP | TYPE | op1 | op2 | op3 | RULES | VREGS | def | op1 | op2 | op3 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | START | | 14 | | | | | | | | |
| 2 | PARAM | I32 | 1 | «a» | 1 | PARAM | R1 | edi | | | |
| 3 | ADD | I32 | 2 | -5 | | LEA 1(%src), %dst | R2 | eax | edi | | |
| 4 | END | | 1 | | | | | | | | |
| 5 | LOOP_BEGIN | | 4 | 11 | | | | | | | |
| 6 | PHI | i32 | 5 | -4 | 7 | PHI | R3 | ecx | | ecx | ecx |
| 7 | ADD | I32 | 6 | 3 | | BINOP | | ecx | ecx | eax | |
| 8 | LT | BOOL | 7 | -7 | | FUSED \| CMP | | | ecx | | |
| 9 | IF | | 5 | 8 | | CMP_AND_BRANCH | | | | | |
| 10 | IF_TRUE | | 9 | | | | | | | | |
| 11 | LOOP_END | | 10 | | | | | | | | |
| 12 | IF_FALSE | | 9 | | | | | | | | |
| 13 | MUL | I32 | 7 | -6 | | LEA (%src, 2), %dst | R4 | eax | ecx | | |
| 14 | RETURN | | 12 | 13 | | RETURN_INT | | | | eax | |




- Now we collect the registers allocated for intervals into a table of registers used by the rules (4 bytes)

# Register Assignment

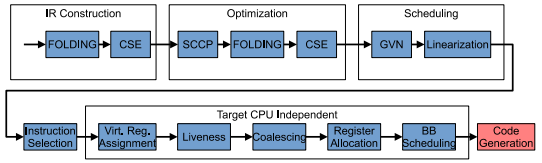



| | | | | | |
|---|---|---|---|---|---|
| -7 | | I32 | | | 10 |
| -6 | | I32 | | | 2 |
| -5 | | I32 | | | 1 |
| -4 | | I32 | | | 0 |
| -3 | | BOOL | | | TRUE |
| -2 | | BOOL | | | FALSE |
| -1 | | ADDR | | | NULL |

| | OP | TYPE | op1 | op2 | op3 | RULES | def | op1 | op2 | op3 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | START | | 14 | | | | | | | |
| 2 | PARAM | I32 | 1 | «a» | 1 | PARAM | edi | | | |
| 3 | ADD | I32 | 2 | -5 | | LEA 1(%src), %dst | eax | edi | | |
| 4 | END | | 1 | | | | | | | |
| 5 | LOOP_BEGIN | | 4 | 11 | | | | | | |
| 6 | PHI | i32 | 5 | -4 | 7 | PHI | ecx | | ecx | ecx |
| 7 | ADD | I32 | 6 | 3 | | BINOP | ecx | ecx | eax | |
| 8 | LT | BOOL | 7 | -7 | | FUSED \| CMP | | ecx | | |
| 9 | IF | | 5 | 8 | | CMP_AND_BRANCH | | | | |
| 10 | IF_TRUE | | 9 | | | | | | | |
| 11 | LOOP_END | | 10 | | | | | | | |
| 12 | IF_FALSE | | 9 | | | | | | | |
| 13 | MUL | I32 | 7 | -6 | | LEA (%src, 2), %dst | eax | ecx | | |
| 14 | RETURN | | 12 | 13 | | RETURN_INT | | | eax | |

- That’s all. We don’t need live intervals any more and can perform code generation.

# Code Generation

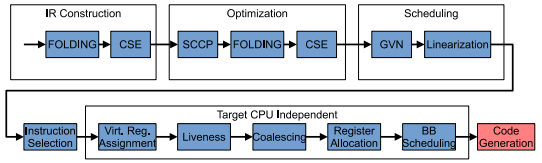



| | | | | | |
|---|---|---|---|---|---|
| -7 | | I32 | | | 10 |
| -6 | | I32 | | | 2 |
| -5 | | I32 | | | 1 |
| -4 | | I32 | | | 0 |
| -3 | | BOOL | | | TRUE |
| -2 | | BOOL | | | FALSE |
| -1 | | ADDR | | | NULL |

- We have rules (CG template), we have inputs for each template (allocated registers and constant values). We have to combine them and emit the code.

| | OP | TYPE | op1 | op2 | op3 | RULES | def | op1 | op2 | op3 | X86_64 CODE |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | START | | 14 | | | | | | | | |
| 2 | PARAM | I32 | 1 | «a» | 1 | PARAM | edi | | | | |
| 3 | ADD | I32 | 2 | -5 | | LEA 1(%src), %dst | eax | edi | | | leal 1(%rdi), %eax |
| 4 | END | | 1 | | | | | | | | xorl %ecx, %ecx |
| 5 | LOOP_BEGIN | | 4 | 11 | | | | | | | .L1: |
| 6 | PHI | i32 | 5 | -4 | 7 | PHI | ecx | | ecx | ecx | |
| 7 | ADD | I32 | 6 | 3 | | BINOP | ecx | ecx | eax | | addl %eax, %ecx |
| 8 | LT | BOOL | 7 | -7 | | FUSED \| CMP | | ecx | | | cmpl $0xa, %ecx |
| 9 | IF | | 5 | 8 | | CMP_AND_BRANCH | | | | | jl .L1 |
| 10 | IF_TRUE | | 9 | | | | | | | | |
| 11 | LOOP_END | | 10 | | | | | | | | |
| 12 | IF_FALSE | | 9 | | | | | | | | |
| 13 | MUL | I32 | 7 | -6 | | LEA (%src, 2), %dst | eax | ecx | | | leal (%rcx, 2), %eax |
| 14 | RETURN | | 12 | 13 | | RETURN_INT | | | eax | | retq |

# Code Generation




| | | | | | |
|---|---|---|---|---|---|
| -7 | | I32 | | | 10 |
| -6 | | I32 | | | 2 |
| -5 | | I32 | | | 1 |
| -4 | | I32 | | | 0 |
| -3 | | BOOL | | | TRUE |
| -2 | | BOOL | | | FALSE |
| -1 | | ADDR | | | NULL |

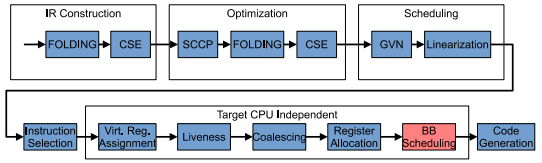
| | OP | TYPE | op1 | op2 | op3 | RULES | def | op1 | op2 | op3 | X86_64 CODE |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | START | | 14 | | | | | | | | |
| 2 | PARAM | I32 | 1 | «a» | 1 | PARAM | edi | | | | |
| 3 | ADD | I32 | 2 | -5 | | LEA 1(%src), %dst | eax | edi | | | leal 1(%rdi), %eax |
| 4 | END | | 1 | | | | | | | | xorl %ecx, %ecx |
| 5 | LOOP_BEGIN | | 4 | 11 | | | | | | | .L1: |
| 6 | PHI | i32 | 5 | -4 | 7 | PHI | ecx | | ecx | ecx | |
| 7 | ADD | I32 | 6 | 3 | | BINOP | ecx | ecx | eax | | addl %eax, %ecx |
| 8 | LT | BOOL | 7 | -7 | | FUSED \| CMP | | ecx | | | cmpl $0xa, %ecx |
| 9 | IF | | 5 | 8 | | CMP_AND_BRANCH | | | | | jl .L1 |
| 10 | IF_TRUE | | 9 | | | | | | | | |
| 11 | LOOP_END | | 10 | | | | | | | | |
| 12 | IF_FALSE | | 9 | | | | | | | | |
| 13 | MUL | I32 | 7 | -6 | | LEA (%src, 2), %dst | eax | ecx | | | leal (%rcx, 2), %eax |
| 14 | RETURN | | 12 | 13 | | RETURN_INT | | | eax | | retq |

- We have rules (CG template), we have inputs for each template (allocated registers and constant values). We have to combine them and emit the code.
- We may need to insert moves (e.g. for SSA deconstruction)

On the fly SSA deconstruction

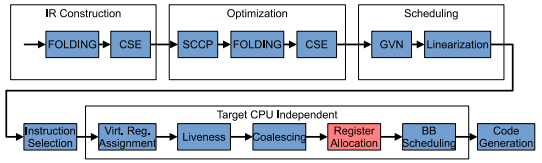
```
int foo(int a) {
    i = 0;
    ...
```

# BB Scheduling



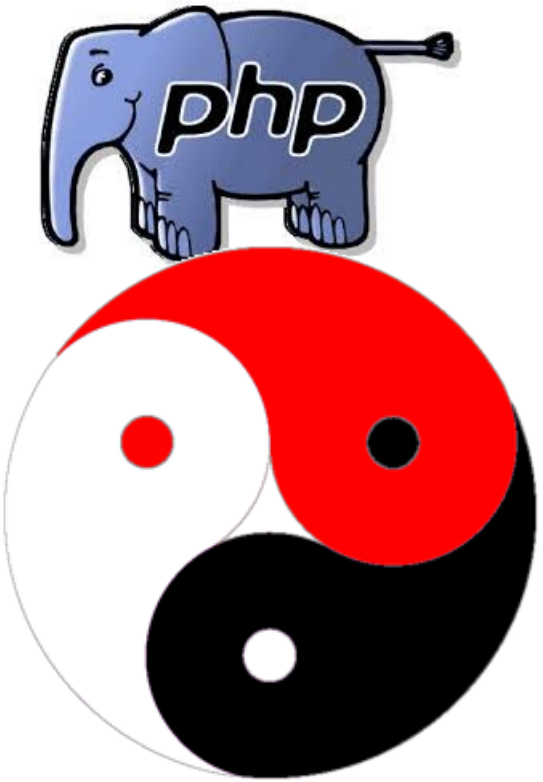

- Reorder Basic Blocks to minimize branches between them
- Karl Pettis and Robert C. Hansen "Profile Guided Code Positioning"
- Top-Down positioning algorithm
  - Starts from the first block
  - Adds the most probable successor
- This doesn’t affect the IR, rules and register tables.

# More About Linear Scan Register Allocation




- C. Wimmer and M. Franz "Linear Scan Register Allocation on SSA Form"
- C. Wimmer "Optimized Interval Splitting in a Linear Scan Register Allocator"
- Similar to RA in Java HotSpot Client Compiler and V8 TurboFan
- Each CG Rule provides a number constrains and hints that we take into account
  - Some rules may require operand in a specific register (e.g. shifts on x86)
  - Some may clobber specific registers (mul, div on x86)
  - Some may require additional temporary register (e.g. use of 64-bit int constant)
- We use 4 sub-positions for each instruction (load, use, def, store)
- Live intervals of inputs of fused instructions lifted to the fusion root instruction
- If we can’t allocate a register for the whole interval, we may split it, allocate registers for the parts and generate moves in split positions
- Finally we may spill interval into a stack slot and generate stores and loads when needed
- Spill slots mat be reused if their intervals don’t overlap (yet another linear scan)

# JIT Specific Extensions



# JIT Specific Extensions - PHP HYBRID VM (direct threaded)

```
register zend_execute_data
    *exexute_data __asm__(«%r14»);
register zend_op
    *opline __asm__(«%r15»);

void execute_ex(
    zend_execute_data *ex)
{
    exexute_data = ex;
    opline = exexcute_data->opline;
    goto *opline->handler;
ZEND_ADD_SPEC_TMPVAR_CONST_LABEL:
    ZEND_ADD_SPEC_TMPVAR_CONST();
    goto *opline->handler;
...
}
```

*handler label*

*jump to next instruction handler*

*real handler call (may be inlined)*

```
static void ZEND_ADD_SPEC_TMPVAR_CONST(void)

{
    zval *op1, *op2, *result;

    op1 = EX_VAR(opline->op1.var);
    op2 = RT_CONSTANT(opline, opline->op2);
    if (Z_TYPE_P(op1)== IS_LONG) {
        if (Z_TYPE_P(op2) == IS_LONG) {
            result = EX_VAR(opline->result.var);
            fast_long_add_function(
                result, op1, op2);
        } else if (Z_TYPE_P(op2) == IS_DOUBLE) {
            ...
        ...
    }
    opline++;
}
```

# JIT Specific Extensions - Fixed Stack Frame

```
register zend_execute_data
    *exexute_data __asm__(«%r14»);
register zend_op
    *opline __asm__(«%r15»);

void execute_ex(
    zend_execute_data *ex)
{
    exexute_data = ex;
    opline = exexcute_data->opline;
    goto *opline->handler;



...
}
```

```
.ENTRY:
    movq 0x50(%r14), %rax ; load int from a PHP var
    addq $42, %rax        ; add 42
    movq $rax, 0x60(%r14) ; store into another var
    movl $4, 0x68(%r14)   ; set IS_LONG type
    addq $32, %r15        ; opline++
    jmpq *(%r15)          ; jmp to next handler
```

*jump to JIT code*

# JIT Specific Extensions - Fixed Stack Frame and Fixed Registers

- IR may specify fixed stack layout
  - **fixed_regset** - registers excluded from regular RA
  - **fixed_stack_red_zone** - amount of CPU stack provided by caller
  - **fixed_stack_frame_size** - stack for spill slots
  - **fixed_call_stack_size** - stack for argument passing
  - **fixed_save_regset** - registers that must be saved/restored in prologue/epilogue
- IR may disable code generation for regular function prologue (**IR_SKIP_PROLOGUE**)
- IR may force using frame pointer register (**IR_USE_FRAME_POINTER**)
- IR provides **RLOAD** and **RSTORE** instructions to access CPU registers
- Non-local jumps implemented through **TAILCALL** or **IJMP**

# JIT Specific Extensions – BIND-ing nodes to VM stack slots

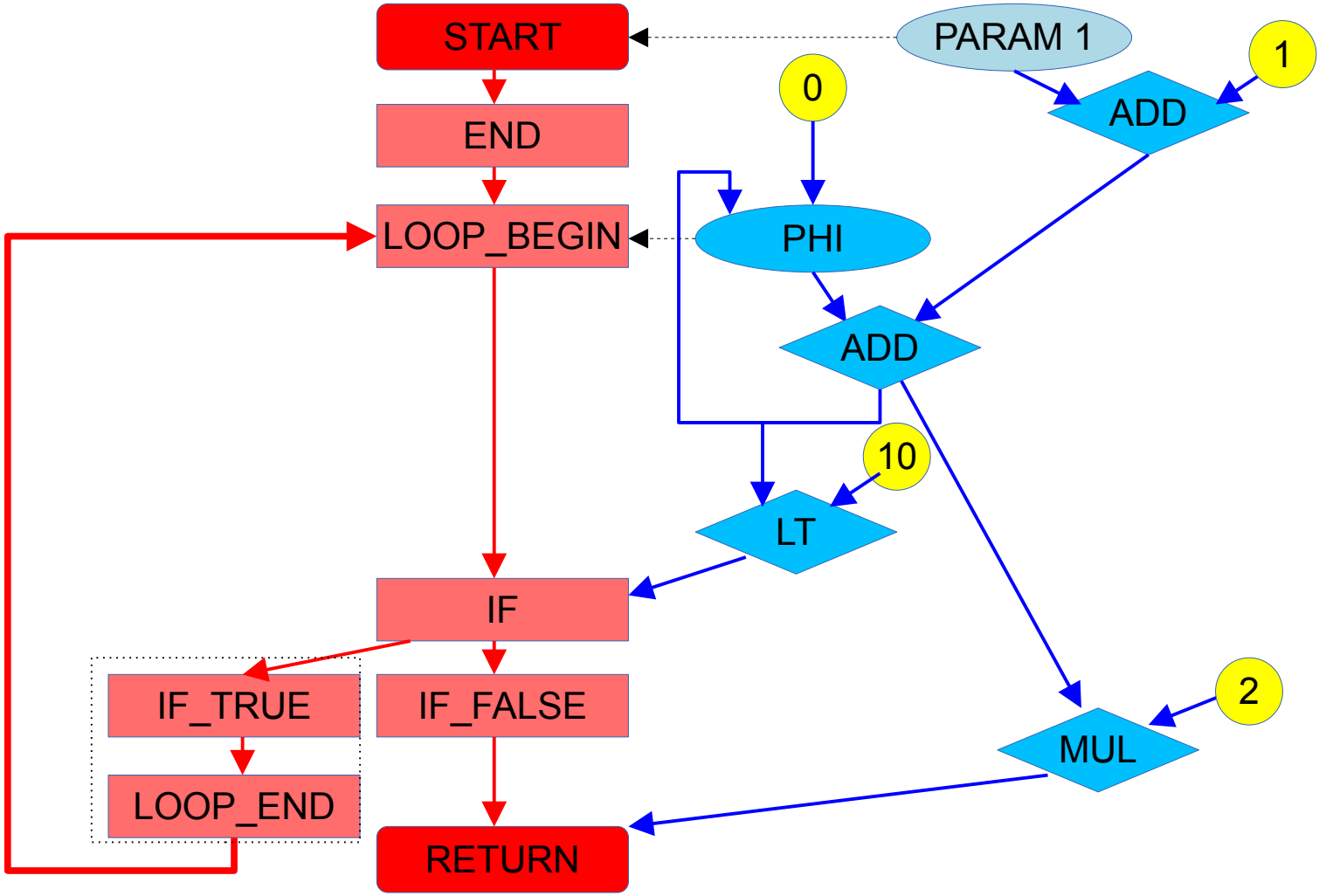
- In case some value can't be kept in a register, it is spilled (usually stored to CPU stack)
- Let spill PHP values to PHP stack slots where they had to be in the first place
- This reduces CPU stack requirement and helps to fit into fixed frame
- This also eliminates a need for transferring values between stacks when OSR

- **ir_bind()** - tells where to spill the node if necessary (PHP stack offset)
- **ir_ctx.spill_base** – the spill base (%r14)

- Few temporary PHP variables may share the same PHP stack slot
- This introduces anti-dependencies and potential problems...

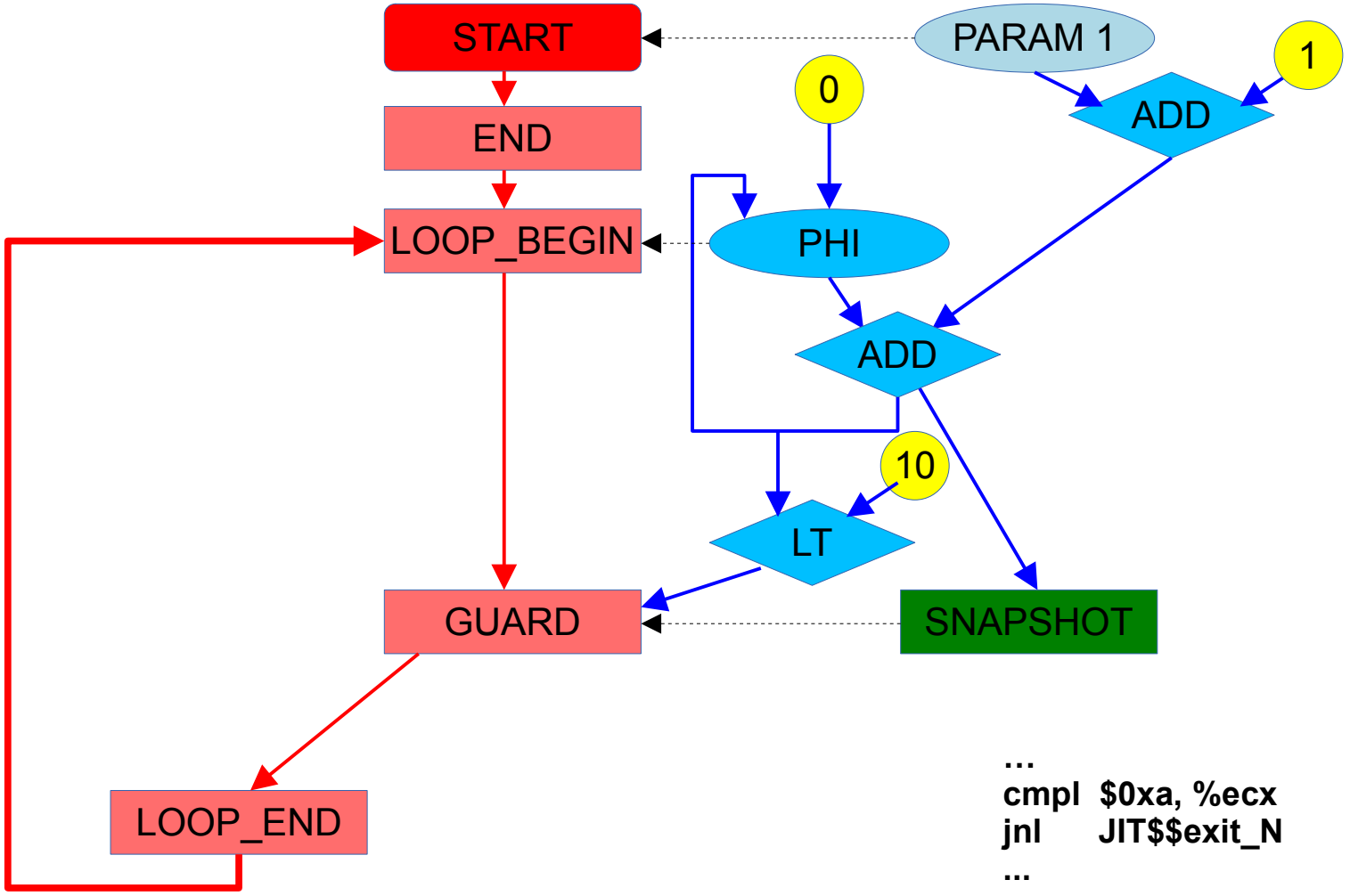
# JIT Specific Extensions – Speculative Compilation and OSR EXIT

- On Stack Replacement
- Let say we don't want to compile the whole function, because some paths are never taken or are not hot
- PHP is dynamically typed, we may want to compile function only for integer numbers and fallback to VM in case of something else
- Fear exit to VM through compensation code and **jmp *(%r15)** is expensive



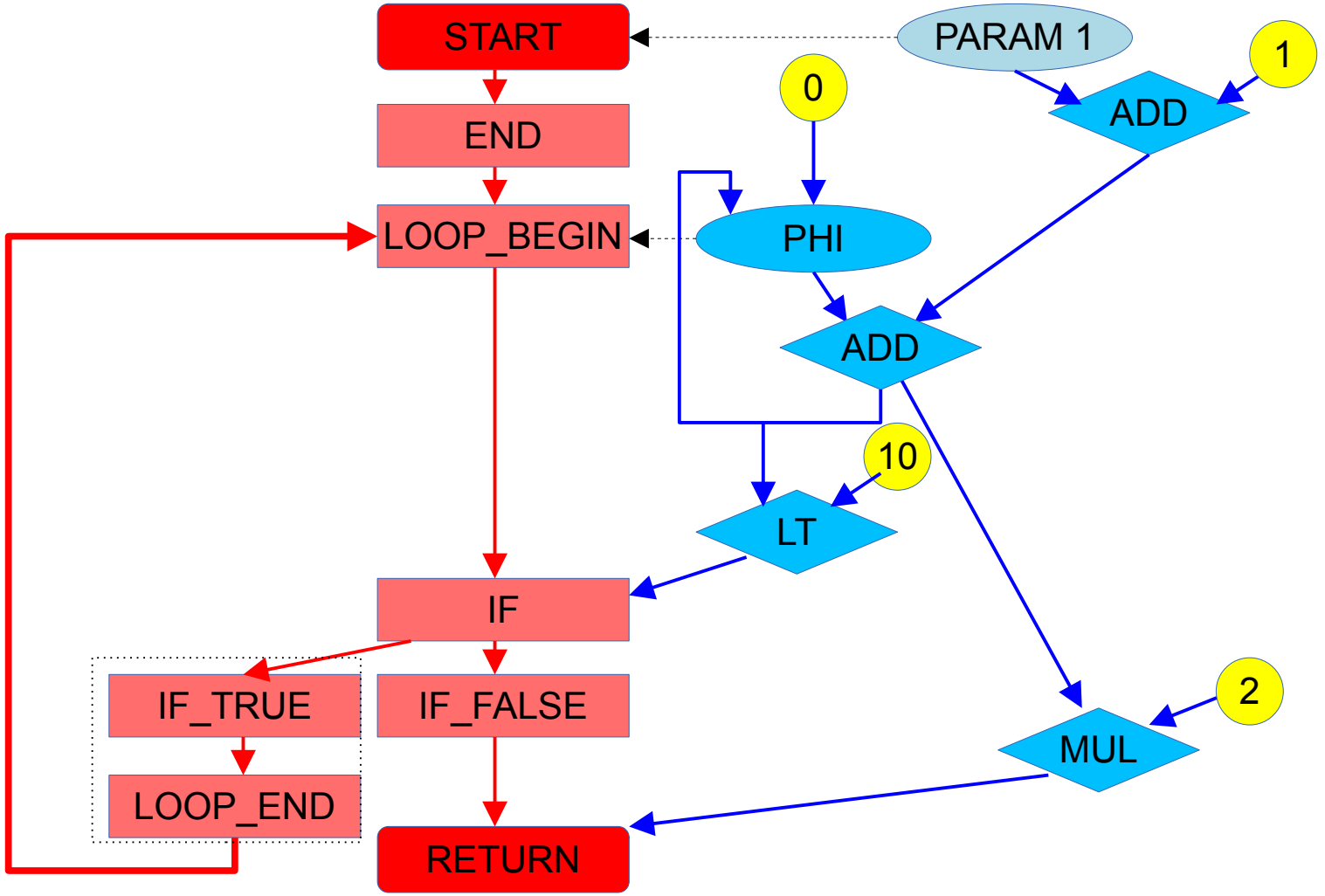
# JIT Specific Extensions – Speculative Compilation and OSR EXIT

- GUARD is a one way IF
- The second way is the EXIT
- SNAPSHOT keeps all the live variables
- EXIT is just a branch to special shared label
- IR Framework provides information about the location of all the live variables that need to be saved back to PHP stack slots to continue interpretation



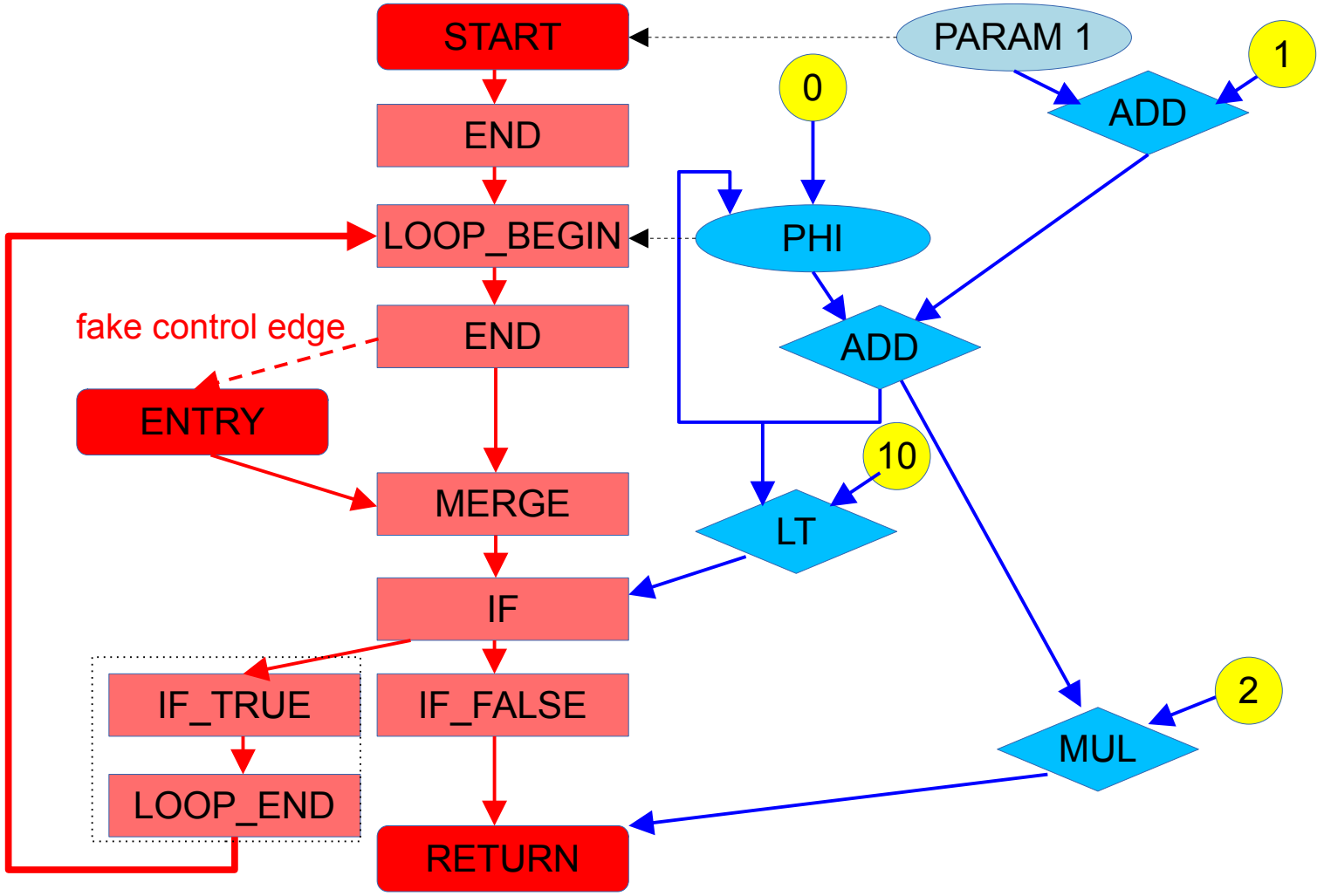
# JIT Specific Extensions – OSR ENTRY and multiple entries

- Now we want to ENTER into JIT code in the middle of the loop.



# JIT Specific Extensions – OSR ENTRY and multiple entries

- ENTRY is inserted without CFG break (thanks to the fake edge)
- During register allocation, we collect all the live intervals intersected with the entry
- For all BIND-ed nodes we generate moves from VM stack slots to CPU registers
- Unfortunately, we can't handle arbitrary nodes and limit some optimization across the ENTRY-es



# The Current IR Framework State

- PHP IR JIT works!
  - Function and tracing JIT
  - Linux, Windows, x86, X86_64, AArch64, ZTS, GCC, CLANG, MSVC, ...
  - All 17K *.phpt tests are passed
  - All PHP CI work-flows are passed
  - IR JIT produces a bit faster and smaller code than PHP 8 JIT (0-5%)
  - Tracing JIT compilation speed is almost the same as in PHP 8
  - Function JIT compilation speed is about 4 times slower
    - Wordpress compilation takes 0.8 sec (PHP 8 without JIT takes 0.1 sec, PHP 8 JIT – 0.2 sec, old LLVM PHP JIT - ~45 sec)
- No any real usage except PHP yet
- IR → LLVM and LLVM → IR (work in progress)
- Not tested enough, edge cases, childhood diseases (it's a 1.5 year old baby)

# Future Directions

- Better code generation (may be switch to BURS)
- Get rid of DynASM. This is a great tool (especially for start-up), but it's expensive.
- More and better IR optimization passes
- Compilation speed improvement

- Support for compilation modules (many functions and data objects all together)
- C → IR front-end
- IR interpreter
- IR template system to simplify run-time IR construction (we might write IR templates in C, then compile them into IR, then apply automatic partial evaluation)

- PHP IR JIT improvement, special support for PHP FFI extension

# Thank you!

Dmitry Stogov

dmitry@php.net

github.com/dstogov