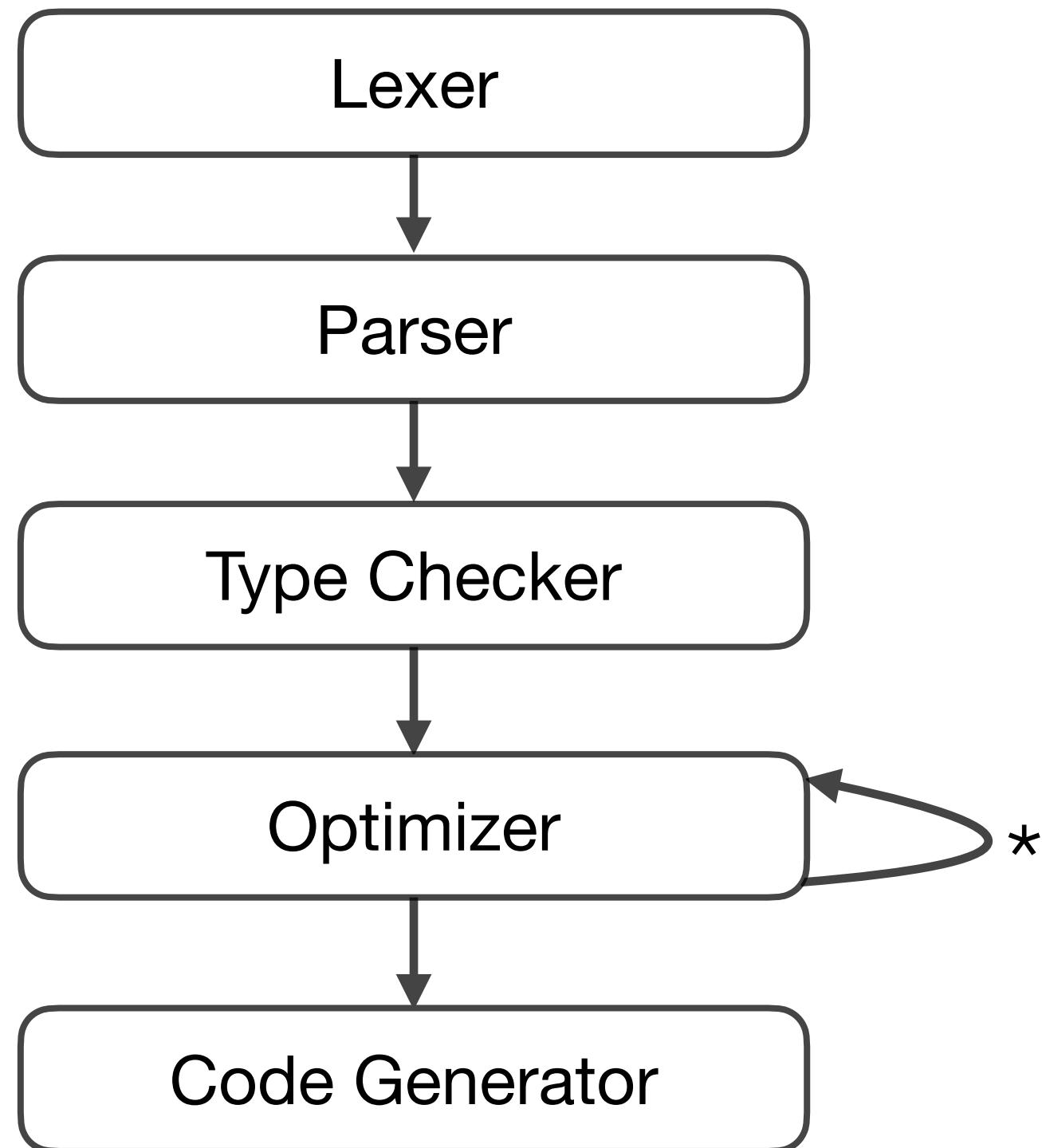


Lecture 10 — Fast Compilation

Stanford CS343D (Winter 2023)
Fred Kjolstad

Classical compiler overview



There is a lot of work to compiling optimized code

Compilation times matter

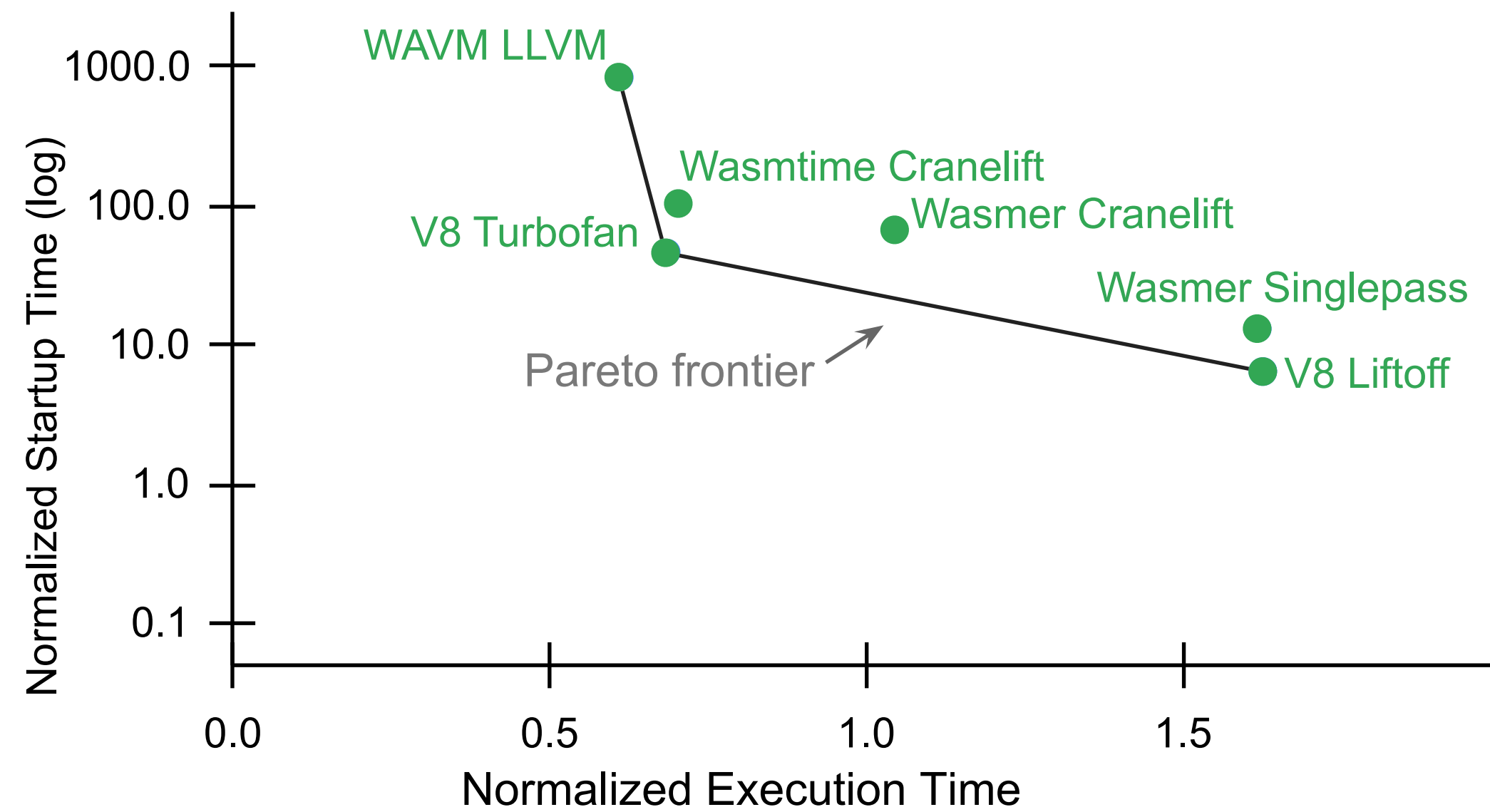
- JIT compilers (compilation at runtime)
- LLVM -O0 vs -O2 (10x difference)
- Scala (large type checking cost)
- JavaScript (teams of engineers)
- WebAssembly (51s for AutoCAD)
- Databases (4.5s for TPC Q19 query)
- Taco (generated expressions)



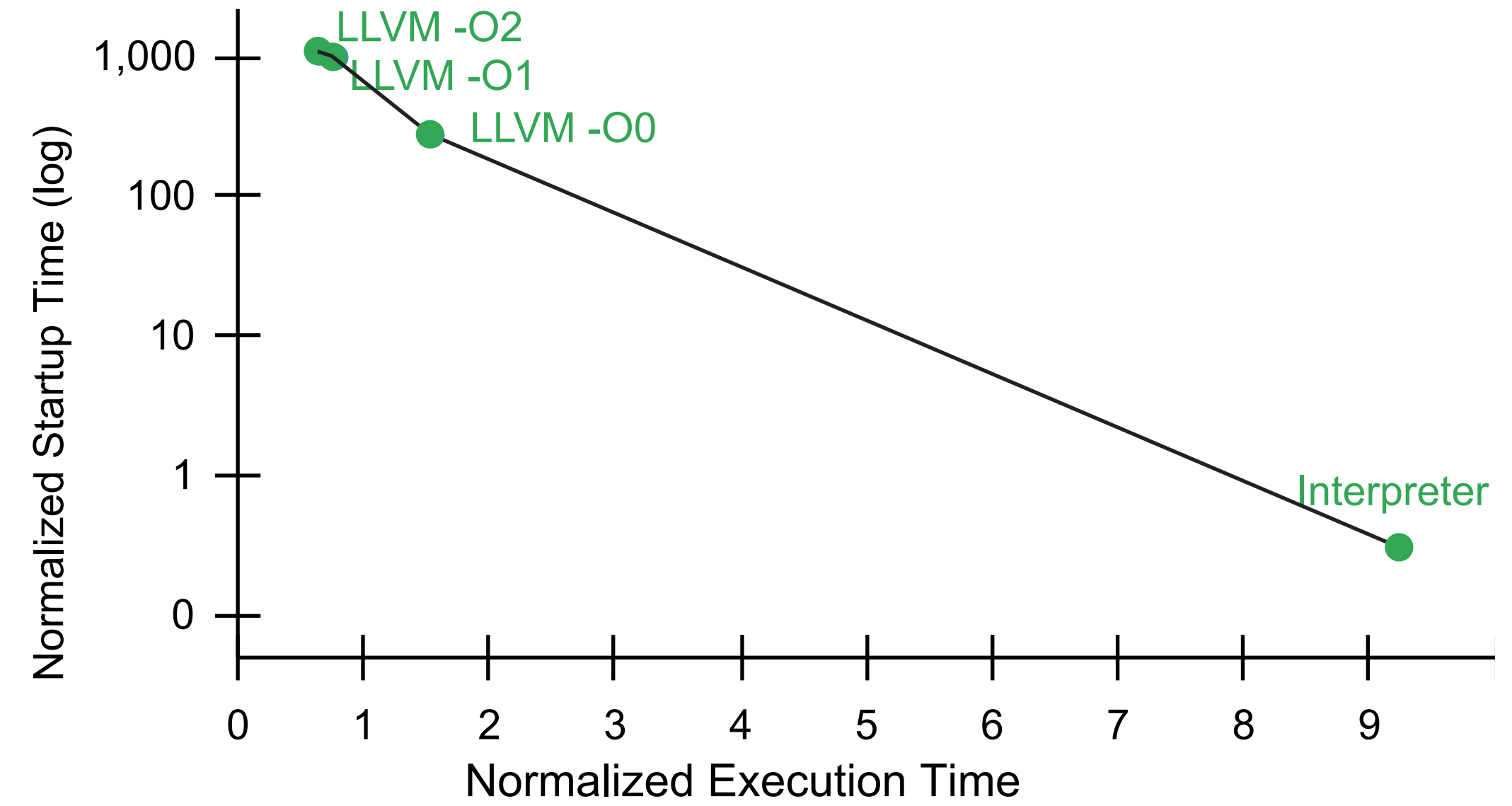
How can we speed up compilation? — Let us brainstorm

- Multithreading
- Turn off optimization
- Interpretation instead of compilation
- Use bytecode for partial pre-compilation
- Change language: e.g., simplify type system

Tradeoff between compilation time and code performance



WebAssembly (PolyBench benchmarks)



Imperative Language (TPC-H Q6)

Idea: Two-tiered execution

Tier 1: Fast startup

- Interpreter
- LLVM -O0
- Baseline compilers (bytecode)

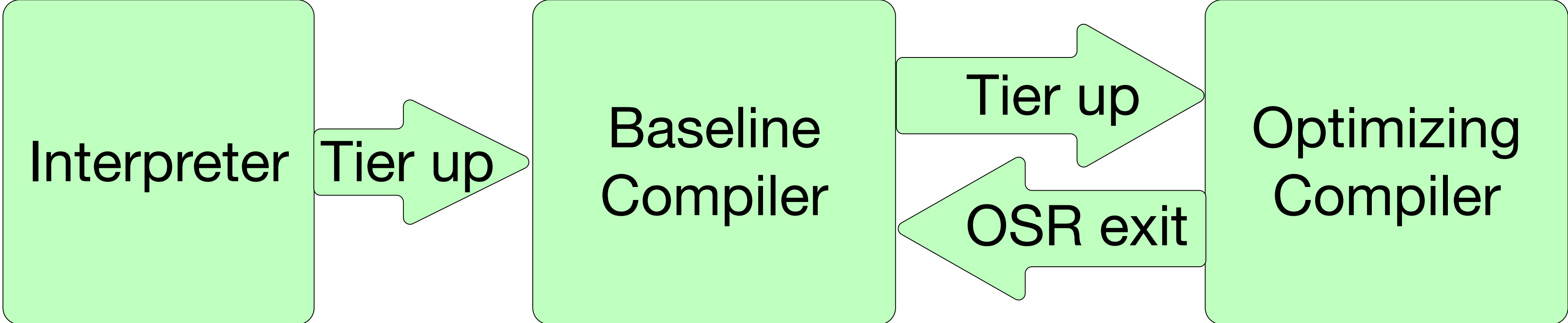
Tier 2: Fast execution

- Java HotSpot JIT Compiler
- LLVM -O2
- Google V8 TurboFan

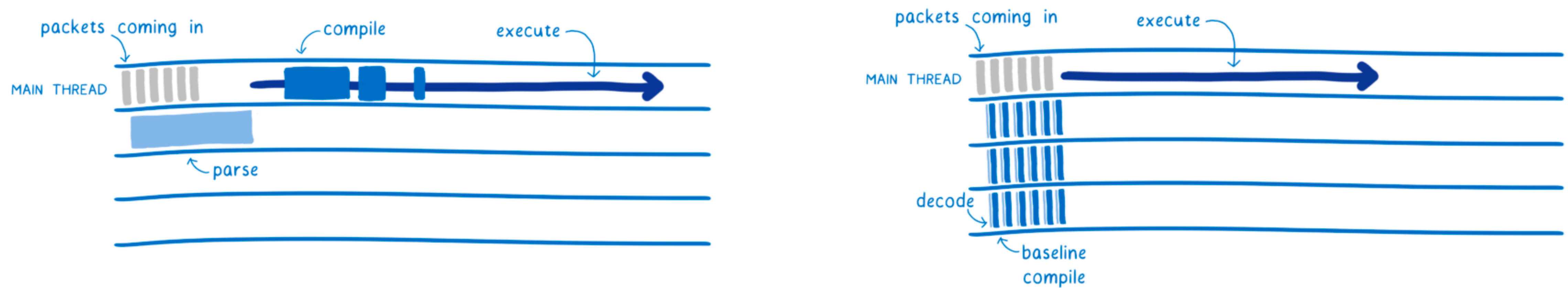
Used in basically all JIT compilers and databases

Examples: Java, JavaScript, WebAssembly, Databases

JavaScript Virtual Machine



Baseline compiler web example



200ms can be perceived by users and cause them to visit a webpage less frequently

Baseline compiler web example

WebAssembly (sent in binary)

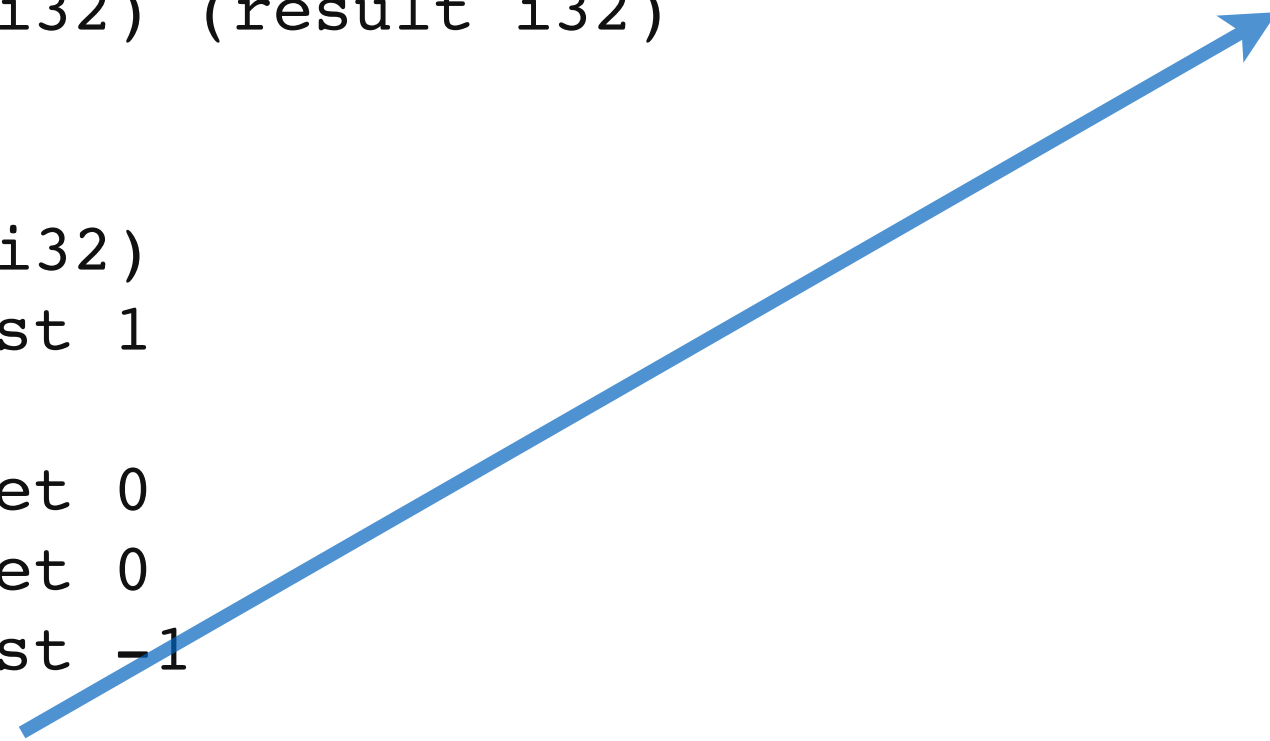
```
(func (param i32) (result i32)
  local.get 0
  i32.eqz
  if (result i32)
    i32.const 1
  else
    local.get 0
    local.get 0
    i32.const -1
    i32.add
    call 0
    i32.mul
  end)
```

Baseline
Compilation

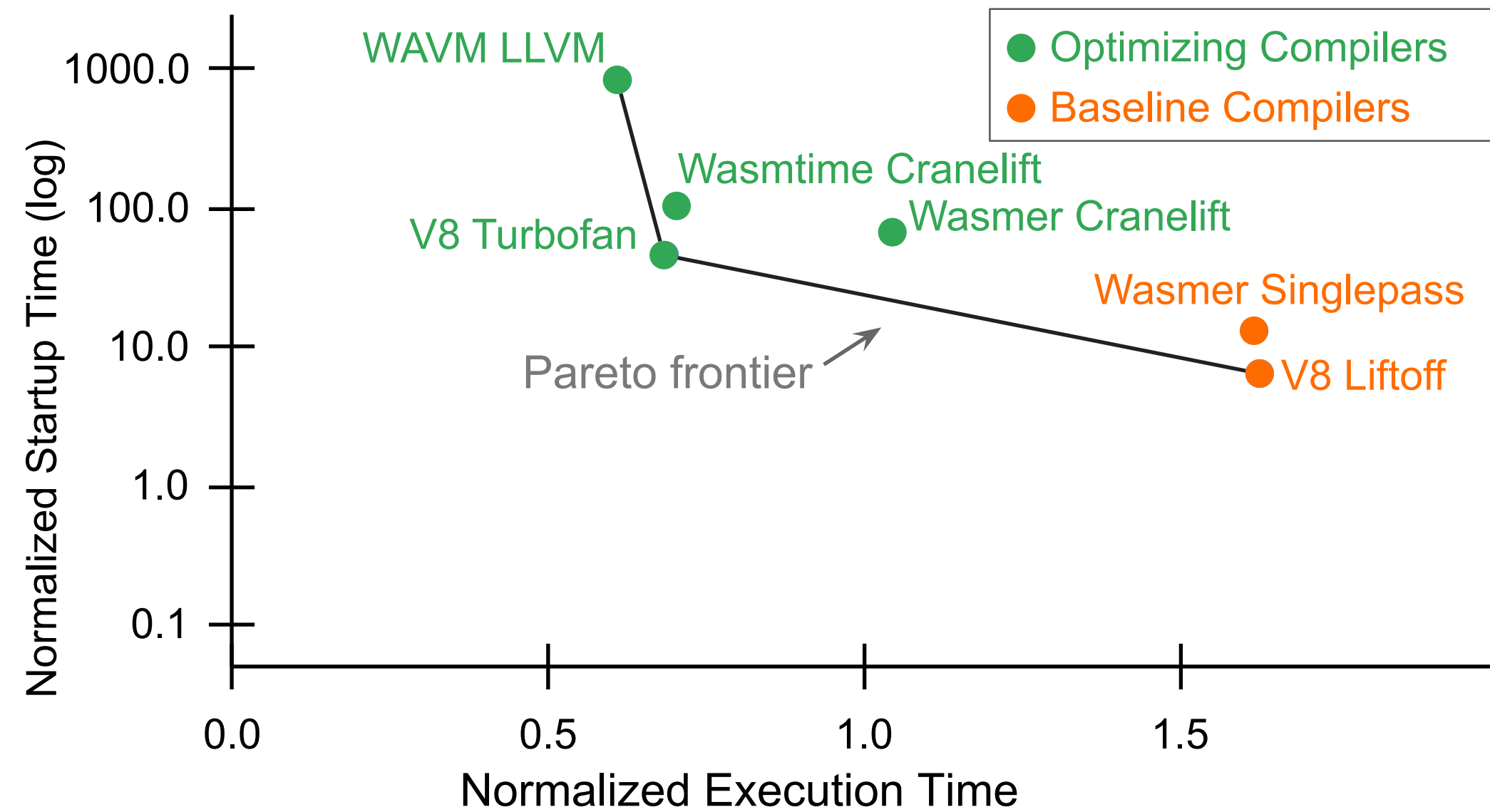


Baseline compiler rule (Firefox)

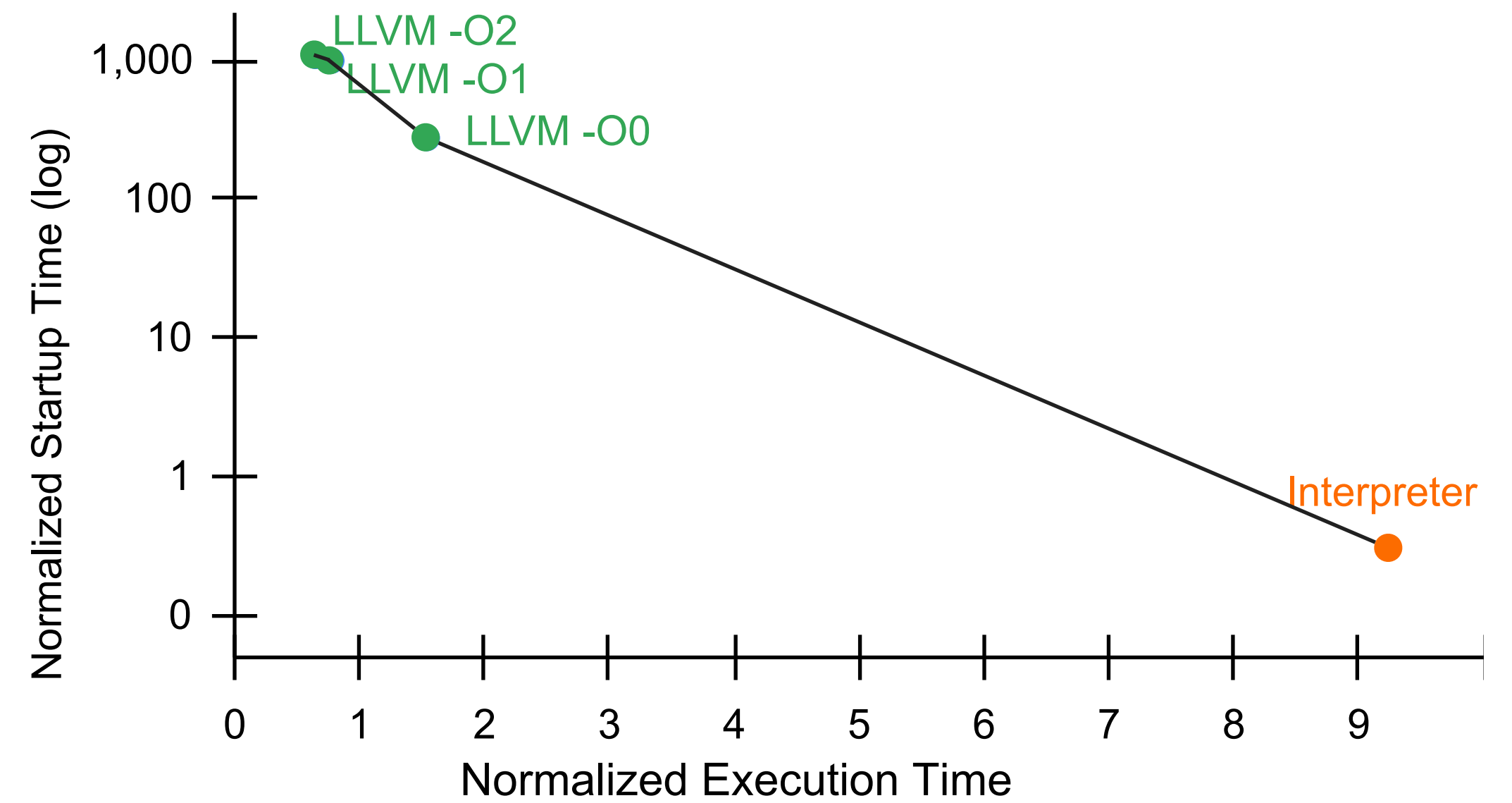
```
void BaseCompiler::emitAddI32() {
  int32_t c;
  if (popConstI32(&c)) {
    RegI32 r = popI32();
    masm.add32(Imm32(c), r);
    pushI32(r);
  } else {
    RegI32 r, rs;
    pop2xI32(&r, &rs);
    masm.add32(rs, r);
    freeI32(rs);
    pushI32(r);
  }
}
```



Copy-and-Patch is a fast baseline compilation algorithm

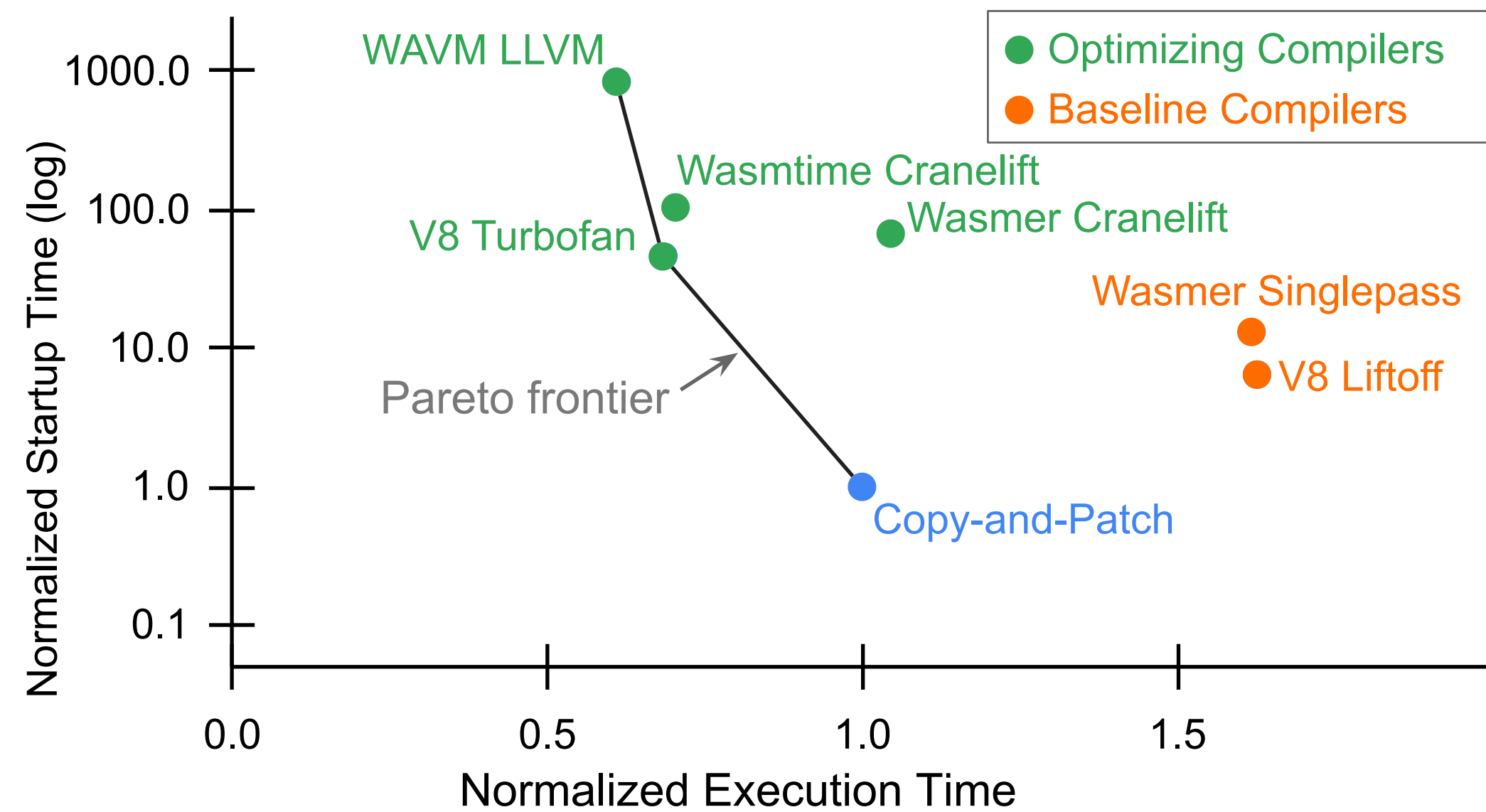


WebAssembly (PolyBench benchmarks)

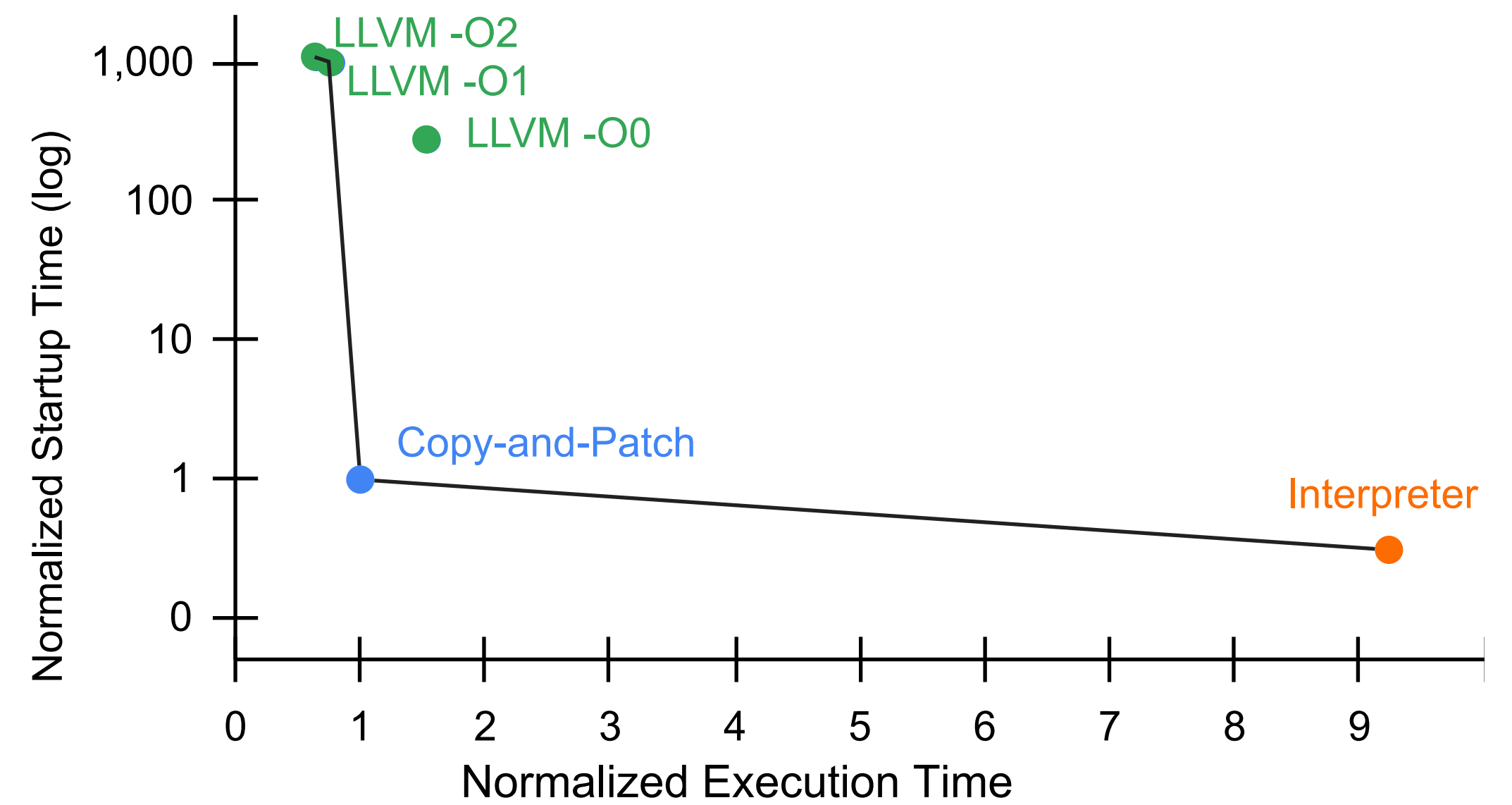


Imperative Language (TPC-H Q6)

Copy-and-Patch is a fast baseline compilation algorithm

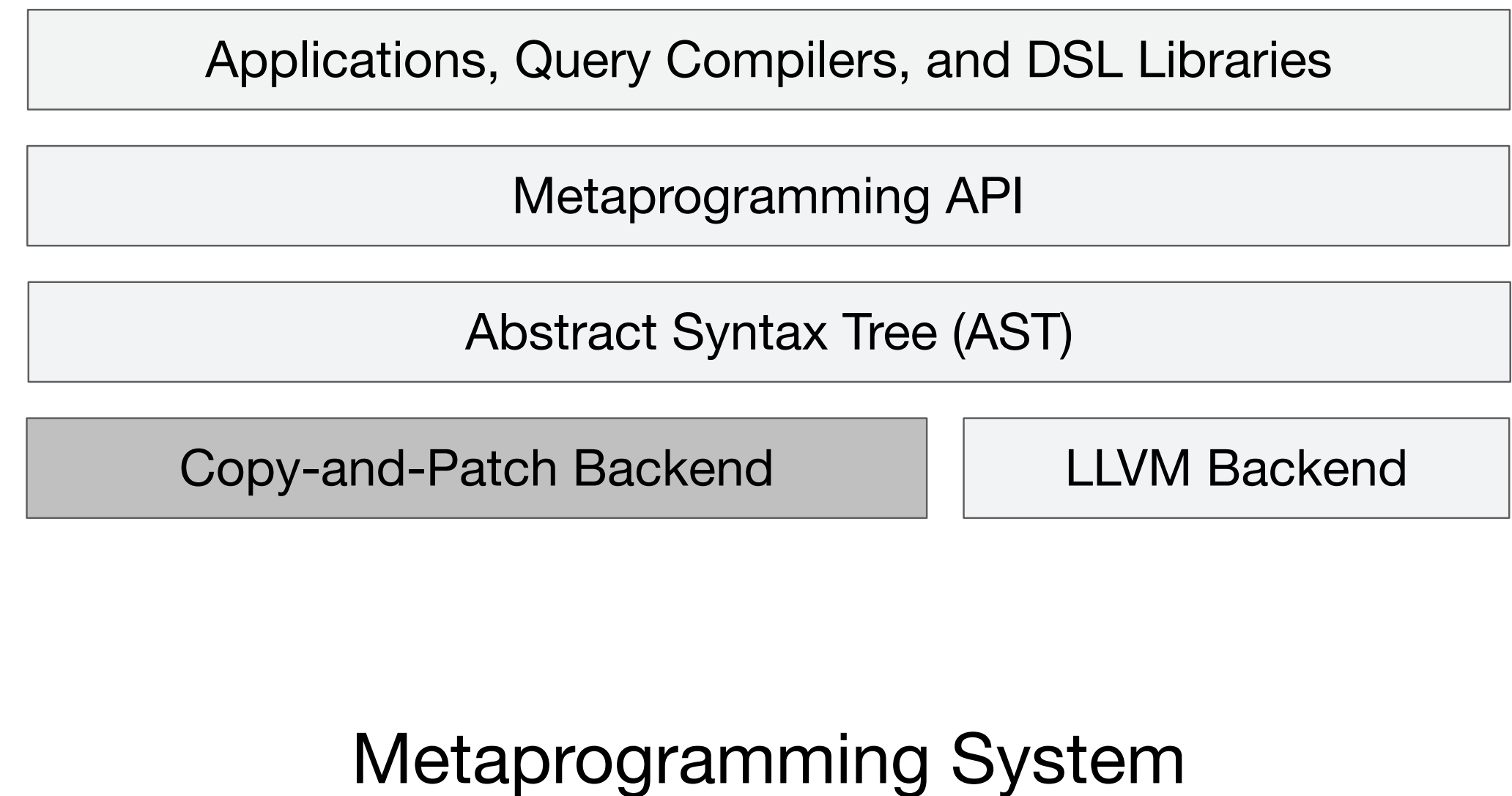
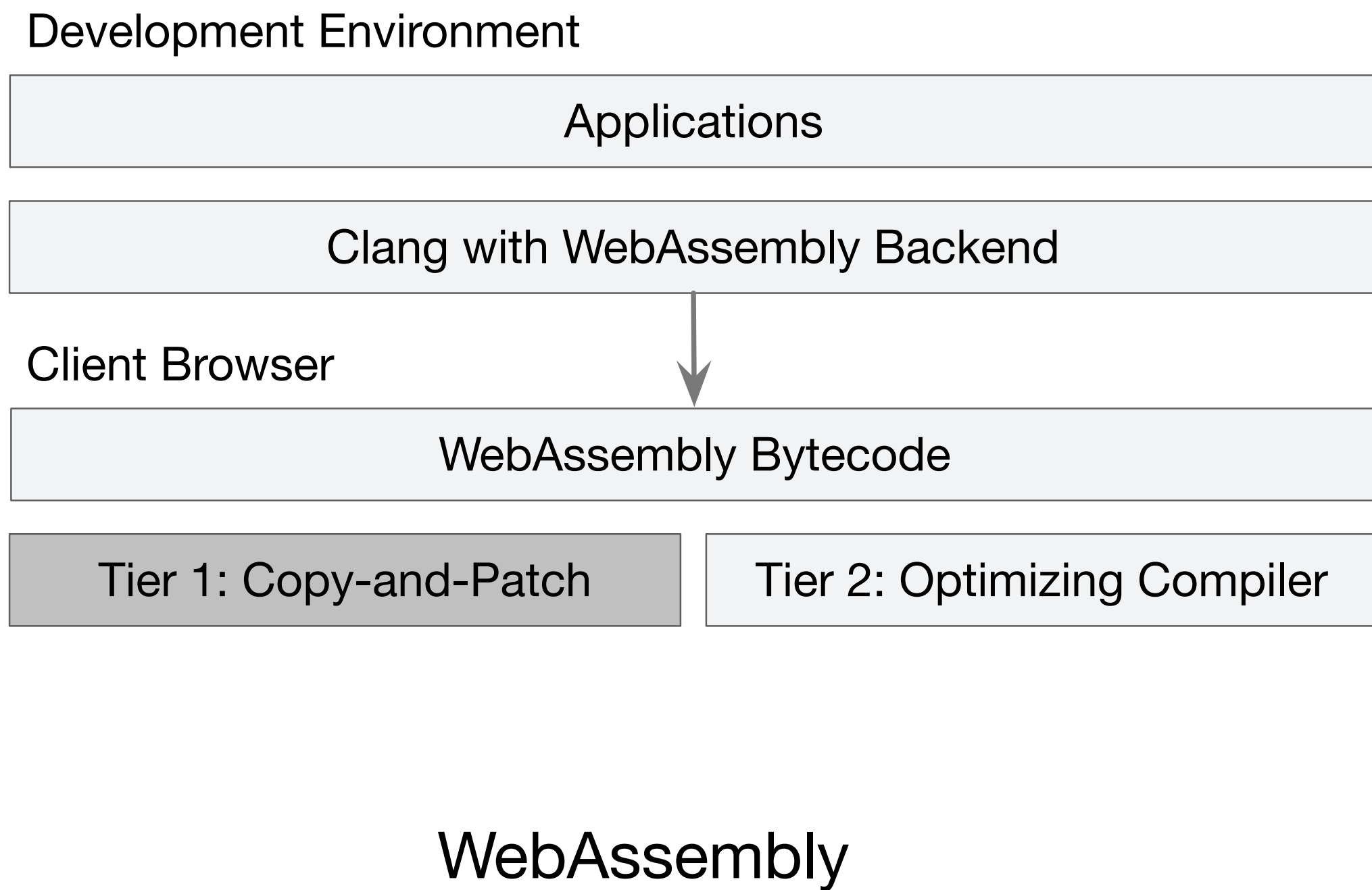


WebAssembly (PolyBench benchmarks)



Imperative Language (TPC-H Q6)

Two use cases



Idea 1: precompile all language constructs

Library of precompiled language constructs

```
        add
      sub  neg
load    mul
for     if
while
      ...
```

(missing stack offsets and jump targets)

At compile-time

For each AST node:

1. Hash lookup
2. Binary code copy
3. Patch in stack offsets and jump targets

Most performance comes from two optimizations (80/20 rule)

- Vilfredo Pareto: “80% of the consequences come from 20% of causes”
- 80% of the performance gain comes from **only two** optimizations
 1. Instruction selection
 2. Register Allocation

Idea 2: Instruction Selection

Precompile specialized stencil variants for super-nodes and constants

Library of precompiled language constructs

add(const, const)

add(stack, const)

mul_add(stack, stack)

load

sub(stack, stack) load_offset

for if if_leq

while

...

At compile-time

For each AST node:

1. Supernode Tree search
2. Hash lookup
3. Binary code copy
4. Patch in stack offsets, jump targets, and constants

Idea 3: Register Allocation

Precompile specialized stencil variants that use different registers

Library of precompiled language constructs

```
add(const, const)
add(stack, const)    add(r1, r2)
mul_add(stack, stack)
                    add(r1, const)
sub(stack, stack)    load
                    load_offset
for    if    if_leq
while
...
```

At compile-time

For each AST node:

1. Supernode Tree search
2. Expression register allocation
3. Hash lookup
4. Binary code copy
5. Patch in stack offsets, jump targets, and constants

Compile a large stencil variant library for use during compilation

Created at compiler build time, used to compile at runtime

WebAssembly

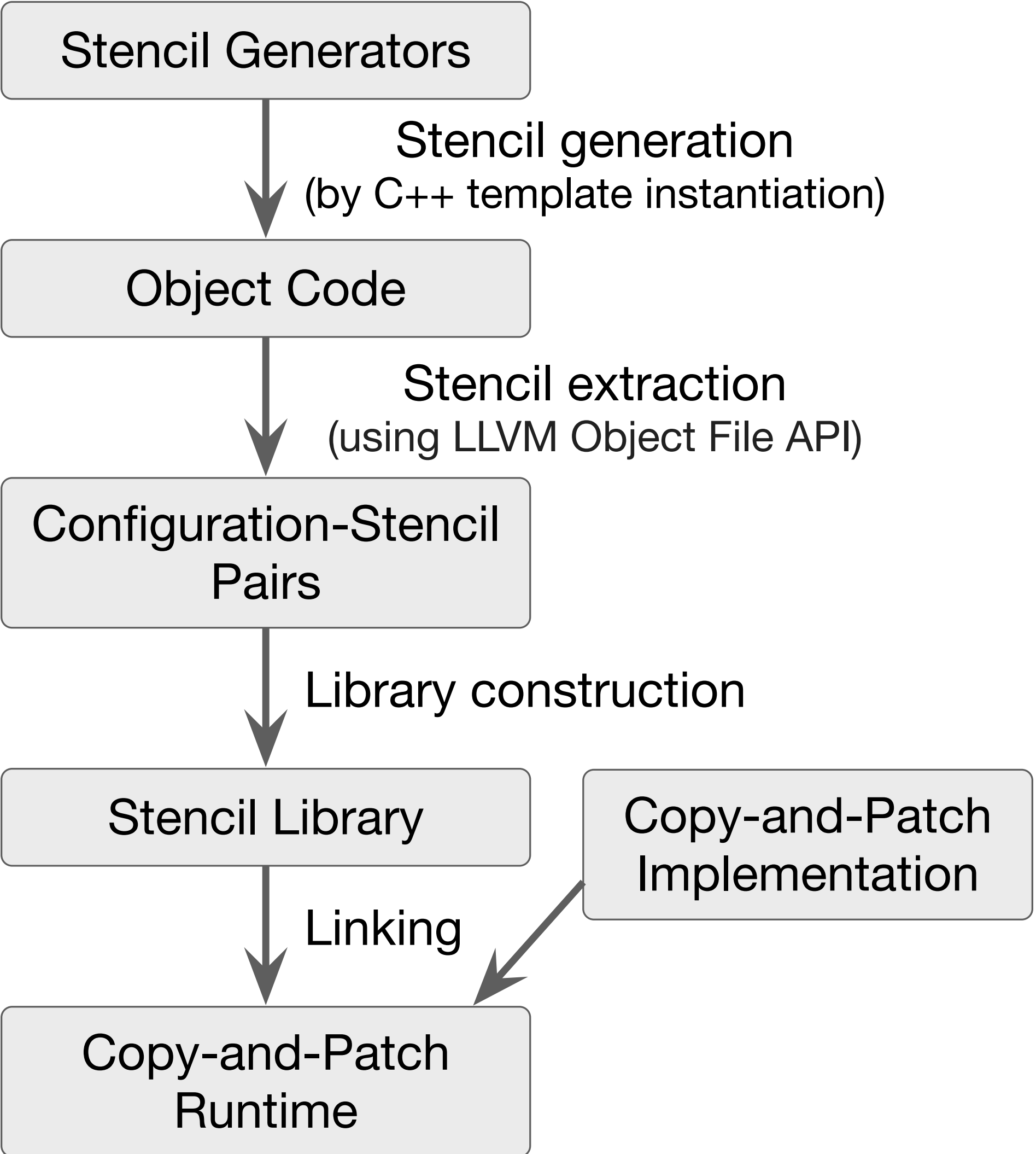
- 1666 stencils
- 30 kilobytes
- <1 minute to compile

High-Level Imperative Language

- 98,831 stencils
- 17.5 megabytes
- 14 minutes to compile

How can we create all of these stencils?

We write variant groups in C++ using templates and Clang+LLVM compiles them for us



Variants

Registers operands lhs and rhs

```
void eq_int(uintptr_t stack, int lhs, int rhs) {  
    bool result = (lhs == rhs);  
    (void*)(uintptr_t, bool) [1](stack, result);  
}
```

Call next operation

```
void if(uintptr_t stack, bool test) {  
    if (test)  
        (void*)(uintptr_t) [1](stack);  
    else  
        (void*)(uintptr_t) [2](stack);  
}
```

```
void eq_int_lvar_rconst(uintptr_t stack) {  
    int lhs = *(int*)(stack + [1]); ← Stack operand  
    int rhs = [2]; ← Constant  
    bool result = (lhs == rhs);  
    (void*)(uintptr_t, bool) [3](stack, result);  
}
```

```
void eq_int_pt(uintptr_t stack, uint64_t r1, int rhs) {  
    int lhs = [1];  
    bool result = (lhs == rhs);  
    (void*)(uintptr_t, uint64_t, bool) [2](stack, r1, result);  
}
```

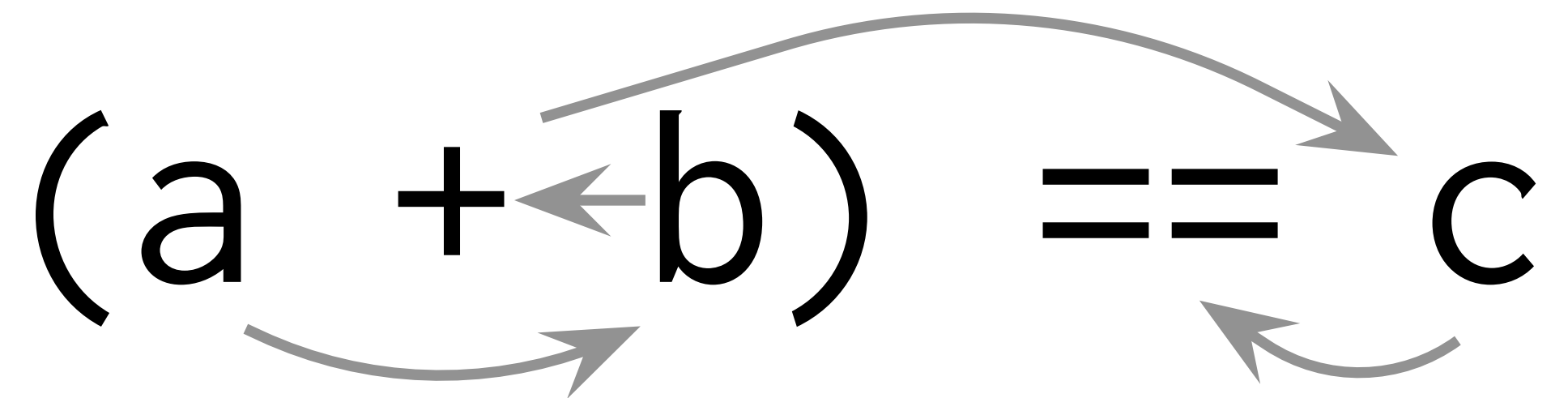
Register communicated
from a previous operation
to a later operations

Continuation-passing style and tail call optimization

Typical recursive interpreter code

```
int evaluate()  
{  
    int lhs = evaluate_lhs();  
    int rhs = evaluate_rhs();  
    return lhs + rhs;  
}
```

Faster continuation-passing style



Register pass-through

```
void stencil1(uintptr_t stack) {  
    int x = /* assign value to x */;  
    (void (*)(uintptr_t, int)  )(stack, x);  
}  
  
void stencil2(uintptr_t stack, uint64_t x) {  
    // computation unrelated to x  
    (void (*)(uintptr_t, uint64_t)  )(stack, x);  
}  
  
void stencil3(uintptr_t stack, int x) {  
    // do something with x  
}
```

Hack: use C++ extern keyword to locate holes in generated code

```
extern int evaluate_lhs();  
extern int evaluate_rhs();  
int evaluate()  
{  
    int lhs = evaluate_lhs();  
    int rhs = evaluate_rhs();  
    return lhs + rhs;  
}
```

1. C++ compiler generates an object file
2. The linker can link object files to any definition of the extern calls
3. The object file thus contains information to locate them in the binary code
4. We can use this information to locate holes in stencils for later patching

Using templates we can generate groups of variants

```
struct ArithAdd {
    template<typename T /* OperandType */,
            bool spillOutput,
            NumPassthroughs numPassthroughs,
            typename... Passthroughs>
    static void g(uintptr_t stack, Passthroughs... pt, T a, T b) {
        T c = a + b;
        if constexpr (! spillOutput) {
            DEF_CONTINUATON_0(void*)(uintptr_t, Passthroughs...,T));
            CONTINUATON_0(stack, pt..., c); // continuation
        } else {
            DEF_CONSTANT_1(uint64_t);
            *(T*)(stack + CONSTANT_1) = c;
            DEF_CONTINUATON_0(void*)(uintptr_t, Passthroughs...));
            CONTINUATON_0(stack, pt...); // continuation
        }
    }
};

template<typename T /* OperandType */,
        bool spillOutput,
        NumPassthroughs numPassthroughs>
static constexpr bool f() {
    if (numPt > numMaxPassthroughs - 2) return false;
    return !std::is_same<T, void>::value;
}

static auto metavaris() {
    return createMetaVarList(
        typeMetaVar(),
        boolMetaVar(),
        enumMetaVar<NumPassthroughs::X_END_OF_ENUM>());
}

};

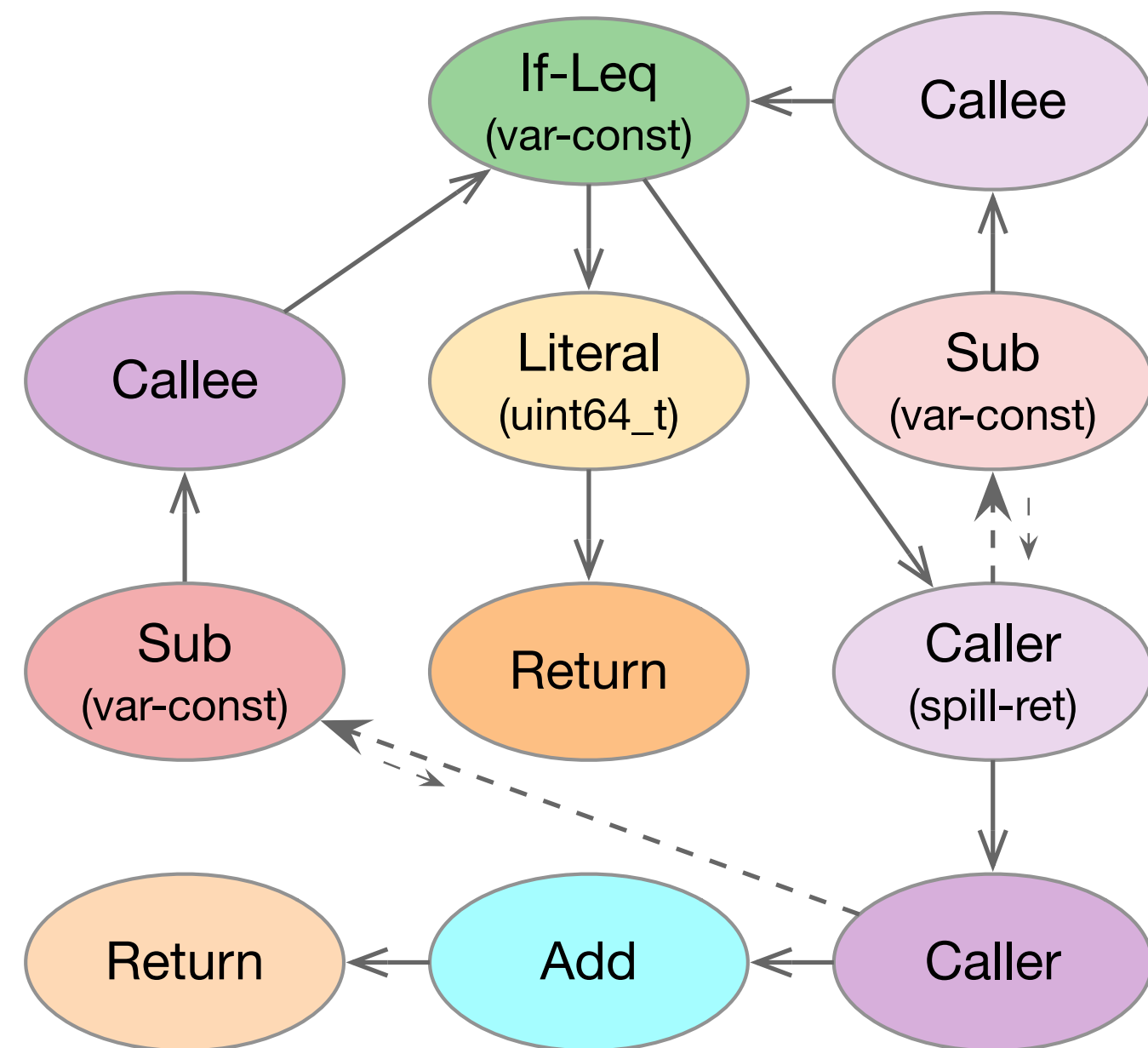
extern "C" void generate(StencilList* result) {
    runStencilGenerator<ArithAdd>(result);
}
```

Fibonacci compilation example

```

If(n <= 2).Then(
  Return(1ULL)
).Else(
  Return(Call<FibFn>("fib", n-1)
    + Call<FibFn>("fib", n-2))
)

```

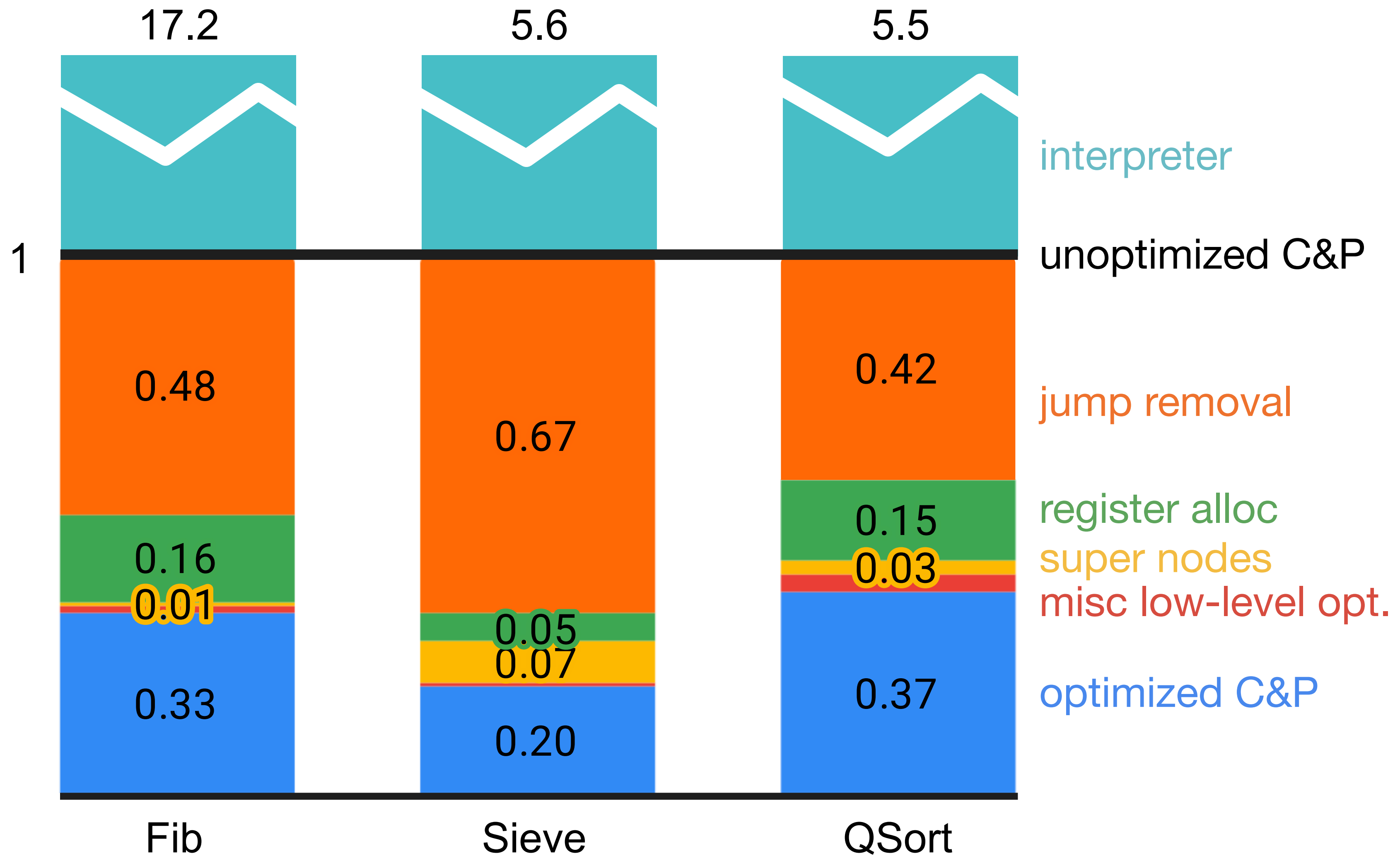


```

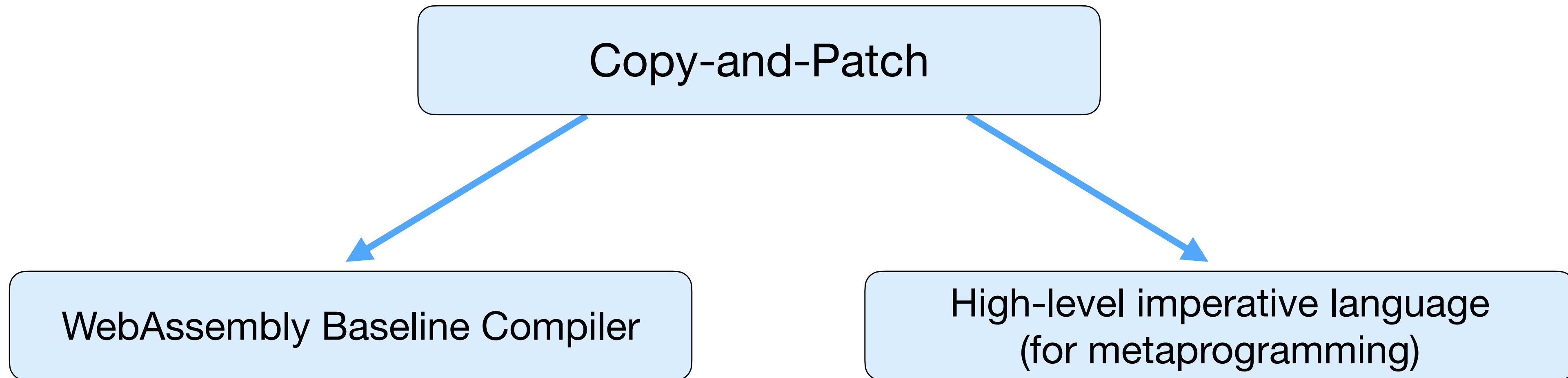
00: mov 0x8(%r13),%r12d
07: mov $0x2,%eax
0c: sub %eax,%r12d
0f: mov %r12d,0x8(%rbp)
13: mov %rbp,%r13
20: mov $0x2,%eax ← fib function entry
25: cmp %eax,0x8(%r13)
2c: jg 40
32: movabs $0x1,%rbp
3c: mov %rbp,%rax
3f: retq
40: sub $0x38,%rsp
44: mov %r13,0x8(%rsp)
49: lea 0x10(%rsp),%rbp
4e: callq 90
53: mov 0x8(%rsp),%r13
58: mov %rax,0x10(%r13) ← only spilled value
5f: add $0x38,%rsp
63: sub $0x38,%rsp
67: mov %r13,0x8(%rsp)
6c: lea 0x10(%rsp),%rbp
71: callq 00
76: mov 0x8(%rsp),%r13
7b: mov %rax,%rbp
7e: add $0x38,%rsp ← jumps between consecutive code blocks are removed
82: add 0x10(%r13),%rbp
89: mov %rbp,%rax
8c: retq
90: mov 0x8(%r13),%r12d
97: mov $0x1,%eax
9c: sub %eax,%r12d
9f: mov %r12d,0x8(%rbp)
a3: mov %rbp,%r13
a6: jmpq 20

```


Execution performance breakdown



Final copy-and-patch performance



	Compilation Speedup	Execution Speedup
Google Chrome Liftoff (baseline compiler)	4.9 – 6.5	1.46 – 1.63
Google Chrome TurboFan (optimizing compiler)	30 – 47 <small>(small module)</small> 88 – 91 <small>(large module)</small>	0.69 – 0.85

	Compilation Speedup	Execution Speedup
Interpreter	0.3 – 0.5	6 – 36
LLVM -O0	79 – 267	1.02 – 1.57
LLVM -O2	936 – 1384	0.61 – 0.96