Just In Time Compilation

Mainack Mondal
Sandip Chakraborty

CS 60203
Autumn 2024

Today's class

- **Interpreted Languages, and VMs**
  - Why Interpreters?
  - Interpreters can be slow :(
- What is Just-In-Time compilation?
  - To JIT, or not to JIT
  - Startup-time vs Execution-time tradeoff
  - Memory Requirements tradeoff
- How to design a JIT Compiler?
  - Case study 1: V8 JIT Explained
  - Case study 2: Copy-and-Patch in CPython JIT

Interpreted Languages and VMs

Compiled languages are a bottleneck

Compilation of source code into object code by the compiler

\- Wikipedia

(…why not say machine code?)

Compiled languages are a bottleneck

Compilation of source code into object code by the compiler

\- Wikipedia

(…why not say machine code?)

Solution: interpreted languages (Python, Java…)

What is an Interpreter? -Wikipedia

In computer science, an interpreter is a computer program that **directly executes instructions written** in a programming or scripting language, **without requiring them previously to have been compiled** into a machine language program

- Wikipedia

What is an Interpreter? –Theory people

An interpreter/VM is a program that consumes a series of instructions, and executes them against an abstract machine

Essentially VM emulates an abstract machine, and the behavior of the abstract machine itself is specified, for operations, and operands

Why (do we need) an Interpreter?

-Systems people

Why Interpreters?

...they **make life easy** :)

As implementing certain features becomes simpler!

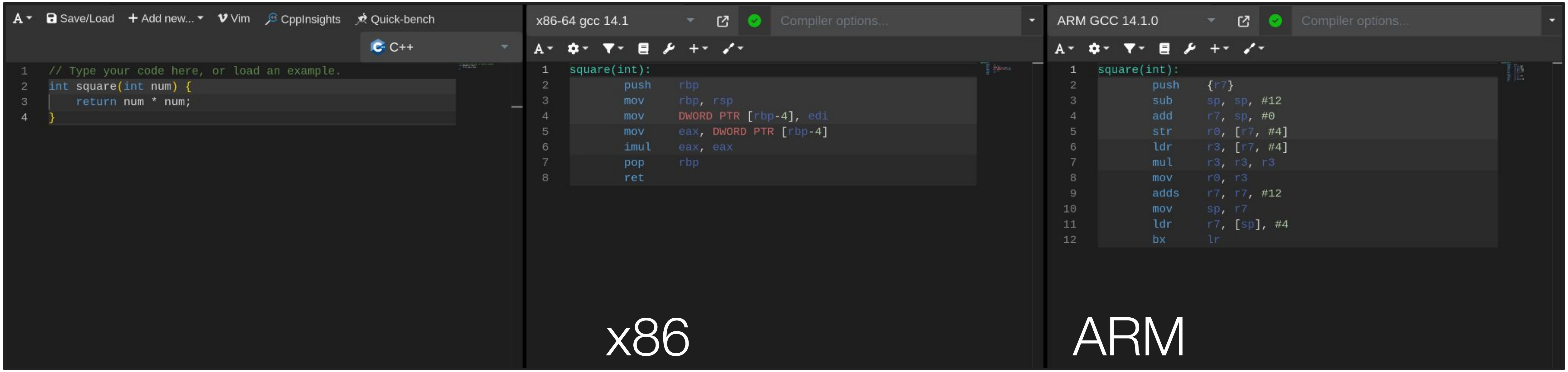
Why Interpreters?

- Platform Independence
- Reflection (we'll talk about this)
- Dynamic Typing (i.e. finding and/or changing types at runtime)
- Easy debugging and profiling
- *Easier* concurrency (concurrency is never easy)
- Small program size
- Automatic memory management (already talked about this)

Case #1 for interpreted languages

Platform Independence

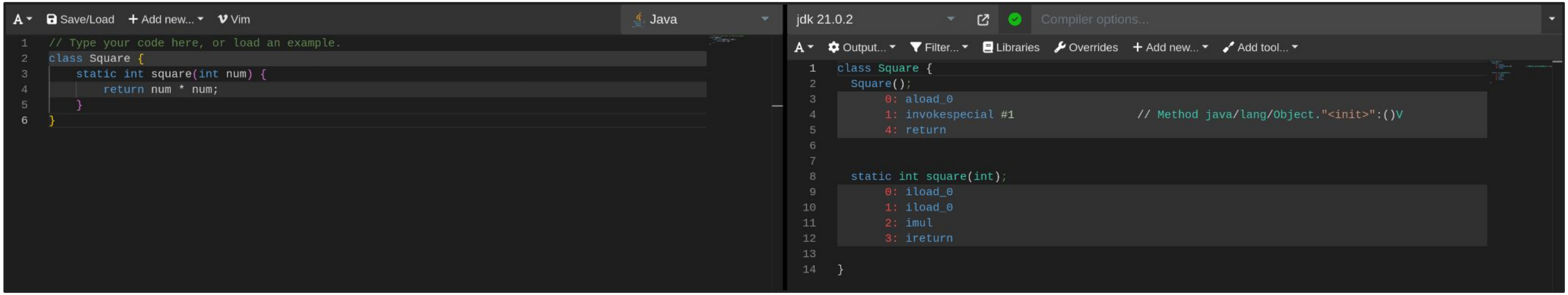
Compiler: Platform dependent code :((



https://godbolt.org/z/YbMebMo47

Even the square function changes :(

Interpreter: Platform independent code :)



Java bytecode is same everywhere :)

(…but what is bytecode?)

Case #2 for interpreted languages

More runtime type-information => Powerful features, and safety

Runtime Type Information

Type Information can be stored as actual **object in the language runtime!**

Allows for **dynamic types, dynamic dispatch, and reflection** (among other things)

Type Objects – Python 3.12.4 documentation

Type Objects

type **PyTypeObject**

Part of the Limited API (as an opaque struct).

The C structure of the objects used to describe built-in types.

PyTypeObject **PyType_Type**

Part of the Stable ABI.

This is the type object for type objects; it is the same object as `type` in the Python layer.

int **PyType_Check**(PyObject *o)

Return non-zero if the object *o* is a type object, including instances of types derived from the standard type object. Return 0 in all other cases. This function always succeeds.

int **PyType_CheckExact**(PyObject *o)

Return non-zero if the object *o* is a type object, but not a subtype of the standard type object. Return 0 in all other cases. This function always succeeds.

unsigned int **PyType_ClearCache**()

Part of the Stable ABI.

Clear the internal lookup cache. Return the current version tag.

unsigned long **PyType_GetFlags**(PyTypeObject *type)

Part of the Stable ABI.

Return the `tp_flags` member of *type*. This function is primarily meant for use with `Py_LIMITED_API`; the individual flag bits are guaranteed to be stable across Python releases, but access to `tp_flags` itself is not part of the limited API.

Added in version 3.2.

Changed in version 3.4: The return type is now `unsigned long` rather than `long`.

Reflection

Essentially **source code that "introspects" / "manipulates" source code**

```
import java.lang.reflect.*;


   public class DumpMethods {
      public static void main(String args[])
      {
         try {
            Class c = Class.forName(args[0]);
            Method m[] = c.getDeclaredMethods();
            for (int i = 0; i < m.length; i++)
            System.out.println(m[i].toString());
         }
         catch (Throwable e) {
            System.err.println(e);
         }
      }
   }
```

getting the methods, using a method

Reflection in C++ is hard!

668

There are several problems with reflection in C++.

- It's a lot of work to add, and the C++ committee is fairly conservative, and don't spend time on radical new features unless they're sure it'll pay off. (A suggestion for adding a module system similar to .NET assemblies has been made, and while I think there's general consensus that it'd be nice to have, it's not their top priority at the moment, and has been pushed back until well after C++0x. The motivation for this feature is to get rid of the `#include` system, but it would also enable at least some metadata).
- You don't pay for what you don't use. That's one of the must basic design philosophies underlying C++. Why should my code carry around metadata if I may never need it? Moreover, the addition of metadata may inhibit the compiler from optimizing. Why should I pay that cost in my code if I may never need that metadata?
- Which leads us to another big point: C++ makes *very* few guarantees about the compiled code. The compiler is allowed to do pretty much anything it likes, as long as the resulting functionality is what is expected. For example, your classes aren't required to actually *be there*. The compiler can optimize them away, inline everything they do, and it frequently does just that, because even simple template code tends to create quite a few template instantiations. The C++ standard library *relies* on this aggressive optimization. Functors are only performant if the overhead of instantiating and destructing the object can be optimized away. `operator[]` on a vector is only comparable to raw array indexing in performance

Why does C++ not have reflection? - Stack Overflow

Reflection in C++ is hard…but not impossible!

Reflection for C++26

Document #:	P2996R0
Date:	2023-10-15
Project:	Programming Language C++
Audience:	EWG
Reply-to:	Wyatt Childers <wcc@edg.com> Peter Dimov <pdimov@gmail.com> Barry Revzin <barry.revzin@gmail.com> Andrew Sutton <andrew.n.sutton@gmail.com> Faisal Vali <faisalv@gmail.com> Daveed Vandevoorde <daveed@edg.com>

Contents

Reflection for C++26

Case #3 for interpreted languages

Additional runtime accessible information, and instrumentation

Runtime Info: Code as a runtime object

```c
static PyCodeObject *
makecode(_PyCompile_CodeUnitMetadata *umd, struct assembler *a, PyObject *const_cache,
         PyObject *constslist, int maxdepth, int nlocalsplus, int code_flags,
         PyObject *filename)
{
    PyCodeObject *co = NULL;
    PyObject *names = NULL;
    PyObject *consts = NULL;
    PyObject *localsplusnames
    PyObject *localspluskinds
    names = dict_keys_inorder(
    if (!names) {
        goto error;
    }
    if (_PyCompile_ConstCacheM
        goto error;
    }

    consts = PyList_AsTuple(co
    if (consts == NULL) {
        goto error;
    }
    if (_PyCompile_ConstCacheM
        goto error;
    }
```

```c
struct _PyCodeConstructor con = {
    .filename = filename,
    .name = umd->u_name,
    .qualname = umd->u_qualname ? umd->u_qualname : umd->u_name,
    .flags = code_flags,

    .code = a->a_bytecode,
    .firstlineno = umd->u_firstlineno,
    .linetable = a->a_linetable,

    .consts = consts,
    .names = names,

    .localsplusnames = localsplusnames,
    .localspluskinds = localspluskinds,

    .argcount = posonlyargcount + posorkwargcount,
    .posonlyargcount = posonlyargcount,
    .kwonlyargcount = kwonlyargcount,

    .stacksize = maxdepth,

    .exceptiontable = a->a_except_table,
};
```

Interpreted languages (can) contain **code as a runtime object** too!

For example, **Python has PyCodeObject**, that “wraps” the bytecode

This is from Python/assemble.c

PyCodeObject, Docs

type **PyCodeObject**

The C structure of the objects used to describe code objects. The fields of this type are subject to change at any time.

PyTypeObject **PyCode_Type**

This is an instance of PyTypeObject representing the Python code object.

int **PyCode_Check**(PyObject *co)

Return true if *co* is a code object. This function always succeeds.

Py_ssize_t **PyCode_GetNumFree**(PyCodeObject *co)

Return the number of free variables in a code object.

int **PyCode_GetFirstFree**(PyCodeObject *co)

Return the position of the first free variable in a code object.

PyCodeObject ***PyUnstable_Code_New**(int argcount, int kwonlyargcount, int nlocals, int stacksize, int flags, PyObject *code, PyObject *consts, PyObject *names, PyObject *varnames, PyObject *freevars, PyObject *cellvars, PyObject *filename, PyObject *name, PyObject *qualname, int firstlineno, PyObject *linetable, PyObject *exceptiontable)

Code Objects — Python 3.12.4 documentation

Instrumentation!

```c
/* Count of all local monitoring events */
#define  _PY_MONITORING_LOCAL_EVENTS 10
/* Count of all "real" monitoring events (not derived from other events) */
#define _PY_MONITORING_UNGROUPED_EVENTS 15
/* Count of all  monitoring events */
#define _PY_MONITORING_EVENTS 17

/* Tables of which tools are active for each monitored event. */
typedef struct _Py_LocalMonitors {
    uint8_t tools[_PY_MONITORING_LOCAL_EVENTS];
} _Py_LocalMonitors;

typedef struct _Py_GlobalMonitors {
    uint8_t tools[_PY_MONITORING_UNGROUPED_EVENTS];
} _Py_GlobalMonitors;
```

Runtime information is valuable to find if something unexpected happened

Or **how often variables / functions are used / executed**

Recall, **instrumentation**

Python runtime also has instrumentation using _Py_*_Monitors

Benefits from keeping code at runtime?

- Easier **debugging**, and **program state inspection**
- Simple to implement **line-by-line profiling**
- Simple to implement **instrumentation**
- *(Spoiler) Just In Time Compilation!*

Takeaway….

Interpreters nice

The issue…

Interpreters can be slow :(

Interpreter vs. Compiler

Let's compare Python and C?

NO

Because its apples to oranges

Compare CPython with Cython

- Cython uses (largely)the same syntax as CPython
- Cython **compiles CPython into C**, using C/Python API and then compiles C, and the executes!

Matrix multiplication: CPython

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

where n is the number of columns in A and rows in B.
A basic implementation in pure Python looks like this:

```
def matmul(A, B, out):
    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            s = 0
            for k in range(A.shape[1]):
                s += A[i, k] * B[k, j]
            out[i,j] = s
```

Matrix multiplication: Cython (simple compilation)

```
def matmul(A, B, out):
    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            s = 0
            for k in range(A.shape[1]):
                s += A[i, k] * B[k, j]
            out[i,j] = s
```

```
tmp = PyTuple_New(2);
if (!tmp) { err_lineno = 21; goto error; }
Py_INCREF(i);
PyTuple_SET_ITEM(tmp, 0, i);
Py_INCREF(k);
PyTuple_SET_ITEM(tmp, 1, k);
A_ik = PyObject_GetItem(A, tmp);
if (!A_ik) { err_lineno = 21; goto error; }
Py_DECREF(tmp);
```

Direct compilation is (only) 1.15x faster

- lookup produces **pointer to Python object**
- and **PyNumber_Multiply** being used for PyObject

The situation gets way worse…
Interpreters get 700x slower …

Reason #1

Type generality prevents optimization!

Matrix multiplication: Cython, machine `types`

```
import numpy as np
cimport numpy as np
ctypedef np.float64_t dtype_t
def matmul(np.ndarray[dtype_t, ndim=2] A,
           np.ndarray[dtype_t, ndim=2] B,
           np.ndarray[dtype_t, ndim=2] out=None):
  cdef Py_ssize_t i, j, k
  cdef dtype_t s
  if A is None or B is None:
      raise ValueError("Input matrix cannot be None")
  for i in range(A.shape[0]):
      for j in range(B.shape[1]):
          s = 0
          for k in range(A.shape[1]):
              s += A[i, k] * B[k, j]
          out[i,j] = s
```

```
tmp_i = i; tmp_k = k;
if (tmp_i < 0) tmp_i += A_shape_0;
if (tmp_i < 0 || tmp_i >= A_shape_1) {
  PyErr_Format(<...>);
  err_lineno = 33; goto error;
}
if (tmp_k < 0) tmp_k += A_shape_1;
if (tmp_k < 0 || tmp_k >= A_shape_1) {
  PyErr_Format(<...>);
  err_lineno = 33; goto error;
}
A_ik = *(dtype_t*)(A_data +
  tmp_i * A_stride_0 + tmp_k * A_stride_1);
```

180-190x faster than CPython!

Bounds checking is slow :(

Reason #2

Interpreters can’t optimize out bounds checks!

(security bros get mad)

Matrix multiplication: Cython, no bounds check

```
cimport cython
@cython.boundscheck(False)
@cython.wraparound(False)
def matmul(np.ndarray[dtype_t, ndim=2] A,
           np.ndarray[dtype_t, ndim=2] B,
           np.ndarray[dtype_t, ndim=2] out=None):
    <...>
```

bounds check removed

700-800x faster!

Today's class

- Interpreted Languages, and VMs
  - Why Interpreters?
  - Interpreters can be slow :(
- **What is Just-In-Time compilation?**
  - To JIT, or not to JIT
  - Startup-time vs Execution-time tradeoff
  - Memory Requirements tradeoff
- How to design a JIT Compiler?
  - Case study 1: V8 JIT Explained
  - Case study 2: Copy-and-Patch in CPython JIT

What is Just-in-Time compilation?

What is Just-in-Time compilation?

Just-In-Time compilation is **compilation (of computer code) during execution** of a program (at run time) rather than before execution

This may consist of source code translation but is more commonly bytecode translation to machine code, which is then executed directly.

- Wikipedia

Refresher: Code as a runtime object

```c
static PyCodeObject *
makecode(_PyCompile_CodeUnitMetadata *umd, struct assembler *a, PyObject *const_cache,
         PyObject *constslist, int maxdepth, int nlocalsplus, int code_flags,
         PyObject *filename)
{
    PyCodeObject *co = NULL;
    PyObject *names = NULL;
    PyObject *consts = NULL;
    PyObject *localsplusnames
    PyObject *localspluskinds
    names = dict_keys_inorder(
    if (!names) {
        goto error;
    }
    if (_PyCompile_ConstCacheM
        goto error;
    }

    consts = PyList_AsTuple(co
    if (consts == NULL) {
        goto error;
    }
    if (_PyCompile_ConstCacheM
        goto error;
    }
```

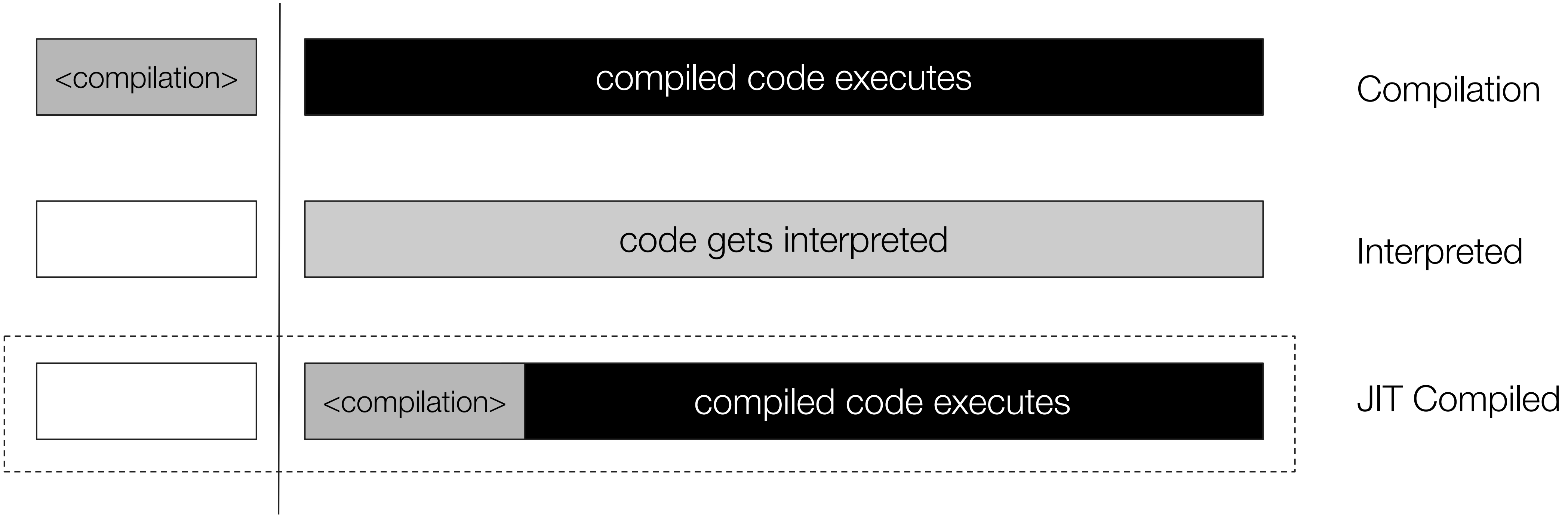
```c
struct _PyCodeConstructor con = {
    .filename = filename,
    .name = umd->u_name,
    .qualname = umd->u_qualname ? umd->u_qualname : umd->u_name,
    .flags = code_flags,

    .code = a->a_bytecode,
    .firstlineno = umd->u_firstlineno,
    .linetable = a->a_linetable,

    .consts = consts,
    .names = names,

    .localsplusnames = localsplusnames,
    .localspluskinds = localspluskinds,

    .argcount = posonlyargcount + posorkwargcount,
    .posonlyargcount = posonlyargcount,
    .kwonlyargcount = kwonlyargcount,

    .stacksize = maxdepth,

    .exceptiontable = a->a_except_table,
};
```

Interpreted languages (can) contain **code as a runtime object** too!

For example, **Python has PyCodeObject**, that “wraps” the bytecode

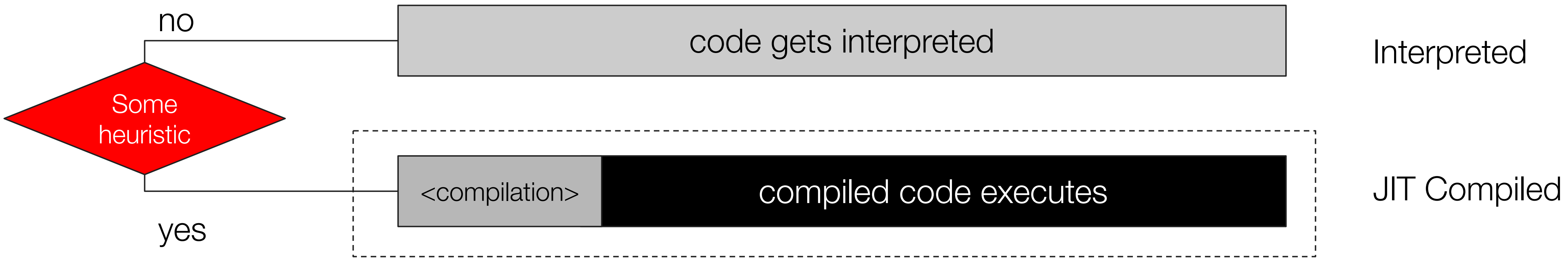
This is from Python/assemble.c

What is Just-in-Time compilation?



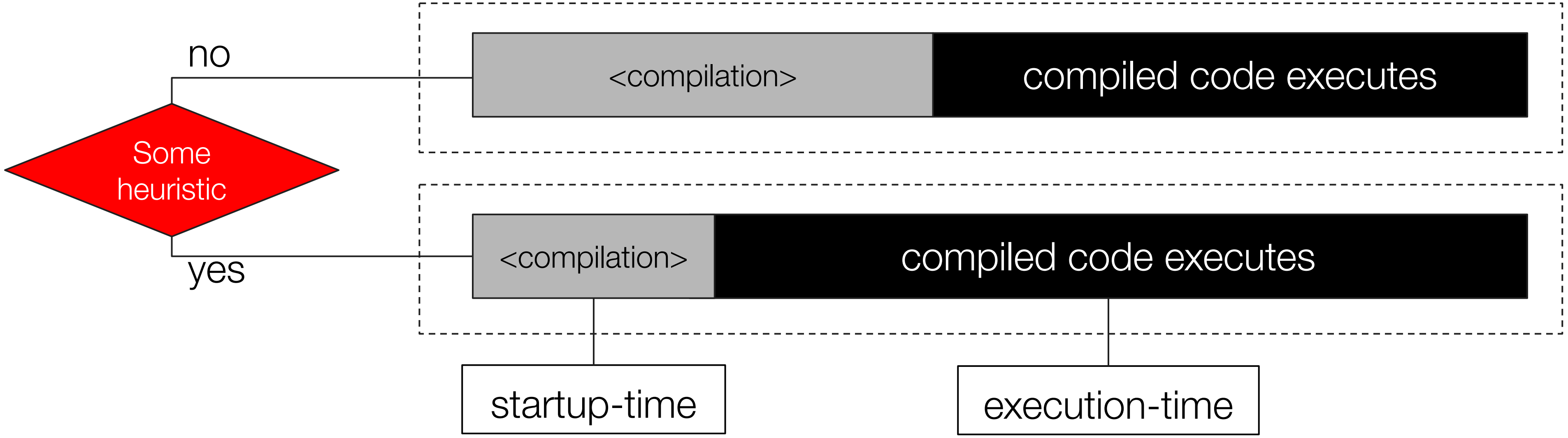
Just-in-Time compilation involves conversion of (a part of) source/bytecode into machine code at runtime (and not in advance)

To JIT, or not to JIT



Interpreted languages get executed line-by-line (or instruction-by-instructions) hence it is **possible to only compile parts of the code and interpret the rest**

Startup-time vs Execution time tradeoff



Start-up time is the time taken by the JIT compiler to **produce the machine code**

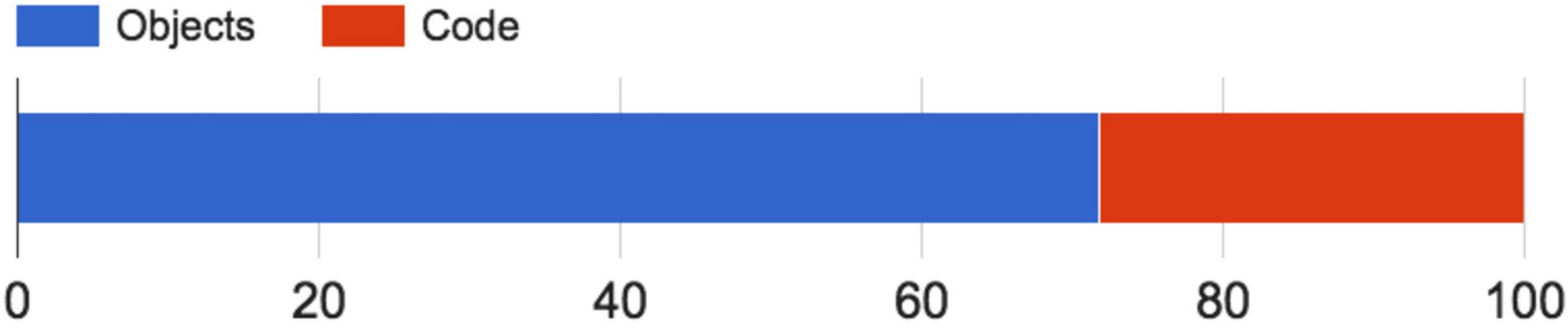
Execution time is time taken by the **machine code to execute**

Startup-time vs Execution time tradeoff

The trade-off exists because it is possible to use sophisticated compilers to produce **optimized machine code**.

But such compilers would be **slow to produce** the machine code.

Remember code objects? They also take up space :(



V8 heap usage by code-objects

V8: Hooking up the Ignition to the Turbofan

Memory requirements tradeoff

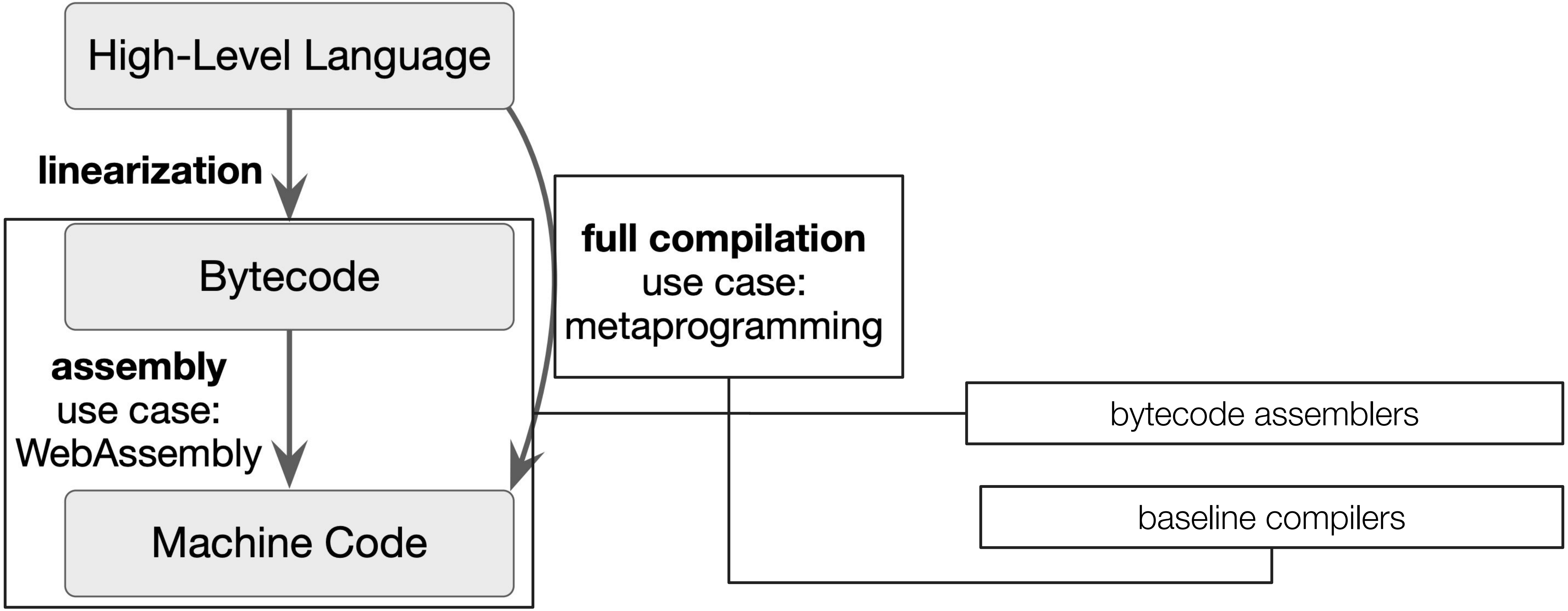
- Code objects take up space.
- Compilers that produce unoptimized code fast, **produce a lot of code**.
- Compilers that produce optimized code are **too slow to run** in user facing scenarios :(

Today's class

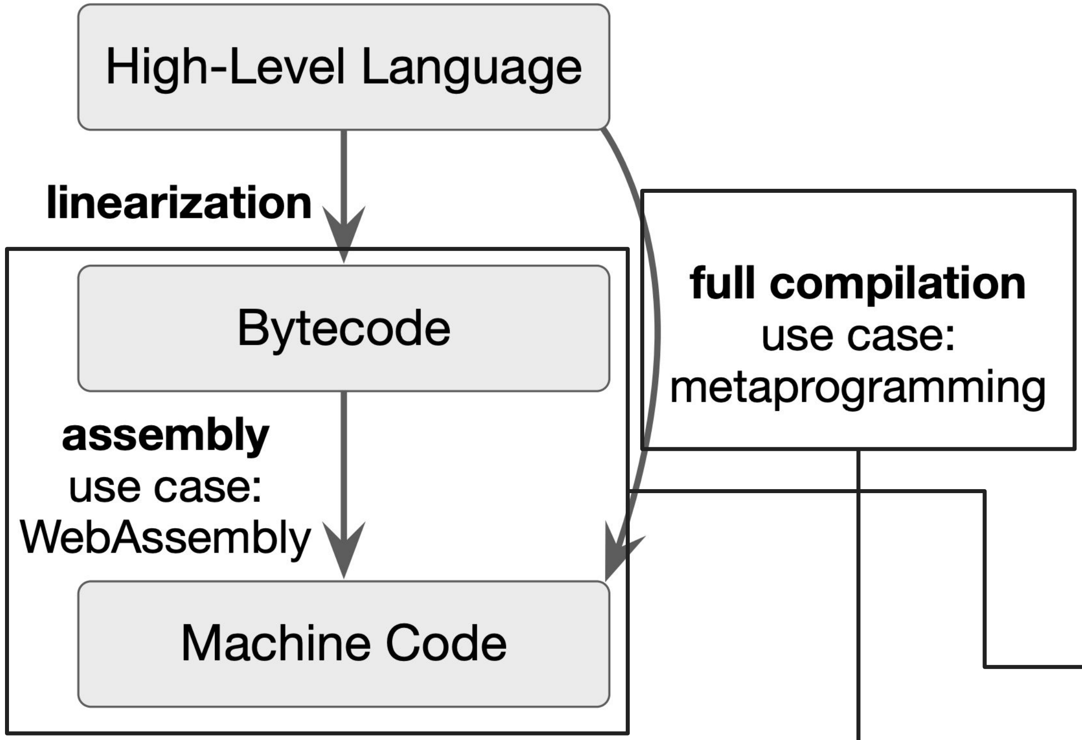
- Interpreted Languages, and VMs
    - Why Interpreters?
    - Interpreters can be slow :(
- What is Just-In-Time compilation?
    - Startup-time vs Execution-time tradeoff
    - Memory Requirements tradeoff
- **How to design a JIT Compiler?**
    - Case study 1: V8 JIT Explained
    - Case study 2: Copy-and-Patch in CPython JIT

How to design a JIT Compiler?

Refresher: Compilation Approaches



Refresher: Compilation Approaches

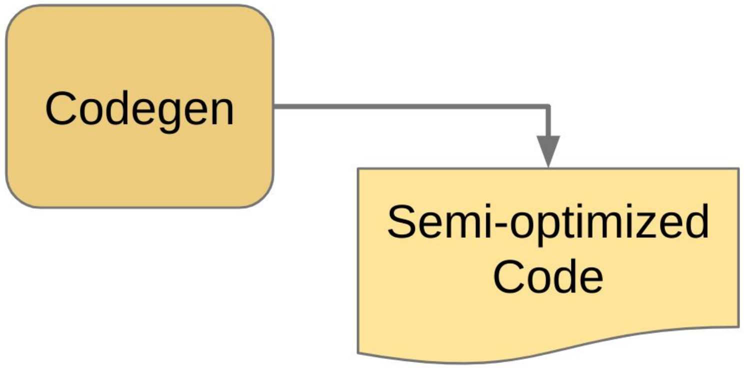


- Most databases go by the bytecode route
- Python also goes the bytecode route

This is because writing bytecode assemblers is easier than baseline compilers

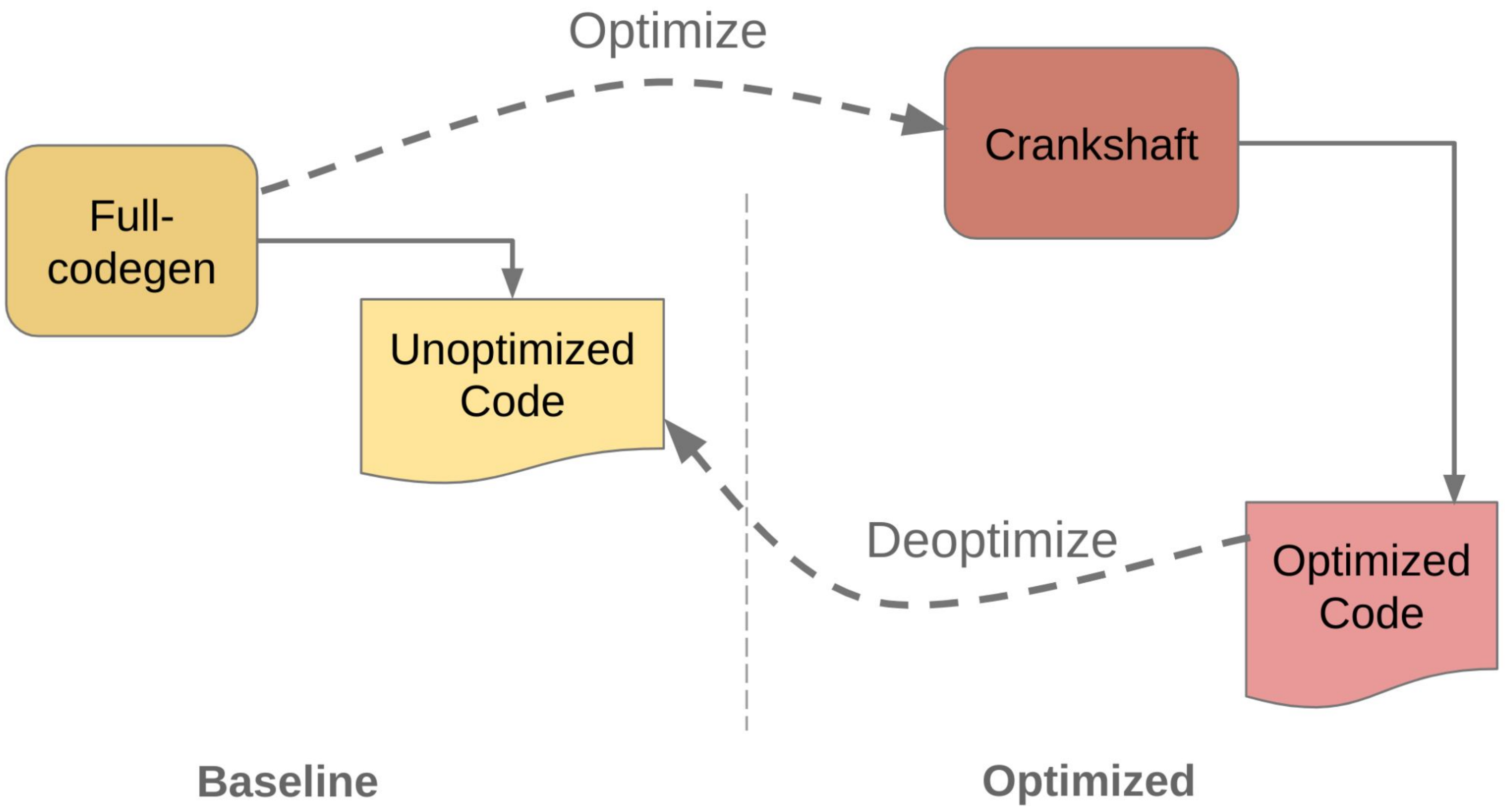
V8 JIT Explained

V8- 2007



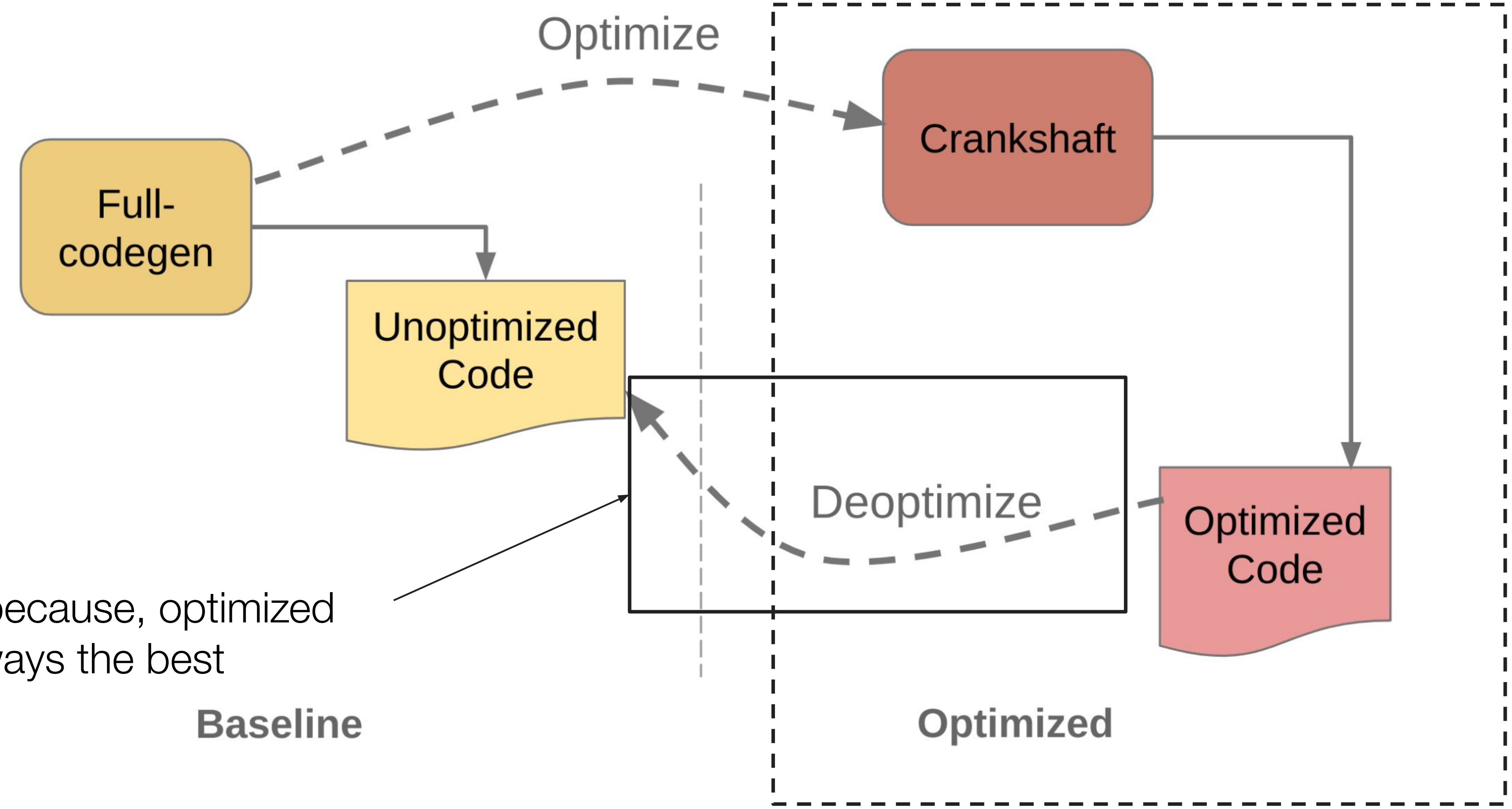
Baseline

V8- 2010


Optimize
Full-codegen
Crankshaft
Unoptimized Code
Deoptimize
Optimized Code
Baseline
Optimized

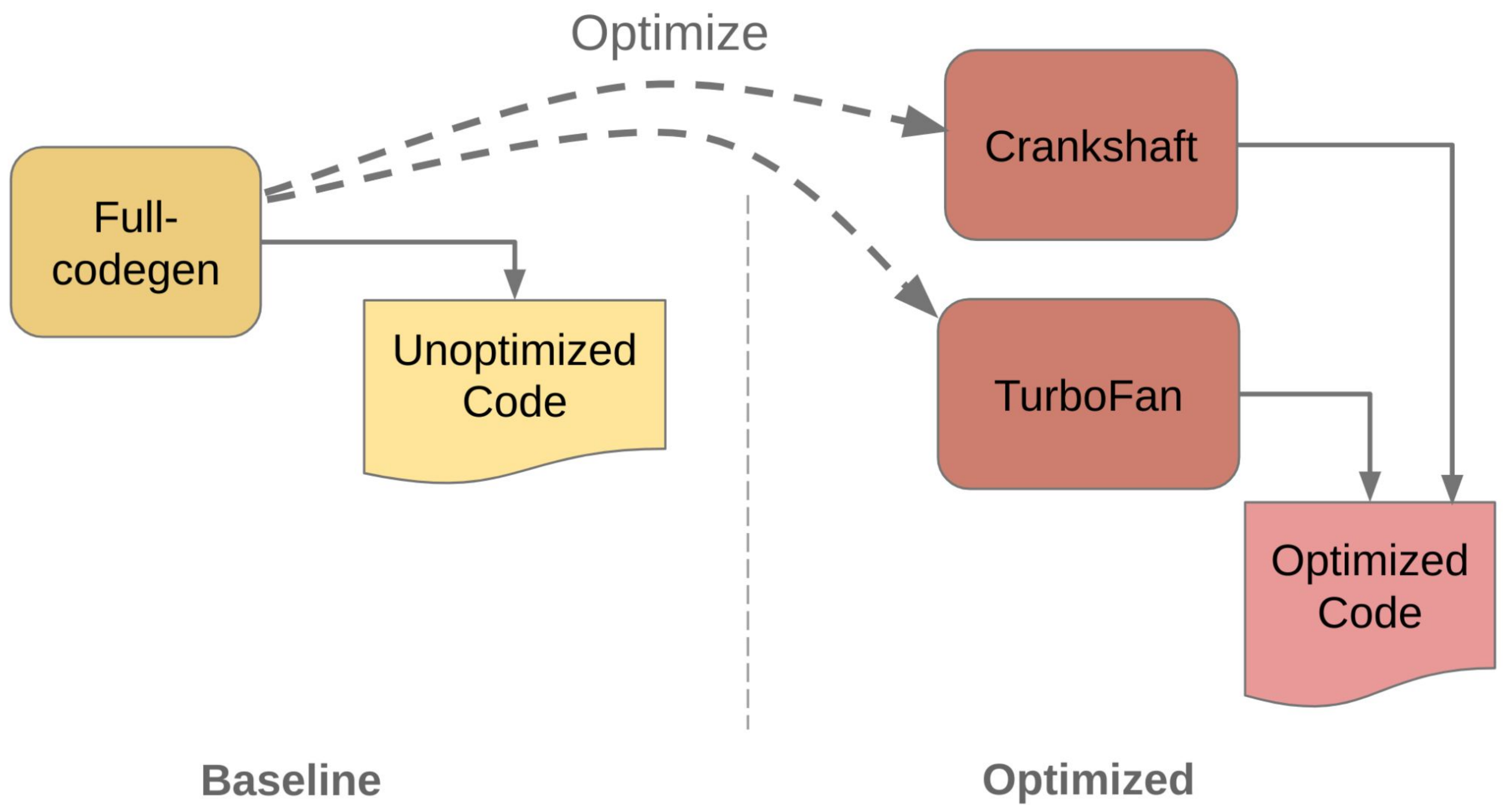
V8- 2010

They added **optimizing compiler**



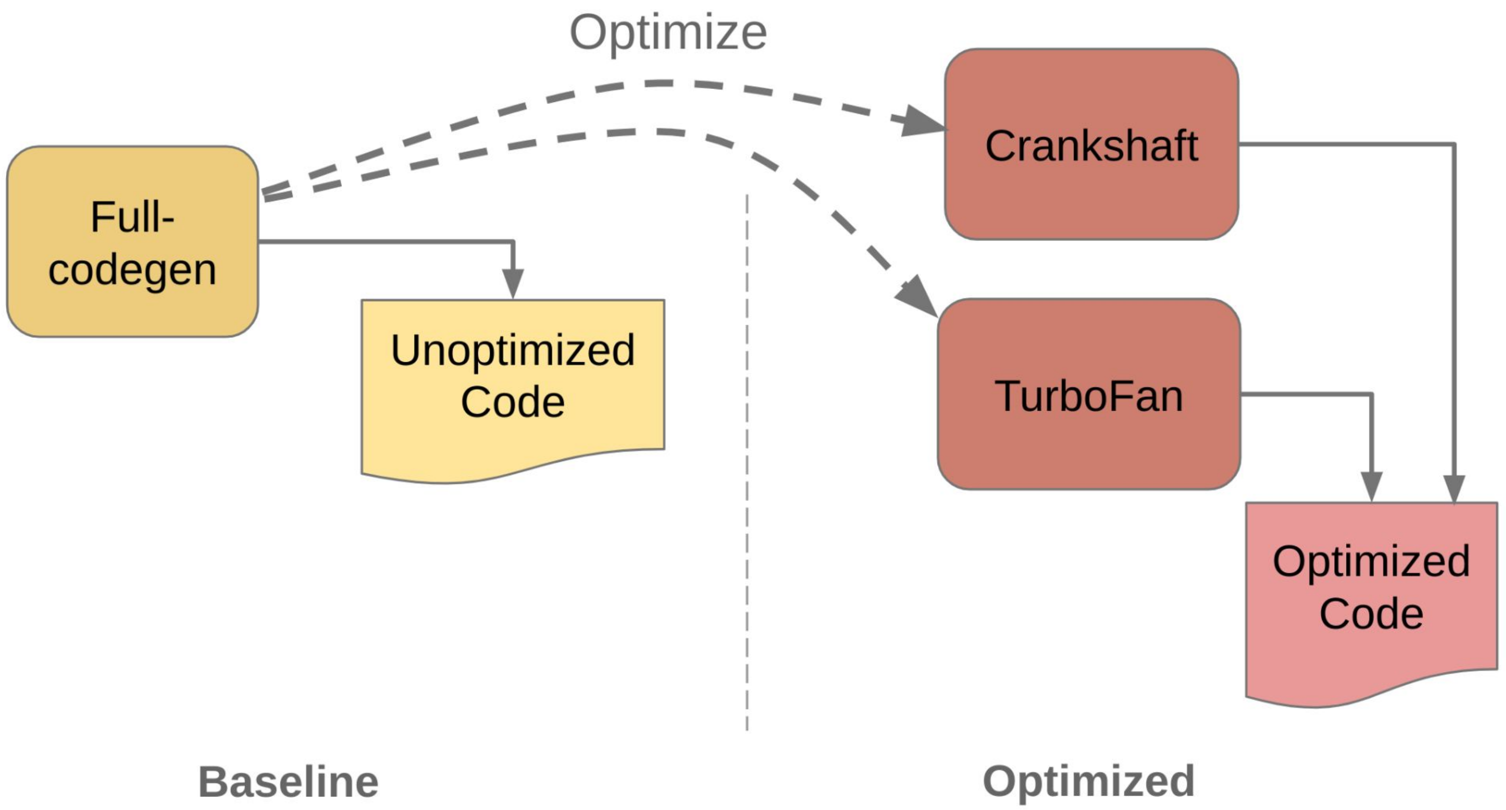
Deoptimizer because, optimized code isn't always the best

V8- 2015



Optimize
Full-codegen
Unoptimized Code
Crankshaft
TurboFan
Optimized Code
Baseline
Optimized

Older, and *simpler* V8



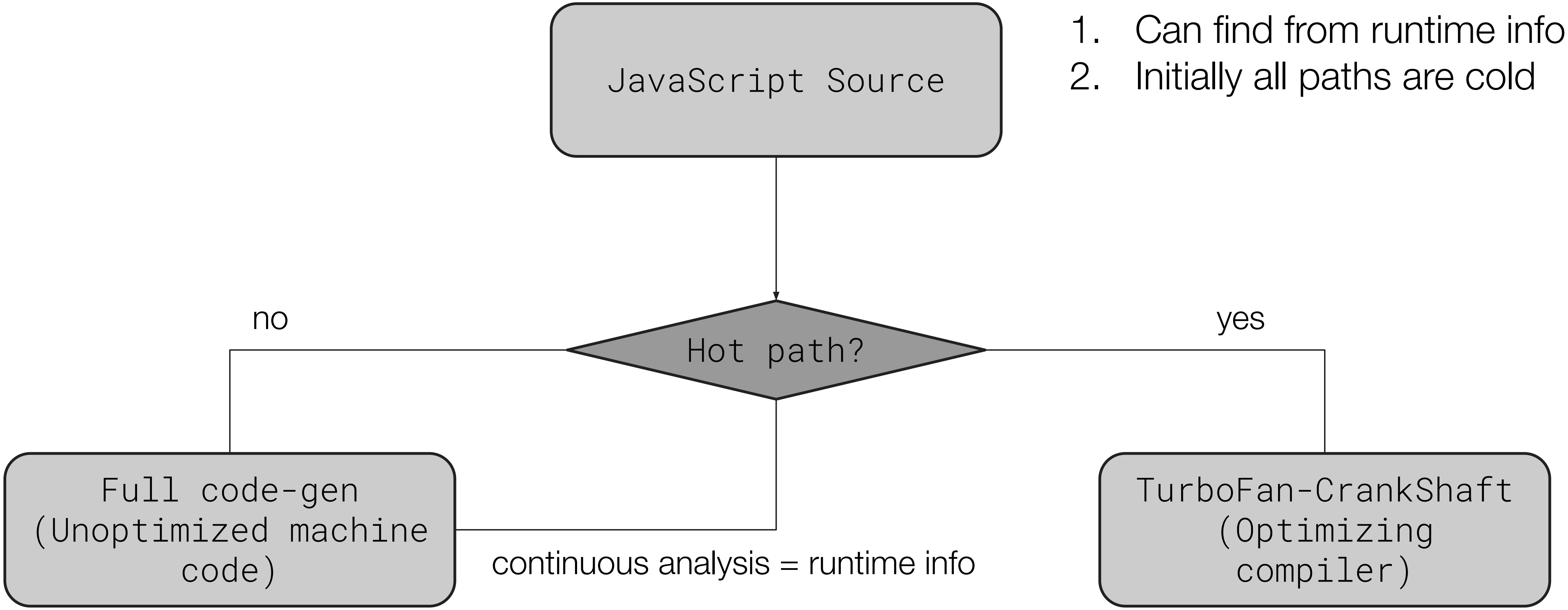
A great **summary of the history of V8 architecture**, because it can be confusing

V8: Hooking up the Ignition to the Turbofan

Older, and *simpler* V8

Hot path: executes often

1. Can find from runtime info!
2. Initially all paths are cold

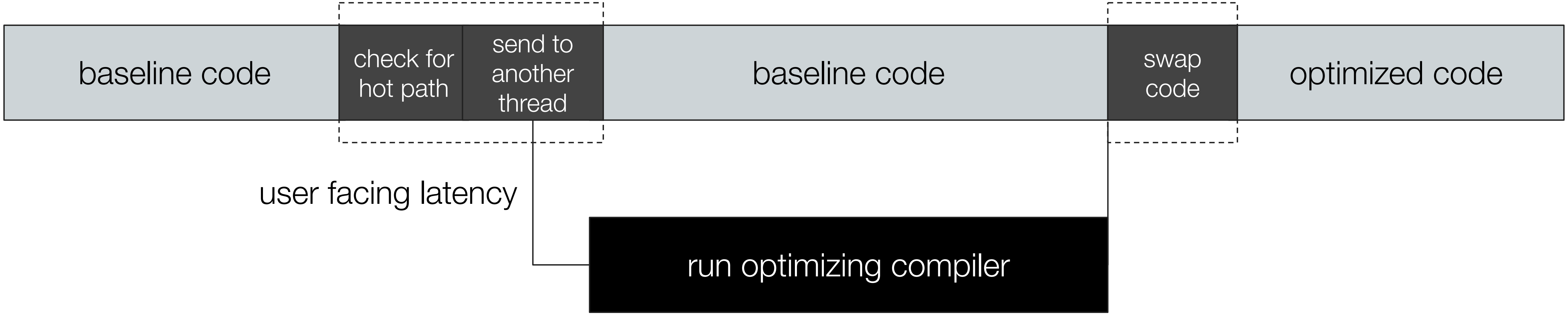


V8 uses a “dumb” full code-gen to **generate code fast, hence leading to slower code**, but faster execution time!

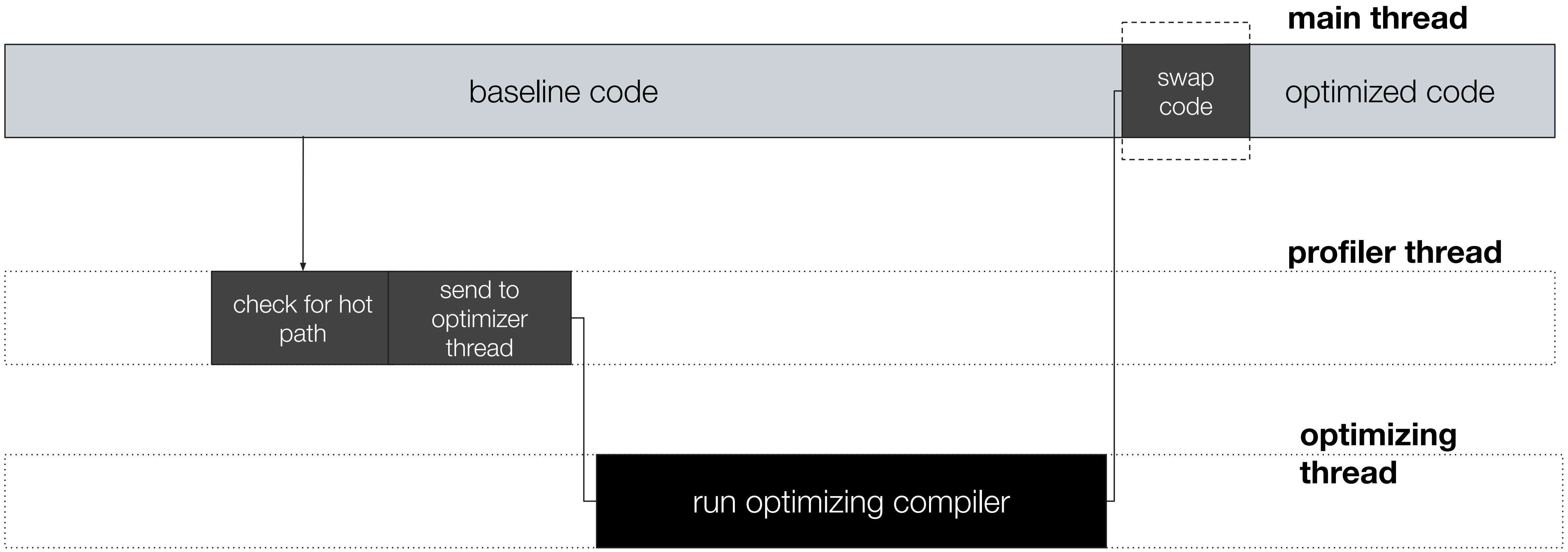
Takeaway: Start up time vs execution time trade-off

Problem: Optimizing compilers are slow :((

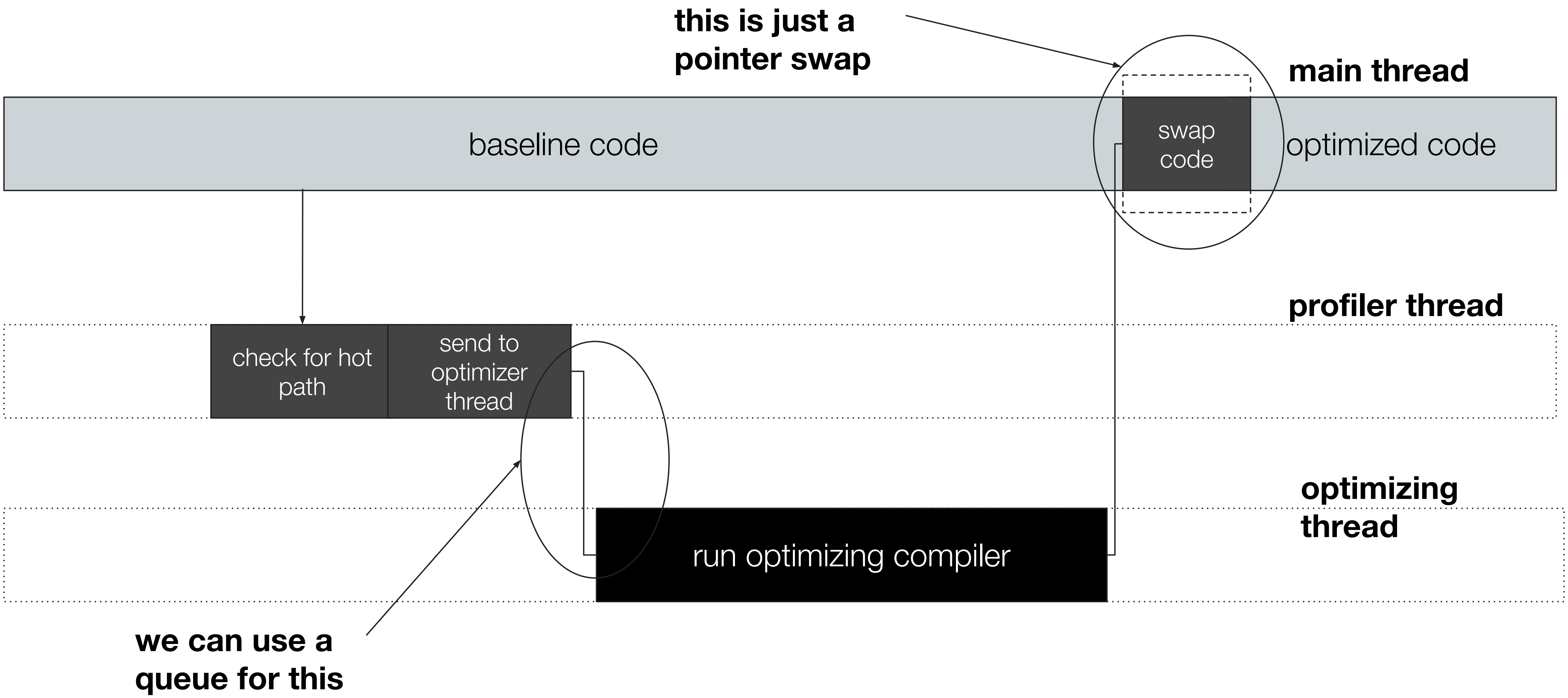
Solution: Optimizing compilers on another thread



Even better: Profiling and Optimizing on other thread(s)

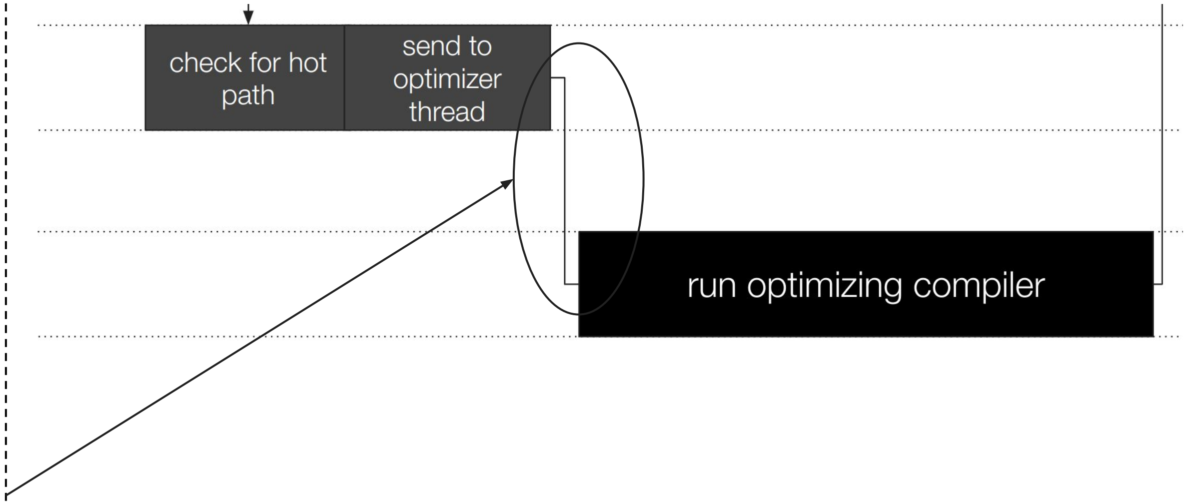


Even better: Profiling and Optimizing on other thread(s)



Code object is submitted to a `DispatcherQueue`

```
// Circular queue of incoming
recompilation tasks (including OSR).
class V8_EXPORT
OptimizingCompileDispatcherQueue {

private:
    …
    TurbofanCompilationJob** queue_;
    int capacity_;
    int length_;
    int shift_;
    base::Mutex mutex_;
};
```



https://chromium.googlesource.com/v8/v8.git/+/refs/heads/main/src/compiler-dispatcher/optimizing-compile-dispatcher.h

Code object is submitted to a DispatcherQueue

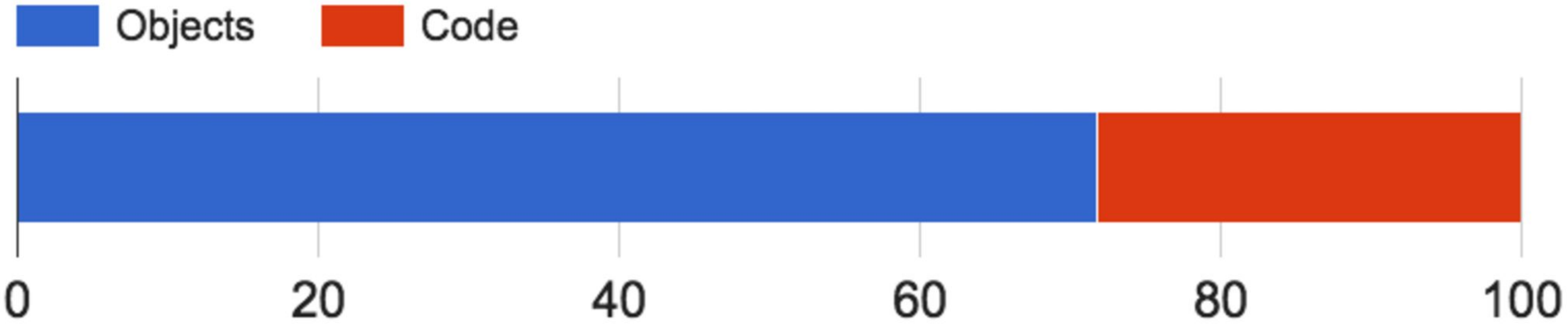
```
// Circular queue of incoming recompilation tasks (including OSR).
class V8_EXPORT OptimizingCompileDispatcherQueue {
public:
…
explicit OptimizingCompileDispatcherQueue(int capacity)
: capacity_(capacity), length_(0), shift_(0) {
   queue_ = NewArray<TurbofanCompilationJob*>(capacity_);
}
~OptimizingCompileDispatcherQueue() { DeleteArray(queue_); }
TurbofanCompilationJob* Dequeue();
void Enqueue(TurbofanCompilationJob* job);
void Flush(Isolate* isolate);
void Prioritize(Tagged<SharedFunctionInfo> function);
```

basic queue ops

V8 moved **optimizing compiler to another thread**, and only did a “dumb” full code-gen in the main thread

Takeaway: Optimization compilers can be run in other threads

Recall: Machine code takes memory :((



V8 heap usage by code-objects

[V8: Hooking up the Ignition to the Turbofan]

Problem: V8 Engine memory issues

- The V8 JavaScript Engine used to do a **`full code-gen`**, using the **baseline compiler**, generating **non-optimized machine code** fast

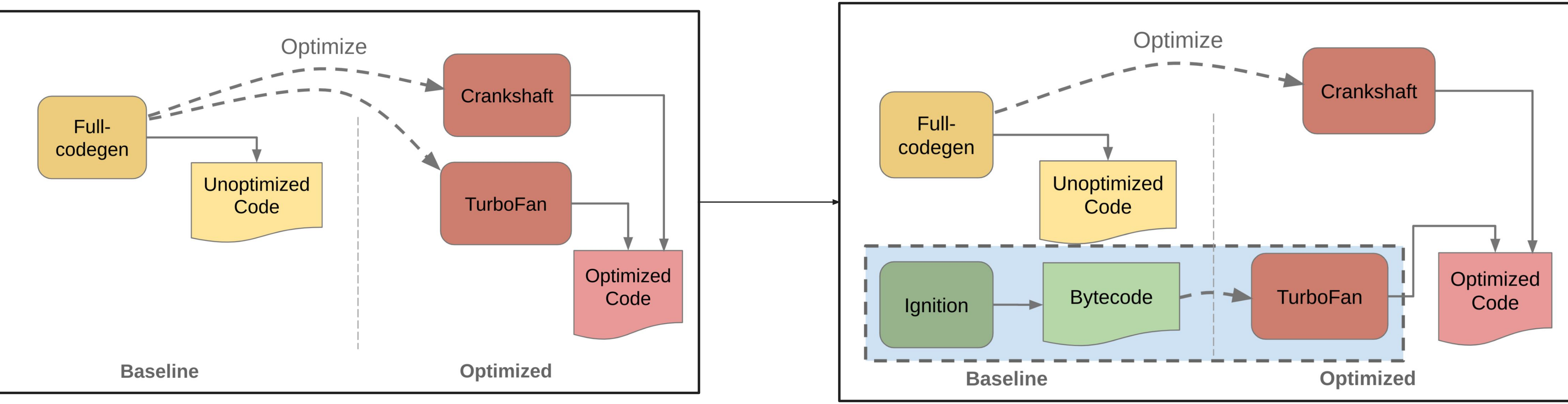
- **JITed machine code can consume a significant amount of memory**, even if the code is only executed once

Solution: Bytecode interpreter instead of full code-gen

- Bytecode is **between 50% to 25% the size** of the equivalent baseline machine code.

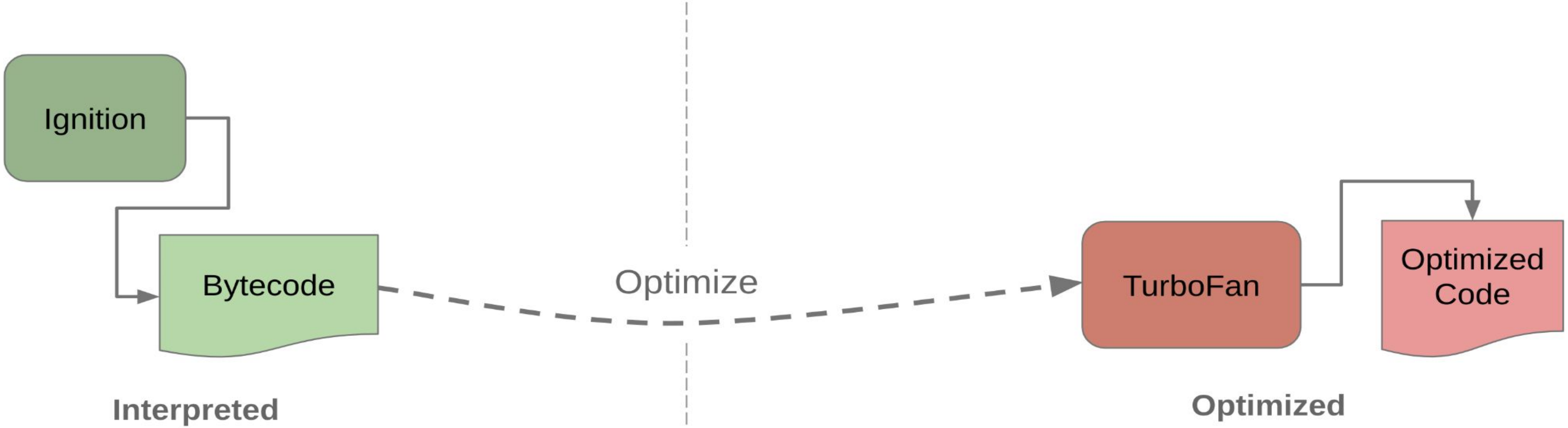
- Bytecode is executed by Ignition which yields **execution speeds** on real-world websites **close to those of code generated by V8's existing baseline compiler**

Introducing Interpreter: Less simple V8



[Firing up the Ignition interpreter]()

Finally in 2017



[Firing up the Ignition interpreter]

Takeaway: Memory requirement of machine code is a trade-off

Copy-and-Patch: CPython JIT

Copy-and-Patch –The paper

(1) The concept of a binary stencil, which is a pre-built implementation of an AST node or bytecode opcode with missing values (immediate literals, stack variable offsets, and branch and call targets) to be patched in at runtime.

(2) An algorithm that uses a library with many binary stencil variants to emit optimized machine code. There are two types of variants: one that enumerates different parameter configurations (whether they are literals, in different registers, or on the stack) and one that enumerates different code patterns (a single AST node/bytecode or a supernode of a common AST subtree/bytecode sequence).

(3) An algorithm that linearizes high-level language constructs like if-statements and loops, and generates machine code by composing multiple binary stencil fragments.

(4) A system called MetaVar for generating binary stencils, which allows the user to systematically generate the binary stencil variants in clean and pure C++, and leverages the Clang+LLVM compiler infrastructure to hide all platform-specific low-level detail.

Copy-and-Patch -**Systems people**

Relatively recent research work on a new way to do JIT compilation!

1. Keep a **table of compiled templates** (called stencils) to “copy” into the code when needed
2. For information available later, **keep “parameters” that you can fill**
3. “**Patch**” the parameter values in the stencil Just-in-Time, and run

Refresher: Python uses bytecode

```python
import dis

# Example function to analyze
def calculate(a, b):
    result = a + b
    return result

# Using dis.dis()
print("Output of dis.dis():")
dis.dis(calculate)
print()
```

```
Output of dis.dis():
4           0 RESUME                   0

  5           2 LOAD_FAST                0 (a)
              4 LOAD_FAST                1 (b)
              6 BINARY_OP                0 (+)
             10 STORE_FAST               2 (result)

  6          12 LOAD_FAST                2 (result)
             14 RETURN_VALUE
```

Refresher: Python uses bytecode

```
replicate(8) pure inst(LOAD_FAST, (-- value)) {
    assert(!PyStackRef_IsNull(GETLOCAL(oparg)));
    value = PyStackRef_DUP(GETLOCAL(oparg));
}

inst(LOAD_FAST_AND_CLEAR, (-- value)) {
    value = GETLOCAL(oparg);
    // do not use SETLOCAL here, it decrefs the old value
    GETLOCAL(oparg) = PyStackRef_NULL;
}

inst(LOAD_FAST_LOAD_FAST, ( -- value1, value2)) {
    uint32_t oparg1 = oparg >> 4;
    uint32_t oparg2 = oparg & 15;
    value1 = PyStackRef_DUP(GETLOCAL(oparg1));
    value2 = PyStackRef_DUP(GETLOCAL(oparg2));
}

pure inst(LOAD_CONST, (-- value)) {
    value = PyStackRef_FromPyObjectNew(GETITEM(FRAME_CO_CONSTS, oparg));
}
```

This is what your Python code compiles to :)

1. Table of Compiled Templates

```
replicate(8) pure inst(LOAD_FAST, (-- value)) {
    assert(!PyStackRef_IsNull(GETLOCAL(oparg)));
    value = PyStackRef_DUP(GETLOCAL(oparg));
}

inst(LOAD_FAST_AND_CLEAR, (-- value)) {
    value = GETLOCAL(oparg);
    // do not use SETLOCAL here, it decrefs the old value
    GETLOCAL(oparg) = PyStackRef_NULL;
}

inst(LOAD_FAST_LOAD_FAST, ( -- value1, value2)) {
    uint32_t oparg1 = oparg >> 4;
    uint32_t oparg2 = oparg & 15;
    value1 = PyStackRef_DUP(GETLOCAL(oparg1));
    value2 = PyStackRef_DUP(GETLOCAL(oparg2));
}

pure inst(LOAD_CONST, (-- value)) {
    value = PyStackRef_FromPyObjectNew(GETITEM(FRAME_CO_CONSTS, oparg));
}
```

entry in table

When building with **--enable-experimental-jit**

C code for bytecode execution is copied. This C code is then built into a shared library.

cpython/Python/bytecodes.c at main

2. Leaving blanks for parameters

```
0000000000000000 <__JIT_ENTRY>:
pushq   %rbp
movq    %rsp, %rbp
movq    (%rdi), %rax
movq    0x28(%rax), %rax
movabsq $0x0, %rcx
000000000000000d:  X86_64_RELOC_UNSIGNED        __JIT_OPARG
movzwl  %cx, %ecx
movq    0x28(%rax,%rcx,8), %rax
movl    0xc(%rax), %ecx
incl    %ecx
je      0x3d <__JIT_ENTRY+0x3d>
movq    %gs:0x0, %r8
cmpq    (%rax), %r8
jne     0x37 <__JIT_ENTRY+0x37>
movl    %ecx, 0xc(%rax)
jmp     0x3d <__JIT_ENTRY+0x3d>
lock
addq    $0x4, 0x10(%rax)
movq    %rax, (%rsi)
addq    $0x8, %rsi
movabsq $0x0, %rax
0000000000000046:  X86_64_RELOC_UNSIGNED        __JIT_CONTINUE
popq    %rbp
jmpq    *%rax
```

For variables determined at runtime, **code is compiled with those parameters left as 0**

All of the **machine code is then stored as a sequence of bytes in the file jit_stencil.h** which is automatically generated by a new build stage

The information of **what goes is the blanks** is available from the runtime!

3. Patch and roll!

Why Copy-and-Patch?

Full JIT compilers convert op-codes to an IR, and then machine code, and are not considered because they're huge, slow, and-

- Java-based JITs for (GraalPy, and Jython) can take up to **100 times longer to start** than normal CPython
- These implementation would also take upto **1GB extra RAM!**

“The WebAssembly compiler uses **1666 stencils taking 35 kB** and the high-level compiler uses 98,831 stencils taking 17.5 MB”

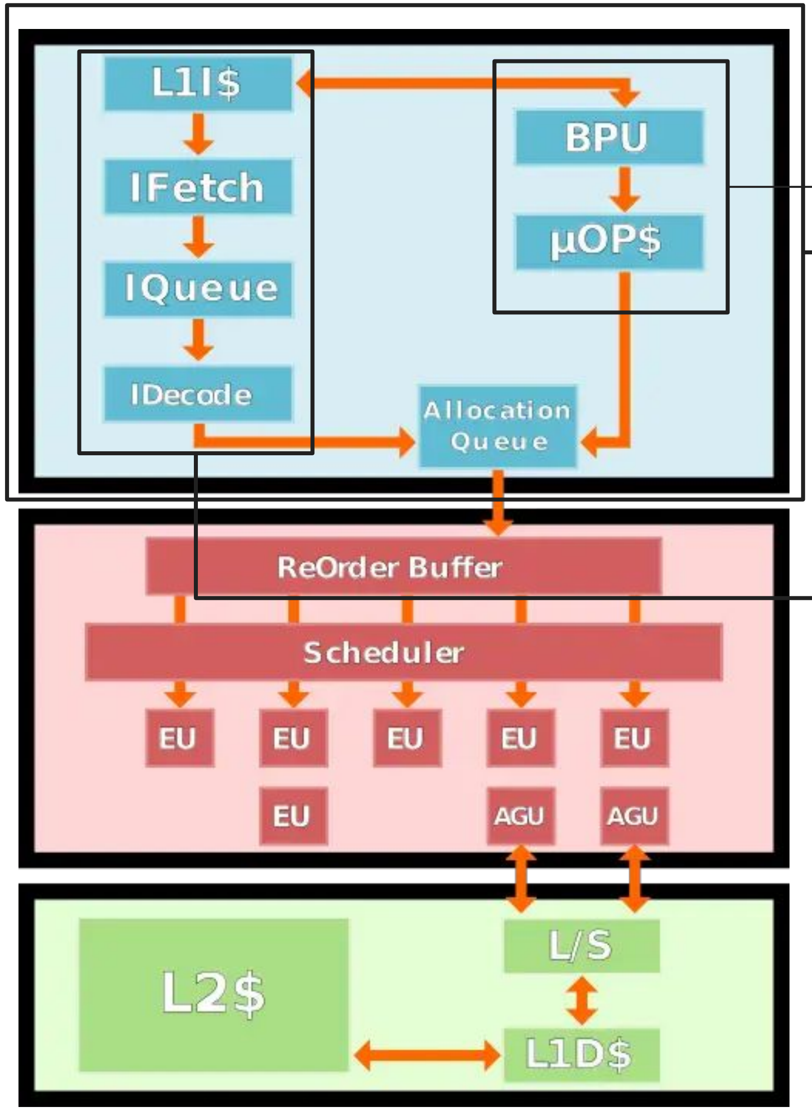
Lesson: Copy-and-Patch compilation can be used for fast compilation with **minimal memory overhead**!

Interesting stuff that did not fit in

Appendix

Parallels between a processor and a VM

How does a real machine work?



The goal of the front-end is to feed the back-end with a sufficient stream of operations which it gets by decoding instructions coming from memory.

The front-end has two major pathways: the **µOPs cache path** and the **legacy path.** The legacy path is the traditional path whereby variable-length x86 instructions are fetched from the level 1 instruction cache, queued, and consequently get decoded into simpler, fixed-length µOPs.

Interpreted Languages

Now that we have a sense for a **hardware machine**, it is **easier to understand how one can emulate** an abstract machine in software

- Python
- Javascript
- SQL
- Java

…

There are many more, but we will talk about these

The JVM Specification

[Chapter 2. The Structure of the Java Virtual Machine]

Table of Contents

Chapter 2. The Structure of the Java Virtual Machine

This document specifies an abstract machine. It does not describe any particular implementation of the Java Virtual Machine.

To implement the Java Virtual Machine correctly, you need only be able to read the `class` file format and correctly perform the operations specified therein. Implementation details that are not part of the Java Virtual Machine's specification would unnecessarily constrain the creativity of implementors. For example, the memory layout of run-time data areas, the garbage-collection algorithm used, and any internal optimization of the Java Virtual Machine instructions (for example, translating them into machine code) are left to the discretion of the implementor.

All references to Unicode in this specification are given with respect to *The Unicode Standard, Version 6.0.0*, available at `http://www.unicode.org/`.

2.1. The class File Format

Compiled code to be executed by the Java Virtual Machine is represented using a hardware- and operating system-independent binary format, typically (but not necessarily) stored in a file, known as the `class` file format. The `class` file format precisely defines the representation of a class or interface, including details such as byte ordering that might be taken for granted in a platform-specific object file format.

Chapter 4, "The `class` File Format", covers the `class` file format in detail.

Fun fact! **JVM doesn't have a native bool type**

2.2. Data Types

Like the Java programming language, the Java Virtual Machine operates on two kinds of types: *primitive types* and *reference types*. There are, correspondingly, two kinds of values that can be stored in variables, passed as arguments, returned by methods, and operated upon: *primitive values* and *reference values*.

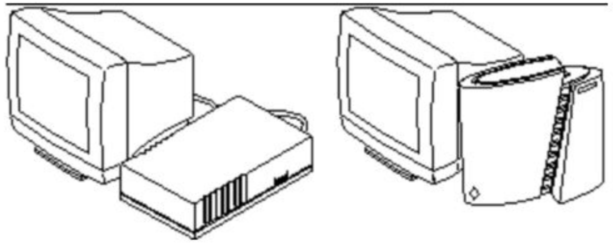
The Java Virtual Machine expects that nearly all type checking is done prior to run time, typically by a compiler, and does not have to be done by the Java Virtual Machine itself. Values of primitive types need not be tagged or otherwise be inspectable to determine their types at run time, or to be distinguished from values of reference t[2.2. Data Types] instruction set of the Java Virtual Machine distinguishes its operand types using instructions intended to operate on values of specific types. For instance, *iadd*, *ladd*, *fadd*, and *dadd* are al[...]hine instructions that add two numeric values and produce numeric results, but each is specialized for its operand type: `int`, `long`, `float`, and `double`, respectively. For a summary of type support in the Java Virtual Machine instruction set, see §2.11.1.

The Java Virtual Machine contains explicit support for objects. An object is either a dynamically allocated class instance or an array. A reference to an object is considered to have Java Virtual Machine type `reference`. Values of type `reference` can be thought of as pointers to objects. More than one reference to an object may exist. Objects are always operated on, passed, and tested via values of type `reference`.

2.3. Primitive Types and Values

The primitive data types supported by the Java Virtual Machine are the *numeric types*, the `boolean` type (§2.3.4), and the `returnAddress` type (§2.3.3).

Aside: Java station! Hardware, running JavaOS



Brick Model

The first-generation brick model JavaStation computer includes the following features:

- **microSPARC-II** - The brick model JavaStation computer is equipped with a 100 MHz microSPARC-II processor.
- **Scalable memory** - The brick model includes 8-64 Mbytes DRAM (64-bit memory bus) and a PC-compatible memory system comprising four SIMM slots (2 logical banks, 2 SIMMs per bank). Memory size can be increased by installing 4-Mbyte or 16-Mbyte SIMMs in the slots.
- **Device connectors** - Connectors for a PS2 mouse, a PS2 keyboard, and a 14-inch or 17-inch monitor are included.
- **Serial port** - A serial port enables local printing to a PostScript(TM) or PCL5 printer.
- **Power switch** - The brick model includes a continuous contact, long life industrial grade rocker switch for power cycling. The power switch is located at the rear of the unit.

JavaStation Hardware

How does CPython execute?

How does CPython execute?

1. Tokenize the source code Parser/lexer/ and Parser/tokenizer/.
2. Parse the stream of tokens into an Abstract Syntax Tree Parser/parser.c.
3. Transform AST into an instruction sequence Python/compile.c.
4. Construct a Control Flow Graph and apply optimizations to it Python/flowgraph.c.
5. Emit bytecode based on the Control Flow Graph Python/assemble.c.

How does CPython execute?

The AST is generated from source code using **_PyParser_ASTFromString() or _PyParser_ASTFromFile()**

```
typedef struct {
    struct tok_state *tok;
    Token **tokens;
    int mark;
    int fill, size;
    PyArena *arena;
    KeywordToken **keywords;
    char **soft_keywords;
    int n_keyword_lists;
    int start_rule;
    int *errcode;
    int parsing_started;
    PyObject* normalize;
    int starting_lineno;
    int starting_col_offset;
    int error_indicator;
    int flags;
    int feature_version;
    growable_comment_array type_ignore_comments;
    Token *known_err_token;
    int level;
    int call_invalid_rules;
    int debug;
} Parser;
```

```
struct assembler {
    PyObject *a_bytecode;  /* bytes containing bytecode */
    int a_offset;              /* offset into bytecode */
    PyObject *a_except_table;  /* bytes containing exception table */
    int a_except_table_off;    /* offset into exception table */
    /* Location Info */
    int a_lineno;          /* lineno of last emitted instruction */
    PyObject* a_linetable; /* bytes containing location info */
    int a_location_off;    /* offset of last written location info frame */
};
```

Parser/peg_api.c.

After some checks, a helper function in Parser/parser.c begins applying production rules

Peeking into CPython: `ast` module

Literals

class ast.**Constant**(*value*)

A constant value. The `value` attribute of the `Constant` literal contains the Python object it represents. The values represented can be simple types such as a number, string or `None`, but also immutable container types (tuples and frozensets) if all of their elements are constant.

```
>>> print(ast.dump(ast.parse('123', mode='eval'), indent=4))
Expression(
    body=Constant(value=123))
```

class ast.**FormattedValue**(*value*, *conversion*, *format_spec*)

Node representing a single formatting field in an f-string. If the string contains a single formatting field and nothing else the node can be isolated otherwise it appears in `JoinedStr`.

- `value` is any expression node (such as a literal, a variable, or a function call).
- `conversion` is an integer:
  - -1: no formatting
  - 115: `!s` string formatting
  - 114: `!r` repr formatting
  - 97: `!a` ascii formatting
- `format_spec` is a `JoinedStr` node representing the formatting of the value, or `None` if no format was specified. Both `conversion` and `format_spec` can be set at the same time.

`ast` is a module in the python standard library.

Python codes need to be converted to an *Abstract Syntax Tree (AST)*

`ast` module: Grammar for Python

[ast — Abstract Syntax Trees — Python 3.12.4 documentation]

```
stmt = FunctionDef(identifier name, arguments args,
                   stmt* body, expr* decorator_list, expr? returns,
                   string? type_comment, type_param* type_params)
      | AsyncFunctionDef(identifier name, arguments args,
                         stmt* body, expr* decorator_list, expr? returns,
                         string? type_comment, type_param* type_params)

      | ClassDef(identifier name,
         expr* bases,
         keyword* keywords,
         stmt* body,
         expr* decorator_list,
         type_param* type_params)
      | Return(expr? value)

      | Delete(expr* targets)
      | Assign(expr* targets, expr value, string? type_comment)
      | TypeAlias(expr name, type_param* type_params, expr value)
      | AugAssign(expr target, operator op, expr value)
      -- 'simple' indicates that we annotate simple name without parens
      | AnnAssign(expr target, expr annotation, expr? value, int simple)
```

```
     -- BoolOp() can use left & right?
expr = BoolOp(boolop op, expr* values)
     | NamedExpr(expr target, expr value)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     | Lambda(arguments args, expr body)
     | IfExp(expr test, expr body, expr orelse)
     | Dict(expr* keys, expr* values)
     | Set(expr* elts)
     | ListComp(expr elt, comprehension* generators)
     | SetComp(expr elt, comprehension* generators)
     | DictComp(expr key, expr value, comprehension* generators)
     | GeneratorExp(expr elt, comprehension* generators)
     -- the grammar constrains where yield expressions can occur
     | Await(expr value)
     | Yield(expr? value)
     | YieldFrom(expr value)
     -- need sequences for compare to distinguish between
     -- x < 4 < 3 and (x < 4) < 3
     | Compare(expr left, cmpop* ops, expr* comparators)
     | Call(expr func, expr* args, keyword* keywords)
     | FormattedValue(expr value, int conversion, expr? format_spec)
     | JoinedStr(expr* values)
     | Constant(constant value, string? kind)
```

```
expr_context = Load | Store | Del

boolop = And | Or

operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
             | RShift | BitOr | BitXor | BitAnd | FloorDiv

unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

comprehension = (expr target, expr iter, expr* ifs, int is_async)
```

`ast` module: types for nodes

ast — Abstract Syntax Trees — Python 3.12.4 documentation

class ast.**Module**(*body*, *type_ignores*)

A Python module, as with file input. Node type generated by `ast.parse()` in the default `"exec"` *mode*.

body is a `list` of the module's Statements.

type_ignores is a `list` of the module's type ignore comments; see `ast.parse()` for more details.

```
>>> print(ast.dump(ast.parse('x = 1'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1))],
    type_ignores=[])
```

class ast.**FunctionType**(*argtypes*, *returns*)

A representation of an old-style type comments for functions, as Python versions prior to 3.5 didn't support PEP 484 annotations. Node type generated by `ast.parse()` when *mode* is `"func_type"`.

Such type comments would look like this:

```
def sum_two_number(a, b):
    # type: (int, int) -> int
    return a + b
```

argtypes is a `list` of expression nodes.

returns is a single expression node.

```
>>> print(ast.dump(ast.parse('(int, str) -> List[int]', mode='func_type'), indent=4))
FunctionType(
    argtypes=[
        Name(id='int', ctx=Load()),
        Name(id='str', ctx=Load())],
    returns=Subscript(
        value=Name(id='List', ctx=Load()),
        slice=Name(id='int', ctx=Load()),
        ctx=Load()))
```

class ast.**Expression**(*body*)

A single Python expression input. Node type generated by `ast.parse()` when *mode* is `"eval"`.

body is a single node, one of the expression types.

```
>>> print(ast.dump(ast.parse('123', mode='eval'), indent=4))
Expression(
    body=Constant(value=123))
```

Peeking into CPython: `dis` the Python disassembler

`dis.dis(x=None, *, file=None, depth=None, show_caches=False, adaptive=False)`

Disassemble the *x* object. *x* can denote either a module, a class, a method, a function, a generator, an asynchronous generator, a coroutine, a code object, a string of source code or a byte sequence of raw bytecode. For a module, it disassembles all functions. For a class, it disassembles all methods (including class and static methods). For a code object or sequence of raw bytecode, it prints one line per bytecode instruction. It also recursively disassembles nested code objects. These can include generator expressions, nested functions, the bodies of nested classes, and the code objects used for annotation scopes. Strings are first compiled to code objects with the `compile()` built-in function before being disassembled. If no object is provided, this function disassembles the last traceback.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

The maximal depth of recursion is limited by *depth* unless it is `None`. `depth=0` means no recursion.

If *show_caches* is `True`, this function will display inline cache entries used by the interpreter to specialize the bytecode.

If *adaptive* is `True`, this function will display specialized bytecode that may be different from the original

class `dis.Instruction`¶

Details for a bytecode operation

`opcode`

numeric code for operation, corresponding to the opcode values listed below and the bytecode values in the Opcode collections.

`opname`

human readable name for operation

`arg`

numeric argument to operation (if any), otherwise `None`

`argval`

resolved arg value (if any), otherwise `None`

`argrepr`

human readable description of operation argument (if any), otherwise an empty string.

Unary operations

Unary operations take the top of the stack, apply the operation, and push the result back on the stack.

`UNARY_NEGATIVE`

Implements `STACK[-1] = -STACK[-1]`.

`UNARY_NOT`

Implements `STACK[-1] = not STACK[-1]`.

`UNARY_INVERT`

Implements `STACK[-1] = ~STACK[-1]`.

`GET_ITER`

Implements `STACK[-1] = iter(STACK[-1])`.

`GET_YIELD_FROM_ITER`

If `STACK[-1]` is a generator iterator or coroutine object it is left as is. Otherwise, implements `STACK[-1] = iter(STACK[-1])`.