A JIT compiler for CPython

Brandt Bucher (October 10th, 2023)

Background

Background

- CPython 3.11:
  - Specializing adaptive interpreter profiles programs and optimizes them on-the-fly
- CPython 3.12:
  - Interpreter is generated from a DSL, allowing analysis and modification at build time
- CPython 3.13:
  - Internal pipeline for detecting, optimizing, and executing hot code paths

Background

```
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a
```

Background

```
for _ in range(n):
    a, b = b, a + b
```

Background

Bytecode

```
for _ in range(n):
    a, b = b, a + b
```

Background

Bytecode

```
for _ in range(n):
    a, b = b, a + b
```

```
FOR_ITER
STORE_FAST
LOAD_FAST_LOAD_FAST
LOAD_FAST
BINARY_OP
STORE_FAST_STORE_FAST
JUMP_BACKWARD
```

Background

Bytecode

```
FOR_ITER
STORE_FAST
LOAD_FAST_LOAD_FAST
```

```
LOAD_FAST
BINARY_OP
STORE_FAST_STORE_FAST
JUMP_BACKWARD
```

Background

CPython 3.11: Specialized Bytecode

```
FOR_ITER
STORE_FAST
LOAD_FAST_LOAD_FAST
```

```
LOAD_FAST
BINARY_OP
STORE_FAST_STORE_FAST
JUMP_BACKWARD
```

Background

CPython 3.11: Specialized Bytecode

```
FOR_ITER
STORE_FAST
LOAD_FAST_LOAD_FAST
```

```
LOAD_FAST
BINARY_OP
STORE_FAST_STORE_FAST
JUMP_BACKWARD
```

Background

CPython 3.11: Specialized Bytecode

```
FOR_ITER_RANGE
STORE_FAST
LOAD_FAST_LOAD_FAST
```

```
LOAD_FAST
BINARY_OP_ADD_INT
STORE_FAST_STORE_FAST
JUMP_BACKWARD
```

Background

CPython 3.11: Specialized Bytecode

```
FOR_ITER_RANGE
STORE_FAST
LOAD_FAST_LOAD_FAST
```

```
LOAD_FAST
BINARY_OP_ADD_INT
STORE_FAST_STORE_FAST
JUMP_BACKWARD
```

Background

CPython 3.13: Micro-Op Traces

```
FOR_ITER_RANGE
STORE_FAST
LOAD_FAST_LOAD_FAST
```

```
LOAD_FAST
BINARY_OP_ADD_INT
STORE_FAST_STORE_FAST
JUMP_BACKWARD
```

Background

CPython 3.13: Micro-Op Traces (`-X uops`)

```
FOR_ITER_RANGE
STORE_FAST
LOAD_FAST_LOAD_FAST
```

```
LOAD_FAST
BINARY_OP_ADD_INT
STORE_FAST_STORE_FAST
JUMP_BACKWARD
```

Background

CPython 3.13: Micro-Op Traces (`-X uops`)

```
FOR_ITER_RANGE

STORE_FAST

LOAD_FAST_LOAD_FAST
```

```
LOAD_FAST

BINARY_OP_ADD_INT

STORE_FAST_STORE_FAST

JUMP_BACKWARD
```

Background

CPython 3.13: Micro-Op Traces (`-X uops`)

```
FOR_ITER_RANGE

STORE_FAST

LOAD_FAST_LOAD_FAST

LOAD_FAST

BINARY_OP_ADD_INT

STORE_FAST_STORE_FAST

JUMP_BACKWARD
```

Background

CPython 3.13: Micro-Op Traces (`-X uops`)

```
_SET_IP
_ITER_CHECK_RANGE
_IS_ITER_EXHAUSTED_RANGE
_POP_JUMP_IF_TRUE
_ITER_NEXT_RANGE

_SET_IP
STORE_FAST

_SET_IP
LOAD_FAST
LOAD_FAST
```

```
_SET_IP
LOAD_FAST

_SET_IP
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT

_SET_IP
STORE_FAST
STORE_FAST

_SET_IP
_JUMP_TO_TOP
```

Background

CPython 3.13: Micro-Op Traces (`-X uops`)

```
_SET_IP
_ITER_CHECK_RANGE
_IS_ITER_EXHAUSTED_RANGE
_POP_JUMP_IF_TRUE
_ITER_NEXT_RANGE

_SET_IP
STORE_FAST

_SET_IP
LOAD_FAST
LOAD_FAST
```

```
_SET_IP
LOAD_FAST

_SET_IP
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT

_SET_IP
STORE_FAST
STORE_FAST

_SET_IP
_JUMP_TO_TOP
```

Background

CPython 3.13: Optimized Micro-Op Traces

```
_SET_IP
_ITER_CHECK_RANGE
_IS_ITER_EXHAUSTED_RANGE
_POP_JUMP_IF_TRUE
_ITER_NEXT_RANGE

_SET_IP
STORE_FAST

_SET_IP
LOAD_FAST
LOAD_FAST
```

```
_SET_IP
LOAD_FAST

_SET_IP
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT

_SET_IP
STORE_FAST
STORE_FAST

_SET_IP
_JUMP_TO_TOP
```

Background

CPython 3.13: Optimized Micro-Op Traces

```
_SET_IP
_ITER_CHECK_RANGE
_IS_ITER_EXHAUSTED_RANGE
_POP_JUMP_IF_TRUE
_ITER_NEXT_RANGE

_SET_IP
STORE_FAST

_SET_IP
LOAD_FAST
LOAD_FAST
```

```
_SET_IP
LOAD_FAST

_SET_IP
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT

_SET_IP
STORE_FAST
STORE_FAST

_SET_IP
_JUMP_TO_TOP
```

Background

CPython 3.13: Optimized Micro-Op Traces

```
_SET_IP
_ITER_CHECK_RANGE
_IS_ITER_EXHAUSTED_RANGE
_POP_JUMP_IF_TRUE
_ITER_NEXT_RANGE


STORE_FAST


LOAD_FAST
LOAD_FAST
```

```
LOAD_FAST

_SET_IP
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT


STORE_FAST
STORE_FAST

_SET_IP
_JUMP_TO_TOP
```

Background

CPython 3.13: Optimized Micro-Op Traces

```
_SET_IP
_ITER_CHECK_RANGE
_IS_ITER_EXHAUSTED_RANGE
_POP_JUMP_IF_TRUE
_ITER_NEXT_RANGE


STORE_FAST


LOAD_FAST
LOAD_FAST
```

```
LOAD_FAST

_SET_IP
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT


STORE_FAST
STORE_FAST

_SET_IP
_JUMP_TO_TOP
```

Background

CPython 3.13: Optimized Micro-Op Traces

```
_IS_ITER_EXHAUSTED_RANGE
_POP_JUMP_IF_TRUE
_ITER_NEXT_RANGE

STORE_FAST

LOAD_FAST
LOAD_FAST
```

```
LOAD_FAST

_BINARY_OP_ADD_INT

STORE_FAST
STORE_FAST

_SET_IP
_JUMP_TO_TOP
```

Background

CPython 3.13: Optimized Micro-Op Traces

```
_IS_ITER_EXHAUSTED_RANGE
_POP_JUMP_IF_TRUE
_ITER_NEXT_RANGE
STORE_FAST
LOAD_FAST
LOAD_FAST
LOAD_FAST
_BINARY_OP_ADD_INT
STORE_FAST
STORE_FAST
_SET_IP
_JUMP_TO_TOP
```

Just-In-Time Compilation

Just-In-Time Compilation

- Technical goals:
  - Remove interpretive overhead
  - Statically compile optimized traces
  - Reduce indirection:
    - "Burn in" constants, caches, and arguments
    - Move data off of frames and into registers
    - Bring hot code paths in-line

- Deployment goals:
  - Broad platform support
  - Few runtime dependencies
  - Low implementation complexity

Just-In-Time Compilation

- Technical goals:
  - Remove interpretive overhead
  - Statically compile optimized traces
  - Reduce indirection:
    - "Burn in" constants, caches, and arguments
    - Move data off of frames and into registers
    - Bring hot code paths in-line

- Deployment goals:
  - Broad platform support
  - Few runtime dependencies
  - Low implementation complexity

Just-In-Time Compilation

Copy-And-Patch Compilation

Copy-And-Patch Compilation

- Haoran Xu and Fredrik Kjolstad. 2021. Copy-and-Patch Compilation: A Fast Compilation Algorithm for High- Level Languages and Bytecode. Proc. ACM Program. Lang. 5, OOPSLA, Article 136 (October 2021), 30 pages. https://doi.org/10.1145/3485513

- Haoran Xu. 2023. Building a baseline JIT for Lua automatically. (12 March 2023). Retrieved from https://sillycross.github.io/2023/05/12/2023-05-12/.

- A way of automatically turning a C interpreter into a fast template JIT compiler

Copy-And-Patch Compilation

- Compared to WebAssembly baseline compiler (`Liftoff`):
  - 5x faster code generation
  - 50% faster code

- Compared to traditional JIT toolchain (`LLVM -O0`):
  - 100x faster code generation
  - 15% faster code

- Compared to an optimizing JIT with hand-written assembly (`LuaJIT`):
  - Faster on 13/44 benchmarks
  - Only 35% slower overall

Copy-And-Patch Compilation

- At runtime, walk over a sequence of bytecode instructions.
- For each:
  - Copy some static, pre-compiled machine code into executable memory
  - Patch up instructions that need to have runtime data encoded into them

Copy-And-Patch Compilation

- At runtime, walk over a sequence of bytecode instructions.
- For each:
  - Copy some static, pre-compiled machine code into executable memory
  - Patch up instructions that need to have runtime data encoded into them

Copy-And-Patch Compilation

- Copy some static, pre-compiled machine code into executable memory
- Patch up instructions that need to have runtime data encoded into them

Copy-And-Patch Compilation

- When linking or loading a relocatable object file (ELF, COFF, Mach-O, etc.):
  - Copy some static, pre-compiled machine code into executable memory
  - Patch up instructions that need to have runtime data encoded into them

Copy-And-Patch Compilation

```
case LOAD_FAST:
    PyObject *value = frame->localsplus[oparg];
    Py_INCREF(value);
    *stack_pointer++ = value;
    break;
```

Copy-And-Patch Compilation

```
PyObject *value = frame->localsplus[oparg];
Py_INCREF(value);
*stack_pointer++ = value;
```

Copy-And-Patch Compilation

```
int MAGICALLY_INSERT_THE_OPARG;
int MAGICALLY_CONTINUE_EXECUTION(_PyInterpreterFrame *frame,
                                 PyObject **stack_pointer);
int
load_fast(_PyInterpreterFrame *frame, PyObject **stack_pointer)
{
    int oparg = MAGICALLY_INSERT_THE_OPARG;
    PyObject *value = frame->localsplus[oparg];
    Py_INCREF(value);
    *stack_pointer++ = value;
    return MAGICALLY_CONTINUE_EXECUTION(frame, stack_pointer);
}
```

Copy-And-Patch Compilation

```
int MAGICALLY_INSERT_THE_OPARG;
int MAGICALLY_CONTINUE_EXECUTION(_PyInterpreterFrame *frame,
                                 PyObject **stack_pointer);
int
load_fast(_PyInterpreterFrame *frame, PyObject **stack_pointer)
{
    int oparg = MAGICALLY_INSERT_THE_OPARG;
    PyObject *value = frame->localsplus[oparg];
    Py_INCREF(value);
    *stack_pointer++ = value;
    return MAGICALLY_CONTINUE_EXECUTION(frame, stack_pointer);
}
```

Copy-And-Patch Compilation

```
extern int MAGICALLY_INSERT_THE_OPARG;
extern int MAGICALLY_CONTINUE_EXECUTION(_PyInterpreterFrame *frame,
                                        PyObject **stack_pointer);
int
load_fast(_PyInterpreterFrame *frame, PyObject **stack_pointer)
{
    int oparg = &MAGICALLY_INSERT_THE_OPARG;
    PyObject *value = frame->localsplus[oparg];
    Py_INCREF(value);
    *stack_pointer++ = value;
    __attribute__((musttail))
    return MAGICALLY_CONTINUE_EXECUTION(frame, stack_pointer);
}
```

Copy-And-Patch Compilation

```
00: 48 b8 00 00 00 00 00 00 00 00 movabsq $0x0, %rax
0a: 48 98                         cltq
0c: 49 8b 44 c5 48                movq    0x48(%r13,%rax,8), %rax
11: 8b 08                         movl    (%rax), %ecx
13: ff c1                         incl    %ecx
15: 74 02                         je      0x19 <load_fast+0x19>
17: 89 08                         movl    %ecx, (%rax)
19: 48 89 45 00                   movq    %rax, (%rbp)
1d: 48 83 c5 08                   addq    $0x8, %rbp
21: e9 00 00 00 00                jmp     0x26 <load_fast+0x26>

02: R_X86_64_64    MAGICALLY_INSERT_THE_OPARG
22: R_X86_64_PLT32 MAGICALLY_CONTINUE_EXECUTION - 0x4
```

Copy-And-Patch Compilation

```
00: 48 b8 00 00 00 00 00 00 00 00 movabsq $0x0, %rax
0a: 48 98                         cltq
0c: 49 8b 44 c5 48                movq    0x48(%r13,%rax,8), %rax
11: 8b 08                         movl    (%rax), %ecx
13: ff c1                         incl    %ecx
15: 74 02                         je      0x19 <load_fast+0x19>
17: 89 08                         movl    %ecx, (%rax)
19: 48 89 45 00                   movq    %rax, (%rbp)
1d: 48 83 c5 08                   addq    $0x8, %rbp
21: e9 00 00 00 00                jmp     0x26 <load_fast+0x26>

02: R_X86_64_64     MAGICALLY_INSERT_THE_OPARG
22: R_X86_64_PLT32 MAGICALLY_CONTINUE_EXECUTION - 0x4
```

Copy-And-Patch Compilation

```
00: 48 b8 00 00 00 00 00 00 00 00 movabsq $0x0, %rax
0a: 48 98                         cltq
0c: 49 8b 44 c5 48                movq    0x48(%r13,%rax,8), %rax
11: 8b 08                         movl    (%rax), %ecx
13: ff c1                         incl    %ecx
15: 74 02                         je      0x19 <load_fast+0x19>
17: 89 08                         movl    %ecx, (%rax)
19: 48 89 45 00                   movq    %rax, (%rbp)
1d: 48 83 c5 08                   addq    $0x8, %rbp
21: e9 00 00 00 00                jmp     0x26 <load_fast+0x26>

02: R_X86_64_64     MAGICALLY_INSERT_THE_OPARG
22: R_X86_64_PLT32 MAGICALLY_CONTINUE_EXECUTION - 0x4
```

Copy-And-Patch Compilation

```
static const unsigned char LOAD_FAST_bytes[38] = {
      48,   b8,   00,   00,   00,   00,   00,   00,
      00,   00,   48,   98,   49,   8b,   44,   c5,
      48,   8b,   08,   ff,   c1,   74,   02,   89,
      08,   48,   89,   45,   00,   48,   83,   c5,
      08,   e9,   00,   00,   00,   00,
};
static const Hole LOAD_FAST_holes[2] = {
    {  02, R_X86_64_64,    MAGICALLY_INSERT_THE_OPARG,   +0x0},
    {  22, R_X86_64_PLT32, MAGICALLY_CONTINUE_EXECUTION, -0x4},
};
```

Copy-And-Patch Compilation

```
static const unsigned char LOAD_FAST_bytes[38] = {
    0x48, 0xb8, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x48, 0x98, 0x49, 0x8b, 0x44, 0xc5,
    0x48, 0x8b, 0x08, 0xff, 0xc1, 0x74, 0x02, 0x89,
    0x08, 0x48, 0x89, 0x45, 0x00, 0x48, 0x83, 0xc5,
    0x08, 0xe9, 0x00, 0x00, 0x00, 0x00,
};
static const Hole LOAD_FAST_holes[2] = {
    {0x02, R_X86_64_64,    MAGICALLY_INSERT_THE_OPARG,   +0x0},
    {0x22, R_X86_64_PLT32, MAGICALLY_CONTINUE_EXECUTION, -0x4},
};
```

brandtbucher/cpython

brandtbucher/cpython
justin

github.com/brandtbucher/cpython/tree/justin

Copy-And-Patch Compilation

- Build time:
  - ~700 lines of complex Python
  - ~100 lines of complex C
  - LLVM dependency

- Run time:
  - ~300 lines of simple C (hand-written)
  - ~3000 lines of simple C (generated)
  - No dependencies

Copy-And-Patch Compilation

- Build time:
  - ~700 lines of complex Python
  - ~100 lines of complex C
  - LLVM dependency

- Run time:
  - ~300 lines of simple C (hand-written)
  - ~3000 lines of simple C (generated)
  - No dependencies

Platform Support

Platform Support

Microsoft Windows

- `i686-pc-windows-msvc/msvc`
- `x86_64-pc-windows-msvc/msvc`

Platform Support

All Tier One Platforms

- i686-pc-windows-msvc/msvc
- x86_64-apple-darwin/clang
- x86_64-pc-windows-msvc/msvc
- x86_64-unknown-linux-gnu/gcc

Platform Support

All Tier One & Tier Two Platforms

- aarch64-apple-darwin/clang
- aarch64-unknown-linux-gnu/clang
- aarch64-unknown-linux-gnu/gcc
- i686-pc-windows-msvc/msvc
- x86_64-apple-darwin/clang
- x86_64-pc-windows-msvc/msvc
- x86_64-unknown-linux-gnu/clang
- x86_64-unknown-linux-gnu/gcc

Platform Support

All Tier One & Tier Two Platforms (Except PowerPC)

- `aarch64-apple-darwin/clang`
- `aarch64-unknown-linux-gnu/clang`
- `aarch64-unknown-linux-gnu/gcc`
- `i686-pc-windows-msvc/msvc`
- `powerpc64le-unknown-linux-gnu/gcc`
- `x86_64-apple-darwin/clang`
- `x86_64-pc-windows-msvc/msvc`
- `x86_64-unknown-linux-gnu/clang`
- `x86_64-unknown-linux-gnu/gcc`

Platform Support

All Tier One & Tier Two Platforms

- `aarch64-apple-darwin/clang`
- `aarch64-unknown-linux-gnu/clang`
- `aarch64-unknown-linux-gnu/gcc`
- `i686-pc-windows-msvc/msvc`
- `x86_64-apple-darwin/clang`
- `x86_64-pc-windows-msvc/msvc`
- `x86_64-unknown-linux-gnu/clang`
- `x86_64-unknown-linux-gnu/gcc`

Platform Support

All Tier One & Tier Two Platforms (Including Cross-Compiles!)

- `aarch64-apple-darwin/clang`
- `aarch64-unknown-linux-gnu/clang`
- `aarch64-unknown-linux-gnu/gcc`
- `i686-pc-windows-msvc/msvc`
- `x86_64-apple-darwin/clang`
- `x86_64-pc-windows-msvc/msvc`
- `x86_64-unknown-linux-gnu/clang`
- `x86_64-unknown-linux-gnu/gcc`