CPython's JIT compiler

Brandt Bucher (September 26th, 2024)

Enabling CPython's JIT compiler

Brandt Bucher (September 26th, 2024)

Enabling CPython's JIT compiler

...but leaving it off by default

Brandt Bucher (September 26th, 2024)

Enabling CPython's JIT compiler

...but leaving it off by default

...but also maybe turning it on by default at some point

Brandt Bucher (September 26th, 2024)

Background

Background

```
def add(a, b):
    return a + b

add(1, 2)
```

Background

Bytecode

```
def add(a, b):
    return a + b

add(1, 2)
```

Background

Bytecode

```
def add(a, b):                    LOAD_FAST_LOAD_FAST     (a, b)
    return a + b                  BINARY_OP               (+)

add(1, 2)
```

Background

Bytecode

```
def add(a, b):                    LOAD_FAST_LOAD_FAST     (a, b)
    return a + b                  BINARY_OP               (+)

add(1, 2)
```

Background

Specialized Bytecode

```
def add(a, b):                    LOAD_FAST_LOAD_FAST     (a, b)
    return a + b                  BINARY_OP               (+)

add(1, 2)
```

Background

Specialized Bytecode

```
def add(a, b):                    LOAD_FAST_LOAD_FAST     (a, b)
    return a + b                  BINARY_OP               (+)

add(1, 2)
```

Background

Specialized Bytecode

```
def add(a, b):                    LOAD_FAST_LOAD_FAST     (a, b)
    return a + b                  BINARY_OP_ADD_INT       (+)

add(1, 2)
```

Background

Specialized Bytecode

```
def add(a, b):                    LOAD_FAST_LOAD_FAST     (a, b)
    return a + b                  BINARY_OP_ADD_INT       (+)

add(1, 2)
```

Background

Micro-Ops

```
def add(a, b):                    LOAD_FAST_LOAD_FAST     (a, b)
    return a + b                  BINARY_OP_ADD_INT       (+)

add(1, 2)
```

Background

Micro-Ops

```
def add(a, b):                    LOAD_FAST_LOAD_FAST     (a, b)
    return a + b                  BINARY_OP_ADD_INT       (+)

add(1, 2)
```

Background

Micro-Ops

```
def add(a, b):
    return a + b

add(1, 2)
```

```
_CHECK_VALIDITY
_SET_IP
_LOAD_FAST                (a)
_LOAD_FAST                (b)
_CHECK_VALIDITY
_SET_IP
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT        (+)
```

Background

Micro-Ops

```
def add(a, b):
    return a + b

add(1, 2)
```

```
_CHECK_VALIDITY
_SET_IP
_LOAD_FAST                (a)
_LOAD_FAST                (b)
_CHECK_VALIDITY
_SET_IP
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT        (+)
```

Background

Optimized Micro-Ops

```
def add(a, b):
    return a + b

add(1, 2)
```

```
_CHECK_VALIDITY
_SET_IP
_LOAD_FAST                (a)
_LOAD_FAST                (b)
_CHECK_VALIDITY
_SET_IP
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT        (+)
```

Background

Optimized Micro-Ops

```
def add(a, b):
    return a + b

add(1, 2)
```

```
_CHECK_VALIDITY
_SET_IP
_LOAD_FAST                (a)
_LOAD_FAST                (b)
_CHECK_VALIDITY
_SET_IP
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT        (+)
```

Background

Optimized Micro-Ops

```
def add(a, b):
    return a + b

add(1, 2)
```

```
_LOAD_CONST_INLINE_BORROW (3)
```

Background

Optimized Micro-Ops

```
def add(a, b):
    return a + b

add(1, 2)
```

```
_LOAD_CONST_INLINE_BORROW (3)
```

Background

Machine Code

```
def add(a, b):
    return a + b

add(1, 2)
```

```
_LOAD_CONST_INLINE_BORROW (3)
```

Background

Machine Code

```
def add(a, b):
    return a + b

add(1, 2)
```

```
_LOAD_CONST_INLINE_BORROW (3)
```

Background

Machine Code

```
def add(a, b):
    return a + b
```

```
_LOAD_CONST_INLINE_BORROW (3)
```

```
48 8b 05 de ad be ef
48 89 45 00
48 83 c5 08
```

The Past Year

The Past Year

- Presented the JIT at the 2023 Core Dev Sprint in Brno
- Wrote PEP 744
- Opened PR #113465 (🎄)
- Merged PR #113465

The Past Year

- Presented the JIT at the 2023 Core Dev Sprint in Brno
- Opened PR #113465 (🎄)
- Merged PR #113465
- Wrote PEP 744

The Past Year

- Presented the JIT at the 2023 Core Dev Sprint in Brno
- Opened PR #113465 (🎄)
- Merged PR #113465
- Wrote PEP 744
- ...?

The Past Year

- ~0% change in performance with the JIT enabled

The Past Year

- ~~~0%~~ ~0% change in performance with the JIT enabled

The Past Year

- ~~~0%~~ ~0% change in performance with the JIT enabled
- ~6% of the benchmark code is run in the JIT

The Past Year

- ~~~0%~~ ~0% change in performance with the JIT enabled
- ~~~6%~~ ~65% of the benchmark code is run in the JIT

The Past Year

- ~~~0%~~ ~0% change in performance with the JIT enabled
- ~~~6%~~ ~~~65%~~ ~91% of the benchmark code is run in the JIT

The Past Year

- ~~~0%~~ ~0% change in performance with the JIT enabled
- ~~~6%~~ ~~~65%~~ ~91% of the benchmark code is run in the JIT
- ~10% increase in total memory consumption

The Past Year

- ~~~0%~~ ~0% change in performance with the JIT enabled
- ~~~6%~~ ~~~65%~~ ~91% of the benchmark code is run in the JIT
- ~~~10%~~ ~8% increase in total memory consumption

The Past Year

- ~~~0%~~ ~0% change in performance with the JIT enabled
- ~~~6%~~ ~~~65%~~ ~91% of the benchmark code is run in the JIT
- ~~~10%~~ ~~~8%~~ ~5% increase in total memory consumption

The Past Year

- 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

The Past Year

- Build time:
  - ~~~700~~ ~1100 lines of complex Python
  - ~100 lines of complex C
  - LLVM dependency

- Run time:
  - ~~~300~~ ~400 lines of simple C (hand-written)
  - ~~~3000~~ ~4000 lines of simple C (generated)
  - No dependencies

The Past Year

- Build time:
  - ~~~700~~ ~1100 lines of complex Python
  - ~100 lines of complex C
  - LLVM dependency

Build time LLVM dependency

jit_stencils.h

`jit_stencils.h`

- A very large C header generated at build-time by LLVM
- Basically a "human-readable compiled extension module"
- Depends on the current platform and build configuration ("internal ABI")
- But... we can *probably* either host them somewhere or track them in Git

jit_stencils.h

~1.2MB

```
void
emit__LOAD_CONST_INLINE_BORROW(
    unsigned char *code, unsigned char *data, _PyExecutorObject *executor,
    const _PyUOpInstruction *instruction, uintptr_t instruction_starts[])
{
    // 0: 48 8b 05 00 00 00 00          movq    (%rip), %rax            # 0x7 <_JIT_ENTRY+0x7>
    // 0000000000000003:  R_X86_64_REX_GOTPCRELX       _JIT_OPERAND-0x4
    // 7: 48 89 45 00                   movq    %rax, (%rbp)
    // b: 48 83 c5 08                   addq    $0x8, %rbp
    // f: ff 25 00 00 00 00             jmpq    *(%rip)                 # 0x15 <_JIT_ENTRY+0x15>
    // 0000000000000011:  R_X86_64_GOTPCRELX   _JIT_CONTINUE-0x4
    const unsigned char code_body[15] = {
        0x48, 0x8b, 0x05, 0x00, 0x00, 0x00, 0x00, 0x48,
        0x89, 0x45, 0x00, 0x48, 0x83, 0xc5, 0x08,
    };
    // 0: OPERAND
    const unsigned char data_body[8] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    };
    memcpy(data, data_body, sizeof(data_body));
    patch_64(data + 0x0, instruction->operand);
    memcpy(code, code_body, sizeof(code_body));
    patch_x86_64_32rx(code + 0x3, (uintptr_t)data + -0x4);
}
```

jit_stencils.h

~1.2MB

```
void
emit__LOAD_CONST_INLINE_BORROW(
    unsigned char *code, unsigned char *data, _PyExecutorObject *executor,
    const _PyUOpInstruction *instruction, uintptr_t instruction_starts[])
{
    // 0: 48 8b 05 00 00 00 00          movq    (%rip), %rax            # 0x7 <_JIT_ENTRY+0x7>
    // 0000000000000003:  R_X86_64_REX_GOTPCRELX       _JIT_OPERAND-0x4
    // 7: 48 89 45 00                   movq    %rax, (%rbp)
    // b: 48 83 c5 08                   addq    $0x8, %rbp
    // f: ff 25 00 00 00 00             jmpq    *(%rip)                 # 0x15 <_JIT_ENTRY+0x15>
    // 0000000000000011:  R_X86_64_GOTPCRELX   _JIT_CONTINUE-0x4
    const unsigned char code_body[15] = {
        0x48, 0x8b, 0x05, 0x00, 0x00, 0x00, 0x00, 0x48,
        0x89, 0x45, 0x00, 0x48, 0x83, 0xc5, 0x08,
    };
    // 0: OPERAND
    const unsigned char data_body[8] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    };
    memcpy(data, data_body, sizeof(data_body));
    patch_64(data + 0x0, instruction->operand);
    memcpy(code, code_body, sizeof(code_body));
    patch_x86_64_32rx(code + 0x3, (uintptr_t)data + -0x4);
}
```

jit_stencils.h

~1.2MB

```
void
emit__LOAD_CONST_INLINE_BORROW(
    unsigned char *code, unsigned char *data, _PyExecutorObject *executor,
    const _PyUOpInstruction *instruction, uintptr_t instruction_starts[])
{
    // 0: 48 8b 05 00 00 00 00          movq    (%rip), %rax            # 0x7 <_JIT_ENTRY+0x7>
    // 0000000000000003:  R_X86_64_REX_GOTPCRELX       _JIT_OPERAND-0x4
    // 7: 48 89 45 00                   movq    %rax, (%rbp)
    // b: 48 83 c5 08                   addq    $0x8, %rbp
    // f: ff 25 00 00 00 00             jmpq    *(%rip)                 # 0x15 <_JIT_ENTRY+0x15>
    // 0000000000000011:  R_X86_64_GOTPCRELX   _JIT_CONTINUE-0x4
    const unsigned char code_body[15] = {
        0x48, 0x8b, 0x05, 0x00, 0x00, 0x00, 0x00, 0x48,
        0x89, 0x45, 0x00, 0x48, 0x83, 0xc5, 0x08,
    };
    // 0: OPERAND
    const unsigned char data_body[8] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    };
    memcpy(data, data_body, sizeof(data_body));
    patch_64(data + 0x0, instruction->operand);
    memcpy(code, code_body, sizeof(code_body));
    patch_x86_64_32rx(code + 0x3, (uintptr_t)data + -0x4);
}
```

jit_stencils.h

~1.2MB

```
void
emit__LOAD_CONST_INLINE_BORROW(
    unsigned char *code, unsigned char *data, _PyExecutorObject *executor,
    const _PyUOpInstruction *instruction, uintptr_t instruction_starts[])
{
    // 0: 48 8b 05 00 00 00 00          movq    (%rip), %rax            # 0x7 <_JIT_ENTRY+0x7>
    // 0000000000000003:  R_X86_64_REX_GOTPCRELX       _JIT_OPERAND-0x4
    // 7: 48 89 45 00                   movq    %rax, (%rbp)
    // b: 48 83 c5 08                   addq    $0x8, %rbp
    // f: ff 25 00 00 00 00             jmpq    *(%rip)                 # 0x15 <_JIT_ENTRY+0x15>
    // 0000000000000011:  R_X86_64_GOTPCRELX   _JIT_CONTINUE-0x4
    const unsigned char code_body[15] = {
        0x48, 0x8b, 0x05, 0x00, 0x00, 0x00, 0x00, 0x48,
        0x89, 0x45, 0x00, 0x48, 0x83, 0xc5, 0x08,
    };
    // 0: OPERAND
    const unsigned char data_body[8] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    };
    memcpy(data, data_body, sizeof(data_body));
    patch_64(data + 0x0, instruction->operand);
    memcpy(code, code_body, sizeof(code_body));
    patch_x86_64_32rx(code + 0x3, (uintptr_t)data + -0x4);
}
```

jit_stencils.h

~1.2MB

```
void emit__LOAD_CONST_INLINE_BORROW(unsigned char*code,unsigned char*data,_PyExecutorObject*executor,const _PyUOpInstruction*inst
ruction,uintptr_t instruction_starts[]){
// 0: 48 8b 05 00 00 00 00 movq (%rip), %rax # 0x7 <_JIT_ENTRY+0x7>
// 0000000000000003: R_X86_64_REX_GOTPCRELX _JIT_OPERAND-0x4
// 7: 48 89 45 00 movq %rax, (%rbp)
// b: 48 83 c5 08 addq $0x8, %rbp
// f: ff 25 00 00 00 00 jmpq *(%rip)# 0x15 <_JIT_ENTRY+0x15>
// 0000000000000011: R_X86_64_GOTPCRELX _JIT_CONTINUE-0x4
const unsigned char code_body[15]={72,139,5,0,0,0,0,72,137,69,0,72,131,197,8};
// 0: OPERAND
const unsigned char data_body[8]={0,0,0,0,0,0,0,0};
memcpy(data,data_body,sizeof(data_body));
patch_64(data+0x0,instruction->operand);
memcpy(code,code_body,sizeof(code_body));
patch_x86_64_32rx(code+0x3,(uintptr_t)data+-0x4);
}
```

jit_stencils.h

~670KB (unformatted)

```
void emit__LOAD_CONST_INLINE_BORROW(unsigned char*code,unsigned char*data,_PyExecutorObject*executor,const _PyUOpInstruction*inst
ruction,uintptr_t instruction_starts[]){
// 0: 48 8b 05 00 00 00 00 movq (%rip), %rax # 0x7 <_JIT_ENTRY+0x7>
// 0000000000000003: R_X86_64_REX_GOTPCRELX _JIT_OPERAND-0x4
// 7: 48 89 45 00 movq %rax, (%rbp)
// b: 48 83 c5 08 addq $0x8, %rbp
// f: ff 25 00 00 00 00 jmpq *(%rip)# 0x15 <_JIT_ENTRY+0x15>
// 0000000000000011: R_X86_64_GOTPCRELX _JIT_CONTINUE-0x4
const unsigned char code_body[15]={72,139,5,0,0,0,0,72,137,69,0,72,131,197,8};
// 0: OPERAND
const unsigned char data_body[8]={0,0,0,0,0,0,0,0};
memcpy(data,data_body,sizeof(data_body));
patch_64(data+0x0,instruction->operand);
memcpy(code,code_body,sizeof(code_body));
patch_x86_64_32rx(code+0x3,(uintptr_t)data+-0x4);
}
```

jit_stencils.h

~670KB (unformatted)

```
void emit__LOAD_CONST_INLINE_BORROW(unsigned char*code,unsigned char*data,_PyExecutorObject*executor,const _PyUOpInstruction*inst
ruction,uintptr_t instruction_starts[]){
// 0: 48 8b 05 00 00 00 00 movq (%rip), %rax # 0x7 <_JIT_ENTRY+0x7>
// 0000000000000003: R_X86_64_REX_GOTPCRELX _JIT_OPERAND-0x4
// 7: 48 89 45 00 movq %rax, (%rbp)
// b: 48 83 c5 08 addq $0x8, %rbp
// f: ff 25 00 00 00 00 jmpq *(%rip)# 0x15 <_JIT_ENTRY+0x15>
// 0000000000000011: R_X86_64_GOTPCRELX _JIT_CONTINUE-0x4
const unsigned char code_body[15]={72,139,5,0,0,0,0,72,137,69,0,72,131,197,8};
// 0: OPERAND
const unsigned char data_body[8]={0,0,0,0,0,0,0,0};
memcpy(data,data_body,sizeof(data_body));
patch_64(data+0x0,instruction->operand);
memcpy(code,code_body,sizeof(code_body));
patch_x86_64_32rx(code+0x3,(uintptr_t)data+-0x4);
}
```

jit_stencils.h

~670KB (unformatted)

```
void emit__LOAD_CONST_INLINE_BORROW(unsigned char*code,unsigned char*data,_PyExecutorObject*executor,const _PyUOpInstruction*inst
ruction,uintptr_t instruction_starts[]){
// 0: 48 8b 05 00 00 00 00 movq (%rip), %rax # 0x7 <_JIT_ENTRY+0x7>
// 0000000000000003: R_X86_64_REX_GOTPCRELX _JIT_OPERAND-0x4
// 7: 48 89 45 00 movq %rax, (%rbp)
// b: 48 83 c5 08 addq $0x8, %rbp
// f: ff 25 00 00 00 00 jmpq *(%rip)# 0x15 <_JIT_ENTRY+0x15>
// 0000000000000011: R_X86_64_GOTPCRELX _JIT_CONTINUE-0x4
const unsigned char code_body[15]={72,139,5,0,0,0,0,72,137,69,0,72,131,197,8};
// 0: OPERAND
const unsigned char data_body[8]={0,0,0,0,0,0,0,0};
memcpy(data,data_body,sizeof(data_body));
patch_64(data+0x0,instruction->operand);
memcpy(code,code_body,sizeof(code_body));
patch_x86_64_32rx(code+0x3,(uintptr_t)data+-0x4);
}
```

jit_stencils.h

~670KB (unformatted)

```
void emit__LOAD_CONST_INLINE_BORROW(unsigned char*code,unsigned char*data,_PyExecutorObject*executor,const _PyUOpInstruction*inst
ruction,uintptr_t instruction_starts[]){
// 0: 48 8b 05 00 00 00 00 movq (%rip), %rax # 0x7 <_JIT_ENTRY+0x7>
// 0000000000000003: R_X86_64_REX_GOTPCRELX _JIT_OPERAND-0x4
// 7: 48 89 45 00 movq %rax, (%rbp)
// b: 48 83 c5 08 addq $0x8, %rbp
// f: ff 25 00 00 00 00 jmpq *(%rip)# 0x15 <_JIT_ENTRY+0x15>
// 0000000000000011: R_X86_64_GOTPCRELX _JIT_CONTINUE-0x4
const unsigned char code_body[15]={72,139,5,0,0,0,0,72,137,69,0,72,131,197,8};
// 0: OPERAND
const unsigned char data_body[8]={0,0,0,0,0,0,0,0};
memcpy(data,data_body,sizeof(data_body));
patch_64(data+0x0,instruction->operand);
memcpy(code,code_body,sizeof(code_body));
patch_x86_64_32rx(code+0x3,(uintptr_t)data+-0x4);
}
```

jit_stencils.h

~670KB (unformatted)

```
void emit__LOAD_CONST_INLINE_BORROW(unsigned char*code,unsigned char*data,_PyExecutorObject*executor,const _PyUOpInstruction*inst
ruction,uintptr_t instruction_starts[]){const unsigned char code_body[15]={72,139,5,0,0,0,0,72,137,69,0,72,131,197,8};const unsig
ned char data_body[8]={0,0,0,0,0,0,0,0};memcpy(data,data_body,sizeof(data_body));patch_64(data+0x0,instruction->operand);memcpy(c
ode,code_body,sizeof(code_body));patch_x86_64_32rx(code+0x3,(uintptr_t)data+-0x4);}
```

jit_stencils.h

~300KB (uncommented)

```
void emit__LOAD_CONST_INLINE_BORROW(unsigned char*code,unsigned char*data,_PyExecutorObject*executor,const _PyUOpInstruction*inst
ruction,uintptr_t instruction_starts[]){const unsigned char code_body[15]={72,139,5,0,0,0,0,72,137,69,0,72,131,197,8};const unsig
ned char data_body[8]={0,0,0,0,0,0,0,0};memcpy(data,data_body,sizeof(data_body));patch_64(data+0x0,instruction->operand);memcpy(c
ode,code_body,sizeof(code_body));patch_x86_64_32rx(code+0x3,(uintptr_t)data+-0x4);}
```

jit_stencils.h

~300KB (uncommented)

```
void emit__LOAD_CONST_INLINE_BORROW(unsigned char*code,unsigned char*data,_PyExecutorObject*executor,const _PyUOpInstruction*inst
ruction,uintptr_t instruction_starts[]){const unsigned char code_body[15]={72,139,5,0,0,0,0,72,137,69,0,72,131,197,8};const unsig
ned char data_body[8]={0,0,0,0,0,0,0,0};memcpy(data,data_body,sizeof(data_body));patch_64(data+0x0,instruction->operand);memcpy(c
ode,code_body,sizeof(code_body));patch_x86_64_32rx(code+0x3,(uintptr_t)data+-0x4);}
```

jit_stencils.h

~300KB (uncommented)

```
void emit__LOAD_CONST_INLINE_BORROW(unsigned char*code,unsigned char*data,_PyExecutorObject*executor,const _PyUOpInstruction*inst
ruction,uintptr_t instruction_starts[]){const unsigned char code_body[15]={72,139,5,0,0,0,0,72,137,69,0,72,131,197,8};const unsig
ned char data_body[8]={0,0,0,0,0,0,0,0};memcpy(data,data_body,sizeof(data_body));patch_64(data+0x0,instruction->operand);memcpy(c
ode,code_body,sizeof(code_body));patch_x86_64_32rx(code+0x3,(uintptr_t)data+-0x4);}
```

`jit_stencils.h`

- On supported builds, *only those working on the JIT* would need LLVM
- Also improves visibility into the code used for any particular build
- We would need to either commit or host these files (potentially very many)
- We would need infrastructure to regenerate them for PRs (GitHub Actions?)

- Do we want to track or host `jit_stencils.h`?

Shipping the JIT

```
$ ./configure --enable-experimental-jit
```

```
$ ./configure --enable-experimental-jit=yes
```

```
$ ./configure --enable-experimental-jit=yes
$ python spam.py
```

```
$ ./configure --enable-experimental-jit=yes
$ python spam.py  # JIT!
```

```
$ ./configure --enable-experimental-jit=yes
$ python spam.py  # JIT!
$ python spam.py
```

```
$ ./configure --enable-experimental-jit=yes
$ python spam.py  # JIT!
$ PYTHON_JIT=0 python spam.py
```

```
$ ./configure --enable-experimental-jit=yes
$ python spam.py  # JIT!
$ PYTHON_JIT=0 python spam.py  # No JIT.
```

```
$ ./configure --enable-experimental-jit=yes
```

```
$ ./configure --enable-experimental-jit=yes-off
```

```
$ ./configure --enable-experimental-jit=yes-off
$ python spam.py
```

```
$ ./configure --enable-experimental-jit=yes-off
$ python spam.py  # No JIT.
```

```
$ ./configure --enable-experimental-jit=yes-off
$ python spam.py  # No JIT.
$ python spam.py
```

```
$ ./configure --enable-experimental-jit=yes-off
$ python spam.py  # No JIT.
$ PYTHON_JIT=1 python spam.py
```

```
$ ./configure --enable-experimental-jit=yes-off
$ python spam.py  # No JIT.
$ PYTHON_JIT=1 python spam.py  # JIT!
```

Shipping the JIT

- At least two Linux distributions (Fedora and OpenSUSE) already do this!
- Allows us to figure out any tricky release issues earlier rather than later
- Makes it easier for us to see how the JIT handles real-world workloads
- It's *still* experimental, so we can always stop shipping it (even in beta)

- Do we want to track or host `jit_stencils.h`?

- Do we want to track or host `jit_stencils.h`?
- Do we want to ship the JIT in releases yet?