Just in Time Compilation

Evgeny Gavrin

JIT Compilation: What is it?

“Compilation done during execution of a program (at run time) rather than prior to execution” -Wikipedia

- JVM
- CLR
- Python/Ruby etc
- JavaScript Engines

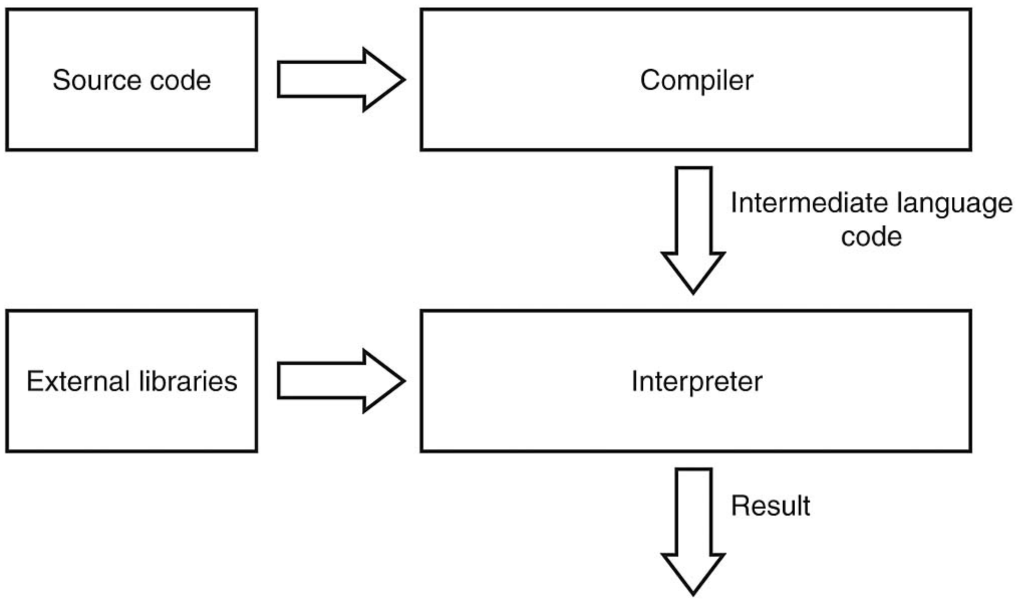
Outline

- Traditional Compilation and Execution
- Interpreters
- JIT Compilation
- Optimization Techniques in JIT
- Simple JIT

Introduction

Traditional Compilation and Execution

- A compiler compiles source code to bytecode readable by VM
- VM interprets bytecode to machine instructions at runtime



Interpreters

- **Switch statement**
- Direct threading
- Indirect threading
- Token threading

```
while (true) {
   switch (opcode) {
     case ADD:
       ...
       break;
     case SUB:
       ...
       break;
     ...
   }
}
```

```
mov    %edx,0xfffffffffffffe4(%rbp)
cmpl   $0x1,0xfffffffffffffe4(%rbp)
je     6e <interpret+0x6e>
cmpl   $0x1,0xfffffffffffffe4(%rbp)
jb     4a <interpret+0x4a>
cmpl   $0x2,0xfffffffffffffe4(%rbp)
je     93 <interpret+0x93>
jmp    22 <interpret+0x22>
...
```

Traditional Compilation and Execution

- Advantages
  - platform independence
  - reflection (modification of program at runtime)
- Drawbacks
  - need memory
  - not as fast as running pre-compiled machine instructions

Speeding up interpretation

- Interpreter optimization
- Compiler optimization
- Just in Time compiler
- Type Inference
- Hidden Type
- Method inline, PICs

Direct threading

```
typedef void *Inst;
Inst program[] = { &&ADD, &&SUB };
Inst *ip = program;
goto *ip++;

ADD:
    ...
    goto *ip++;

SUB:
    ...
    goto *ip++;
```

```
mov     0xffffffffffffffe8(%rbp),%rdx
lea     0xffffffffffffffe8(%rbp),%rax
addq    $0x8,(%rax)
mov     %rdx,0xffffffffffffffd8(%rbp)
jmpq    *0xffffffffffffffd8(%rbp)

ADD:
   ...
   mov     0xffffffffffffffe8(%rbp),%rdx
   lea     0xffffffffffffffe8(%rbp),%rax
   addq    $0x8,(%rax)
   mov     %rdx,0xffffffffffffffd8(%rbp)
   jmp     2c <interpreter+0x2c>
```

Inline Threading

```
ICONST_1_START: *sp++ = 1;
ICONST_1_END: goto **(pc++);
INEG_START: sp[-1] = -sp[-1];
INEG_END: goto **(pc++);
DISPATCH_START: goto **(pc++);
DISPATCH_END: ;

size_t iconst_size = (&&ICONST_1_END - &&ICONST_1_START);
size_t ineg_size = (&&INEG_END - &&INEG_START);
size_t dispatch_size = (&&DISPATCH_END - &&DISPATCH_START);

void *buf = malloc(iconst_size + ineg_size + dispatch_size);
void *current = buf;
memcpy(current, &&ICONST_START, iconst_size); current += iconst_size;
memcpy(current, &&INEG_START, ineg_size); current += ineg_size;
memcpy(current, &&DISPATCH_START, dispatch_size);
...

goto **buf;
```

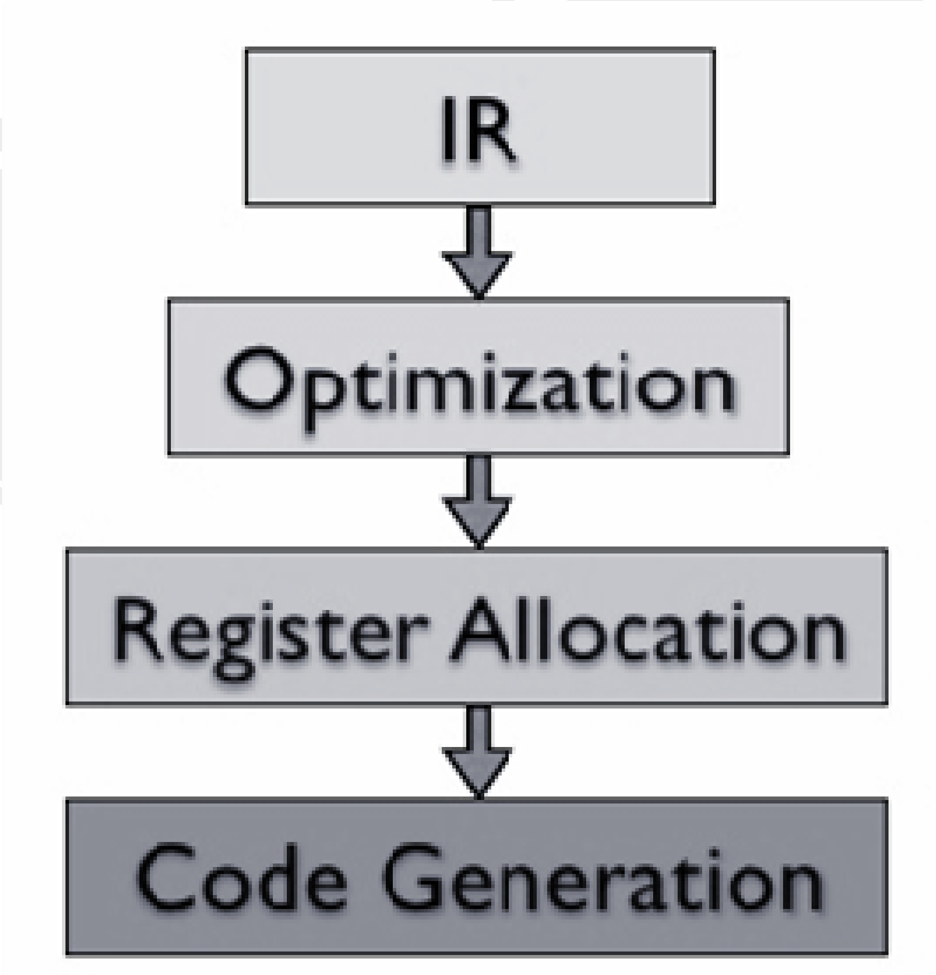
Interpreter? JIT!

Goals in JIT Compilation

- Combine speed of compiled code with flexibility of interpretation

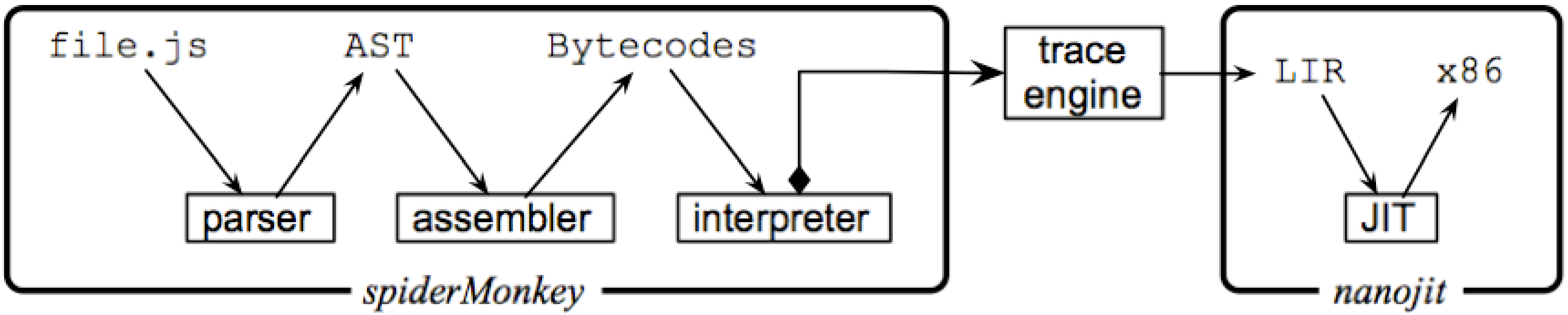
Goal: "surpass the performance of static compilation, while maintaining the advantages of bytecode interpretation" -Wikipedia

Astronaut View on JIT



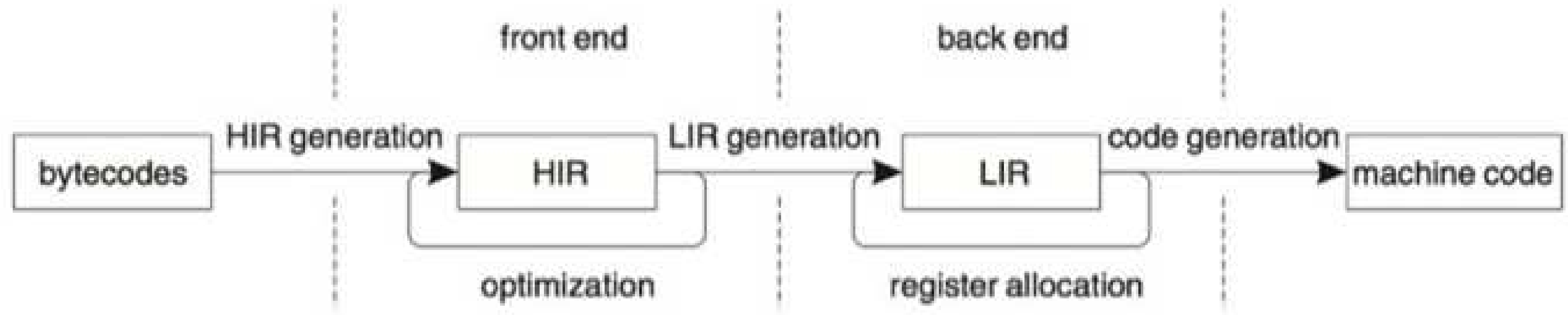
JIT Compilation

- Compiler compiles source code to bytecode readable by VM
- VM compiles bytecode at runtime into machine instructions as opposed to interpreting
- Run compiled code

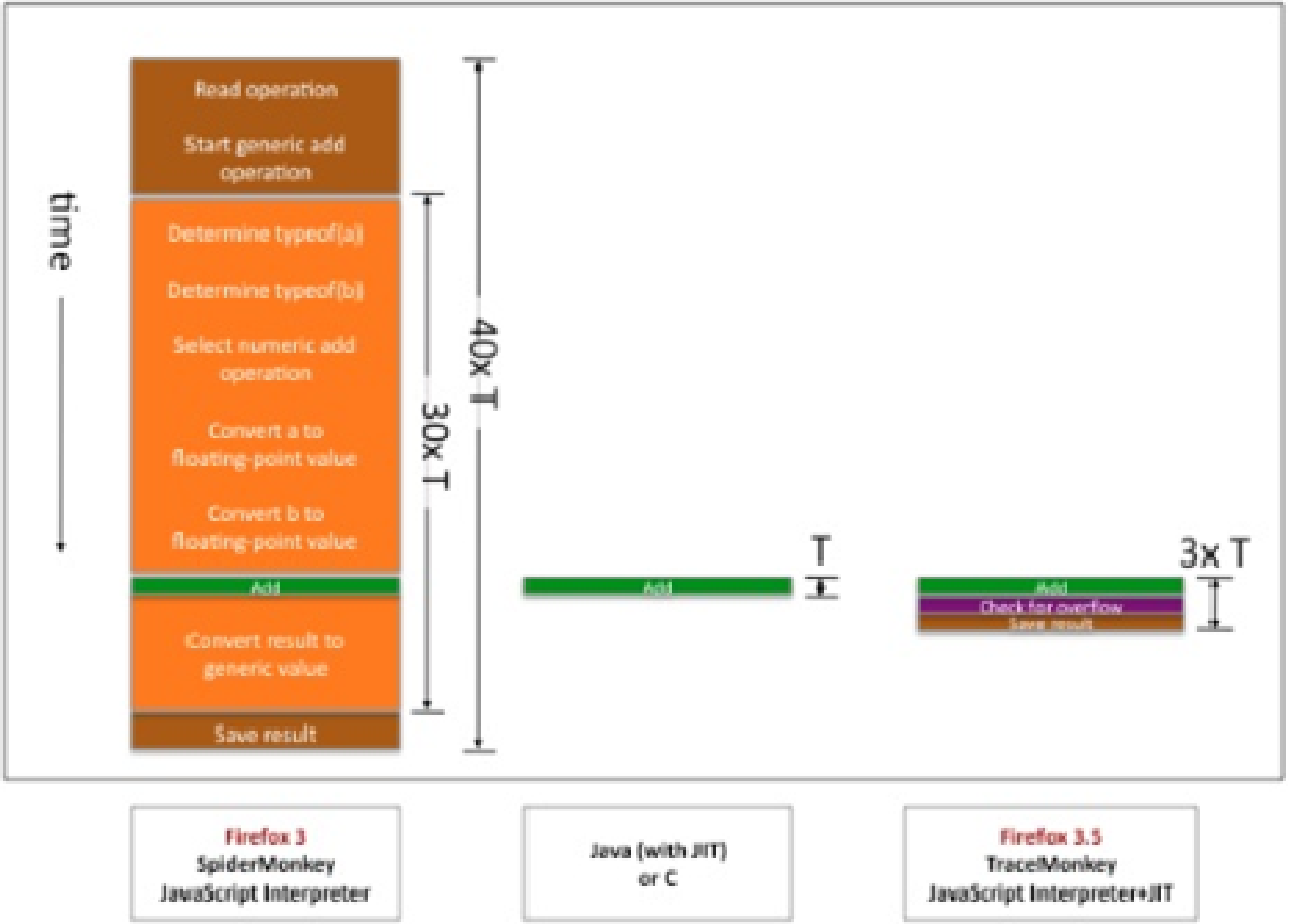


Advantages of JIT Compilation

- Can perform optimizations
  - more than 70 available
- Native code execution is faster than bytecode interpretation



Hotspot client compiler


time
Read operation
Start generic add operation
Determine typeof(a)
Determine typeof(b)
Select numeric add operation
Convert a to floating-point value
Convert b to floating-point value
Add
Convert result to generic value
Save result
30x T
40x T
Add
T
Add
Check for overflow
Save result
3x T
Firefox 3
SpiderMonkey
JavaScript Interpreter
Java (with JIT)
or C
Firefox 3.5
TraceMonkey
JavaScript Interpreter+JIT

Drawback of JIT compilation

- Startup Delay
  - compilation of bytecode -> machine code takes time
  - bytecode interpretation may run faster on earlier stages

- Limited set of optimization b/c of time
  - Regular JSE applies no more than 10 opts
  - Server-side JVM applies more than 70+

- JIT has to be implemented for each targeted architecture

Optimization

Optimization techniques

- Detect frequently used bytecode instructions & optimize
  - # of times a method executed
  - detection of loops
- Combine interpretation with JIT Compilation
- Useful in longer running programs
  - have more runtime information
  - have time to perform more optimization

Optimizations in JIT

speculative (profile-based) techniques
- optimistic nullness assertions
- optimistic type assertions
- optimistic type strengthening
- optimistic array length strengthening
- untaken branch pruning
- optimistic N-morphic inlining
- branch frequency prediction
- call frequency prediction

proof-based techniques
- exact type inference
- memory value inference
- memory value tracking
- constant folding
- reassociation
- operator strength reduction
- null check elimination
- type test strength reduction
- type test elimination
- algebraic simplification
- common subexpression elimination
- integer range typing

flow-sensitive rewrites
- conditional constant propagation
- dominating test detection
- flow-carried type narrowing
- dead code elimination

language-specific techniques
- class hierarchy analysis
- devirtualization
- symbolic constant propagation
- autobox elimination
- escape analysis
- lock elision
- lock fusion
- de-reflection

memory and placement transformation
- expression hoisting
- expression sinking
- redundant store elimination
- adjacent store fusion
- card-mark elimination

loop transformations
- loop unrolling
- loop peeling
- safepoint elimination
- iteration range splitting
- range check elimination
- loop vectorization

global code shaping
- inlining (graph integration)
- global code motion
- heat-based code layout
- switch balancing
- throw inlining

control flow graph transformation
- local code scheduling
- local code bundling
- delay slot filling
- graph-coloring register allocation
- linear scan register allocation
- live range splitting
- copy coalescing
- constant splitting
- copy removal
- address mode matching
- instruction peepholing
- DFA-based code generator

Optimizations in V8

- Dynamic type feedback
- Inlining
- Representation inference
- Static type inference
- UInt32 analysis
- Canonicalization
- Global value numbering (GVN)
- Loop invariant code motion (LICM)
- Range analysis
- Redundant bounds check elimination
- Array index dehoisting
- Dead code elimination

Tiered compilation

- level 0 - Interpreter
- level 1 - Baseline JIT (part. opt/no profile)
- level 2 - Baseline JIT (part. opt/profile)
- level 3 - Optimizing JIT (full opt/profile)

Register allocation

- Linear Scan
- Graph Coloring

How to create own JIT compiler?

How JIT work?

- Method JIT, Trace JIT, RegExp JIT
- Code generation
- Register allocation

- mmap/new/malloc (mprotect)
- generate native code
- c cast/resinterpret_cast
- call the function

JavaScriptCore

```
// Execute the code!
inline JSValue execute(RegisterFile* registerFile,
                       CallFrame* callFrame,
                       JSGlobalData* globalData)
{
    JSValue result = JSValue::decode(
      ctiTrampoline(
        m_ref.m_code.executableAddress(),
        registerFile,
        callFrame,
        0,
        Profiler::enabledProfilerReference(),
        globalData));
    return globalData->exception ? jsNull() : result;
}
```

```
asm (
".text\n"
".globl " SYMBOL_STRING(ctiTrampoline) "\n"
HIDE_SYMBOL(ctiTrampoline) "\n"
SYMBOL_STRING(ctiTrampoline) ":" "\n"
    "pushl %ebp" "\n"
    "movl %esp, %ebp" "\n"
    "pushl %esi" "\n"
    "pushl %edi" "\n"
    "pushl %ebx" "\n"
    "subl $0x3c, %esp" "\n"
    "movl $512, %esi" "\n"
    "movl 0x58(%esp), %edi" "\n"
    "call *0x50(%esp)" "\n"
    "addl $0x3c, %esp" "\n"
    "popl %ebx" "\n"
    "popl %edi" "\n"
    "popl %esi" "\n"
    "popl %ebp" "\n"
    "ret" "\n"
);
```

Example of Simple JIT

```
int main(int argc, char *argv[]) {
  // Machine code for:
  //   mov eax, 0
  //   ret
  unsigned char code[] = {0xb8, 0x00, 0x00, 0x00, 0x00, 0xc3};

  // Overwrite immediate value "0" in the instruction
  // with the user's value.  This will make our code:
  //   mov eax, <user's value>
  //   ret
  int num = atoi(argv[1]);
  memcpy(&code[1], &num, 4);

  // Allocate writable/executable memory.
  // Note: real programs should not map memory both writable
  // and executable because it is a security risk.
  void *mem = mmap(NULL, sizeof(code), PROT_WRITE | PROT_EXEC,
                   MAP_ANON | MAP_PRIVATE, -1, 0);
  memcpy(mem, code, sizeof(code));

  // The function will return the user's value.
  int (*func)() = mem;
  return func();
}
```