JUST-IN-TIME COMPILATION

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira
fernando@dcc.ufmg.br
A Toy Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

int main(void) {
    char* program;
    int (*fnptr)(void);
    int a;
    program = mmap(NULL, 1000, PROT_EXEC | PROT_READ |
                    PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
    program[0] = 0xB8;
    program[1] = 0x34;
    program[2] = 0x12;
    program[3] = 0;
    program[4] = 0;
    program[5] = 0xC3;
    fnptr = (int (*)(void)) program;
    a = fnptr();
    printf("Result = %X\n",a);
}
```

1) What is the program on the left doing?
2) What is this API all about?
3) What does this program have to do with a just-in-time compiler?
Just-in-Time Compilers

- A JIT compiler translates a program into binary code while this program is being executed.
- We can compile a function as soon as it is necessary.
  - This is Google's V8 approach.
- Or we can first interpret the function, and after we realize that this function is hot, we compile it into binary.
  - This is the approach of Mozilla's IonMonkey.

1) When/where/why are just-in-time compilers usually used?

2) Can a JIT compiled program run faster than a statically compiled program?

3) Which famous JIT compilers do we know?
There are many JIT compilers around

- Java Hotspot is one of the most efficient JIT compilers in use today. It was released in 1999, and has been in use since then.
- V8 is the JavaScript JIT compiler used by Google Chrome.
- IonMonkey is the JavaScript JIT compiler used by the Mozilla Firefox.
- LuaJIT ([http://luajit.org/](http://luajit.org/)) is a trace based just-in-time compiler that generates code for the Lua programming language.
- The .Net framework JITs CIL code.
- For Python we have PyPy, which runs on Cpython.
Can JITs compete with static compilers?

How can an interpreted language run faster than a compiled language?

http://benchmarksgame.alioth.debian.org/
Tradeoffs

• There are many tradeoffs involved in the JIT compilation of a program.
• The time to compile is part of the total execution time of the program.
• We may be willing to run simpler and faster optimizations (or no optimization at all) to diminish the compilation overhead.
• And we may try to look at runtime information to produce better codes.
  – Profiling is a big player here.
• The same code may be compiled many times!

Why would we compile the same code many times?
Example: Mozilla's IonMonkey

IonMonkey is one of the JIT compilers used by the Firefox browser to execute JavaScript programs.

This compiler is tightly integrated with SpiderMonkey, the JavaScript interpreter.

SpiderMonkey invokes IonMonkey to JIT compile a function either if it is often called, or if it has a loop that executes for a long time.

Why do we have so many different intermediate representations here?
When to Invoke the JIT?

• Compilation has a cost.
  – Functions that execute only once, for a few iterations, should be interpreted.

• Compiled code runs faster.
  – Functions that are called often, or that loop for a long time should be compiled.

• And we may have different optimization levels...

As an example, SpiderMonkey uses three execution modes: the first is interpretation; then we have the baseline compiler, which does not optimize the code. Finally IonMonkey kicks in, and produces highly optimized code.
TheCompilationThreshold

Many execution environments associate counters with branches. Once a counter reaches a given threshold, that code is compiled. But defining which threshold to use is very difficulty, e.g., how to minimize the area of the curve below? JITs are crowded with magic numbers.

Optimized Native Code
Unoptimized Native Code
Interpreted Code
Baseline Compiler
Optimizing Compiler

Time
Number of instructions processed (either natively or via interpretation)
Number of instructions compiled
The Million-Dollars Question

- When to invoke the JIT compiler?

1) Can you come up with a strategy to invoke the JIT compiler that optimizes for speed?

2) Do you have to execute the program a bit before calling the JIT?

3) How much information do you need to make a good guess?

4) What is the price you pay for making a wrong prediction?

5) Which programs are easy to predict?

6) Do the easy programs reflect the needs of the users?
SPECULATION
Speculation

• A key trick used by JIT compilers is *speculation*.
• We may assume that a given property is true, and then we produce code that capitalizes on that speculation.
• There are many different kinds of speculation, and they are always a *gamble*:
  – Let's assume that the type of a variable is an integer,
    • but if we have an integer overflow...
  – Let's assume that the properties of the object are fixed,
    • but if we add or remove something from the object...
  – Let's assume that the target of a call is always the same,
    • but if we point the function reference to another closure...
Inline Caching

• One of the earliest, and most effective, types of specialization was *inline caching*, an optimization developed for the Smalltalk programming language.
• Smalltalk is a dynamically typed programming language.
• In a nutshell, objects are represented as *hash-tables*.
• This is a very flexible programming model: we can add or remove properties of objects at will.
• Languages such as Python and Ruby also implement objects in this way.
• Today, inline caching is the key idea behind JITs's high performance when running JavaScript programs.

◊: Efficient implementation of the smalltalk-80 system, POPL (1984)
Virtual Tables

In Java, C++, C#, and other programming languages that are compiled statically, objects contain pointers to virtual tables. Any method call can be resolved after two pointer dereferences. The first dereference finds the virtual table of the object. The second finds the target method, given a known offset.

```java
class Animal {
    public void eat() {
        System.out.println(this + " is eating");
    }
    public String toString () { return "Animal"; }
}

class Mammal extends Animal {
    public void suckMilk() {
        System.out.println(this + " is sucking");
    }
    public String toString () { return "Mammal"; }
    public void eat() {
        System.out.println(this + " is eating like a mammal");
    }
}

class Dog extends Mammal {
    public void bark() {
        System.out.println(this + " is barking");
    }
    public String toString () { return "Dog"; }
    public void eat() {
        System.out.println(this + ", is eating like a dog");
    }
}
```

1) In order for this trick to work, we need to impose some restrictions on virtual tables. Which ones?

2) What are the virtual tables of?
   Animal a = new Animal();
   Animal m = new Mammal();
   Mammal d = new Dog()
Virtual Tables

class Animal {
    public void eat() {
        System.out.println(this + " is eating");
    }
    public String toString () { return "Animal"; }
}

class Mammal extends Animal {
    public void suckMilk() {
        System.out.println(this + " is sucking");
    }
    public String toString () { return "Mammal"; }
    public void eat() {
        System.out.println(this + " is eating like a mammal");
    }
}

class Dog extends Mammal {
    public void bark() {
        System.out.println(this + " is barking");
    }
    public String toString () { return "Dog"; }
    public void eat() {
        System.out.println(this + ", is eating like a dog");
    }
}

How to locate the target of d.eat()?
Virtual Call

First, we need to know the table d is pointing to. This requires one pointer dereference:

Second, we need to know the offset of the method eat, inside the table. This offset is always the same for any class that inherits from Animal, so we can jump blindly.

Can we have virtual tables in duck typed languages?

```
public void eat() {
    System.out.println("Eats like a dog");
}
```
Objects in Python

INT_BITS = 32

def getIndex(element):
    index = element / INT_BITS
    offset = element % INT_BITS
    bit = 1 << offset
    return (index, bit)

class Set:
    def __init__(self, capacity):
        self.capacity = capacity
        self.vector = range(1+capacity/INT_BITS)
        for i in range(len(self.vector)):
            self.vector[i] = 0
    def add(self, element):
        (index, bit) = getIndex(element)
        self.vector[index] |= bit

class ErrorSet(Set):
    def checkIndex(self, element):
        if (element > self.capacity):
            raise IndexError(str(element) + " is out of range.")
    def add(self, element):
        self.checkIndex(element)
        Set.add(self, element)
        print element, "successfully added."
def fill(set, b, e, s):
    for i in range(b, e, s):
        set.add(i)

s0 = Set(15)
fill(s0, 10, 20, 3)

s1 = ErrorSet(15)
fill(s1, 10, 14, 3)

class X:
    def __init__(self):
        self.a = 0

fill(X(), 1, 10, 3)

>>> AttributeError: X instance has no attribute 'add'
def fill(set, b, e, s):
    for i in range(b, e, s):
        set.add(i)

class Num:
    def __init__(self, num):
        self.n = num
    def add(self, num):
        self.n += num
    def __str__(self):
        return str(self.n)

n = Num(3)
print n
fill(n, 1, 10, 1)
...
Duck Typing

The program works just fine. The only requirement that `fill` expects on its first parameter is that it has a method `add` that takes two parameters. Any object that has this method, and can receive an integer on the second argument, will work with `fill`. This is called duck typing: if it squeaks like a duck, swims like a duck, eats like a duck, then it is a duck!

```python
class Num:
    def __init__(self, num):
        self.n = num
    def add(self, num):
        self.n += num
    def __str__(self):
        return str(self.n)

n = Num(3)
print(n)
fill(n, 1, 10, 1)
print(n)

>>> 3
>>> 48
```
The Price of Flexibility

- Objects, in these dynamically typed languages, are for the most part implemented as hash tables.
  - That is cool: we can add or remove methods without much hard work.
  - And mind how much code we can reuse?
- But method calls are pretty expensive.

```python
def fill(set, b, e, s):
    for i in range(b, e, s):
        set.add(i)
```

How can we make these calls cheaper?
The first time we generate code for a call, we can check the target of that call. *We know this target, because we are generating code at runtime!*

```
class Num:
    def __init__(self, n):
        self.n = n
    def add(self, num):
        self.n += num

def fill(set, b, e, s):
    for i in range(b, e, s):
        set.add(i)

n = Num(3)
print n
fill(n, 1, 10, 1)
print n
>>> 3
>>> 48
```

Could you optimize this code even further using classic compiler transformations?
Inlining on the Method

• We can also speculate on the method name, instead of doing it on the calling site:

```python
fill(set, b, e, s):
    for i in range(b, e, s):
        _f_add(set, i)

    _f_add(o, e):
        if isinstance(o, Num):
            o.n += e
        else:
            f = lookup(o, "add")
            f(o, e)
```

1) Is there any advantage to this approach, when compared to inlining at the call site?

2) Is there any disadvantage?

3) Which one is likely to change more often?
Polymorphic Calls

• If the target of a call changes during the execution of the program, then we have a polymorphic call.
• A monomorphic inline cache would have to be invalidated, and we would fall back into the expensive quest.

```python
>>> l = [Num(1), Set(1), Num(2), Set(2), Num(3), Set(3)]
>>> for o in l:
...   o.add(2)
...```

Is there anything we could do to optimize this case?
Polymorphic Calls

• If the target of a call changes during the execution of the program, then we have a polymorphic call.
• A monomorphic inline cache would have to be invalidated, and we would fall back into the expensive quest.

```python
>>> l = [Num(1), Set(1), Num(2), Set(2), Num(3), Set(3)]
>>> for o in l:
...   o.add(2)
...   ...

Would it not be better just to have the code below?

```
The Speculative Nature of Inline Caching

• Python – as well as JavaScript, Ruby, Lua and other very dynamic languages – allows the user to add or remove methods from an object.

• If such changes in the layout of an object happen, then the representation of that object must be recompiled. In this case, we need to update the inline cache.

```python
from Set import INT_BITS, getIndex, Set
def errorAdd(self, element):
    if (element > self.capacity):
        raise IndexError(str(element) + " is out of range."
    else:
        (index, bit) = getIndex(element)
        self.vector[index] |= bit
        print element, "added successfully!"

Set.add = errorAdd
s = Set(60)
s.errorAdd(59)
s.remove(59)
```
The Benefits of the Inline Cache

These numbers have been obtained by Ahn et al. for JavaScript, in the Chrome V8 compiler:\footnote{Improving JavaScript Performance by Deconstructing the Type System, PLDI (2014)}:

- Monomorphic inline cache hit:
  - 10 instructions
- Polymorphic Inline cache hit:
  - 35 instructions if there are 10 types
  - 60 instructions if there are 20 types
- Inline cache miss: 1,000 – 4,000 instructions.

Which factors could justify these numbers?
PARAMETER SPECIALIZATION
Functions are often called with same arguments

- Costa et al. have instrumented the Mozilla Firefox browser, and have used it to navigate through the 100 most visited webpages according to the Alexa index.

- They have also performed these tests on three popular benchmark suites:
  - Sunspider 1.0, V8 6.0 and Kraken 1.1

Empirical finding: most of the JavaScript functions are always called with the same arguments. This observation has been made over a corpus of 26,000 functions.

• http://www.alexa.com
• Just-in-Time Value Specialization (CGO 2013)
Many Functions are Called only Once!

Almost 50% of the functions have been called only once during a mock user session.
Most Functions are Called with Same Arguments

Almost 60% of the functions are called with only one set of arguments throughout the entire web session.
A Similar Pattern in the Benchmarks

The three upper histograms show how often each function is called.

The three lower histograms shows how often each function is called with the same suite of arguments.
So, if it is like this...

- Let's assume that the function is always called with the same arguments. We generate the best code specific to those arguments. If the function is called with different arguments, then we re-compile it, this time using a more generic approach.

```javascript
function sum(N) {
    if(typeof N != 'number')
        return 0;

    var s = 0;
    for(var i=0; i<N; i++)
    {
        if(N % 3 == 0)
            s += i*N;
        else
            s += i;
    }
    return s;
}
```

If we assume that N = 59, then we can produce this specialized code:

```javascript
function sum() {
    var s = 0;
    for(var i=0; i<59; i++)
    {
        s += i;
    }
    return s;
}
```
A Running Example – Linear Similarity Search

Given a vector \( v \) of size \( n \), holding integers, find the shortest difference \( d \) from any element in \( v \) to a parameter \( q \).

We must iterate through \( v \), looking for the smallest difference \( v[i] - q \). This is an \( O(q) \) algorithm.

```javascript
function closest(v, q, n) {
  if (n == 0) {
    throw "Error";
  } else {
    var i = 0;
    var d = 2147483647;
    while (i < n) {
      var nd = abs(s[i] - q);
      if (nd <= d) {
        d = nd;
        i++;
      }
    }
    return d;
  }
}
```

What could we do if we wanted to generate code that is specific to a given tuple \((v, q, n)\)?
function closest(v, q, n) {
    if (n == 0) {
        throw "Error";
    } else {
        var i = 0;
        var d = 2147483647;
        while (i < n) {
            var nd = abs(s[i]-q);
            if (nd <= d)
                d = nd;
            i++;
        }
        return d;
    }
}

This CFG has two entries. The **first** marks the ordinary starting point of the function. The **second** exists if we compiled this binary while it was being interpreted. This entry is the point where the interpreter was when the JIT was invoked.
Identifying the Parameters

Because we have two entry points, there are two places from where we can read arguments.

To find the values of these arguments, we just ask them to the interpreter. We can ask this question by inspecting the interpreter's stack at the very moment when the JIT compiler was invoked.
Dead Code Elimination (Dead Code Avoidance?)

If we assume that the function will not be called again, we do not need to generate the function's entry block.

We can also remove any other blocks that are not reachable from the "On-Stack-Replacement" block.
Array Bounds Checking Elimination

In JavaScript, every array access is guarded against out-of-bounds accesses. We can eliminate some of these guards.

Let's just check the pattern \( i = \text{start}; \ i < n; \ i++ \). This pattern is easy to spot, and it already catches many induction variables. Therefore, if we have \( a[i] \), and \( n \leq a.length \), then we can eliminate the guard.
Loop Inversion

Loop inversion consists in changing a while loop into a do-while:

```
while(cond) {
  .
  .
  .
}
```

```
if(cond) {
  do {
    ...
  } while (cond);
}
```

1) Why do we need the surrounding conditional?
2) Which info do we need, to eliminate this conditional?

On stack replacement

```
v = load[0]
q = 42
n = 100
i_3 = 40
d_4 = 2147483647
```

```
L3: i_1 = φ (i_2, i_3)
d_1 = φ (d_3, d_4)
if (i_1 < n) goto L7
```

```
L7: t_0 = 4 * i_1
t_1 = v[t_0]
nd = abs(t_1, q)
if (nd > d_1) goto L9
```

```
L4: return d_3
```

```
L6: d_2 = nd
```

```
L9: d_3 = φ (d_1, d_2)
i_2 = i_1 + 1
goto L3
```
The standard loop inversion transformation requires us to surround the new repeat loop with a **conditional**. However, we can evaluate this conditional, if we know the value of the variables that are being tested, in this case, \( i \) and \( n \). Hence, we can avoid the cost of creating this code.

Otherwise, if we generate code for the conditional, then a subsequent *sparse conditional constant propagation* phase might still be able to remove the surrounding 'if'.

```javascript
while (i < n) {
    var nd = abs(s[i] - q);
    if (nd <= d)
        d = nd;
    i++;
}
```

```javascript
if (i < n) {
    do {
        var nd = abs(s[i] - q);
        if (nd <= d)
            d = nd;
        i++;
    } while (i < n)
}
```
Constant Propagation + Dead Code Elimination

We close our suite of optimizations with constant propagation, followed by a new dead-code-elimination phase.

This is possibly the optimization that gives us the largest profit.

It is often the case that we can eliminate some conditional tests too, as we can evaluate the condition "statically".

On stack replacement

\[
\begin{align*}
v &= \text{load}[0] \\
q &= 42 \\
n &= 100 \\
i_3 &= 40 \\
d_4 &= 2147483647
\end{align*}
\]

\[
\begin{align*}
\text{L3: } i_1 &= \phi (i_2, i_3) \\
d_1 &= \phi (d_3, d_4)
\end{align*}
\]

\[
\begin{align*}
\text{L7: } t_0 &= 4 \times i \\
t_1 &= v[t_0] \\
nd &= \text{abs}(t_1, q) \\
\text{if } (nd > d_1) \text{ goto L7}
\end{align*}
\]

On stack replacement

\[
\begin{align*}
v &= \text{load}[0] \\
q &= 42 \\
n &= 100 \\
i_3 &= 40 \\
d_4 &= 2147483647
\end{align*}
\]

\[
\begin{align*}
\text{L7: } t_0 &= 4 \times i \\
t_1 &= v[t_0] \\
nd &= \text{abs}(t_1, q) \\
\text{if } (nd > d_1) \text{ goto L7}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } t_0 &= 4 \times i \\
t_1 &= v[t_0] \\
nd &= \text{abs}(t_1, q) \\
\text{if } (nd > d_1) \text{ goto L7}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } t_0 &= 4 \times i \\
t_1 &= v[t_0] \\
nd &= \text{abs}(t_1, q) \\
\text{if } (nd > d_1) \text{ goto L7}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } t_0 &= 4 \times i \\
t_1 &= v[t_0] \\
nd &= \text{abs}(t_1, q) \\
\text{if } (nd > d_1) \text{ goto L7}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } t_0 &= 4 \times i \\
t_1 &= v[t_0] \\
nd &= \text{abs}(t_1, q) \\
\text{if } (nd > d_1) \text{ goto L7}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } t_0 &= 4 \times i \\
t_1 &= v[t_0] \\
nd &= \text{abs}(t_1, q) \\
\text{if } (nd > d_1) \text{ goto L7}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } t_0 &= 4 \times i \\
t_1 &= v[t_0] \\
nd &= \text{abs}(t_1, q) \\
\text{if } (nd > d_1) \text{ goto L7}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } t_0 &= 4 \times i \\
t_1 &= v[t_0] \\
nd &= \text{abs}(t_1, q) \\
\text{if } (nd > d_1) \text{ goto L7}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } t_0 &= 4 \times i \\
t_1 &= v[t_0] \\
nd &= \text{abs}(t_1, q) \\
\text{if } (nd > d_1) \text{ goto L7}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } t_0 &= 4 \times i \\
t_1 &= v[t_0] \\
nd &= \text{abs}(t_1, q) \\
\text{if } (nd > d_1) \text{ goto L7}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } t_0 &= 4 \times i \\
t_1 &= v[t_0] \\
nd &= \text{abs}(t_1, q) \\
\text{if } (nd > d_1) \text{ goto L7}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } t_0 &= 4 \times i \\
t_1 &= v[t_0] \\
nd &= \text{abs}(t_1, q) \\
\text{if } (nd > d_1) \text{ goto L7}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } t_0 &= 4 \times i \\
t_1 &= v[t_0] \\
nd &= \text{abs}(t_1, q) \\
\text{if } (nd > d_1) \text{ goto L7}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } t_0 &= 4 \times i \\
t_1 &= v[t_0] \\
nd &= \text{abs}(t_1, q) \\
\text{if } (nd > d_1) \text{ goto L7}
\end{align*}
\]

\[i_2 = i_1 + 1 \\
\text{if } (i_2 > n) \text{ goto L4}
\]

\[
\begin{align*}
\text{L7: } d_3 &= \phi (d_1, d_2) \\
i_2 &= i_1 + 1 \\
\text{if } (i_2 > n) \text{ goto L4}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } d_3 &= \phi (d_1, d_2) \\
i_2 &= i_1 + 1 \\
\text{if } (i_2 > n) \text{ goto L4}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } d_3 &= \phi (d_1, d_2) \\
i_2 &= i_1 + 1 \\
\text{if } (i_2 > n) \text{ goto L4}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } d_3 &= \phi (d_1, d_2) \\
i_2 &= i_1 + 1 \\
\text{if } (i_2 > n) \text{ goto L4}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } d_3 &= \phi (d_1, d_2) \\
i_2 &= i_1 + 1 \\
\text{if } (i_2 > n) \text{ goto L4}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } d_3 &= \phi (d_1, d_2) \\
i_2 &= i_1 + 1 \\
\text{if } (i_2 > n) \text{ goto L4}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } d_3 &= \phi (d_1, d_2) \\
i_2 &= i_1 + 1 \\
\text{if } (i_2 > n) \text{ goto L4}
\end{align*}
\]

\[
\begin{align*}
\text{L7: } d_3 &= \phi (d_1, d_2) \\
i_2 &= i_1 + 1 \\
\text{if } (i_2 > n) \text{ goto L4}
\end{align*}
\]
The Specialized Code

• The specialized code is much shorter than the original code.
• This code size reduction also improves compilation time. In particular, it speeds up register allocation.

Are there other optimizations that could be applied in this context of runtime specialization?
function inc(x) {  
  return x + 1;  
}

function map(s, b, n, f) {  
  var i = b;  
  while (i < n) {  
    s[i] = f(s[i]);  
    i++;  
  }  
  return s;  
}

print(map(new Array(1, 2, 3, 4, 5), 2, 5, inc));

1) Which call could we inline in this example?
2) How would the resulting code look like?
Function Call Inlining

function inc(x) {
    return x + 1;
}

function map(s, b, n, f) {
    var i = b;
    while (i < n) {
        s[i] = f(s[i]);
        i++;
    }
    return s;
}

print (map(new Array(1, 2, 3, 4, 5), 2, 5, inc));

We can also perform loop unrolling in this example. How would be the resulting code?
Loop unrolling, in this case, is very effective, as we know the initial and final limits of the loop. In this case, we do not need to insert prolog and epilogue codes to preserve the semantics of the program.

```javascript
function map(s, b, n, f) {
    var i = b;
    while (i < n) {
        s[i] = s[i] + 1;
        i++;
    }
    return s;
}
print (map(new Array(1, 2, 3, 4, 5), 2, 5, inc));
```
TRACE COMPILATION
What is a JIT trace compiler?

• A trace-based JIT compiler translates only the most executed paths in the program’s control flow to machine code.
• A trace is a linear sequence of code, that represents a hot path in the program.
• Two or more traces can be combined into a tree.
• Execution alternates between traces and interpreter.

What are the advantages and disadvantages of trace compilation over traditional method compilation?
The anatomy of a trace compiler

• TraceMonkey is the trace based JIT compiler used in the Mozilla Firefox Browser.
function foo(n) {
    var sum = 0;
    for (i = 0; i < n; i++) {
        sum += i;
    }
    return sum;
}

Can you see the correspondence between source code and bytecodes?

jsparser → AST → jsemitter
The trace engine kicks in

- TraceMonkey interprets the bytecodes.
- Once a loop is found, it may decide to ask Nanojit to transform it into machine code (e.g., x86, ARM).
  - Nanojit reads LIR and produces x86
- Hence, TraceMonkey must convert this trace of bytecodes into LIR
Bytecodes

00: getname n
02: setlocal 0
06: zero
07: setlocal 1
11: zero
12: setlocal 2
16: goto 35 (19)
19: trace
20: getlocal 1
23: getlocal 2
26: add
27: setlocal 1
31: localinc 2
35: getlocal 2
38: getlocal 0
41: lt
42: ifne 19 (-23)
45: getlocal 1
48: return
49: stop

Nanojit LIR

L: load "i" %r1
load "sum" %r2
add %r1 %r2 %r1
%p0 = ovf()
bra %p0 Exit1
store %r1 "sum"
inc %r2
store %r2 "i"
%p0 = ovf()
bra %p0 Exit2
load "i" %r0
load "n" %r1
lt %p0 %r0 %r1
bne %p0 L

Overflow tests are required by operations such as add, sub, inc, dec, mul.

From bytecodes to LIR
Why do we have overflow tests?

- Many scripting languages represent numbers as floating-point values.
  - Arithmetic operations are not very efficient.
- The compiler sometimes is able to infer that these numbers can be used as integers.
  - But floating-point numbers are larger than integers.
  - This is another example of speculative optimization.
  - Thus, every arithmetic operation that might cause an overflow must be preceded by a test. If the test fails, then the runtime engine must change the number's type back to floating-point.
The overflow tests are also translated into machine code.

Can you come up with an optimization to eliminate some of the overflow checks?
How to eliminate the redundant tests

• We use range analysis:
  – Find the range of integer values that a variable might hold during the execution of the trace.

```javascript
function foo(n) {
    var sum = 0;
    for (i = 0; i < n; i++) {
        sum += i;
    }
    return sum;
}
```

**Example:** if we know that \( n \) is 10, and \( i \) is always less than \( n \), then we will never have an overflow if we add 1 to \( i \).
Cheating at runtime

- **A static analysis** must be very conservative: if we do not know for sure the value of \( n \), then we must assume that it may be anywhere in \([-\infty, +\infty]\).
- **However, we are not a static analysis!**

```javascript
function foo(n) {
    var sum = 0;
    for(i = 0; i < n; i++){
        sum+=i;
    }
    return sum;
}
```

- We are compiling at runtime!
- To know the value of \( n \), just ask the interpreter.
How the algorithm works

• Create a constraint graph.
  – While the trace is translated to LIR.

• Propagate range intervals.
  – Before sending the LIR to Nanojit.
  – Using infinite precision arithmetic.

• Eliminate tests whenever it is safe to do so.
  – We tell Nanojit that code for some overflow tests should not be produced.
The constraint graph

- We have four categories of vertices:

**Name**

- $N_x$

**Arithmetic**

- inc
- dec
- add
- sub
- mul

**Relational**

- lt
- leq
- gt
- geq
- eq

**Assignment**
Building the constraint graph

19: trace
20: getlocal sum
23: getlocal i
26: add
27: setlocal sum
31: localinc i
35: getlocal n
38: getlocal i
41: lt
42: ifne 19 (-23)

- We start building the constraint graph once TraceMonkey starts recording a trace.
- TraceMonkey starts at the branch instruction, which is the first instruction visited in the loop.
  - Although it is at the end of the trace.

In terms of code generation, can you recognize the pattern of bytecodes created for the test if n < i goto L?
19: trace
20: getlocal sum
23: getlocal i
26: add
27: setlocal sum
31: localinc i
35: getlocal n
38: getlocal i
41: lt
42: ifne 19 (-23)

We check the interpreter’s stack to find out that n is 10.
19: trace
20: getlocal sum
23: getlocal i
26: add
27: setlocal sum
31: localinc i
35: getlocal
38: getlocal i
41: lt
42: ifne 19 (-23)

Why do we not initialize i with 0 in our constraint graph?

i started holding 0, but at this point, its value is already 1.
19: trace
20: getlocal sum
23: getlocal i
26: add
27: setlocal sum
31: localinc i
35: getlocal n
38: getlocal i
41: lt
42: ifne 19 (-23)

Comparisons are great! We can learn new information about variables.

Now we know that i is less than 10, so we rename it to $i_1$.
We also rename n dynamically?

We are renaming after conditionals. Which program representation are we creating?
19: trace
20: getlocal sum
23: getlocal i
26: add
27: setlocal sum
31: localinc i
35: getlocal n
38: getlocal i
41: lt
42: ifne 19 (-23)

The current value of sum on the interpreter's stack is 1.
For the sum, we get the last version of variable $i$, which is $i_1$.

Why do we pair up $\text{sum}_0$ with $i_1$, and not with $i_0$?

19: trace
20: getlocal $sum$
23: getlocal $i$
26: add
27: setlocal $sum$
31: localinc $i$
35: getlocal $n$
38: getlocal $i$
41: lt
42: ifne 19 (-23)
19: trace
20: getlocal sum
23: getlocal i
26: add
27: setlocal sum
31: localinc i
35: getlocal n
38: getlocal i
41: lt
42: ifne 19 (-23)

The last version of variable i is still $i_1$ so we use it as a parameter of the increment.
Some variables, like \( n \), have not been updated inside the trace. We know that they are \textit{constants}.

The comparisons allows us to infer \textbf{bounds} on the variables that are updated in the trace.

Other variables, like \( \text{sum} \), have been \textit{updated} inside the trace.
The next phase of our algorithm is the propagation of range intervals.

We start by assigning conservative i.e., \([-\infty, +\infty]\), bounds to the ranges of variables updated inside the trace.
We propagate ranges through arithmetic and relational nodes in topological order. 

No need to sort the graph topologically. Follow the order in which nodes are created.

Why does the order of creation of nodes give us the topological ordering of the constraint graph?
We are doing abstract interpretation. We need an abstract semantics for every operation in our constraint graph. As an example, we know that \([-\infty, +\infty] + [-\infty, 9] = [-\infty, +\infty]\)
After range propagation we can check which overflow tests are really necessary.

1) How many overflow checks do we have in this constraint graph?

2) Is there any overflow check that is redundant?
After range propagation we can check which overflow tests are really necessary.

This increment operation will never cause an overflow, for it receives at most the integer 9.

The add operation might cause an overflow, for one of the inputs – sum – might be very large.
L: load “i” %r1
  load “sum” %r2
  add %r1 %r2 %r1
  %p0 = ovf()
  bra %p0 Exit1
  store %r1 “sum”
  inc %r2
  store %r2 “i”
  %p0 = ovf()
  bra %p0 Exit2
  load “i” %r0
  load “n” %r1
  lt %p0 %r0 %r1
  bne %p0 L

✓

✗
A Bit of History

• A comprehensive survey on just-in-time compilation is given by John Aycock.
• Class Inference was proposed by Chambers and Ungar.
• The idea of Parameter Specialization was proposed by Costa et al.
• The overflow elimination technique was taken from a paper by Sol et al.

• Sousa, R., Guillon, C., Pereira, F. and Bigonha, M. "Dynamic Elimination of Overflow Tests in a Trace Compiler", CC, p 2-21 (2011)