# Performance comparisons of Java and Groovy

Jochen Theodorou
Groovy Project Tech Lead
SpringSource Germany

Groovy is a strong and dynamic typed language with static elements

- Dynamic language
- Has an MOP (add/remove/update methods)
- Instance based multimethods
- Multi threaded (uses java threads)
- Runtime class generation or compilation to file
- Joint compilation of Groovy and Java (or Scala)
- Compiles to normal classes with all signatures visible

# Groovy Intro

- Tight integration with Java (Groovy extends Java extends Groovy)
- Support for generic signature
- Support for annotations
- In Groovy 1.7: Inner classes
- Overloaded Methods
- Support for closures
- Duck typing

# Groovy Intro

- Dynamic typing
- Static typing possible, but with a different concept
- Supports java security model
- Native Java Bean property support

# Groovy Intro

## Differences to Java:

- Array init syntax is not supported
- Semis are optional
- No generics testing in expressions
- Parents are partially optional
- Native lists and maps
- Additional loop constructs
- Additional methods on standard classes

## Important Projects:

**Grails** for Web Applications

**Griffon** for Swing Applications

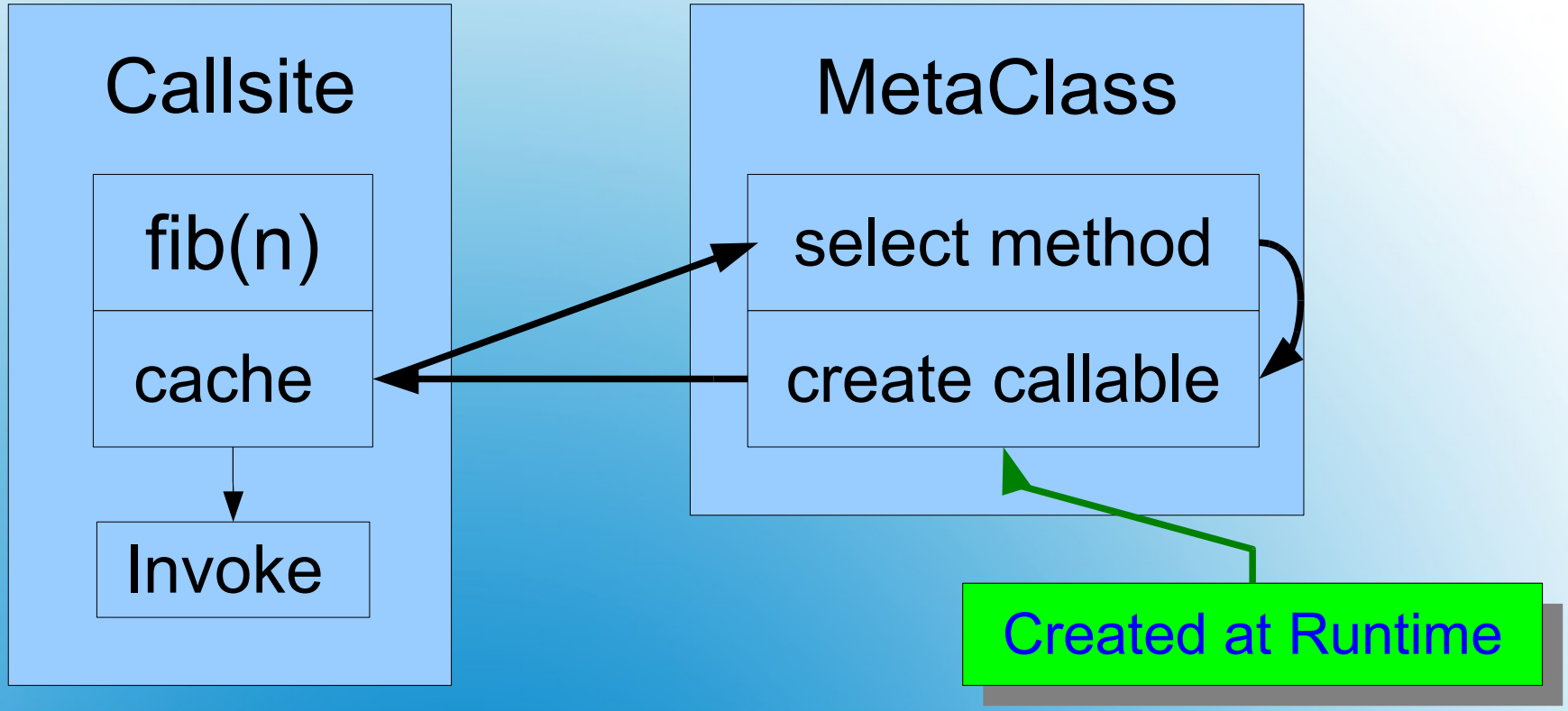
**Gradle** for Buildsystems

**Gparalizer** for Grid Computing

## Groovy 1.6 Callsite Caching:

- Class stores an CallSite[]
- Callsite becomes invalid on meta class operations
- meta class might be changed from a different thread
- Execution method might be precreated, use reflection or runtime generated

# Callsite caching

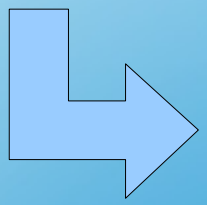


Problem:

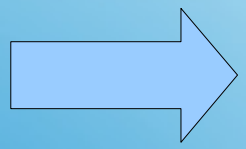
Multi threaded changes to meta classes require a volatile or synchronized checke at the call site

Loops are often optimized by loop unrolling

```
int x = 0;
for(int i=0; i<3; i++) x++;
```



```
int x = 0;
x++;
x++;
x++;
```



```
x=3;
```

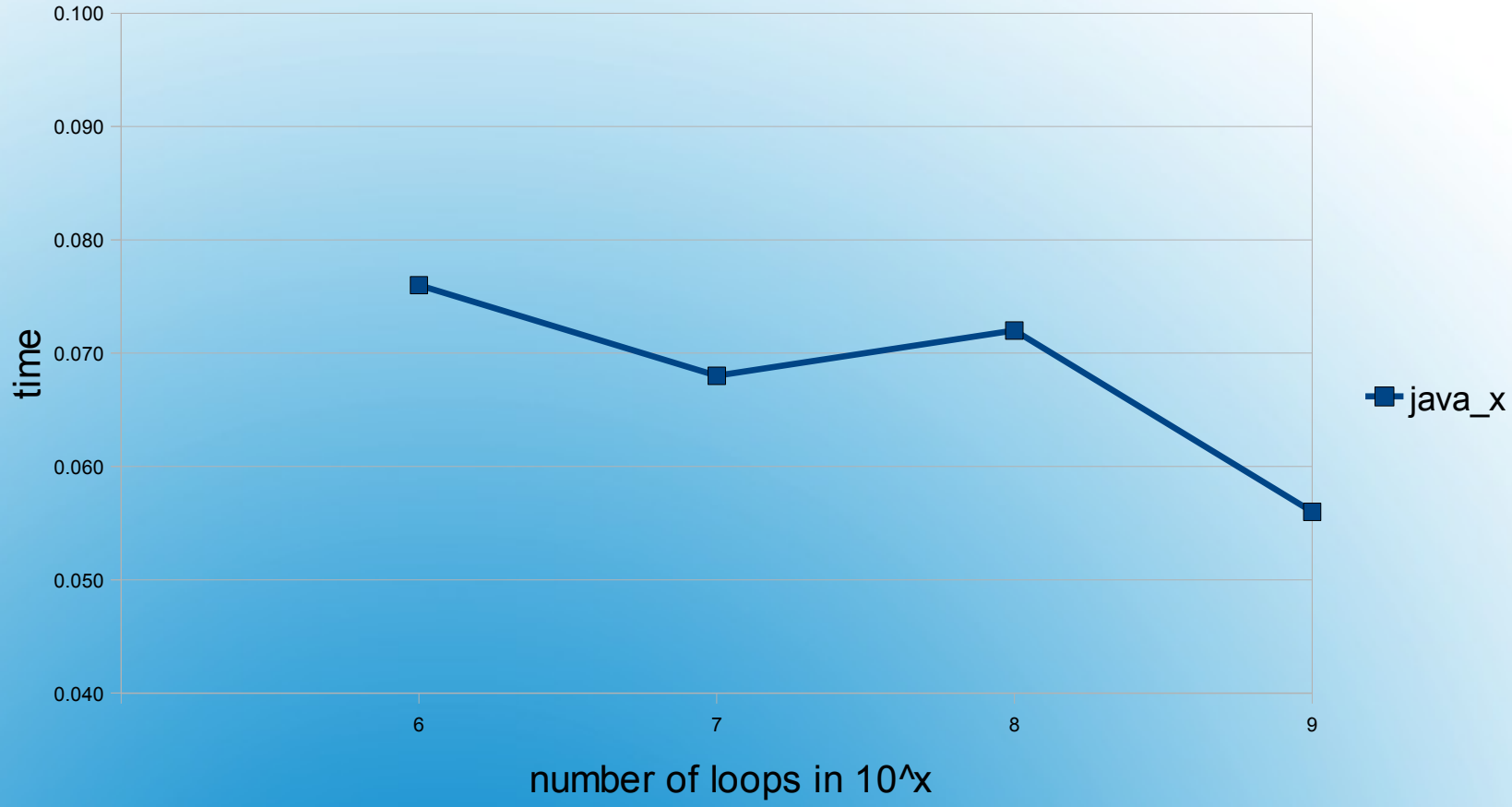
# Loops

My Example:

```
int c = Integer.parseInt(args[0]);
int x = 0;
while (x<c) x++;
```

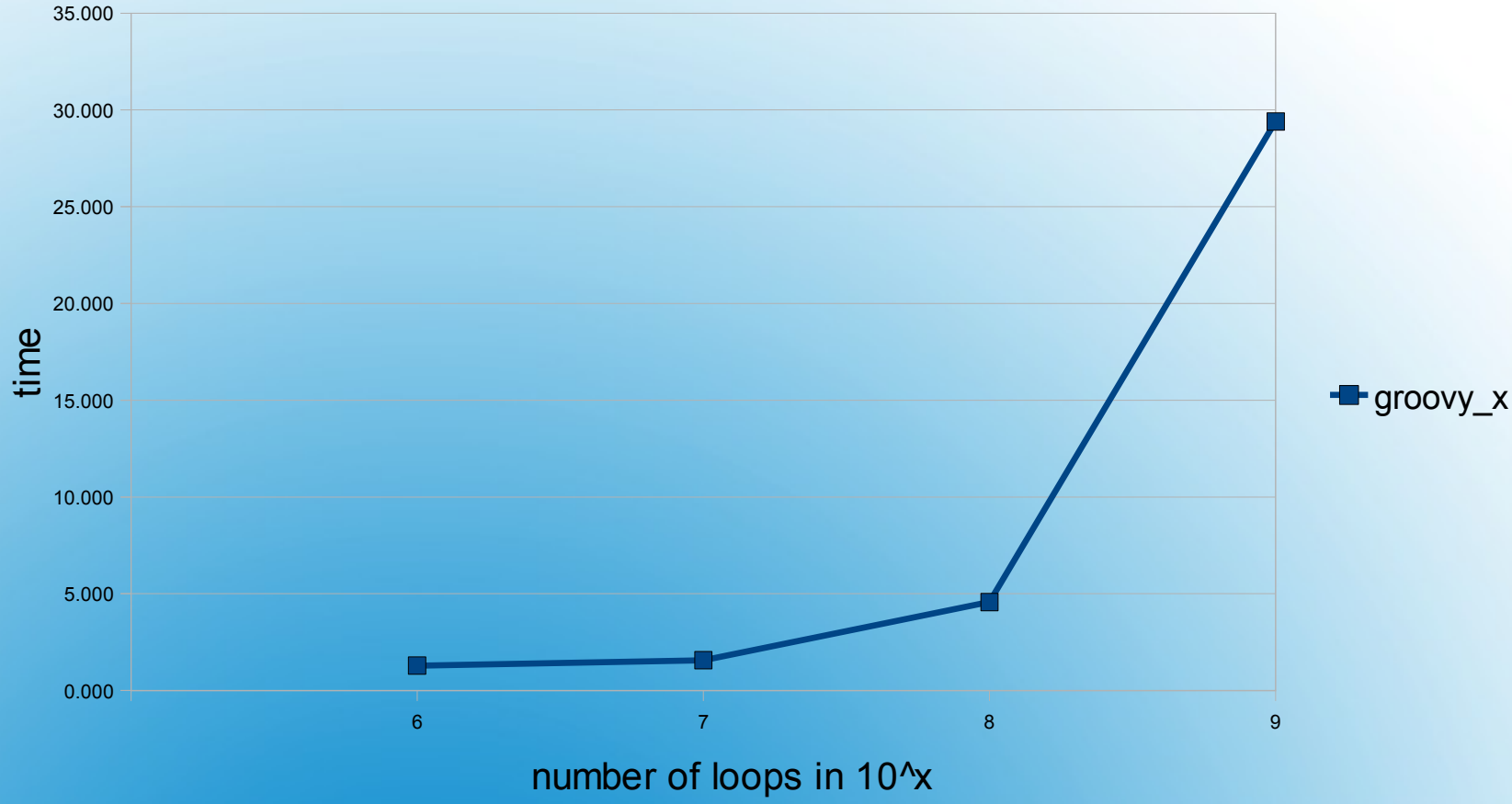
**This loop can be removed at runtime!**

# Proof:


0.100
0.090
0.080
0.070
0.060
0.050
0.040
time
6
7
8
9
number of loops in 10^x
java_x

# Loops

# Situation in Groovy:




# Loops

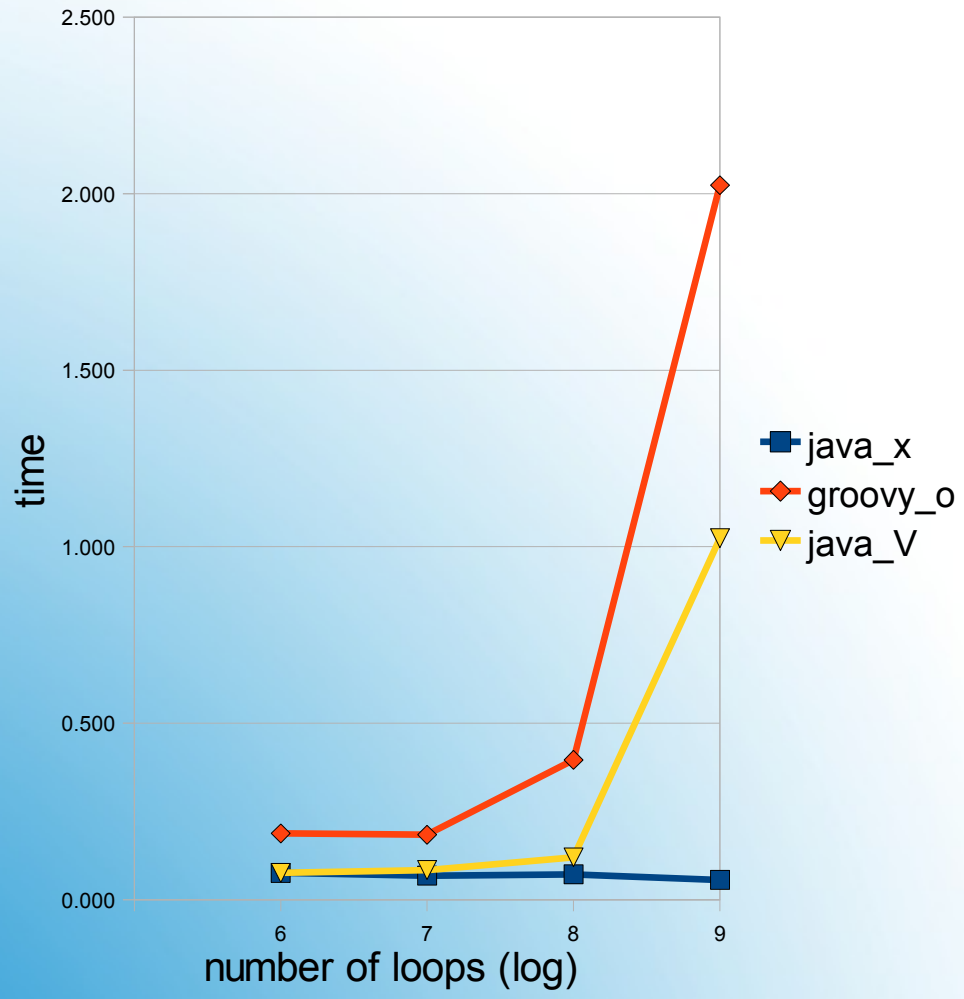
- Loop unrolling might be possible
- Removing the code is not
- This makes code blocks larger than needed
- Does allow less optimizations

# Loops

# Loops

java_V

```
private volatile int t = 0;
public void loop(int n) {
  int x = 0;
  while  (x<n) {
    if (t==0) x++
  }
}
```

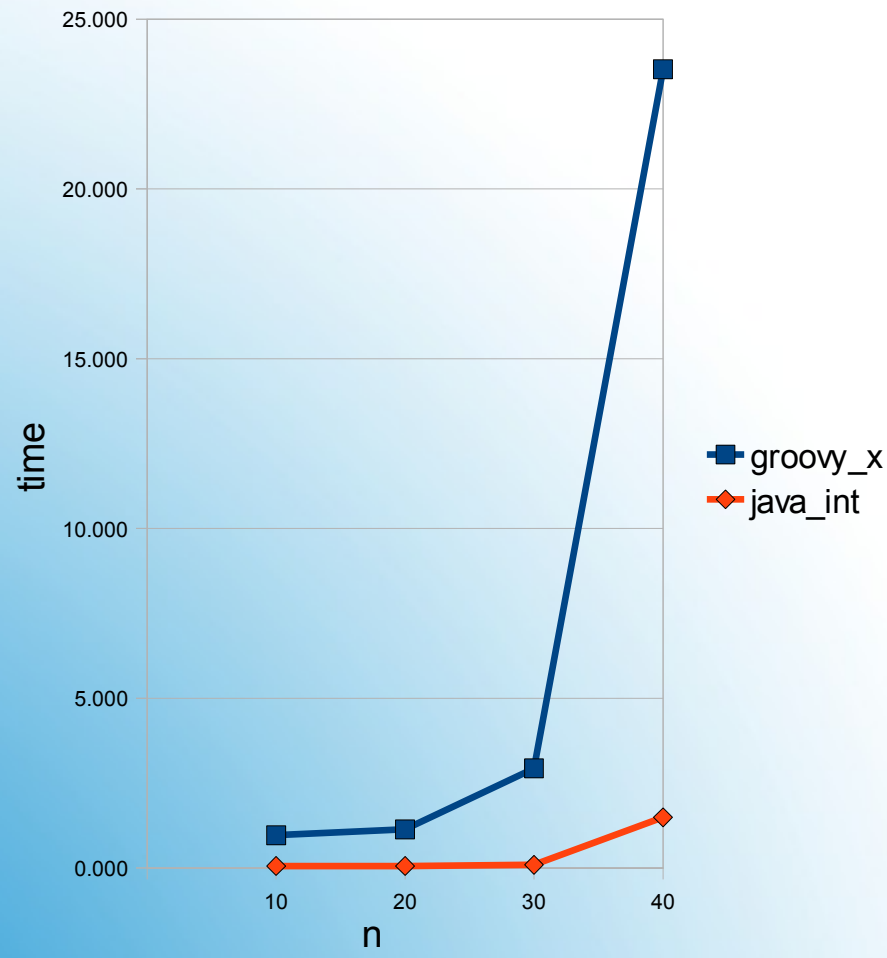


The usage of volataile prevents the code being optimized away

No solution to this!?

java_int

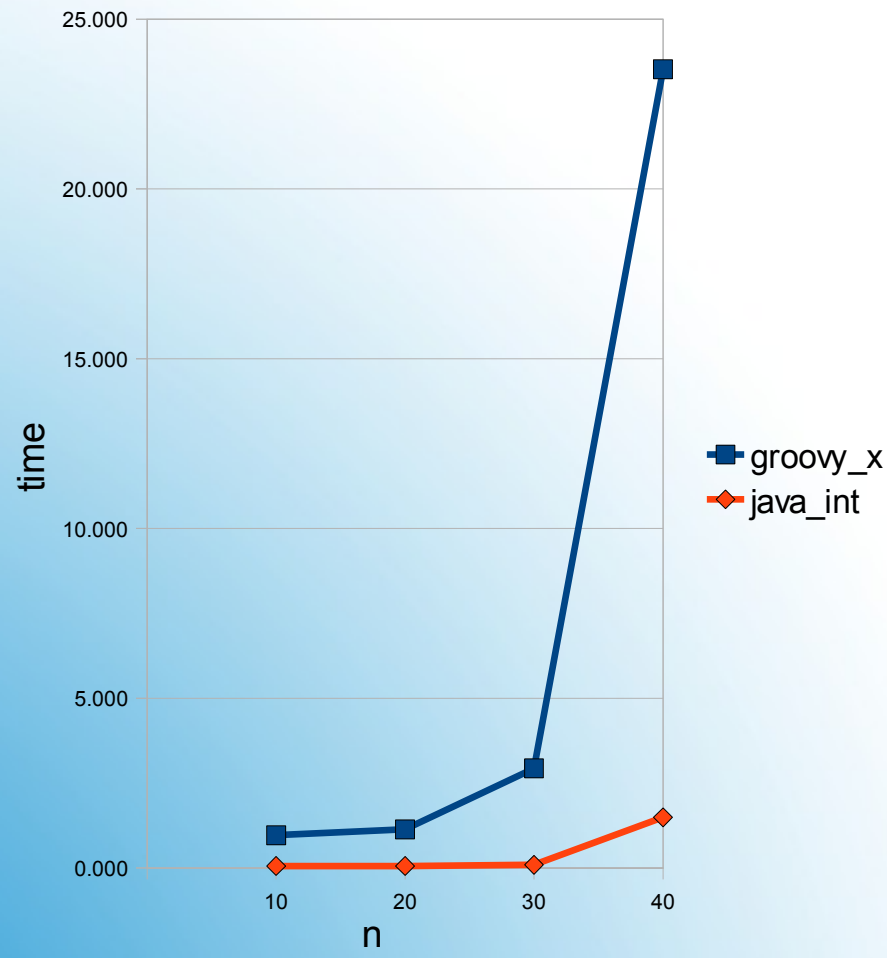
```
public int fib(int n) {
  if(n<2) return n;
  return fib(n-1) +
         fib(n-2);
}
```




# Fibonacci

groovy_x

```
def fib(n) {
  if(n<2) return n
  return fib(n-1) +
         fib(n-2)
}
```

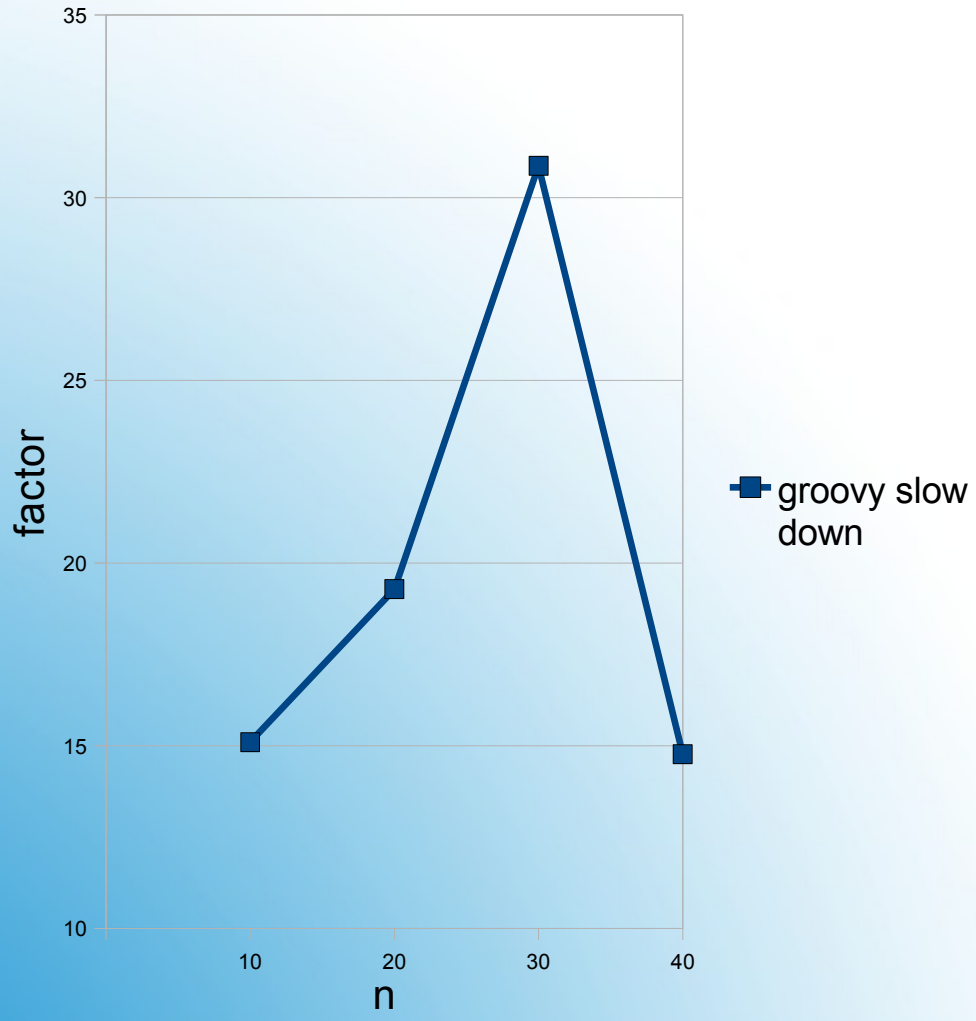



# Fibonacci

Is Groovy slow?

Are the programs equal?

No.




# Fibonacci

To perform n<2 in Groovy we actually do:

n.compareTo(2)<0

- n is Integer
- compareTo will be called directly
- Still the bytecode version with primitives is faster

# Fibonacci

To perform x+y in Groovy we actually call:

```
DGM#plus(int x,int y) {
    return x+y;
}
```

- x and y exist as Integer on the stack
- to do x+y, we have to unbox x and y
- the result needs to be boxed again
- dynamic method call to this method
- x and y are stored in Object[]

Even if the Java program is changed to use Integer, the performance is about the same.

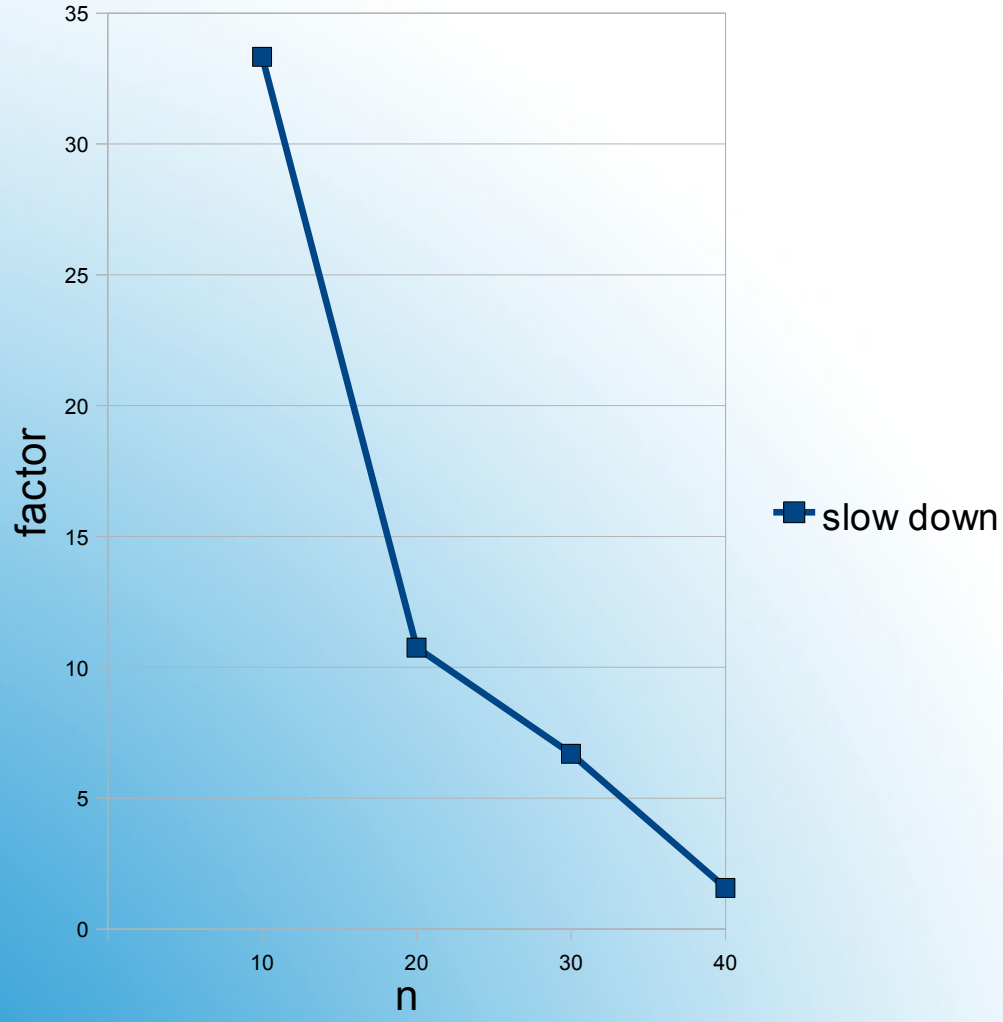
Boxing does cost, but not as much to explain the low speed

Are method calls responsible?

# Fibonacci

Java using BigInteger compared with Groovy using BigInteger

In the end only 57% slower



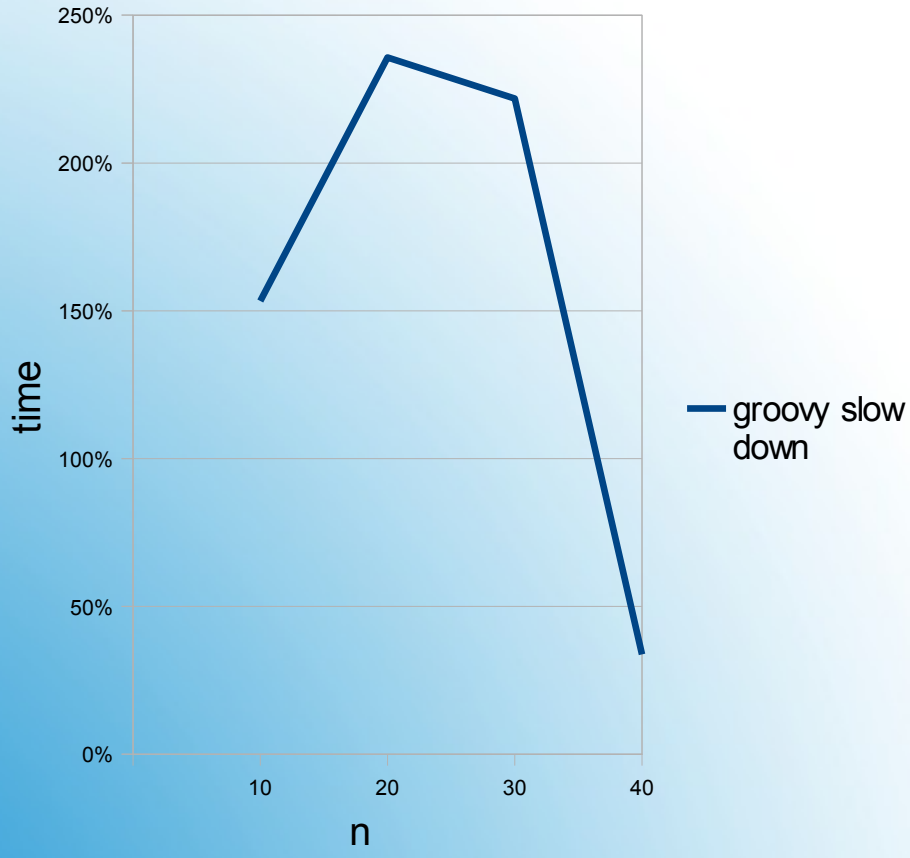

# Fibonacci

- Hotspot needs much longer for Groovy
- Groovy has an addional startup penalty
- Method calls are about 50% slower

# Fibonacci

# fast path compiler:

- "private" allows a direct method call
- optional typing allows usage of primitive values
- meta programing still possible




**FPC**

groovy_o

by private enabled
direct method call

optional types

```
def fib(n) {
  return fib_p(n)
}
private int fib_p(int x) {
  if (n<2) return n
  return fib_p(n-1) +
         fib_p(n-2)
}
```

- more clean stack trace
- less bytecode generation at runtime
- less class loading problems
- lower initial costs compared to generating

# Instrumentation based Hotspot

- Agent cannot attach itself ot its own VM
- Continously rewriting methods seems to cause problems

This causes Problems if groovy is used:

- In a restricted environment
- As library

GSoc 2008
Chanwit Kaewkasi
http://code.google.com/p/gjit/

## Replacing the method content with a callable is not enough

- Stack trace will be even more problematic to read (line number and file can be retained, class name not)
- Requires runtime byte code generation with its class loading  and permgen problems (annok?)
- Tricking with sun.reflect package

# Runtime generated Callables

- Microbenchmarks are EVIL!
- What do we need that speed for?
- If you are trying to be as fast as Java, you have to fight smallest problems
- Possible good solutions for us, are not always good for hotspot engineer minds

# Conclusion