Who should be doing these fancy optimizations anyway?

Fredrik Öhrström
Principal Member of Technical Staff
ORACLE JRockit

Features of JRockit

- Targets long running servers
- No interpreter
- Advanced profiling tools

JIT and Optimizer

JIT synchronous

OPT asynchronous, background thread

- Use the same code pipeline,
  most optimizations are simply turned off in the JIT.

JIT and Optimizer

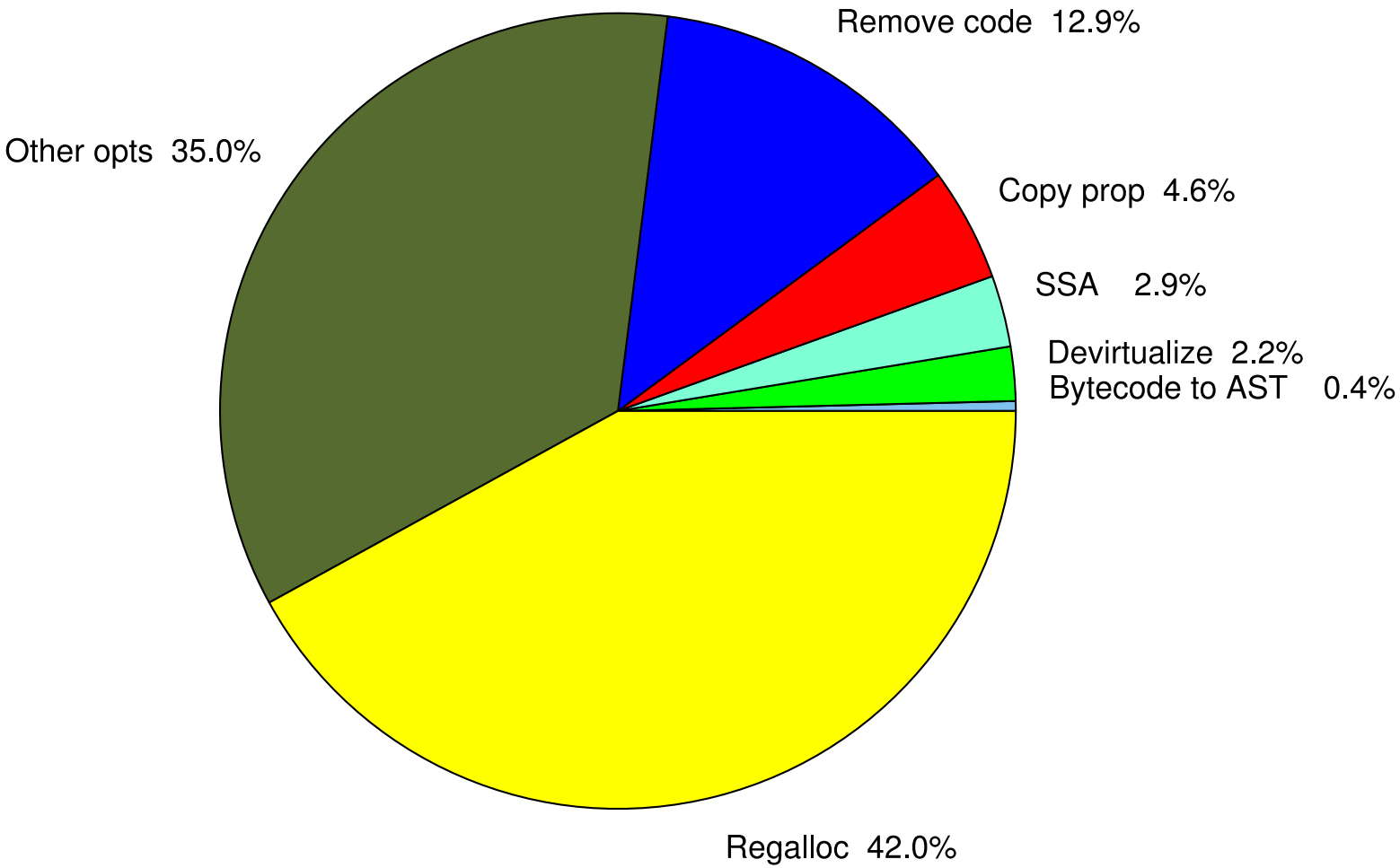
JIT synchronous

OPT asynchronous, background thread

- Use the same code pipeline,
  most optimizations are simply turned off in the JIT.
- Always turns the bytecode into an AST!

A JRockit developer might never see an actual bytecode.

SPECjAppServer optimization times



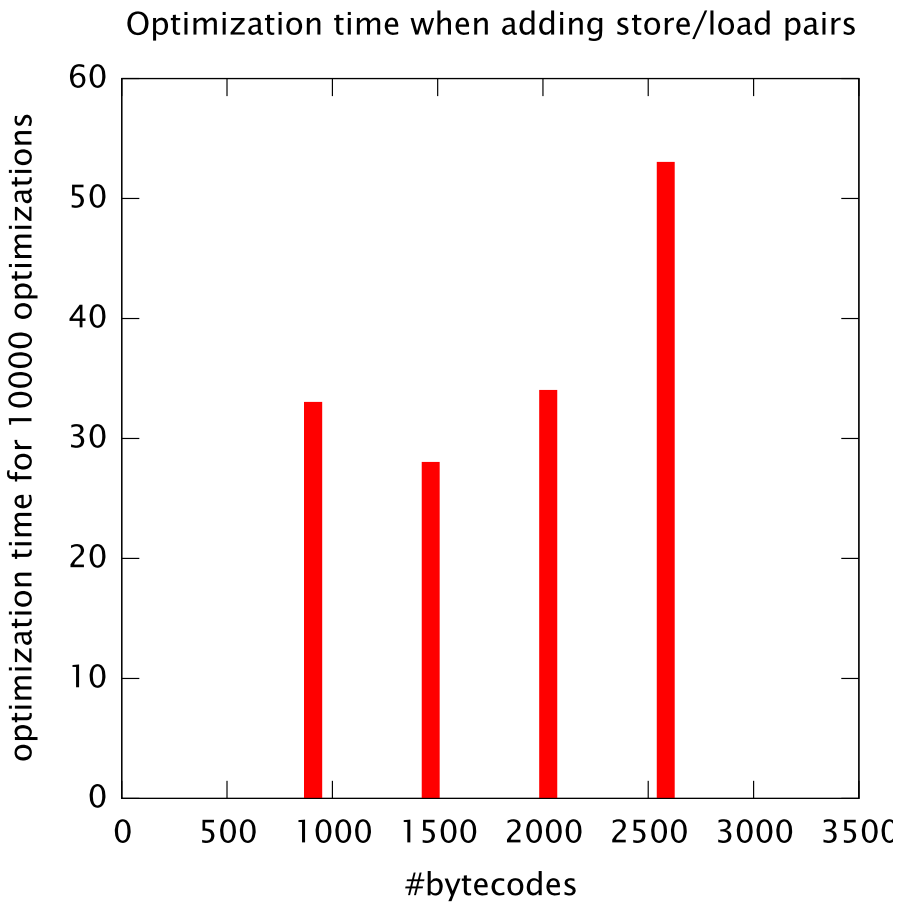
Anti-optimization

Anti-optimization

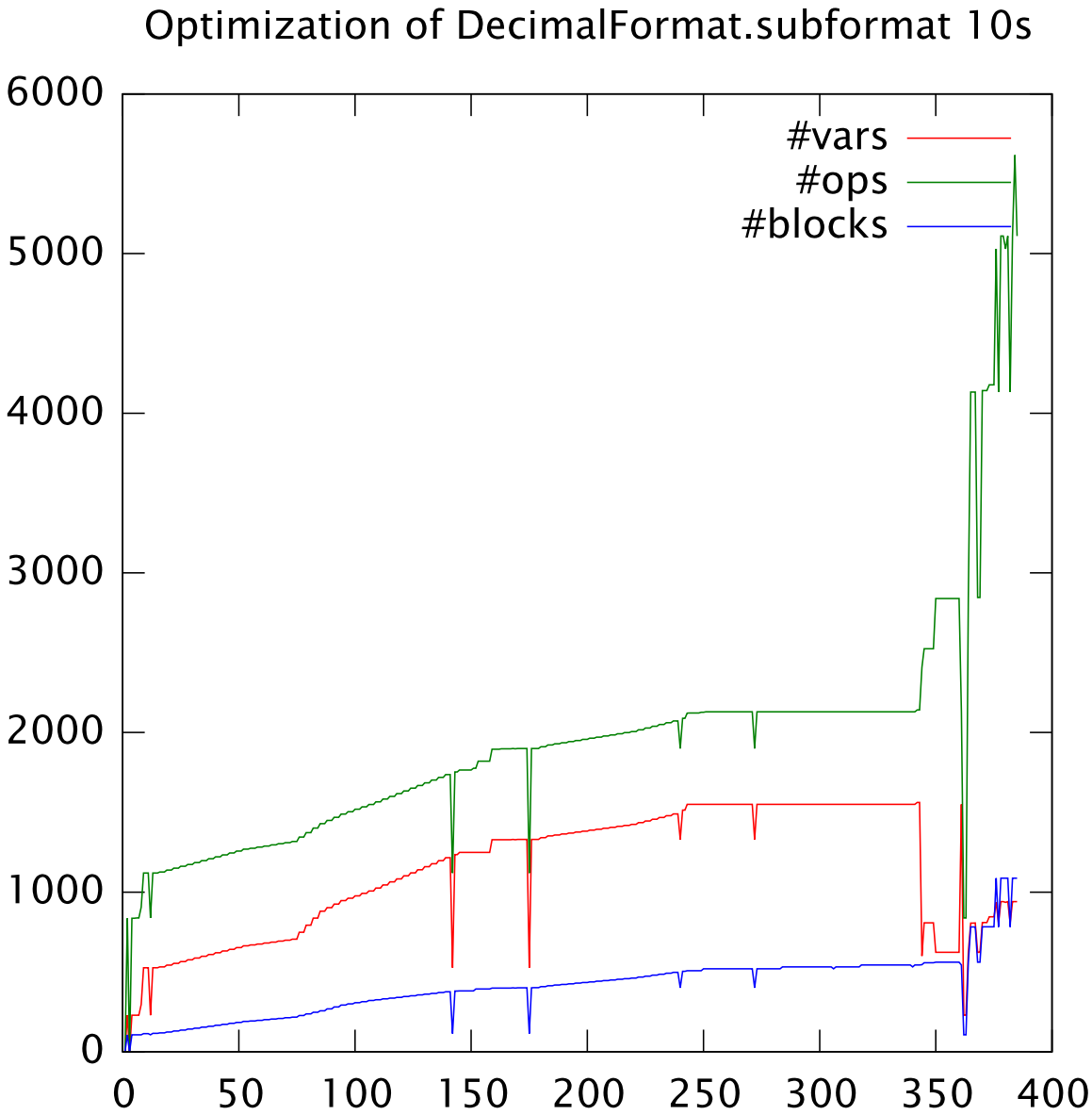
- Optimize 10000 random programs
- Original length 1000 bytecodes
- Add unnecessary ISTORE/ILOAD pairs

Anti-optimization

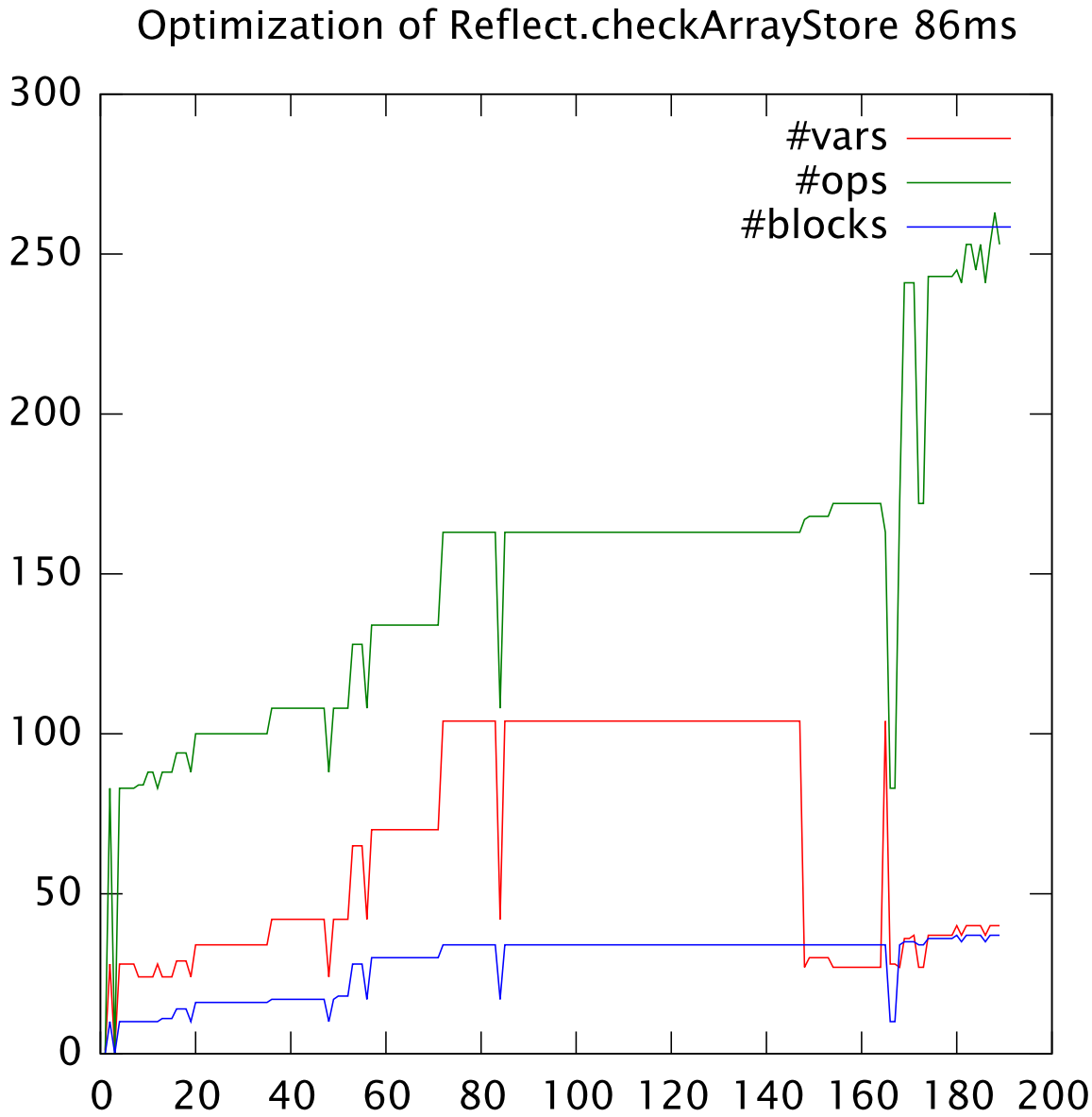
- Optimize 10000 random programs
- Original length 1000 bytecodes
- Add unnecessary ISTORE/ILOAD pairs



Number of AST ingredients



Number of AST ingredients



A DLR should not...

- A DLR should not do any of the optimzations JRockit does.

A DLR should not...

- A DLR should not do any of the optimzations JRockit does.
- JRockit has approx 100 optimization types.

A DLR should not...

- A DLR should not do any of the optimzations JRockit does.
- JRockit has approx 100 optimization types.
- All the standard compiler techniques and then some.

A DLR should not...

- A DLR should not do any of the optimzations JRockit does.
- JRockit has approx 100 optimization types.
- All the standard compiler techniques and then some.
- That leaves what?

A DLR should...

- Think of the bytecode as an AST.

A DLR should...

- Think of the bytecode as an AST.
- Assume by default, that local objects and variables are merely abstract representation of state that can be realized in any way the JVM feels is the most efficient.

A DLR should...

- Think of the bytecode as an AST.
- Assume by default, that local objects and variables are merely abstract representation of state that can be realized in any way the JVM feels is the most efficient.
- Use this assumption to reduce the dynamicness of the code.

Hard problem, Optimizing mathematical operations on dynamic Number objects

```
interface Num
{
    Num add(Num a);
    Num add(IntNum a);
    Num add(LongNum a);
    int intValue();
    long longValue();
}
```

The Integer Range Immutable Object

```
final public class IntNum implements Num
{
   final int i;
...
   public Num add(Num a) { return a.add(this); }
   public Num add(IntNum a) {
      int s = i+a.intValue();
      if (s >= -32768 && s <= 32767) {
         return new IntNum(s); }
      return new LongNum(s);
   }
...
}
```

The Long Range Immutable Object

```
final public class LongNum implements Num
{
   final long l;
...
   public int intValue() { return ((int)l)&0x7fff; }
   public Num add(Num a) { return a.add(this); }
   public Num add(IntNum a) {
      return new LongNum(l+a.intValue());
   }
...
}
```

A calculation

```
static int test(int i, int j)
{
    Num a = new IntNum(i);
    Num b = new IntNum(j);
    Num c = new IntNum(151);
    return a.add(b).add(c).intValue();
}
```

A calculation

```
static int test(int i, int j)
{
    Num a = new IntNum(i);
    Num b = new IntNum(j);
    Num c = new IntNum(151);
    return a.add(b).add(c).intValue();
}
```

It is difficult to optimize this code to make sure the integer path performs no memory allocations.

Help the JVM, create Large Immutable Object, that contribute AST and reduce dynamicness!

```
final public class FastNum implements Num
{
   final boolean isint; final int i; final long l;

   public FastNum(int x) {
      i = x; l = 0;
      isint = true;
  }

   public FastNum(long x) {
      l = x; i = 0;
      isint = false;
   }
...
```

```
public Num add(FastNum a) {
    if (isint && a.isint) {
        int s = i+a.intValue();
        if (s >= -32768 && s <= 32767) {
            return new FastNum(s);
        }
        return new FastNum((long)s);
    }
    return new FastNum(longValue()+a.longValue());
}
```

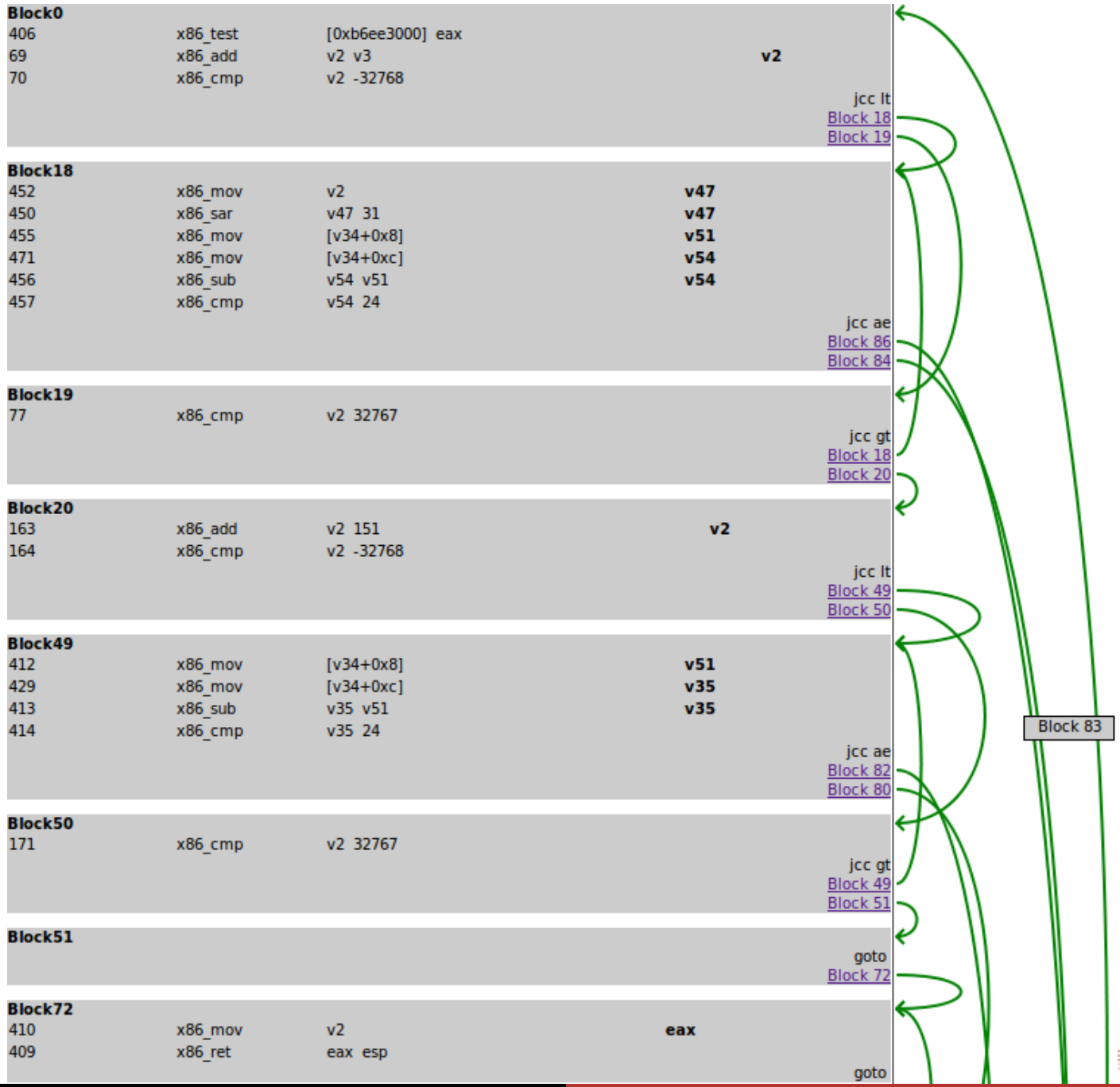
Inject this AST object into calculations that are hot!

```
static int test(int i, int j)
{
    Num a = new FastNum(i);
    Num b = new FastNum(j);
    Num c = new FastNum(151);
    return a.add(b).add(c).intValue();
}
```

The initial AST

Block0					
0		alloc		**v3**	FastNum
5	F	call	v3 v1		FastNum.<init>(I)V
6		mov	v3	**v4**	
7		alloc		**v5**	FastNum
12	F	call	v5 v2		FastNum.<init>(I)V
13		mov	v5	**v6**	
14		alloc		**v7**	FastNum
19	F	call	v7 151		FastNum.<init>(I)V
20		mov	v7	**v8**	
23	I	call	v4 v6	**v9**	Num.add(LNum;)LNum;
26	I	call	v9 v8	**v10**	Num.add(LNum;)LNum;
28	I	call	v10	**v11**	Num.intValue()I
30		return	v11		

goto


Block0
406 x86_test [0xb6ee3000] eax
69 x86_add v2 v3 v2
70 x86_cmp v2 -32768
jcc lt
Block 18
Block 19
Block18
452 x86_mov v2 v47
450 x86_sar v47 31 v47
455 x86_mov [v34+0x8] v51
471 x86_mov [v34+0xc] v54
456 x86_sub v54 v51 v54
457 x86_cmp v54 24
jcc ae
Block 86
Block 84
Block19
77 x86_cmp v2 32767
jcc gt
Block 18
Block 20
Block20
163 x86_add v2 151 v2
164 x86_cmp v2 -32768
jcc lt
Block 49
Block 50
Block49
412 x86_mov [v34+0x8] v51
429 x86_mov [v34+0xc] v35
413 x86_sub v35 v51 v35
414 x86_cmp v35 24
jcc ae
Block 82
Block 80
Block50
171 x86_cmp v2 32767
jcc gt
Block 49
Block 51
Block51
goto
Block 72
Block72
410 x86_mov v2 eax
409 x86_ret eax esp
goto
Block 83

Conclusions

- JVM uses standard optimizing techniques en masse.

Conclusions

- JVM uses standard optimizing techniques en masse.
- DLR reduce dynamicness to help the JVM, it does so by relying on the underlying JVM optimizations.

Conclusions

- JVM uses standard optimizing techniques en masse.
- DLR reduce dynamicness to help the JVM, it does so by relying on the underlying JVM optimizations.
- Create standardized performance tests and compete!