

bBerger Technologies, Inc.



Tuning Java Code

Robert Berger
bBerger Technologies, Inc.
<http://www.bberger.net>
bob@bberger.net

Performance Issues



- System Interactions
- Algorithms
- Code



Performance Issues

- System Interactions
 - Database
 - Other Server
 - Component Framework
 - Web Client Connection
 - User



Performance Issues

- System Interactions
 - Database
 - Query Optimization
 - Connection Pooling
 - Data Caching

Performance Issues



- System Interactions
 - Other Server
 - Connection Pooling
 - Data Caching

Performance Issues



- System Interactions
- Component Framework
 - EJB Container
 - Session Facade pattern to control transaction and remote method granularity
 - Bean instance pool sizes



Performance Issues

- System Interactions
 - Web Client Connection
 - DHTML to reduce page loads
 - User
 - Javascript to pre-validate data



Performance Issues

- Algorithms
 - Linear Search
 - Tree, Hash Table
 - `java.util` Collections
 - Application Specific Collections

Code Tuning



- Profile to find “Hot Spots”
- “90/10” Rule
 - 90% of time spent in 10% of code
 - Concentrate tuning on hot spots identified by the profiler.

Example Application



- Business Rules Engine
 - HaleyRules for Java Platform
 - Haley Systems, Inc.
 - <http://www.haley.com/>

Business Rules Engine



An application should be referred if the requested coverage is more than 80% of the applicant's income.

```
(defrule statement04
  (_is_ ?applicant1 applicant 083)
  (_is_ ?application2 application 081)
  (PersonIncome ?applicant1 ?Income3)
  (ApplicationRequestedCoverage ?application2 ?RequestedCoverage4
   &:(> RequestedCoverage4 (/ (* 80. ?Income3) 100.)))
=>
  (infer (ApplicationShouldBeReferred ?application2))
)
```

Business Rules Engine



- Performance goal: within 2x of existing C implementation
- Initial benchmark: > 10x slower than C
 - Profiler results: 80% of time spent in one area:
 - Object creation / collection

Minimizing Object Creation



- Reduce, Reuse, Recycle

Reduce



```
static String append(String[] array) {
    String result = "";
    for (int i = 0; i < array.length; i++) {
        result = result + array[i];
    }
    return result;
}
```

The above code creates at least `array.length * 3` objects. Each pass of the loop creates a `StringBuffer` (for the append operation), at least one array for the `StringBuffer`, and a `String` for the new value of `result`. We can do better with:

```
static String append(String[] array) {
    StringBuffer result = new StringBuffer();
    for (int i = 0; i < array.length; i++) {
        result.append(array[i]);
    }
    return result.toString();
}
```



Reuse

```
StringBuffer sb = new StringBuffer();  
    .  
    .  
    .  
sb.setLength(0);  
    .  
    .  
    .  
sb.setLength(0);
```



Reuse

```
double d;  
java.io.Writer writer;  
  
writer.write(Double.toString(d));
```

- Creates a new string every time
- Solution: reuse a `java.text.NumberFormat`

```
import java.io.*;
import java.text.*;
```

```
public class NumberWriter extends FilterWriter {
    private StringBuffer stringBuffer = new StringBuffer();
    private char[] chars = new char[64];
    private NumberFormat realFormat = NumberFormat.getInstance();
    private FieldPosition integerField =
        new FieldPosition(NumberFormat.INTEGER_FIELD);

    NumberWriter(Writer writer) {
        super(writer);
    }

    public void write(double value) throws IOException {
        stringBuffer.setLength(0);
        realFormat.format(value, stringBuffer, integerField);
        writeStringBuffer();
    }

    private void writeStringBuffer() throws IOException {
        int size = stringBuffer.length();
        stringBuffer.getChars(0, size, chars, 0);
        write(chars, 0, size);
    }
}
```





```
class Complex {
    double real, imaginary;

    Complex() {}

    Complex(double real, double imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }

    final static Complex add(Complex c1, Complex c2) {
        Complex result = new Complex();
        result.real = c1.real + c2.real;
        result.imaginary = c1.imaginary + c2.imaginary;
        return result;
    }
}

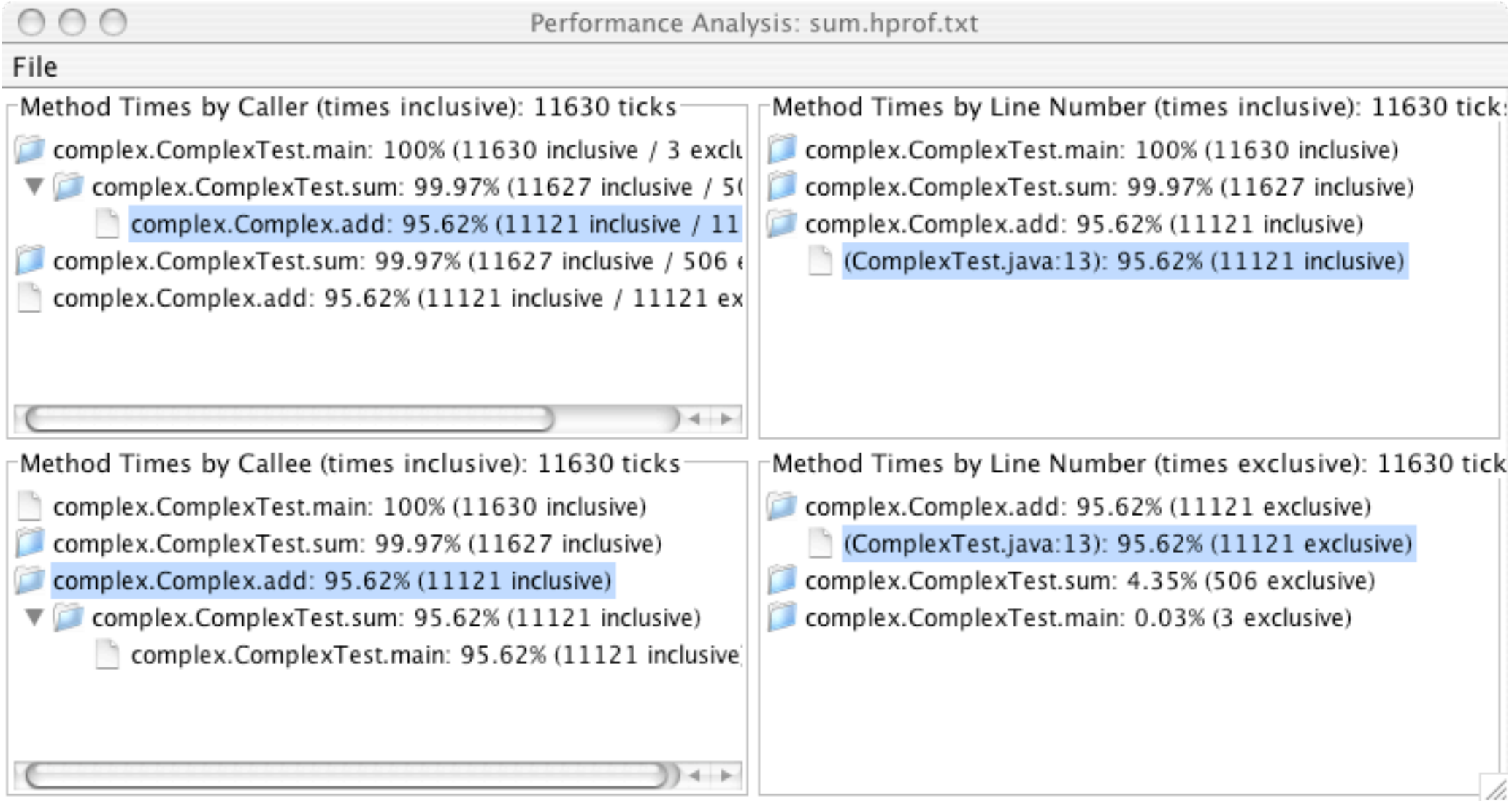
static Complex sum(Complex[] array) {
    Complex result = new Complex(0, 0);
    for (int i = 0; i < array.length; i++) {
        result = Complex.add(result, array[i]);
    }
    return result;
}
```

Sum of 1000 element array: 330 microseconds

Profiling



- `java -Xrunhprof:cpu=samples`
- PerfAnal
- *Java Programming on Linux* Nathan Meyers
- <http://javainlinux.rallylobster.com/CDROM/Chapter60/PerfAnal/>
- <http://java.sun.com/developer/technicalArticles/Programming/perfanal/>



```
final static Complex add(Complex c1, Complex c2) {
    Complex result = new Complex();
```

Reuse



```
final static void add(Complex c1, Complex c2, Complex result) {  
    result.real = c1.real + c2.real;  
    result.imaginary = c1.imaginary + c2.imaginary;  
}
```

```
static Complex sum1(Complex[] array) {  
    Complex result = new Complex(0, 0);  
    for (int i = 0; i < array.length; i++) {  
        Complex.add(result, array[i], result);  
    }  
    return result;  
}
```

Sum of 1000 element array: 54 microseconds

File

Method Times by Caller (times inclusive): 1525 ticks

complex.ComplexTest.main: 99.8% (1522 inclusive / 0 exclusive)
complex.ComplexTest.sum1: 99.8% (1522 inclusive / 1522 exclusive)
java.lang.ClassLoader.loadClass: 0.13% (2 inclusive / 1 exclusive)
java.util.zip.ZipFile.getInflater: 0.07% (1 inclusive / 0 exclusive)
java.util.zip.ZipFile.getInputStream: 0.07% (1 inclusive / 0 exclusive)
java.lang.ClassLoader.loadClassInternal: 0.07% (1 inclusive / 0 exclusive)
java.util.zip.Inflater.<clinit>: 0.07% (1 inclusive / 1 exclusive)
java.net.URLClassLoader.findClass: 0.07% (1 inclusive / 0 exclusive)
java.security.AccessController.doPrivileged: 0.07% (1 inclusive / 0 exclusive)

Method Times by Line Number (times inclusive): 1525 ticks

complex.ComplexTest.main: 99.8% (1522 inclusive)
complex.ComplexTest.sum1: 99.8% (1522 inclusive)
java.lang.ClassLoader.loadClass: 0.13% (2 inclusive)
java.util.zip.ZipFile.getInflater: 0.07% (1 inclusive)
java.util.zip.ZipFile.getInputStream: 0.07% (1 inclusive)
java.lang.ClassLoader.loadClassInternal: 0.07% (1 inclusive)
java.util.zip.Inflater.<clinit>: 0.07% (1 inclusive)
java.net.URLClassLoader.findClass: 0.07% (1 inclusive)
java.security.AccessController.doPrivileged: 0.07% (1 inclusive)

Method Times by Callee (times inclusive): 1525 ticks

complex.ComplexTest.main: 99.8% (1522 inclusive)
complex.ComplexTest.sum1: 99.8% (1522 inclusive)
java.lang.ClassLoader.loadClass: 0.13% (2 inclusive)
java.util.zip.ZipFile.getInflater: 0.07% (1 inclusive)
java.util.zip.ZipFile.getInputStream: 0.07% (1 inclusive)
java.lang.ClassLoader.loadClassInternal: 0.07% (1 inclusive)
java.util.zip.Inflater.<clinit>: 0.07% (1 inclusive)
java.net.URLClassLoader.findClass: 0.07% (1 inclusive)
java.security.AccessController.doPrivileged: 0.07% (1 inclusive)

Method Times by Line Number (times exclusive): 1525 ticks

complex.ComplexTest.sum1: 99.8% (1522 exclusive)
(ComplexTest.java:40): 91.61% (1397 exclusive)
(ComplexTest.java:39): 8.2% (125 exclusive)
java.lang.ClassLoader.loadClass: 0.07% (1 exclusive)
java.util.zip.Inflater.<clinit>: 0.07% (1 exclusive)
java.security.AccessController.doPrivileged: 0.07% (1 exclusive)
complex.ComplexTest.main: 0% (0 exclusive)
java.util.zip.ZipFile.getInflater: 0% (0 exclusive)
java.util.zip.ZipFile.getInputStream: 0% (0 exclusive)

```
for (int i = 0; i < array.length; i++) {
```

Cache Static and Instance Variables



```
for (int i = 0; i < array.length; i++) {  
  
int length = array.length;  
for (int i = 0; i < length; i++) {
```

Sum of 1000 element array: 29 microseconds

The Java language specification allows the compiler/JVM to perform this type of optimization automatically.

However, as of JDK 1.4.2, Sun's implementation does not perform this optimization.

Caching values in local variables can therefore produce a significant improvement (almost a factor of 2 in this example).

Reuse



- *Value Object*: a small, preferably immutable object whose equality depends solely on the contained value.
- `java.lang.String`
- `java.lang.Integer`

Note: some J2EE practitioners use *Value Object* as the term for what other OO designers were already calling *Data Transfer Object*.

See *Patterns of Enterprise Application Architecture*, Martin Fowler, page 486.

Reuse



- Flyweight Pattern
 - Use singleton instances of commonly used values.
 - *Design Patterns*, Gamma, Helm, Johnson, Vlissides

Flyweight Pattern



- `java.lang.Boolean`
- `Boolean.TRUE`, `Boolean.FALSE`

```
boolean b;
```

```
new Boolean(b)
```

```
b ? Boolean.TRUE : Boolean.FALSE
```

Flyweight Pattern



```
public class IntegerFactory {
    private static final int MIN_CACHED = -10;
    private static final int MAX_CACHED = 100;
    private static Integer cached[];

    static {
        cached = new Integer[MAX_CACHED-MIN_CACHED+1];
        for (int i = MIN_CACHED; i <= MAX_CACHED; i++) {
            cached[i-MIN_CACHED] = new Integer(i);
        }
    }

    public static Integer getInteger(int i) {
        if (i >= MIN_CACHED && i <= MAX_CACHED) {
            return cached[i-MIN_CACHED];
        } else {
            return new Integer(i);
        }
    }
}
```



Flyweight Pattern

- Java 5.0 added static `valueOf` methods for classes in `java.lang` that wrap primitive types.
- `valueOf` methods are used for automatic “boxing” of primitive types when passed to methods that expect objects.
- `Boolean`, `Char`, `Byte`, `Short`, `Int`, and `Long` use the flyweight pattern in `valueOf`.

```
public static Integer valueOf(int i) {
    final int offset = 128;
    if (i >= -128 && i <= 127) { // must cache
        return IntegerCache.cache[i + offset];
    }
    return new Integer(i);
}
```

Recycle



```
class ObjectPool {
    private Class objectClass; // Class of objects managed by this factory
    PooledObject freeObjects; // List of previously used objects

    public ObjectPool(Class objectClass) {this.objectClass = objectClass; }

    public PooledObject get() throws InstantiationException, IllegalAccessException {
        PooledObject pooledObject = null;

        if (freeObjects == null) {
            pooledObject // Allocate a new object
                = (PooledObject) objectClass.newInstance();
        } else {
            pooledObject = freeObjects; // Get an existing object from the free list
            freeObjects = freeObjects.nextFree;
        }
        pooledObject.factory = this; pooledObject.nextFree = null;
        return pooledObject;
    }
}

abstract class PooledObject {
    PooledObject nextFree; // Next object on free list
    ObjectPool factory;

    void free() {
        nextFree = factory.freeList; // Add object to free list
        factory.freeList = this;
    }
}
```

Recycle



- Issues with object pooling
 - Increased memory usage
 - Lack of automatic object initialization
 - Threading issues
 - Per-thread object pools minimize synchronization overhead at the cost of increased memory usage.



Reflection

- Used to call methods of classes whose definition is not available when calling code is compiled
- Used to implement languages, extensible systems
- Two steps:
 - Use class and method names to lookup class and method at runtime
 - Call method using `java.lang.reflect.Method.invoke()`



Reflection

```
static void test() {}
```

2.7 GHz Intel Celeron

All times are in nanoseconds

	Windows XP		Suse Linux 9.1	
	1.3.1	1.4.2	1.3.1	1.4.2
Direct call	7	6	6	6
Cached java.util.Method	814	821	1728	1133
Lookup every call	13377	5940	61510	14729

Reflection Recommendations



- Avoid reflection if possible
 - Define interfaces for extension classes
 - Automatic code generation
- Do lookups once and cache `java.lang.reflect.Method` object
- Use latest JVM
- Lobby Sun to improve JVM performance on Linux



Exceptions

```
try {  
    throw new Exception("An error occurred");  
} catch (Exception e) {}
```

12.8 microseconds

```
try {  
    throw cachedException;  
} catch (Exception e) {}
```

1.2 microseconds

- Disadvantage: inaccurate output from `Exception.printStackTrace()`

Casts



- Minimize casts.
 - Cast once and store in local variable.
- Cost depends on depth of type hierarchy.
- Interfaces are more expensive.
- Recent JVM's are better.
- Don't change design for minor performance gains.
- Java 5.0 generics eliminate casts from source code, but not from generated bytecode.

Cache Frequently Computed Values



```
public final class String {
    private int hash = 0;

    public int hashCode() {
        int h = hash;
        if (h == 0) {
            .
            .
            .
            hash = h;
        }
        return h;
    }
}
```

Quotes



Donald Knuth

“We should forget about small efficiencies, say about 97% of the time. Premature optimization is the root of all evil.”

M. A. Jackson

Rules of Optimization:

Rule 1:

Don't do it.

Rule 2: (for experts only)

Don't do it yet.

William A. Wulf

“More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.”

Jon Bentley

“On the other hand, we cannot ignore efficiency.”



Resources

Java Performance Tuning

Jack Shirazi

<http://www.oreilly.com/catalog/javapt2/>

Effective Java Programming Language Guide

Joshua Bloch

<http://java.sun.com/docs/books/effective/>

Better, Faster, Lighter Java

Bruce A. Tate, Justin Gehtland

<http://www.oreilly.com/catalog/bfljava/>