United Business Media

APPLICATION LIFECYCLE MANAGEMENT

Dr. Dobb's JOURNAL

www.ddj.com The World of Software Development

ISSUE NO. 412
September 2008

Erik Demaine on GEOMETRIC ALGORITHMS: COPY, FOLD, & CUT

ALM Meets Model-Driven Development

Matching Wildcards: An Elegant Algorithm

The Android Mobile Phone Platform

Managing Application Thread Use

Signalling Integer Overflows in Java

.NET Development & the IBM WebSphere Portal Server

Scott W. Ambler: The Agile Edge

Herb Sutter: Effective Concurrency

by Frederic Bapst and François Kilchoer

Signalling Integer Overflows in Java

A tool for checking overflows in Java code

Frederic and François *are members of the computer science department at the University of Applied Sciences of Western Switzerland. They can be contacted at francois.kilchoer@hefr.ch and frederic.bapst @hefr.ch, respectively.*

When computer scientists manipulate numbers, they know that from a mathematical point of view their numbers sometimes behave strangely. For programmers, it's no surprise that a floating-point number may not vary when it is increased by 1.0, or that adding two positive integers can lead to a negative result. The latter phenomenon is known as "integer overflow"—the fixed-length representation of signed integers (that is, 32 bits) leads to a model of so-called "circular arithmetic", in which the greatest positive integer is followed by the smallest negative integer.

Most CPUs usually maintain an "overflow flag" that can be used to detect this anomaly. On the other hand, most popular programming languages (C++, Java...) define arithmetic operations so that an overflow is never signalled. That's why integer overflow is known to be an insidious source of bugs. Of course, overflow is, in fact, an announced feature, and can be used deliberately to obtain interesting results (for instance, to perform a modulo operation inside a random-number generator). But most of the time, when programmers use integers, they would prefer not to be threatened by overflows. If nothing else, it would be useful if overflows could automatically be reported during development and testing.

Instrumentation

To detect an overflow at runtime, one way is to replace every arithmetic operator by a call to a method that performs the equivalent operation, but also checks whether the result suffers from overflow. This is illustrated in Example 1 for the addition of *int*s. This kind of transformation could be automated at the source-code level, but then you need a full parser for Java. In fact, it is easier to apply the replacement after compilation at the bytecode level, which has a much simpler structure, where every integer operation directly corresponds to a particular bytecode instruction.

Understanding the instrumentation requires an informal description of the Java Virtual Machine (java.sun.com/docs/books/jvms). Local data and partial results are stored in a frame. Every time a method is invoked, a new frame is created, and destroyed upon method exit (whether normal or not). Frames are allocated on the stack; for our purposes, each frame has its own array of local variables and an operand stack. The array of local variables is organized as 32-bit variables, regardless of the type of the variables.

The array of local variables is determined at compile time. The type of a local variable is one of *boolean, byte, char, short, int, float, reference* (arrays and objects), or *returnAddress*; for *long*s and *double*s, a pair of consecutive local variables are required.

The size of the operand stack is also determined at compile time, and this size is included in the code associated with the method. When the frame for the method is created, the stack is initially empty. The JVM provides instructions to load values from local variables or constants into the stack; other instructions take operands from the operand stack, perform the required operation, and push the results back on the stack.

For example, the *iadd* instruction adds two integer values. It requires that two *int* values be present on the stack, pushed there by other instructions. Both values are popped off the stack, added, and the sum is

pushed back on the stack. A class verifier ensures that attempts to operate on values inappropriate for the instruction are caught. For example, it is illegal to load a *long* on the stack and attempt to use it as two *int*s.

As the term "bytecode" suggests, the JVM model defines operation codes (or opcodes) as single bytes, thus limiting the number of instructions the virtual machine can provide. The operations that are provided are not, therefore, orthogonal: there is no *add* instruction for shorts or bytes. Instead, to perform addition on bytes or shorts, the value must first be converted to *int*, and only then can the addition be performed. Converting shorts or bytes to integer "widens" the integer without loss of data.

The JVM does not indicate overflow of integer operations (the only exception that JVM generates as a result of integer arithmetic operations is *ArithmeticException* when dividing by zero). So undetected overflow can occur in several places:

- The arithmetic instructions *add, sub, mul, div,* and *neg* (the last two for the special cases of dividing *MIN_VALUE* by -1 or negating *MIN_VALUE*, respectively).
- During narrowing operations from *int* to short or byte.
- During increment operations.

Instrumenting the code therefore involves adding code to the following instructions:

- Arithmetic instructions. *iadd, isub, imul, idiv, ineg, ladd, lsub, lmul, ldiv, lneg, iinc.*
- Narrowing instructions. *i2b, i2s, l2i.*

We consciously decided that left shifting would be left alone—we think that you are more likely to be aware of possible overflows in this case.

Using the ASM Library

While it is possible to process bytecode files with no other support than reading/writing bytes, there are well-defined libraries that considerably simplify the work, by managing the details and bookkeeping, such as computing the maximal size of the stack operand, the index of a literal in the constant pool, or the precise offsets when inserting jump operations. ASM (asm.objectweb.org) is such a library, particularly small yet powerful and efficient, making heavy use of design patterns. ASM offers methods to process bytecode streams, either in the form of an editable structure holding the entire code (analogous to a DOM-like parser for XML files), or through an event model (a SAX-like parser).

The event model makes filtering easy; for instance, to replace every *iadd* by a method call. The library also offers a program named ASMifier, which takes an existing *.class file and generates the Java+ASM source code that generates the same bytecode sequence at runtime. Integrated in an Eclipse plug-in, this tool quickly becomes a must-have.

Example 2 is a fragment of Java source code, and the corresponding bytecode instructions, first in an abstracted syntax, then as an output of ASMifier.

Example 3 illustrates how to write a program that replaces every *iadd* by a call to the presumably existing method used in Example 1.

The ASM library makes it easy to write a bytecode instrumentation program, but sketching the details of overflow management requires a careful analysis. For instance, you might not at first notice that unary integer negation or integer division are "dangerous" operations, or that incrementation has its own bytecode instruction. When playing with the operand stack instructions, it appears that some manipulations are possible on *int* values but not on *long* values, because the latter occupy two slots in the stack. The identifiers of added methods must be unique, even when processing an already instrumented class.

COJAC Tool

With the help of two students, we developed COJAC (short for "Checking Overflows in JAva Code"), a freely available tool (http://home.hefr.ch/bapst/cojac) that instruments any existing Java bytecode for overflow detection. It takes as input any *.class file, and outputs an instrumented version that will detect overflows and signal them at runtime. Here are some features of COJAC:

- COJAC is a portable Java program with a command-line interface.
- Instrumentation is applied on individual classes or entire *.jar archives.
- Any type of arithmetic integer overflow can be processed.
- COJAC can also check when narrowing type casting (*int2short*, for example) leads to a data loss (such as when the original value is out of target type bound).
- The options offer a fine control over the particular bytecode instructions to instrument (six operations on *int*s, five on *long*s, four kinds of typecasting).

(a)
```
int a, b, r;
 ...
r = a + b;
 ...
```

(b)
```
int a, b, r;
 ...
r = checkedIADD(a,b);
 ...
```

(c)
```
package utils;
public class SecuredArithmetics {
  static int checkedIADD(int a, int b) {
    int r=a+b;
    if (( (a^r) & (b^r) ) < 0)
      System.err.println("Overflow!");
    return r;
  }
}
// (a^r)&(b^r))<0) is a shorthand for:
// (a<0 && b<0 && x>=0) ||
// (a>0 && b>0 && x<=0) )
```

***Example 1:** Adding overflow checking: (a) before, (b) after instrumentation, (c) possible detection algorithm.*

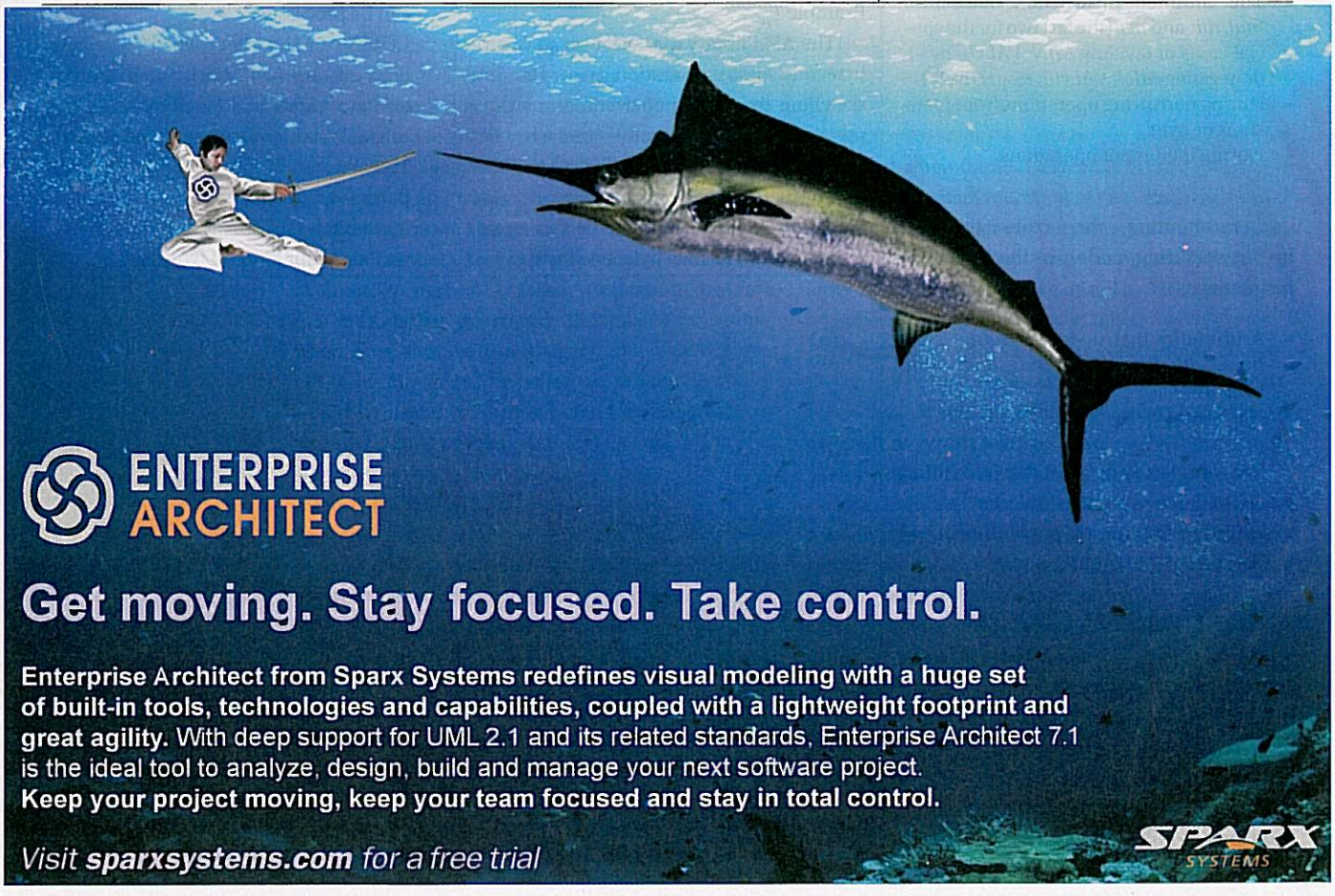
What About C/C++ ?

Java and C/C++ differ in their models of arithmetic expression evaluation, but the problem of arithmetic overflow is absolutely similar. Some C/C++ compilers offer an option to generate code that will check integer operations and report overflows. For instance, GCC comes with the *-ftrapv* flag, but it used to be implemented in a way that does not cover every situation: for instance, it doesn't signal overflows on shorts, and it is known to miss some overflows on *int* multiplications. Maybe this is why it doesn't seem to be popular. We found another attempt to provide instrumentation in GCC as a separate module using GEM, but the information is a bit laconic about the current state of this project (www.ecsl.cs.sunysb.edu/iop/index.html).

- COJAC lets you choose between four reporting policies: printing a message on the console (*stderr*), showing the location in the source code (with or without a detailed stacktrace); writing a similar message to a log file; throwing an *ArithmeticException*; and applying a user-defined reaction (the user indicates the name of an existing static method that will be used as an error reporting function (Example 4).
- It is possible to instrument only a single selected method, instead of the entire class.

The instrumentation of course introduces overhead, both in speed and space (bytecode size). Consider first how many bytes are added to the bytecode file. Internally, we do not add any new classes, but in any instrumented class we do add:

- Instructions in existing methods (a method call instead of an *iadd*).
- Some items in the constant pool.



- New methods, which are used for overflow reporting (unless a callback is given) and for overflow detection (unless the user chooses to inline the tests, which makes the code faster and often bigger).

For each instrumented class, the corresponding memory footprint of these additions typically amounts to a constant of less than 5KB, plus a couple of bytes for each arithmetic operation. To give an idea, when activating every possible check, the Java2Demo.jar program grows from 400KB to 625KB.

It is a bit harder to predict the slowdown incurred by COJAC, because it depends on factors like the hardware and OS, the JVM version and parameters (with/without JIT), or the proportion of integer arithmetic operations in the instrumented code. We measured the slowdown on a reasonable configuration for the four operators on *int*s. The overhead per arithmetic operation ranges from 20 percent (for *idiv*), to 950 percent (for *imul*). When inlining the detection method, this overhead is reduced and reaches between 10 percent (for *idiv*) and 580 percent (for *iadd*). As a typical Java program does other things than integer operations, we can expect that the runtime will roughly grow by a factor of two (as we observed with sorting algorithms). This is an important overhead, but similar to other means used for discovering bugs (think about using memory checkers like Valgrind in C/C++).

Perspectives

Using the ASM library, COJAC was not really hard to implement. It nevertheless opens some interesting perspectives:

- It is straightforward to implement a new classloader that performs on-the-fly instrumentation.
- Java is not the only programming language that produces code for the JVM. Thus, COJAC is expected to be suitable for other languages equipped with a bytecode translator, like Python or Ruby.
- COJAC could be packaged as an Eclipse plug-in that automatically performs instrumentation on entire projects. Similarly, it would of course be possible to integrate the functionality as a Java compiler option (maybe considered in a next release of Java?).
- Floating-point operations do not suffer from overflow, but also have pitfalls; sometimes it would be useful to discover that a number did not change after the addition of a nonzero term or the product with a nonone factor. It would be easy to extend COJAC to report such situations.

COJAC is designed to be simple and robust, and this causes some inherent limitations:

- Compile-time evaluated expressions are not detected. The compiler evaluates expressions such as *(3*Integer.MAX_VALUE)*, and only the result is stored in the bytecode.
- The instrumentation could potentially be confusing for the debugger and other tools that exploit the correspondence between source code and bytecode (e.g. a profiler).
- Because we didn't want to add a new class in the instrumented code, the

(a)

```
public static double arraySum(double [] t) {
  double sum=0.0;
   for(int i=0; i<t.length; i++)
    sum+=t[i];
   return sum;
}
```

(b)

```
public static arraySum(double[]):double
L0 (0) LINENUMBER 11 L0
        DCONST_0
        DSTORE 1: sum
L1 (3) LINENUMBER 12 L1
        ICONST_0
        ISTORE 3: i
L2 (6) GOTO L3
L4 (8) LINENUMBER 13 L4
        DLOAD 1: sum
        ALOAD 0: t
        ILOAD 3: i
        DALOAD
        DADD
        DSTORE 1: sum
L5 (15) LINENUMBER 12 L5
        IINC 3: i 1
L3 (17) ILOAD 3: i
        ALOAD 0: t
        ARRAYLENGTH
        IF_ICMPLT L4
L6 (22) LINENUMBER 14 L6
        DLOAD 1: sum
        DRETURN
L7 (25) LOCALVARIABLE t double[] L0 L7 0
        LOCALVARIABLE sum double L1 L7 1
        LOCALVARIABLE i int L2 L6 3
        MAXSTACK = 4
        MAXLOCALS = 4
```

(c)

```
MethodVisitor mv;
mv = cw.visitMethod(ACC_PUBLIC + ACC_STATIC,
  "arraySum", "([D)D", null, null);
mv.visitCode();
  Label l0=new Label(), l1=new Label(), l2=new Label(),...;
mv.visitLabel(l0); mv.visitLineNumber(11, l0);
mv.visitInsn(DCONST_0);
mv.visitVarInsn(DSTORE, 1);
mv.visitLabel(l1); mv.visitLineNumber(12, l1);
mv.visitInsn(ICONST_0);
mv.visitVarInsn(ISTORE, 3);
mv.visitJumpInsn(GOTO, l3);
mv.visitLabel(l4); mv.visitLineNumber(13, l4);
mv.visitVarInsn(DLOAD, 1);
mv.visitVarInsn(ALOAD, 0);
mv.visitVarInsn(ILOAD, 3);
mv.visitInsn(DALOAD);
mv.visitInsn(DADD);
mv.visitVarInsn(DSTORE, 1);
mv.visitLabel(l5); mv.visitLineNumber(12, l5);
mv.visitIincInsn(3, 1);
mv.visitLabel(l3);
mv.visitVarInsn(ILOAD, 3);
mv.visitVarInsn(ALOAD, 0);
mv.visitInsn(ARRAYLENGTH);
mv.visitJumpInsn(IF_ICMPLT, l4);
mv.visitLabel(l6); mv.visitLineNumber(14, l6);
mv.visitVarInsn(DLOAD, 1);
mv.visitInsn(DRETURN);
mv.visitLabel(l7);
mv.visitLocalVariable("t", "[D", null, l0, l7, 0);
mv.visitLocalVariable("sum", "D", null, l1, l7, 1);
mv.visitLocalVariable("i", "I", null, l2, l6, 3);
mv.visitMaxs(4, 4);
mv.visitEnd();
```

Example 2: (a) Source code, (b) bytecode, (c) ASMifier output.

(a)

```
import org.objectweb.asm.*;
import java.io.*;
public class MyInstrumentation {
  public static void main(String[] args) throws IOException {
    String filename = args[0];
    FileInputStream fis = new FileInputStream(filename);
    ClassReader cr = new ClassReader(fis);
    ClassWriter cw = new ClassWriter(cr, ClassWriter.COMPUTE_FRAMES);
    ClassAdapter ca = new MyClassAdapter(cw);
    cr.accept(ca, 0);
    byte[] newByteCode = cw.toByteArray();
    fis.close();
    FileOutputStream fos = new FileOutputStream(filename);
    fos.write(newByteCode);
    fos.close();
  }
}
//--------------------
class MyMethodAdapter extends MethodAdapter implements Opcodes {
  public MyMethodAdapter(MethodVisitor mv) { super(mv); }
   //---------------------------
   public void visitInsn(int opcode) {
     final String METHOD_NAME= "checkedIADD";
     final String METHOD_LOCATION="utils/SecuredArithmetics";
     final String METHOD_SIGNATURE="(II)I";
     if (opcode == IADD) {
       mv.visitMethodInsn(INVOKESTATIC, METHOD_LOCATION,
                 METHOD_NAME, METHOD_SIGNATURE);
     } else {
       mv.visitInsn(opcode);
     }
  }
}
//--------------------
class MyClassAdapter extends ClassAdapter {
   public MyClassAdapter(ClassVisitor cv) { super(cv); }
   //---------------------------
   public MethodVisitor visitMethod(int access, String name,
          String desc, String signature, String[] exceptions) {
     MethodVisitor mv;
     mv = cv.visitMethod(access, name, desc, signature, exceptions);
     if (mv != null) {
       mv = new MyMethodAdapter(mv);
     }
     return mv;
  }
}
```

(b)

```
public class Hello {
  public static void main(String[] args) {
    int a=3, b=5, c=Integer.MAX_VALUE;
    System.out.println(a+c);
    System.out.println(a+b);
  }
}
```

(c)

```
prompt> javac Hello.java
prompt> java Hello
  -2147483646
  8
prompt> java MyInstrumentation "Hello.class"
prompt> java Hello
  Overflow!
  -2147483646
  8
```

***Example 3:** Simple instrumentation example: (a) instrumentation program, (b) instrumented test program, (c) commands and result.*

(a)

```
import java.util.HashSet;
package logging;
...
  }
}
```

(b)

```
prompt> java -jar cojac.jar -call logging/BetterLog/log Java2Demo.jar
```

***Example 4:** Using a reaction callback: (a) code, (b) command.*

default logging will repeatedly report the same location every time it causes an overflow. It is straightforward to define a better logging callback that filters already-seen locations, as in Example 4.

Our approach of bytecode instrumentation is not the only way of attacking the overflow problem. An idea to reduce the slowdown would be a probabilistic technique that decides at runtime whether an operation has to be checked or not. An alternative would be to write nonportable C code to consult the CPU overflow flag, and bind this code with JNI. Another interesting approach is static analysis: We could, for instance, extend JML tools (like ESCJava) to report at compile-time some of the potential overflows, as is already done for bad array indices. By the way, it can be argued that reporting overflows is a poor goal and that we should aim at completely suppressing the overflow risk, for example, automatically converting *int/long* to *BigInteger* objects; this is not trivial because it would cause deep transformations in the code (e.g. converting arrays to *ArrayLists*).

Finally, a totally different approach would be to rely on virtualization: Is it difficult to adapt QEmu or Xen, for example, so that the virtual machine signals to the host OS any overflow occurring in any running process of the guest OS?

Conclusion

Arithmetic overflow is an annoying feature of most programming languages: On rare occasions, it is exploited by the programmers as a computation property; but most of the time, it is simply a nasty source of bugs. Any attempt to help discovering bugs and deliver robust code is worth trying. That is why Java developers should keep an eye on recent developments. Just to arbitrarily name a few free new tools, look to Eclipse TPTP, JML verifiers (www.cs.ucf.edu/~leavens/JML), delta debugging support (www.st.cs.uni-sb.de/eclipse), the ODB "omniscient debugger" (www.lambdacs.com/debugger)—or COJAC. DDJ