Analysis of
“Proposal for Extension to Java Floating Point Semantics, Revision 1”

by
Joseph D. Darcy
and
William Kahan

Computer Science Division,
University of California, Berkeley

email: {darcy, wkahan}@CS.Berkeley.EDU
WWW: http://www.cs.berkeley.edu/~{darcy, wkahan}

Summary of PEJFPS

- PEJFPS has two goals:
  - allow some access to extended precision where it is supported by hardware
  - ameliorate Java's floating point performance implications on the x86 (double rounding on underflow problem)
- These goals are accomplished by allowing (in certain contexts) `float` and `double` local variables and parameters to be stored as and operated on as extended format floating point values.
- New class and method qualifiers `widefp` and `strictfp` control contexts where extended formats can and cannot be used.
- Existing Java source code and `class` files are subject to the revised semantics (the default is implicit `widefp` instead of implicit `strictfp`).
- Only operand stack, local variables, parameters, and method return values can use extended formats. There are no extended format arrays or object fields.
- The virtual machine is granted wide latitude in choosing when and if to use extended formats.

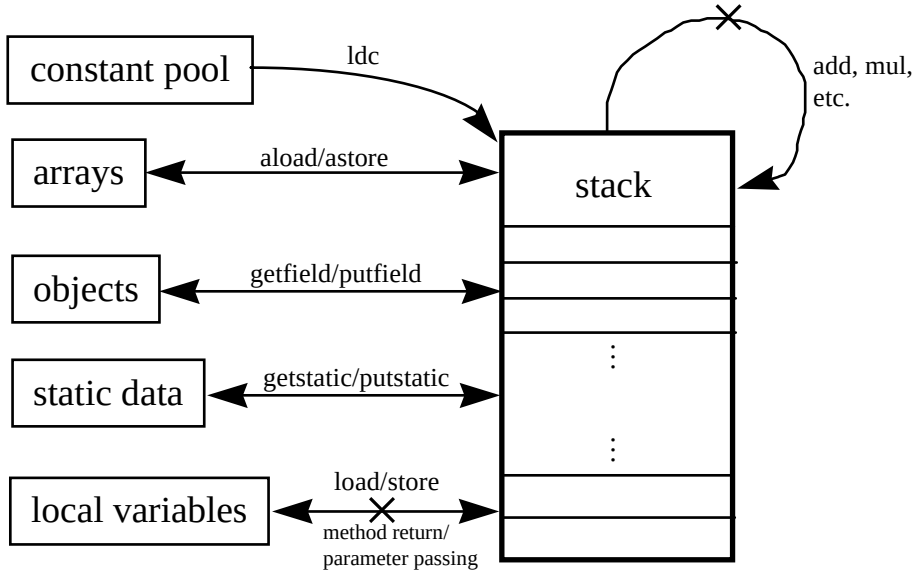
Background on the JVM

Where do floating point values in the JVM come from?

- × returning a value from a method
- × passing a parameter
- × generated by a floating point operation
- × local variables
- • arrays (allocated on the heap)
- • fields in objects (allocated on the heap)
- • static class fields
- • constants in the constant pool

Data movement in the JVM

- All data movement must go through the stack
- No direct path from, say, an array element to an object field



A change in philosophy

Java allows application developers to write a program once and then be able to run it everywhere on the Internet.

Except for timing dependencies or other non-determinisms and given sufficient time and sufficient memory space, a Java program should compute the same result on all machines and in all implementations.

—Preface to The Java™ Language Specification

- For both intrinsic and practical reasons, Java code does not live up to it "write once, run anywhere" slogan
- But, Java is much more *predictable* than other contemporary languages. The sizes of the types are given, expression evaluation order is specified, etc.
- PEJFPS removes Java's predictability for floating point

Compiler Latitude

Under PEJFPS, the compiler can decide to use or not use extended precision at its discretion. From PEJFPS,

Section 5.1.8, Format Conversion
Within an FP-wide expression, format conversion *allows an implementation, at its option, to perform any of the following operations on a value:*

- float → float extended ***and*** float extended → float
- double → double extended ***and*** double extended → double

Conclusions:

- extended formats can be used *inconsistently* at the compiler's whim
- `fp`, `femax`, and `femin` (PEJFPS §4.2.3) can *misleadingly* indicate extended formats are in use when in actuality they are not

A Legal Perversion

Can the multiply overflow?

```
static double mul(float f1, float f2)
{ double d1, answer;
  d1 = f1;
  answer = d1 * f2;
  return answer;
}
```

Can the assignment overflow?

Even FORTRAN 77 and C guarantee
width(`float`) ≤ width(`double`).
PEJFPS does not.

Potential Surprises

When will extended precision by used?

```
a[] = {Double.MAX_FLOAT, -Double.MAX_FLOAT, 1.0}
b[] = {Double.MAX_FLOAT,  Double.MAX_FLOAT, 1.0}

widefp static double dot(double a[], double b[]){
  double sum;
  for(int i = 0; i <= a.length; i++)
      sum += a[i] * b[i];

  return sum;
}
```

- When run on an x86, when will `dot` return a NaN (strict floating point) and when will `dot` return 1.0 (wide floating point)?

VM = interpreter + JIT

```
// arrays a and b and method dot from previous slide

widefp static double dot(double a[], double b[])
{...}

public static void main(String[] args)
{
  double a[] = {Double.MAX_FLOAT, -Double.MAX_FLOAT, 1.0};
  double b[] = {Double.MAX_FLOAT,  Double.MAX_FLOAT, 1.0};

  for(int i = 0; i < 10; i++)
    System.out.println(dot(a, b));
  Other code...
  System.out.println(dot(a, b));
}
```

- Why do the first 5 calls to `dot` print NaN and the next 5 print 1.0?
- Why does the last call to `dot` print NaN?

A problem for compiler writers?

Want to compile the following to native code:

```
class A {
  strict fp double foo(){...}
}

class B extends class A {
  widefp double foo(){...}
}

static double bar() {
  A x = (random?new B():
                new A());
  return x.foo();  //does strict or wide foo get called?
}
```

- Floating point format returned by `x.foo` might be unknowable at compile time
- This is tolerable on the x86 due to the register architecture
- Problematic on machines with orthogonal support for 3 floating point formats
- Solution: promote all stack values to the same format

Do cry over spilt registers

Will breaking an expression into pieces change the value computed?

```
// widefp context
double a, t1, t2;
a = BigExpression1 * BigExpression2;
foo(); // foo could modify global vars, but doesn't
t1 = BigExpression1;
t2 = BigExpression2;
if(a == t1 *t2)
  ...
```

- faster to register spill 64 bit `double` values instead of 80 bit `double extended` values is (lower latency instruction and less memory traffic)
- breaks referential transparency

Using extended precision arrays

- Eigenproblem refinement requires a residual

$$\left[R\right] = \left[\quad A \quad\right] \cdot \left[X\right] - \left[X\right] \cdot \left[H\right]$$

Everything old is new again

- Many Java ideas have been used before, bytecode (UCSD P-code), garage collection (LISP et. al.), strong static typing, etc.

Those who cannot remember the past are condemned to repeat it.
—George Santayana
The Life of Reason, vol. 1, Reason in Common Sense

- Sun III compilers used extended precision for anonymous values but had no language type corresponding to `double extended`
- Lack of a language type caused problems since the (`doubled extended`) value of an expression assigned to a `double` variable could be used in place of the rounded `double` value stored in the variable

Miscellaneous Problems

- In `widefp` contexts gradual is underflow not required (PEJFPS p. 30) $\Rightarrow$ `widefp` variables might be able to represent *fewer* floating point values than corresponding `strictfp` ones
- Can fused mac be used? How would this be indicated to the programmer?
- The `widefp` default can break existing Java programs that rely on Java's strict floating point semantics (admittedly such programs are in the minority, although codes such as doubled-`double` will break)
- PEJFPS doesn't appear to follow the stated Java evolution plans, changing the minor revision number in the `class` file, etc.
- What about library support? Printing out and reading in extended values? JNI support, Bitwise conversion? ...

Another Way

- Two goals:
  - Grant access to extended precision where available
  - Allow the x86 to run Java's floating point as fast as the floating point of other languages
- Two constraints:
  - Java and the JVM can be modified *but*
  - Keep the language *predictable*
- Speed and extended precision are separate issues:
  - The speed problem can be removed by allowing only the significand to be rounded (giving `float` and `double` values stored in registers greater exponent range). Alternatively, the current practice of storing after each floating point operation can be canonized.
  - A separate type can be used to refer to double extended

`indigenous` and `anonymous`

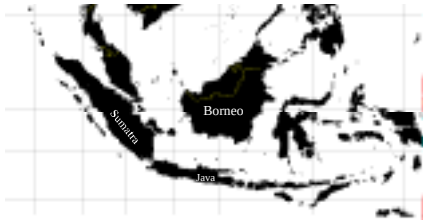
- The Borneo language proposal addresses the issues dealt with by PEJFPS
- Borneo adds a third primitive floating point type, `indigenous`. The `indigenous` type corresponds to the largest IEEE 754 format that executes directly on the underlying processor (80 bit `double extended` on the x86, `double` elsewhere).
- `indigenous` values and variables, can be used in all the usual ways, local variables, object fields, arrays, parameters, etc.
- Borneo has a new declaration, `anonymous` *FloatingPointType*. The *FloatingPointType* indicates the compiler should widen all narrower floating point operands to be of that type (pre-ANSI C style evaluation). Therefore, `anonymous indigenous` allows gives good hardware utilization at the highest precision supported by the hardware.
- This proposal maintains predictability at the cost of more noticeable JVM modifications (to support `indigenous`).

What can you do?

Sun claims to want feedback for their open process.
Send your thoughts before September 15, 1998 to:
`javasoft-spec-comments@eng.sun.com`

Self-promotion and References

For more information on Borneo:
`http://www.cs.berkeley.edu/~darcy/Borneo`



For a discussion of Java' floating point support:
`http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf`

References

Jerome Coonen, "A Note On Java Numerics,"
http://www.validgh.com/numeric-interest/numeric-interest.archive/numeric-interest.9701

Roger A. Golliver, "First-implementation artifacts in Java™"

James Gosling, Bill Joy, and Guy Steele, *The Java™ Language Specification*, Addison-Wesley, 1996.

"Proposal for Extension of Java™ Floating Point Semantics, Revision 1," May 1998
http://java.sun.com/feedback/fp.html