Support for Complex Numbers in Habanero

Revision 1.0

Jisheng Zhao, Rice University, Houston ,Texas
Vincent Cave, Rice University, Houston ,Texas
Yonghong Yan, Rice University, Houston ,Texas
Zoran Budimlic, Rice University, Houston ,Texas
Vivek Sarkar, Rice University, Houston ,Texas

Technical Report TR 14-02
Department of Computer Science
Rice University

This work was supported by BHP Billiton (from Sep 2008 to Sep 2009).

Table of Contents

BACKGROUND 3

COMPILER ARCHITECTURE 3

DESIGN AND IMPLEMENTATION 4

FRONTEND 4
LEXER EXTENSION 4
TYPE SYSTEM EXTENSION 5
AST NODE GENERATION 5
BACKEND 5
IR GENERATION 6
OBJECT INLINING 6
NOTES FOR OPTIMIZATION ISSUES **ERROR! BOOKMARK NOT DEFINED.**

MICRO-BENCHMARKS ERROR! BOOKMARK NOT DEFINED.

CHARACTERIZATION 8
EVALUATED IMPLEMENTATIONS 8
COMPARISON OF PRIMITIVE COMPLEX AND OBJECT COMPLEX 9
COMPARISON OF PRIMITIVE AND OBJECT COMPLEX BASED ON JAVA 1.5 9
COMPARISON OF PRIMITIVE AND OBJECT COMPLEX BASED ON JAVA 1.6 10
COMPARISON OF JAVA 1.5 AND JAVA 1.6 RESULTS 11
COMPARISON TO FORTRAN 90 13

DIPOLE1D EXPERIMENTS 15

METHOD'S BODY SIZE AND NUMBER OF VARIABLES 15
INLINING VERSUS BOXED RETURN VALUE 16
CODE MOTION 19
ARRAY ACCESS 20
LOAD ELIMINATION 20

Background

Habanero Java (HJ) Compiler translates Habanero Java language (based on X10 1.5) source code to Java bytecode (.class files). In order to support Fortran-style complex numbers in the language, we have designed and implemented two approaches:

- Implementing a Complex number class (Java code + library).
- semantic expansion of complex numbers at the language level.

Considering the performance impact of the first approach, we have chosen the semantic expansion to as the main support complex numbers in Habanero Java language and the compiler. The complex number are implemented as the new primitive data type in HJ language. For example, the following code fragment is a valid HJ code:

```
complex_d cx;

cx = (1.0d, 2.0d);

complex cy = cx * cx;
```

The complex number support in HJ compiler includes two primitive data types: double precision (64-bits) and single precision (32-bits).

As the runtime system is based on a Java Virtual Machine, the HJ compiler generates Java bytecode. The complex number are translated to pairs of floating point numbers. For example, given the source HJ code:

```
complex_d cy = cx * cz;
```

The translated Java code is:

```
double cy_r = cx_r * cz_r – cx_i * cz_i;

double cy_i = cx_r * cz_i + cx_i * cz_r;
```

Compiler Architecture

HJ compiler is composed of two major elements (shown on Figure 1):

- Polyglot based front-end: parse HJ source code and generate AST nodes.
- Soot based IR backend: IR analysis and optimization, bytecode generation.

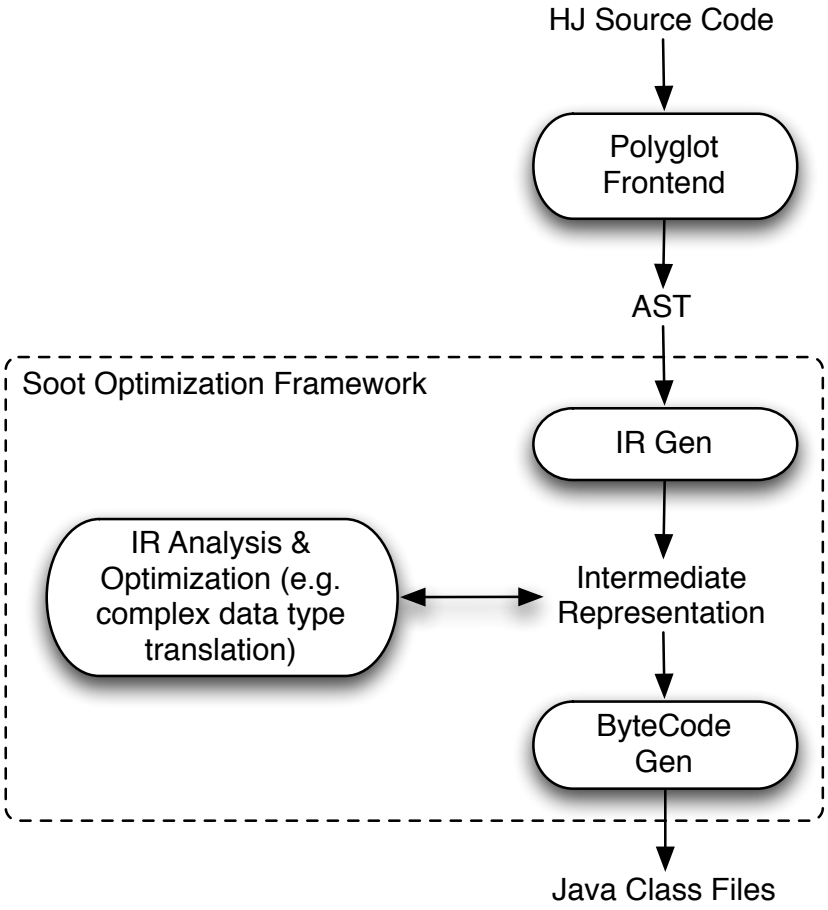

Figure 1. HJ Compiler Architecture

Design and Implementation

Complex number support is in HJ consists of two parts: front-end (parsing, type checking and type transformation) and backend (complex to double translation).

Frontend

The polyglot-based front end is extended to parse the complex numbers as a new primitive type (similar to as int, float, double). There are three steps in handling complex number in the HJ compiler front end:

1. Lexer extension for parsing the complex number related operations.
2. Type system extension for enabling type checking and type transformation.
3. AST Node generation.

Lexer Extension

Focuses on enabling the parser to process all of the complex number related constructs the HJ language, including:

1. Definitions.

2. Binary operations.
3. Array operations.
4. Real/imag data operations, e.g. AREAL, AIMAG.
5. Function call, i.e. input parameter and return value.

Type System Extension

The HJ type system is extended to enable the type checking and translation of complex data type, including:

- Internal type checking, an extension of the current HJ type check ingmechanism.
- Implicit type casts, for example in `complex_d cx = (i, 1.0d)`, if `i` is an integer it should be translated to double implicitly.
- Explicit type casts.
- Type translations for language intrinsic, e.g. AREAL, AIMAG.

AST Node Generation

The interface between front-end and backend is AST Node tree, the front end generates AST nodes for all complex number related constructs, after parsing and processing the HJ source code. The proposed new AST nodes include:

1. Complex PrimaryType.
2. ComplexLiteral which contains two sub literals: real literal and imag literal, e.g. (1.0d, 2.0d), (1.0f, 2.0f), (dx, 2.0d) dx is a double variable.
3. ComplexReal and ComplexImag, for support of intrinsic functions (i.e. AREAL, AIMAG).

Backend

The soot based backend focuses on complex to double/float type translation (see Figure 2). There are two stages: IR generation (translating of AST nodes to soot IR) and storage translation (translate of complex into double/single precision floating point pairs using Object Inlining).

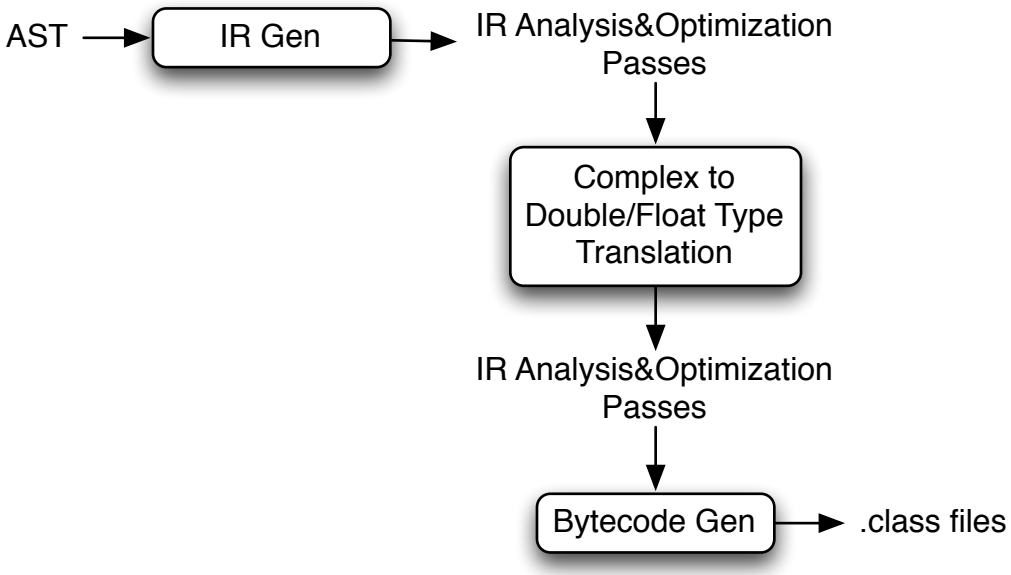

Figure 2. Backend Type Translation

IR Generation

In order to translate the AST nodes with complex numbers to soot IR, we have extended the Soot IR to support complex data type. The extensions are:

1. Primary data type extension for complex.
2. Two new box types for complex double and complex float which correspond to ComplexLiteral AST node, where the two sub literals (i.e. real and imag) are boxed.
3. The IR code extension for constructs containing complex data types, e.g. binary operations, method invocations.
4. Two static intrinsic operations: complexreal and complexiмag corresponding to and ComplexReal/ComplexImag AST nodes.

Object Inlining

Translation of complex numbers into floating point pairs is done in a separate Soot compilation pass using the Object Inlining(Budi mlić & Kennedy, 1997) technique. Here are the translation rules:

1. Complex variable is translated to two double/float variables,
   e.g. `cx = cy + cz` → `cx_r = cy_r + cz_r; cx_i = cy_i + cz_i;`
   `cx, cy, cz` are of type `complex_d`, `cx_r, cy_r, cz_r, cx_i, cy_i, cz_i` are of type `double`.
2. Complex class field is translated to two double/float class fields, the class structure is modified (eliminate the complex field and add the two double/float fields).
   e.g. class `Foo { complex_d cx;}` → `class Foo { double cx_i, double cx_i;}`

3. The complex class field load/store operations are translated to two load/store operations.
4. Complex array is translated to double length double/float array, the real and imag values of complex array element are stored in the adjacent elements in the new double/float array (see Figure 3). The array construction need to be translated as well, e.g. complex_d cArray = new complex_d[n] → double cArray = new double[n * 2]
5. The complex array index value is doubled after the translation. e.g. cArray[i] = cx; → i_ = i * 2; cArray[i_] = cx_r; i_ = i_ + 1; cArray[i_] = cx_i; cArray is a complex_d array before translation and double array after translation, cx is complex_d
6. Comparison between two complex number or a complex number and a float/double/int

   e.g. `if (cx == cy)` → `if (cx.real == cy.real && cx.imag == cy.imag)` where cx, cy are complex type

   `if (cx == d)` → `if (cx.imag == 0.0 && cx.real == d)` // cx is complex and d is double

7. The complex array reference type is translated to double/float array reference type, e.g. `foo(complex_d[] cArray)` → `foo(double[] cArray)`
8. Map the complexreal/complexiмag intrinsic functions to direct access of the inlined data,
   e.g. `d = complexreal(cx)` → `d = cx_r;`
   d is double, cx is complex
9. Reconstruct the method definition to support complex numbers as input parameter.
   e.g. definition: `void foo(complex_d cx, double d)` → `void foo(double cx_r, double cx_i, double d)`
   method invocation: `foo(cx, d)` → `foo(cx_r, cx_i, d)`
10. Reconstruct the method definition to support complex numbers as return values.
    e.g. definition: `complex_d foo(double d)` → `void foo(BoxComplex_d bc, double d)`
    method invocation: `cx = foo(d)` →
    `BoxComplex_d bc = new BoxComlex_d(); foo(bc, d); cx_r = bc.r; cx_i = bc.i;`
    the `BoxComplex_d` is a Java class which boxes the two double values, the class is defined as: `class BoxComplex_d {public double r, i;}`

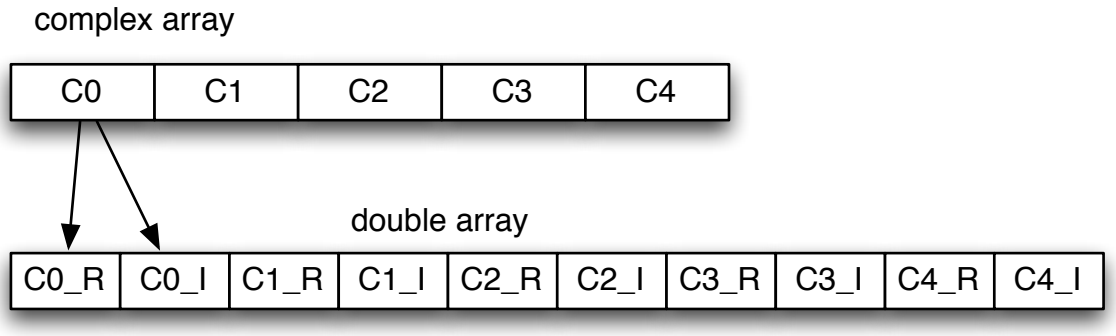

Figure 3. Array Element Mapping

Performance Evaluation

We have developed a set of micro-benchmark in order to evaluate the performance of our current implementation of primitive complex.

Characterization

The micro-benchmarks evaluate the time it takes to perform basic arithmetic operations on complex numbers. These includes basic arithmetic operations such as add, substract, multiply, divide and some more advanced arithmetic operations such as square root, exponential, absolute value, cosinus and sinus (of double). Each of these operations are applied and or stored to complex located in arrays.

Evaluated implementations

The micro-benchmarks have been developed for the following implementations:

- Fortran-based on primitive complex
- Hj-based on primitive complex (Java 1.5)
- Hj-based on primitive complex (Java 1.6)
- Java-based on object complex (Java 1.5)
- Java-based on object complex (Java 1.6)

The fortran version has been compiled with the –O3 optimization flag, java and hj implementation are run with the –server option.

Note that experiments have been carried out on the MacOS X platform, results may vary on other platforms.

Comparison of primitive complex and object complex

Regardless or whether java 1.5 or java 1.6 is used, we can see that the primitive complex implementation outperforms the object one in all cases. Especially when the runtime has to create new complex object frequently. Worthy to note that timing does not include complex array creation and initialization, which are also in disfavor of the complex object implementation.

These benchmarks raise the question of which java version is better to use. It seems from the benchmarks that java 1.6 is slower than java 1.5 on simple arithmetic operations. However there is a significant difference for advanced arithmetic functions that are much more time consuming than simple ones. For instance experiments shows a 1.7 and 2.5 performance improvement for exponential benchmarks.

Comparison of primitive and object complex based on Java 1.5

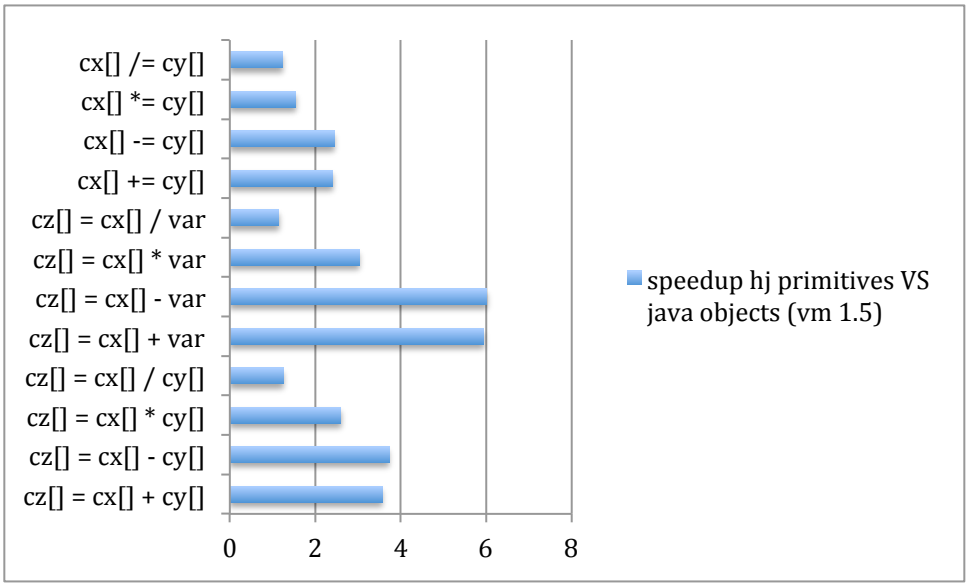

Figure 1: Primitive complex implementation speedup compared to object complex implementation with Java 1.5

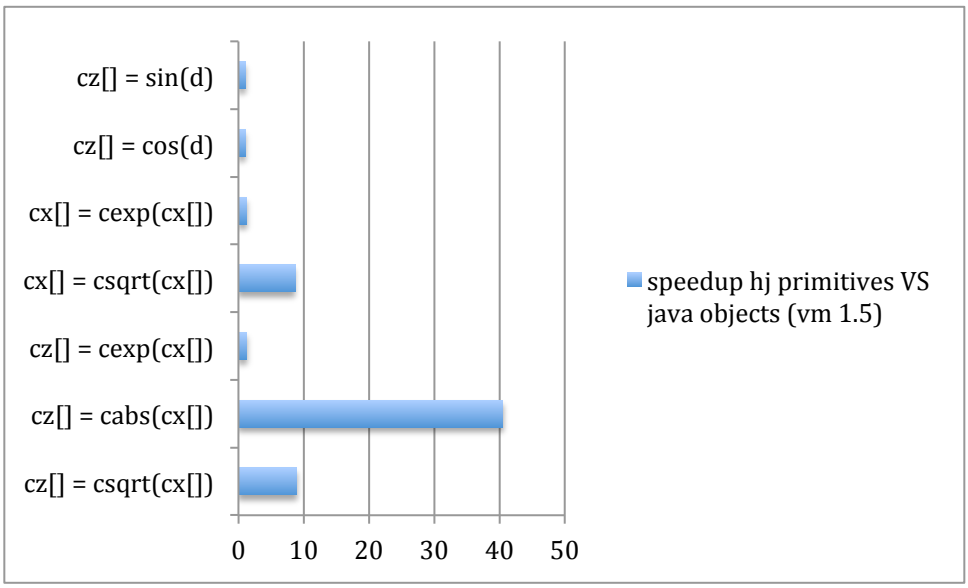

Figure 2: Primitive complex implementation speedup compared to object complex implementation with Java 1.5 (continued)

Comparison of primitive and object complex based on Java 1.6

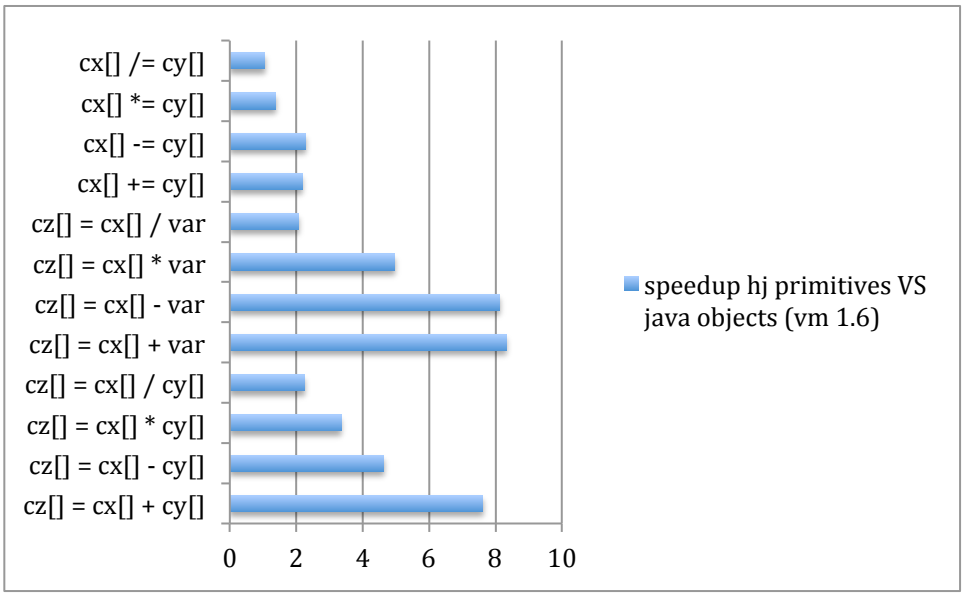

Figure 3: Primitive complex implementation speedup compared to object complex implementation with Java 1.6

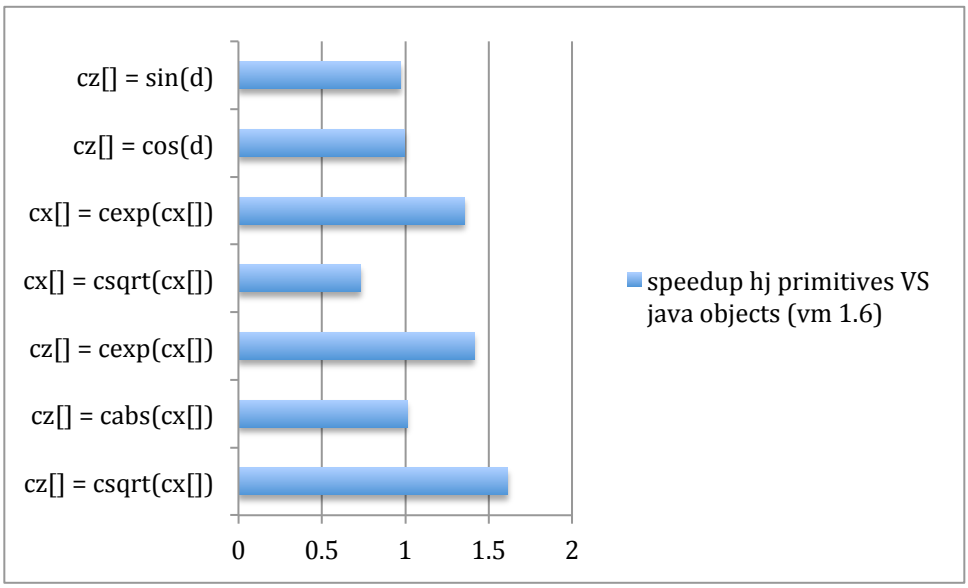

Figure 4: Primitive complex implementation speedup compared to object complex implementation with Java 1.6 (continued)

Comparison of Java 1.5 and Java 1.6 results

We can note that in general the java 1.6 version is slower than the Java 1.5 except in few particular cases involving square root and exponential, which are then much more efficient.

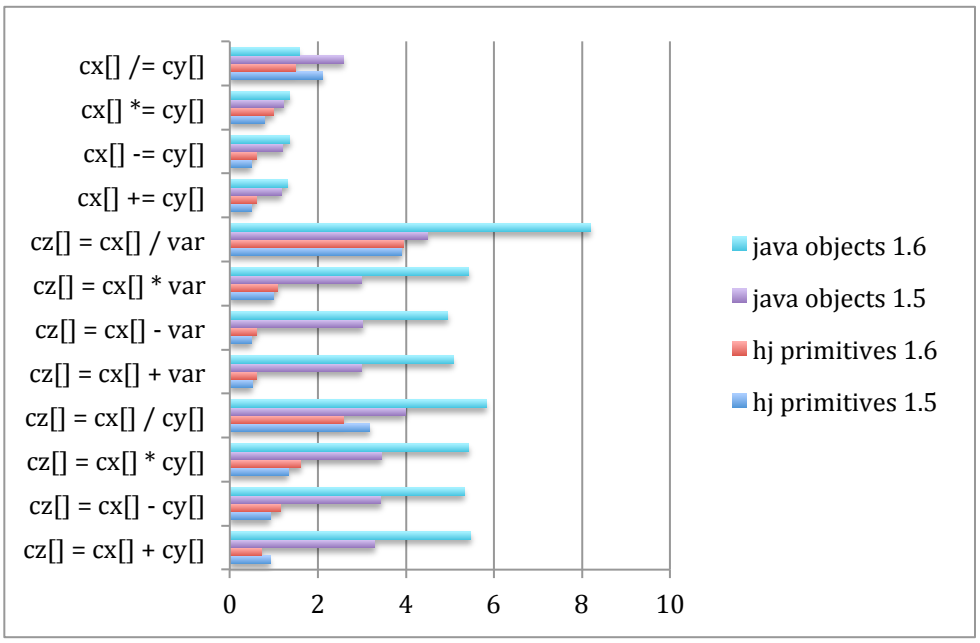

Figure 5: Run durations comparison of java 1.5 and java 1.6

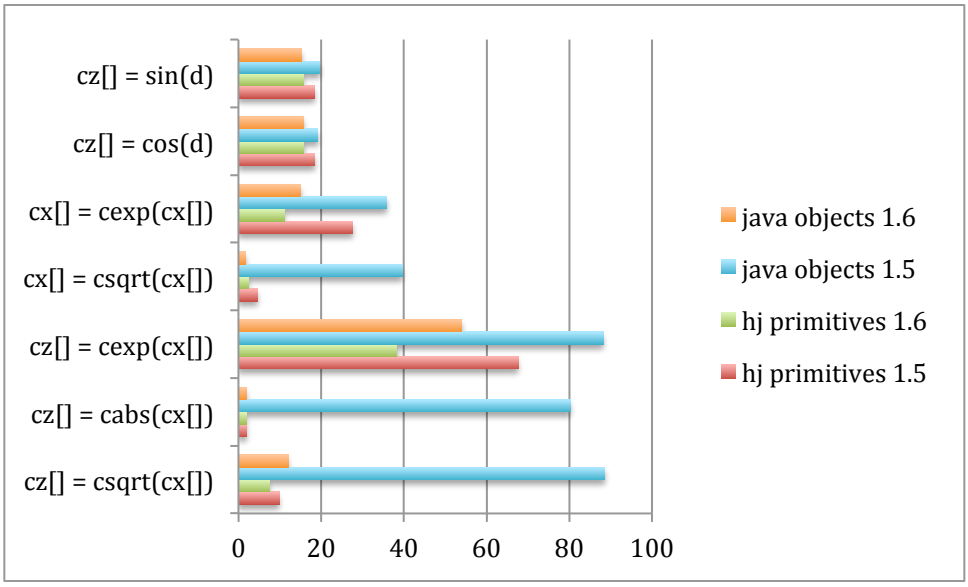

Figure 6:Run durations comparison of java 1.5 and java 1.6 (continued)

Comparison to fortran 90

We have also compared our HJ complex implementation with native Fortran 90 implementation, and we demonstrate on Figure 7 that our implementation is within a factor of 2 of the native Fortran.

We can also notice a consequent difference between executions in java 1.5 and java 1.6 on some micro-benchmarks. In average the java 1.5 and the java 1.6 versions are respectively 55% and 33% slower than the fortran implementation.

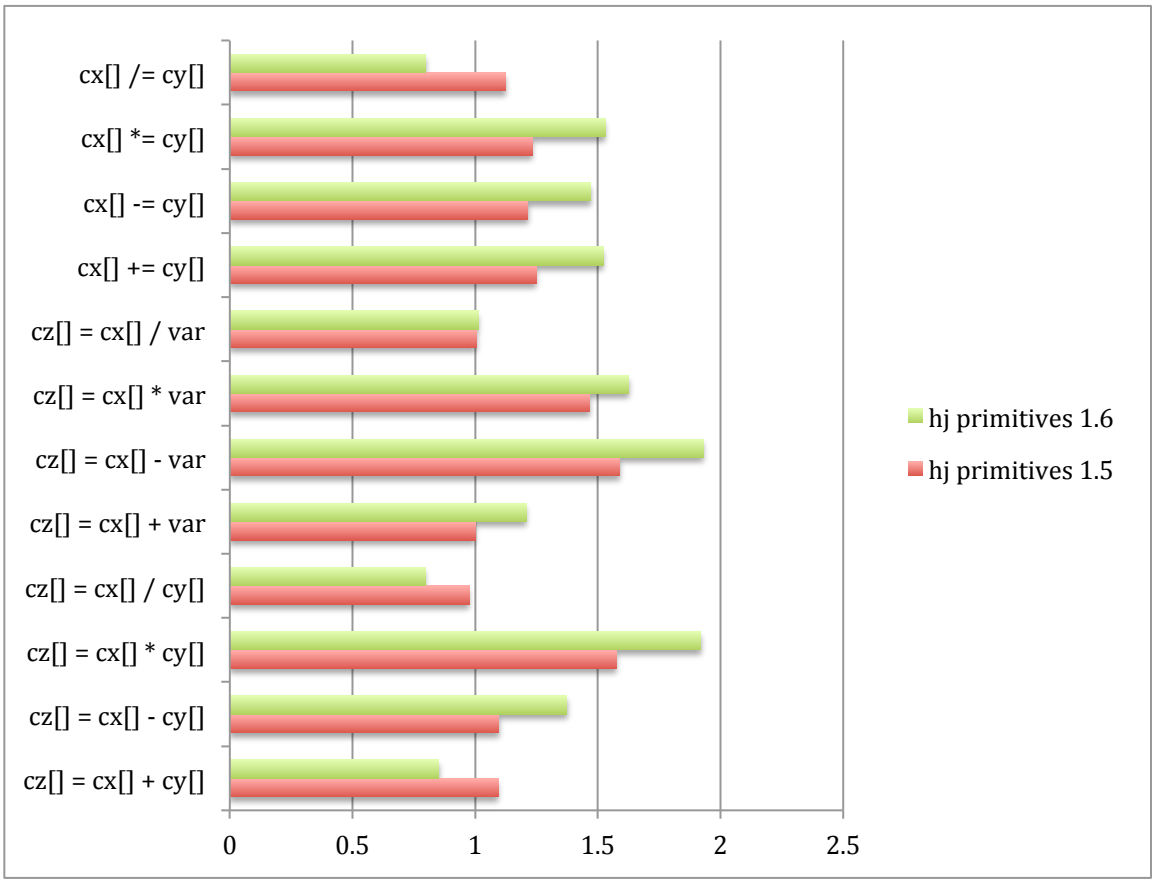

Figure 7: Overhead of hj primitive complex compared to fortran f90 (> 1 means hj primitive is slower than fortran)

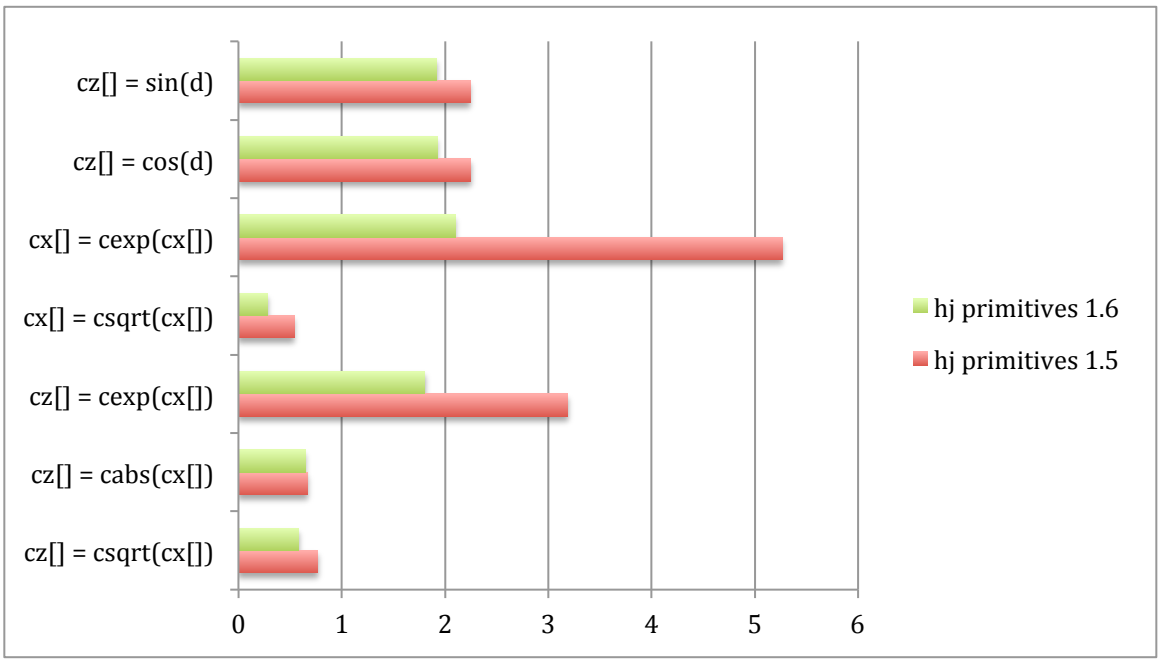

Figure 8: Overhead of hj primitive complex compared to fortran f90 (> 1 means hj primitive is slower than fortran) (continued)

Hand optimizations and performance of Dipole1D

Analyzing the micro-benchmarks alone, we can conclude that our implementation of complex numbers achieves very good performance when compared to object-based implementations, and reaches a level that is comparable (within a factor of 2) to a native Fortran implementation. When analyzing a fairly large real-world application (Dipole1D benchmark), we have discovered some additional areas for improvement. The following sections describe some coding practices that may affect the performance of applications written in HJ using the complex primitive types. They describe the hand-coded optimizations we have performed on Dipole1D to avoid introducing those performance penalties. They can be also viewed as guidelines for writing efficient code in HJ using complex primitive types.

The following are the issues one needs to be aware of when writing HJ code in order to avoid introducing performance bottlenecks:

- Method's body size and number of variables
- Inlining versus boxed return value
- Code motion
- Array Accesses
- Load elimination

The following sections describe these issues and how to handle them.

Method's body size and number of variables

The size of method body and the number of variables inside the method can produce register pressure (JVM's in general use use linear-scan register allocators, which is are much more easily effected by the number of variables).

Method bodies in Fortran version of Dipole1D program are fairly large. Our complex number implementation splits one complex variable into two double/float variables; this can double the number of local variables in the worst case and introduce much more register pressure. For example, the Dipole1D's computation kernel is Dipole1D.hed_coef_nlayer, in which there are 54 local variables in the bytecode generated by our compiler, and only 32 local variables in the bytecode generated by normal javac compiler using the object-based complex implementation.

The solution is to split large methods into several smaller methods. Dipole1D.hed_coef_nlayer has 6 loops that have complex64 computation, we have split it into 8 smaller methods, this has resulted in an overall 2X~2.05X performance improvement over the original version of code.

Inlining versus boxed return value

Inlining is a typical optimization for object-orient language, as it can reduce the overhead for saving/restoring the calling context. But it could be harmful for performance if the method body is too large (as we have explained above). In our current compiler implementation, we inline all of the methods that return complex numbers. This does not result in better performance in all cases, however. Using of the boxed return value resulted in improved performance in some case. For example, the util.exp and util.sqrt called in Dipole1D.hed_coef_nlayer produced better performance if we used a boxed return value, because the method Dipole1D.hed_coef_nlayer is large, and increasing the size produces more register pressure.

In future work, we plan to develop some compiler heuristics that will decide in which cases to inline the method that returns a complex value (thus avoiding the overhead of saving/restoring the calling context and of boxing/unboxing of the complex number being returned), and in which cases to use a boxed complex for the return value (in order to avoid increasing the register pressure in the caller method).

Reusing the boxed return value

In situations where we do use boxed objects to handle return values of complex types, a naïve translation could result in some inefficiency. We can perform some code motion and inter-procedure analysis to optimize the type translation.

Here are two examples. Given the code:

```
for (int i = 0; i < n; i ++) cArray[i] = foo(d); // foo returns a
complex?d value, cArray is a complex_d  array
```

translated code will be:

```
for (int i = 0; i < n; i ++) {

      BoxComplex_d bc = new BoxComplex_d();

      foo(bc, d); int i_ = i * 2;
```

```
      cArray[i_] = bc_r; cArray[i_] = bc_i;}
```

To avoid creating a boxed object in each each loop iteration (which can be prohibitively expensive), we can perform loop invariant code motion.

The optimized code is:

```
BoxComplex_d bc = new BoxComplex_d();

for (int i = 0; i < n; i ++) {

      foo(bc, d); int i_ = i * 2;

      cArray[i_] = bc_r; cArray[i_] = bc_i;}
```

Another example:

```
for (int i = 0; i < n; i ++) bar(cArray); // cArray is complex_d
array

void bar(complex_d[] cArray) {

      cArray[i] = foo(d);}
```

the translated code is:

```
for (int i = 0; i < n; i ++) bar(cArray); // cArray is complex_d
array

void bar(double[] cArray) {

      BoxComplex_d bc = BoxComplex_d();

      foo(bc, d); int i_ = i * 2;

      cArray[i_] = bc_r; cArray[i_] = bc_i }
```

The overhead is similar as in the first example. In this case, we need to perform inter-procedural analysis in order to allow code motion.

The optimized code is:

```
BoxComplex_d bc = BoxComplex_d();

for (int i = 0; i < n; i ++) bar(bc, cArray); // cArray is
complex double array

void bar(BoxComlex_c bc, double[] cArray) {
```

```
      foo(bc, d); int i_ = i * 2;

      cArray[i_] = bc_r; cArray[i_] = bc_i }
```

Now the caller and callee share a box object, there is only small overhead for passing the box object reference.

A general optimization would be to share the boxed object in the whole method scope and call path. For example:

```
void foo(BoxComplex_d bc, …) {

      … …;

      bar0(bc, …);

      … …;

      bar1(bc, …);

}
```

The intra-procedure analysis would be needed here to identify the asynchronous execution case.

For example:

```
void foo(BoxComplex_d bc, …) {

      … …;

      async bar0(bc, …);

      … …;

      bar1(bc, …);

}
```

As bar0 will be spawn in a parallel thread, we have to create a new box object for it. The translated should be:

```
void foo(BoxComplex_d bc, …) {

      … …;

      BoxComplex_d bc1 = new BoxComplex_d();

      async bar0(bc1, …);

      … …;
```

```
      bar1(bc, …);
}
```

Code motion

Dipole1D uses some global array references to store and share data between methods. Each time a method wants to access an array element, it needs to do class field load operations first (the array references are stored in class field, e.g. Dipole1D.gamma). If the array access is inside a loop body, this will introduce some overhead. Here's an example:

```
class Test {
      ... ...
      complex64[] carray;
      public void run(complex64 x) {
            for (int i = 0; i < carray.length; i ++)
                  carray[i] += x;
      }
}
```

The compiler generated code will be (java code):

```
class Test {
... ...
double[] carray_d;
public void run(double x_real, double x_imag) {
      for (int i = 0; i < carray_d.lenght/2; i ++) {
            double[] temp_carray = carray;
            temp_carray[i * 2] += x_real;
            temp_carray[i * 2 + 1] += x_imag;
      }
}
}
```

If the code in red can be moved out of the loop body, the performance can be improved. Unfortunately, both our compiler and javac are unable to do that (i.e. compiler presumes it would be unsafe, especially in the case of multi-thread programming). We achieved some speed improvement (about 4~6% overall) when we performed the code motion by hand.

Array access

In our compiler implementation, the complex number array is replaced by a double-sized double/float number. The compiler has to correctly manage the array indexing. For example, if i is the index used in the access of a complex number array, compiler will generate new index i * 2 and i * 2 + 1 (see the code example above) to access the complex number array element's real and imaginary parts. This is an additional overhead.

The common subexpression elimination (both soot and javac support this) can eliminate some redundant index calculations. For example, for a statement `a[i] = b[i] + c[i];` where a, b, c are all complex number arrays, the translated code (java code) would be:

```
temp = i * 2;
b_real = b[temp]; b_imag = b[temp + 1]; c_real = c[temp];
c_imag = c[temp + 1];
a[temp] = b_real + c_real; a[temp + 1] = b_imag + c_imag;
```

here we only need to calculate i * 2 once, and can reduce some overhead.

Load elimination

Similarly to code motion, both soot and javac do not handle load elimination well. To load a complex value, the complier will generate two load operations (i.e. load real and imag value). If a heap variable (complex type) is used more than once, load elimination will improve performance. We achieved some performance improvements by doing hand-coded load elimination. We plan to develop a full load elimination optimization pass in the Soot backend of our HJ compiler in the future.

Dipole1D Performance Results

Below are the result of our experimental evaluation of the Dipole1D program. As the execution time for 1 run is about 1 second (for Java and HJ versions), we have increased the granularity of the program by increasing the number of computation kernel's iterations (the `comp_spatial` function) to better illustrate the performance differences.

Figure 2.1 shows the executions time for native Fortran, Java and Habanero Java versions of Dipole1D. The Fortran90 code is built using gfortran compiler. The Java and HJ code are run on top of the Sun JDK 1.6 runtime with '-server' option

(optimizing JIT compilation). The x axis shows the number of times comp_spatial kernel has been run, while the y axis shows the total execution time in seconds.

We can observe that while the HJ version of the code is not as fast as native Fortran (about 40% slower), it significantly outperforms (by about 70%) the Java version of the code. We can conclude that the performance which is comparable to native Fortran combined with the much better programming model, programmability, maintainability and productivity makes the Habanero Java platform with the support for primitive complex numbers a much more attractive alternative to Fortran for seismic computation.

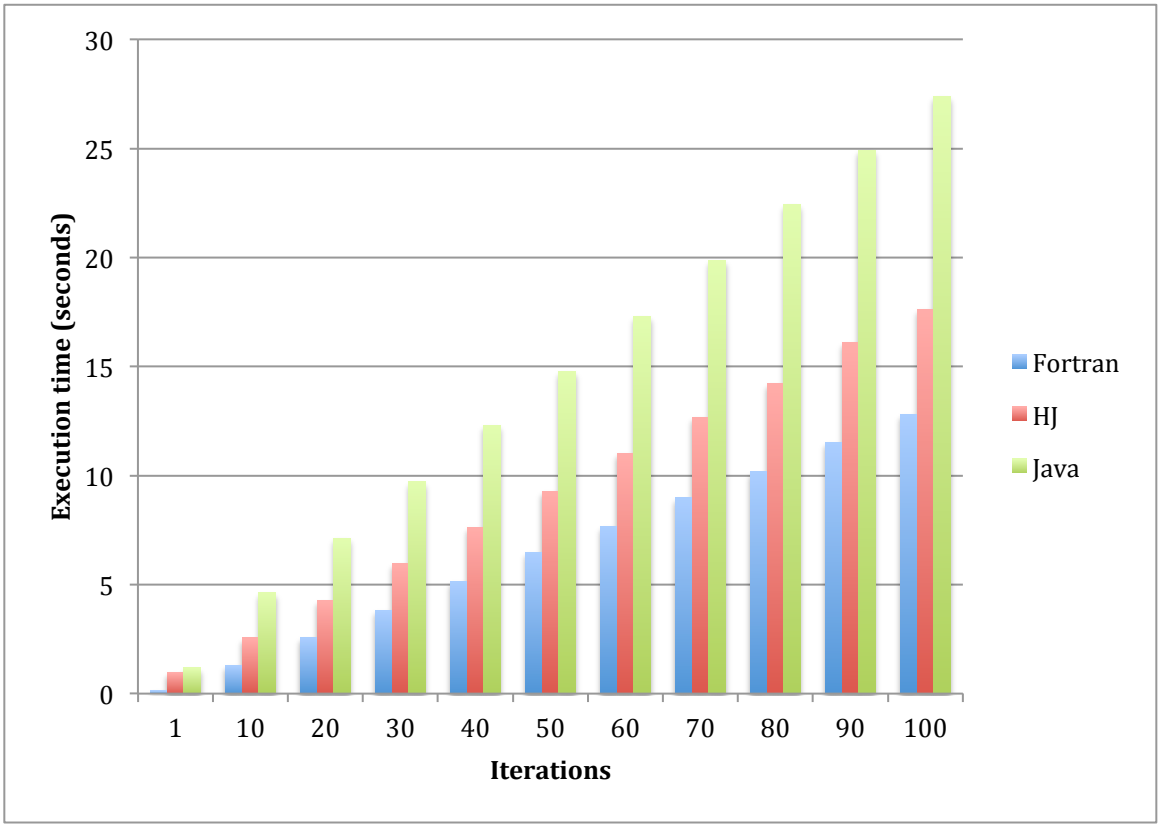

Figure 2.1