A Design of Deep Reification

Shigeru Chiba YungYu Zhuang Maximilian Scherr
Graduate School of Information Science and Technology
The University of Tokyo, Japan
{chiba,zhuang,scherr}@csg.ci.i.u-tokyo.ac.jp

Abstract

This short paper presents our design of deep reification, which is a reflection mechanism for computation offloading. As heterogeneous computing is getting popular, several systems have been proposed and implemented for offloading a part of Java program to an external processor such as GPU. Deep reification provides the common functionality among those systems so that it will help their development. It dynamically extracts abstract syntax trees of offloaded methods as well as offloaded objects and class definitions. These methods, objects, and classes are *deeply* copied to be sent to an external processor. Deep reification also allows user programmers to annotate offloaded code to give optimization hints to the offloading system built with deep reification.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

Keywords Reflection, Meta programming

1. Introduction

As the effectiveness of GPU computing emerges, a number of computation offloading systems from Java to CUDA or OpenCL have been developed. For example, JCUDA(Yan et al. 2009), Rootbeer (Pratt-Szeliga et al. 2012), JConqurr (Ganegoda et al. 2009), JCudaMP (Dotzler et al. 2010), and JaBEE (Zaremba et al. 2012) belong to this category. These systems except JCUDA translate Java code into CUDA on the fly during runtime and execute the generated CUDA code on GPUs to exploit GPUs' massively parallel computing capability. Computation offloading is effective with not only GPUs but also many-core accelerators, FPGA chips, and even native CPUs. A Java program sometime runs faster if part of the program is translated into C code and run directly on a native CPU after being compiled by an external C compiler to exploit, for example, the latest SIMD instructions such as AVX2 and AVX-512 and/or multi-threading by OpenMP. The translation into C is also effective when the just-in-time compiler of the Java virtual machine (JVM) cannot produce highly optimized code for the target processor. For example, we cannot expect highly optimized just-in-time compilation by the JVM on the K supercomputer at the RIKEN AICS in Japan since its node processors are SPARC64 VIIIfx, which supports non-standard SIMD extension. The OpenJDK for SPARC does not support that extension. It is also effective to offload computation from the JVM to multiple-nodes parallel supercomputers such as the K computer, where a number of nodes communicate through MPI.

This paper presents *Bytespresso*, which provides a reflection mechanism used as a basis of systems for offloading computation from the JVM. This mechanism is called *deep reification* and (1) it dynamically extracts a self-contained snapshot of the current execution environment for executing an offloaded method. It contains abstract syntax trees of necessary methods, necessary class definitions, and objects that will be accessed during the execution of the offloaded method. (2) The mechanism *deeply* extracts (*i.e.* reify) all methods that may be directly or indirectly invoked from an offloaded method unless they are explicitly excluded. The mechanism traces a call graph including dynamic method dispatch. (3) The construction of an abstract syntax tree is customizable and its node objects are also customizable.

Reflective computation (Smith 1984) consists of two operations: *reify* and *reflect*. Reifying is to convert an abstract concept in a program into a first-class object, which a program can process. For example, program code and type definitions in a virtual machine can be reified. Reflecting is to convert a first-class object back to an abstract concept so that program behavior is affected by changes to a reified first-class object. Deep reification is our operation for reifying offloaded code and data.

Developers can use Bytespresso and its deep reification mechanism to build an offloading system. Bytespresso runs on the standard JVM. Its deep reification reads Java bytecode, decompiles it, and constructs an abstract syntax tree corresponding to decompiled Java source code. Hence the developers have only to implement a code generator from the abstract syntax trees and an executor of the generated code. If needed, the abstract syntax tree can be translated into target code without preserving the original Java semantics. Custom node objects will help this custom translation. For example, accesses to a particular `static` field might be translated as accesses to some native language construct of the target system. Developers of GPU offloading systems might want to provide a `static` field, `Cuda.idx` (`Cuda` is a library class), for their users. Unlike accesses to normal `static` fields, an access to that `static` field would be specially translated into an access to a built-in variable `threadIdx` in CUDA as seen in typical GPU-offloading systems like Rootbeer (Pratt-Szeliga et al. 2012). This design enables the users to directly access `threadIdx` in their offloaded Java code while their code is still within the confines of the standard Java syntax. In this fashion, language extensions can be expressed by a similar idiom of the host language Java. A preprocessor is not needed for making these extensions available.

2. Deep Reification and Bytespresso

If a method that will be offloaded is given, deep reification extracts a self-contained snapshot of the current execution environment. It

MODULARITY Companion'16, March 14–17, 2016, Málaga, Spain
ACM. 978-1-4503-4033-5/16/03...
http://dx.doi.org/10.1145/2892664.2892695

is not a whole environment but only a minimum part of the environment that is necessary to execute that "root" method on an isolated platform. This snapshot includes all necessary methods, type definitions, and objects. Deep reification is similar to object serialization or marshalling since object serialization also extracts a self-contained object graph for restoring it on a different platform. However, unlike deep reification, the serialized data contain only type names but not type definitions. The type definitions are expected to be available on a destination platform.

This design of deep reification is based on the fact that a platform where an offloaded program runs has an isolated memory space and hence accessing data on the host platform is expensive; explicit data transfer and data conversion are required. Even program code has to be explicitly transferred with code translation. Offloading systems manage this data/code transfer and conversion. Deep reification supports the implementation of this management.

Using Bytespresso, our deep-reification implementation for Java, is simple. For example, the following code extracts a snapshot for executing method m:

```
Object[] args = ...;
Reifier reifier = new Reifier(m, args);
Snapshot image = reifier.snap();
```

The snap method on reifier performs deep reification. Here, m refers to a method[1] used as the root of the extraction and args refers to the runtime values of the arguments passed to that method. Deep reification extracts a snapshot of the environment that is needed when invoking m with the argument args. An object passed as an argument is included in the snapshot.

The snapshot returned by snap contains abstract syntax trees of not only the body of the method m but also other methods' that may be invoked during the execution of the method m. Deep reification extracts not a single tree but a forest. The snapshot also contains a table of classes referred in the abstract syntax trees. Furthermore, it contains a table of objects accessed through static fields as well as the objects passed as args.

If a node object of the abstract syntax tree represents a method call, it holds references to the abstract syntax trees of all the methods potentially invoked on that call. To correctly enumerate these methods, snap refers to the types of the objects contained in the snapshot. It also refers to classes that will be instantiated when the program contained in the snapshot is executed.

Since computing the method m passed to snap as an argument is not intuitive for end users, an offloading system built with Bytespresso may provide a wrapper class providing a higher-level programming interface. For example, if the system is for offloading to a GPU, its users could write the following code:

```
double[] p = ...;
double[] q = ...;
double d = new GpuOffloader().run(() -> {
    double r = dotProduct(p, q); return r;
});
```

The run method in the GpuOffloader class takes a function closure (or a Java lambda). In the code above,

```
() -> {
    double r = dotProduct(p, q); return r;
}
```

is a function closure. It does not take any arguments but executes the body {...} with the current values of p and q. The implementation of run would be as follows:

```
import java.util.function.DoubleSupplier;
   :
double run(DoubleSupplier f) {
    CtMethod m = getOffloadMethod();
    Object[] args = { f };
    Reifier reifier = new Reifier(m, args);
    Snapshot image = reifier.snap();
    return translateIntoCUDAandExecute(image);
}
   :
static double offload(DoubleSupplier f) {
    return f.getAsDouble();
}
   :
```

The implementation of run would first compute the method object representing offload by calling getOffloadMethod. Then it would perform deep reification by snap with that method object. Since abstract syntax trees of the body of the offload method, the body of the function closure, and the body of dotProduct (and others called by dotProduct) are extracted as well as p and q, the run method could translate the trees into CUDA code, compile it by an external CUDA compiler, and execute it.

3. Abstract Syntax Tree Construction

The implementation of deep reification needs as a primitive mechanism to be able to read the bytecode of a specified method body. Some virtual machines (VM) such as Graal (Oracle 2012), OpenJIT (Ogawa et al. 2000), and classic Smalltalk VMs already provide a reflection mechanism for reading bytecode loaded in a virtual machine. Another approach is to rely on existing facilities for locating and reading a class file to obtain bytecode. Since this approach is available with the standard Java 8 (and earlier) VM, Bytespresso adopts this approach. A limitation of this approach is that all bytecode has to be stored in class files. So if a program uses a function closure, -Djdk.internal.lambda.dumpProxyClasses option has to be given to the Java VM running Bytespresso. It forces the Java VM to dynamically produce a class file containing the bytecode of a function closure.

After reading bytecode, deep reification decompiles the bytecode to construct an abstract syntax tree of that method. Since the decompilation may not reconstruct all Java statements, an abstract syntax tree obtained by deep reification does not exactly represent the original Java source. Bytespresso's Deep reification mechanism reconstructs only expressions and control structures so that the obtained tree can be easily converted into C or C-like language code. We attempt to reconstruct control-flow structures, in particular, for-loops. Where this fails, we fall back to representing control-flow by Goto and (conditional) Branch nodes in an abstract syntax tree.

Deep reification does not extract only a single abstract syntax tree. It rather extracts a forest; the trees of methods invoked from a root method. Hence it traces a call graph to collect all the methods. Since Java supports dynamic method dispatch, a method call in an extracted method body may dynamically select and invoke one of multiple methods declared in different classes. The selected method is not statically determined. Hence, deep reification extracts abstract syntax trees of all the methods potentially selected on each call. It can exploit runtime values for a technique known as devirtualization (Aigner and Hölzle 1996) to minimize a set of potentially selected methods.

For each call to method m, deep reification examines all the classes of the objects reachable from arguments given to a "root" method given to snap of Bytespresso, or a static field accessed in a method body that has been extracted so far. It also examines a class instantiated in such a method body. Then, if the examined class overrides m, the overriding method m' is added to the set of potentially selected methods. The body of the added method m'

[1] The variable m does not refer to java.lang.reflect.Method. It refers to a CtMethod object provided by Javassist bytecode engineering library (Chiba 2000).

is examined and, if it instantiates a new class or accesses a static field, the instantiated class or the object that the static field refers to is also examined. If a new method call is found in the body, then all the potentially-selected methods for this call are collected. This algorithm is repeatedly performed until no new class or method call is found. Deep reification does not support the use of Java's reflection API in extracted methods since the algorithm above is unable to find a class that will be loaded later by reflection.

4. Abstract Syntax Tree Customization

Besides customizing the reification process by overriding methods in Reifier, Bytespresso allows customizing the construction of abstract syntax trees. The node objects of abstract syntax trees are instances of ASTree or its subclasses. Since they support the Visitor pattern (Gamma et al. 1994), a code generator can be written as a visitor traversing trees. Bytespresso allows programmers to change the class for a particular node object representing a method body from the default class Function to its subclass. Programmers can add an annotation @Metaclass to a method so that the node object representing the body of that method will be changed to an subclass instance. For example,

```
@Metaclass(type=CUDA.Global.class)
public static void gpuKernel(int blk, int th) { ... }
```

This will change the class of a node object for the body of the gpuKernel method. The node object will be created by a factory object CUDA.Global.instance. A code generator can exploit this to change how to generate the code. An offloading system can rely on this feature to allow user-programmers to control code generation by adding this annotation, for example, so that the gpuKernel method will become a __global__ function in CUDA.

Once an @Metaclass annotation is given to a method, the specified factory object continues to create node objects for methods directly or indirectly called from the method that @Metaclass is given to. This feature is useful when an offloading system to CPU and GPU automatically determines whether a given Java method should be translated into a __host__ function or a __device__ function in CUDA.

Programmers can also add an @Metaclass annotation to a class declaration in Java. The annotation changes the class for an object representing the class declaration. Recall that a snapshot returned by the snap method in Reifier contains a table of classes appearing in abstract syntax trees. The annotation added to a class declaration changes the class of a table entry representing that class declaration. For example,

```
@Metaclass(type=ImmutableClass.class)
public final class Vec3 {
  public final float x, y, z;
}
```

for the Vec3 class, Bytespresso creates an instance of ImmutableClass and puts it in a class table. A code generator can exploit such a custom entry in a class table so that custom code will be generated for an expression related to Vec3, the class represented by that table entry. This design is similar to compile-time metaobject protocols or compile-time reflection (Chiba 1995).

For example, when a code generator encounters a new expression for object creation, the visit(New expr) method is invoked on the code generator. New is the class for node objects representing new expressions. It may look up a class table and call a method on a table-entry object representing the instantiated class. The method will generate appropriate code for the new expression. If the expression is new Vec3(), then a method on an ImmutableClass object will be invoked. By implementing a method in ImmutableClass to override the default behavior, the code generator can implement custom object creation effective only to Vec3 and other classes associated with ImmutableClass.

A code generator may also exploit a table-entry object in a class table when it generates the code defining a data structure for every class type. For example, if target code is generated for the C language, a class type in Java may be translated into a struct type in C. A code generator may generate a custom struct type for the Vec3 class since it is an ImmutableClass (a class for immutable objects).

An @Metaclass annotation given to a class declaration may be inherited by subclasses. A table-entry object in a class table can supply an @Metaclass annotation to subclass declarations. For example, an ImmutableClass object supplies an annotation to subclasses of Vec3 to use ImmutableClass. Programmers do not have to explicitly add @Metaclass to every subclass. This is useful when building a class library, where an @Metaclass annotation is hidden in a super class. The library users can define a subclass without considering @Metaclass.

5. Delimiting Extraction

Although deep reification extracts the abstract syntax trees of all the methods included in a call graph from a root method, programmers may want to delimit this traversal of a call graph. Deep reification allows developers of offloading systems to control how to delimit the traversal.

Bytespresso stops the traversal when it encounters a native or abstract method since it does not have a method body. In addition, an offloading system built on Bytespresso may provide a user-defined "native" method, whose body is not traversed. For this aim, Bytespresso provides an annotation @Native:

```
@Native("return sqrtf(v1);")
public static float squareRoot(float f) {
  return (float)Math.sqrt(f);
}
```

Since a method squareRoot has an @Native annotation, its body is not traversed. An abstract syntax tree for the sqrt method in Math class is not extracted. A node object representing the method body of squareRoot is an instance of a subclass of the default class Function, that is, a custom class that does not contain an abstract syntax tree of the method body. It only contains an argument to the annotation "return sqrtf(v1);". A code generator can use this argument to generate code for the squareRoot method. It may generate a function in the C language whose body is the same as that argument and has a parameter v1 referring to the first argument to that function.

6. Optimization

As part of deep reification, Bytespresso transforms extracted abstract syntax trees for performance optimization of offloaded code. Since deep reification uses runtime values when extracting abstract syntax trees, constant propagation (and constant folding, or simple partial evaluation) based on these runtime values is possible. This is not a mandatory feature of deep reification but a useful extension.

Bytespresso recognizes that a final static field has a constant value. If a reference to an object is a constant value and the object has a final field, then this final field is also recognized as having a constant value. Note that a free variable in a function closure is currently implemented as a final field of the function object. Bytespresso also recognizes that the arguments passed to the root method (*i.e.* the second argument to the constructor of Reifier) are constant values. If a variable is initialized with a constant value and does not change its value, the variable is recognized as having a constant value.

When a method is called with an argument and the argument is recognized as a constant value, Bytespresso specializes the body of the called method with that constant value, *i.e.* it constructs a specialized abstract syntax tree for that method body. In this specialized method body, that argument is recognized as a constant value.

If a variable or field has a constant value, it is available from the node object representing that variable or field in an abstract syntax tree. A code generator using Bytespresso can exploit this constant value. Furthermore, Bytespresso exploits this for devirtualization (Aigner and Hölzle 1996). If Bytespresso encounters a method call and a reference to a called object is constant, it statically resolves method dispatch and binds the method call directly to a particular method declared in the class of the called object. It extracts only an abstract syntax tree of this particular method.

Bytespresso also performs dead code elimination. If a constant value is the value of a branch condition of `if` statement, its then or else block not executed is eliminated from an abstract syntax tree as dead code. A method call in the eliminated block is not examined to find a called method or further extract an abstract syntax tree for the method. Dead code elimination can be also used for delimiting extraction. Bytespresso currently performs dead code elimination only if a branch condition is either a constant value of `boolean` type or an expression testing a constant value of `int` is equal to `0`.

Although Bytespresso tends to specialize a method body, this has a risk of code explosion. To avoid excessive specialization and infinite regression due to recursive calls, if the number of the specialized instances of the same Java method reaches a threshold, Bytespresso stops specialization for that method. For the same reason, arguments of primitive types such as `int` are not considered for the specialization. Only if an argument of reference type is a constant value, Bytespresso performs specialization.

7. Concluding Remarks

This paper presents the design for deep reification and Bytespresso as its implementation for Java. Deep reification is a reflection mechanism for helping the development of a computation-offloading system. It extracts a snapshot of the current environment necessary for executing offloaded code. An offloading system can be developed by implementing only a code generator from the code and objects contained in the snapshot.

There have been a few systems providing functionality similar to deep reification or Bytespresso. Lancet (Rompf et al. 2014) allows programmers to directly control just-in-time (JIT) compilation in Scala. Although it does not explicitly provide deep reification as presented in this paper, programmers can access the bytecode of a specified method at runtime. So they can also build an offloading system on top of Lancet; a customized JIT compiler on Lancet can generate GPU code instead of native CPU code. However, since Lance is for customizing a JIT compiler like OpenJIT (Ogawa et al. 2000), making an extracted snapshot of the environment self-contained is not a main concern. In the context of JIT compilation, it is customary to have escape hatches and interfaces to account for dynamic situations. The resulting code of JIT compilation does not run on a different platform; it can access missing objects and code on demand.

Graal (Oracle 2012), which is used as the back-end of Lancet, makes bytecode accessible from a running program through a graph-based internal representation (IR). Although deep reification adopts abstract syntax trees as its IR for straightforward translation into a C-like language, the graph-based IR is good for optimization. It is used, for example, for partial evaluation by Truffle (Würthinger et al. 2013).

As we have already mentioned, there have been a number of systems for offloading computation from the JVM to GPUs. Although a certain degree of deep reification (as we define it) is internally performed in most systems, it is tightly embedded in the systems and hence its implementation is not reusable. An article (Nystrom et al. 2011) on Firepile mentions how Firepile constructs abstract syntax trees while supporting dynamic method dispatch. According to the article, Firepile seems to conservatively enumerate methods on each call. Unlike deep reification, it seems to collect methods in all available subclasses of the type of a called object.

Acknowledgement

This work was partly supported by JST CREST funding program.

References

G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. In *ECOOP '96 — Object-Oriented Programming*, LNCS 1098, pages 142–166, 1996.

S. Chiba. A metaobject protocol for C++. In *Proc. of ACM OOPSLA*, pages 285–299. ACM, 1995.

S. Chiba. Load-time structural reflection in Java. In *ECOOP 2000*, LNCS 1850, pages 313–336. Springer-Verlag, 2000.

G. Dotzler, R. Veldema, and M. Klemm. JCudaMP: OpenMP/Java on CUDA. In *Proc. of the 3rd Int. Workshop on Multicore Software Engineering*, IWMSE '10, pages 10–17. ACM, 2010.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

G. A. C. P. Ganegoda, D. M. A. Samaranayake, L. S. Bandara, and K. A. D. N. Wimalawarne. JConqurr — a multi-core programming toolkit for Java. *Int. Journal of Computer and Information Engineering*, 3(7):596–603, 2009.

N. Nystrom, D. White, and K. Das. Firepile: Run-time compilation for GPUs in Scala. In *Proc. of the 10th ACM Int. Conf. on Generative Programming and Component Engineering*, GPCE '11, pages 107–116. ACM, 2011.

H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and F. Kimura. Openjit : An open-ended, reflective jit compiler framework for java. In *ECOOP 2000*, LNCS 1850, pages 362–387. Springer-Verlag, 2000.

Oracle. Openjdk: Graal project. `http://openjdk.java.net/projects/graal`, 2012.

P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch. Rootbeer: Seamlessly using GPUs from Java. In *Proc. of the 2012 IEEE 14th Int. Conf. on High Performance Computing and Communication & 2012 IEEE 9th Int. Conf. on Embedded Software and Systems*, HPCC '12, pages 375–380. IEEE Computer Society, 2012.

T. Rompf, A. K. Sujeeth, K. J. Brown, H. Lee, H. Chafi, and K. Olukotun. Surgical precision jit compilers. In *Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '14, pages 41–52. ACM, 2014.

B. C. Smith. Reflection and semantics in Lisp. In *Proc. of ACM Symp. on Principles of Programming Languages*, pages 23–35, 1984.

T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One vm to rule them all. In *Proc. of ACM Onward! 2013*, pages 187–204. ACM, 2013.

Y. Yan, M. Grossman, and V. Sarkar. JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In *Proc. of the 15th Int. Euro-Par Conf. on Parallel Processing*, Euro-Par '09, pages 887–899. Springer-Verlag, 2009.

W. Zaremba, Y. Lin, and V. Grover. JaBEE: framework for object-oriented Java bytecode compilation and execution on graphics processor units. In *Proc. of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 74–83. ACM, 2012.