Deeply Reifying Running Code for Constructing a Domain-Specific Language

Shigeru Chiba
Graduate School of Information Science and Technology
The University of Tokyo
chiba@acm.org

YungYu Zhuang
Graduate School of Information Science and Technology
The University of Tokyo
zhuang@csg.ci.i.u-tokyo.ac.jp

Maximilian Scherr
Graduate School of Information Science and Technology
The University of Tokyo
scherr@csg.ci.i.u-tokyo.ac.jp

ABSTRACT

This paper presents *deep reification*, which is a language mechanism for reflective computing. It reifies a self-contained partial snapshot of the current execution environment. The snapshot contains not only data but also code and type definitions. This mechanism can be used as a common component of execution systems of embedded domain-specific languages (DSLs). Unlike typical implementations of embedded DSLs, the mechanism enables a DSL to borrow the host-language's syntax yet execute under a different semantics on different platforms from the host language's. DSL implementation can allow programmers to first construct a function closure as DSL code, reify a snapshot necessary for executing the function closure, transform the code in the snapshot into a program for a target platform, and finally execute the program. As a prototype system of deep reification, we have implemented *Bytespresso* for Java. This paper shows Bytespresso and also *Bytespresso-C*, our DSL built on top of Bytespresso. The target of this DSL is numerical computing on cluster computers and GPUs.

CCS Concepts

•**Software and its engineering** → **Translator writing systems and compiler generators; Domain specific languages;** *Abstraction, modeling and modularity; Object oriented languages;*

Keywords

Reflection, Meta programming, Embedded DSL, Parallel computing.

1. INTRODUCTION

When writing a program for large-scale scientific computing, or high-performance computing (HPC), execution performance has been often its primary concern. As the system architecture of modern high-performance computers is getting more complicated, however, programming techniques for achieving the best execution performance are getting also complicated. Using rich abstractions when writing a program would be a solution to avoid convoluted programming but the use of rich abstraction itself often involves a severe performance penalty. This is true if the abstraction is implemented as a library in a general-purpose language.

Domain-specific languages (DSLs) are a promising option to address this problem. A DSL for HPC can provide such rich abstraction as built-in language constructs and apply a domain-specific compiler-optimization to it. Since the development cost of DSLs is a drawback, a category of DSLs called *embedded DSLs* [16] are widely used. An embedded DSL is a DSL that is embedded in the host language and uses a large part of the host language infrastructure such as the syntax and the type system. To reduce the development cost, thus the DSL author can reuse the development tools of the host language such as an editor. By reusing the parser of the host language, she does not have to implement the parser of her DSL.

For domain-specific optimization, the DSL author has to be able to implement the transformation of the DSL code. Deep embedding is a technique that enables the extraction of the intermediate representation (IR) of DSL code written in an embedded DSL. The DSL author can transform the extracted IR for optimization. In deep embedding, DSL code is treated as regular host-language code but, when it runs, it constructs the IR of the DSL code, transforms it for optimized execution, and finally executes it. An example of the IR is an abstract syntax tree. The IR transformation gives DSL authors a chance of performance optimization. A drawback of deep embedding is that the DSL users are aware of two-stage computation; the IR construction and transformation and the execution of it. The representation of the DSL code is not satisfactory due to this awareness.

This paper presents an alternative approach, we name *deep reification*. It is a reflection mechanism for extracting the IR. Reification is an operation of reflective computing [31]. It extracts and converts entities in an execution environment (or meta-level entities) into program-manipulatable values (or base-level entities) [10].

Our deep reification reifies the IR of a code block typically given as a function closure. DSL authors (who implement a DSL compiler) exploit this mechanism to retrieve DSL code embedded in a host language and compile it into a program

PPPJ '16, August 29-September 02, 2016, Lugano, Switzerland

DOI: http://dx.doi.org/10.1145/2972206.2972219

running on a target platform. The DSL code is expressed with the host-language syntax. It can contain any host-language expression/statement including a method call. If a method is called, the body of the invoked method is regarded as part of the DSL code. Since the host-language syntax is borrowed, the DSL code is naturally embedded in its host-language code. However, the semantics of the DSL code is given by the DSL compiler although it is expected to reuse a large part of the host-language semantics. To support DSL compilation, the deep reification extracts a self-contained partial snapshot of the current execution environment. It contains not only the IR of the extracted code but also that data and types that the code refers to.

The design of the deep reification is illustrated through *Bytespresso*, our prototype system for Java. This paper also presents a DSL built on top of Bytespresso. This DSL, called *Bytespresso-C*, is designed for performance-sensitive numerical computing on various platforms such as GPUs and cluster computers. The DSL code is compiled into a C or CUDA program and then native binary code. The paper presents that a program expressed in this DSL can exploit high-level abstraction while showing good performance comparable to a program written by hand in C or CUDA. Since the deep reification implies runtime overheads, it is not appropriate for DSL programs that are relatively short and repeatedly invoked by the host-language code. The deep reification best fits standalone, largely self-contained DSL programs, which run for sufficiently long time with minimal data exchanges with the host-language code.

2. MOTIVATION

The recent advancement of parallelization hardware such as GPUs and complex memory hierarchy for hiding access latency are rendering performance-conscious programming extremely difficult for average scientists. Introducing richer abstraction, for example, by object orientation should be a promising solution but existing general-purpose object-oriented languages do not show acceptable performance. For example, the all-pairs approach to N-body simulation is to compute all pair-wise interactions among the N bodies and its kernel computation is represented by the following formula [21]:

$$\boldsymbol{a_i} = \textstyle\sum_j w_j(\boldsymbol{r}\cdot\boldsymbol{r}+\varepsilon^2)^{-\frac{3}{2}}\boldsymbol{r} \quad \text{where } \boldsymbol{r} = \boldsymbol{p_j} - \boldsymbol{p_i} \tag{1}$$

$$\boldsymbol{v_i'} = \delta(\boldsymbol{v_i} + \Delta t \cdot \boldsymbol{a_i}) \tag{2}$$

$$\boldsymbol{p_i'} = \boldsymbol{p_i} + \Delta t \cdot \boldsymbol{v_i'} \tag{3}$$

Here, $\boldsymbol{p_i}$ is the position of the i-th body and $\boldsymbol{v_i}$ is its velocity. w_i is the mass of its body. $\boldsymbol{p_i'}$ and $\boldsymbol{v_i'}$ are the values at the next time step.

A program for this simulation would be simplified and easy to maintain if it is written with abstractions such as vector types and map functions. Listing 1 shows a Java program computing the formula above by using these abstractions. Vec3 is a class for three-dimensional vectors. Vec4Array is a class for arrays of four-dimensional vectors (used as an array of bodies $[x_i^{(p)}, y_i^{(p)}, z_i^{(p)}, w_i]$). Vec3 provides methods for basic arithmetic such as add and sub (subtraction). It also provides mult for scalar product and scale for scalar multiplication. Vec4Array provides the sum and map methods as well as getters and setters. Line 9 to 13 compute the formula (1), line 14 and 15 compute (2), and line 16 computes (3). The variables softening, damping, and

```
1 final Vec4Array pos1 = ...
2 final Vec4Array pos2 = ...
3 final Vec4Array vel = ...
4 final float softening = 0.01f;
5 final float damping = 1.0f;
6 final float deltaTime = 0.016f;
7
8 final Func f = (Vec4Array pos, int i, Vec3 pi, float wi) -> {
9   Vec3 a=pos.sum((Vec4Array p,int j,Vec3 pj,float wj)->{
10    Vec3 r = pi.sub(pj);
11    float ra = VecDSL.reciprocalSqrt(r.mult(r) + softening);
12    return r.scale(wj * (ra * ra * ra));
13  });
14  Vec3 v2=vel.get(i).add(a.scale(deltaTime)).scale(damping);
15  vel.set(i, v2);
16  return pi.add(v2.scale(deltaTime));
17 };
18    :
19 pos2.map(f, pos1);
```

Listing 1: N-body simulation in Java

deltaTime correspond to constants ε, δ, and Δt, respectively. Line 19 applies the function f to every element of an array of bodies pos1 and stores the resulting values into pos2. It implements a single step of the simulation. Note that an expression such as `(Vec4Array p, ...)->{...}` is a function object called a lambda expression in Java. `(Vec4Array p, ...)` specifies function parameters and `{...}` does a function body. The sum method in line 9 receives a function object, which implements the expression $w_j(\boldsymbol{r}\cdot\boldsymbol{r}+\varepsilon^2)^{-\frac{3}{2}}\boldsymbol{r}$, and computes the sum of that expression. Some parameters such as p in line 9 are not used. This is because the same type Func is used for both map and sum.

The abstractions such as Vec4Array are not only for simplifying code but also useful for hiding parallel computing under the hood. However, the abstractions often involve extra data copying and indirections to access data. Even a modern optimizing compiler has difficulties in eliminating performance penalties due to the abstractions.

Domain Specific Languages

Domain-specific languages (DSLs) are an option to enable rich abstraction with acceptable execution overheads. They provide abstractions for their specific domains. The programs using the abstractions are compiled by their compilers with domain-specific optimization. An example is Physis [20], which is a DSL for parallel stencil computation. A drawback of this approach is their development costs. Developing a compiler, an intelligent editor, and other development tools are relatively expensive. The development cost of a DSL can be paid off only if we can expect a large user base of that DSL.

To minimize the development costs of such DSLs, *deep embedding* is a well-known approach. In a deeply embedded DSL, the DSL code is embedded in a host-language program and the syntax of the DSL is directly mapped into language constructs such as method calls in the host language. The DSL code is compiled and executed as part of the host-language program within the confines of the host-language semantics. No new parser is needed. The existing tool chain for the host language is used. Such deeply embedded DSLs are also called library-based DSLs.

With deep embedding, a language construct that the DSL

syntax maps to does not interpret the semantics of the DSL but extracts the intermediate representation (IR). This provides opportunities for optimized execution and sets it apart from its simpler counterpart, *shallow embedding*. For example,

```
v2 = vi.add(a.scale(t)).eval();
```

this DSL code deeply embedded in Java represents vector arithmetic $\boldsymbol{v_2} = \boldsymbol{v_i} + t \cdot \boldsymbol{a}$. Here, add and scale are Java methods. They do not immediately perform addition or scalar multiplication but they construct the IR, such as an abstract syntax tree, for these arithmetic operations. So the execution of the expression vi.add(a.scale(t)) extracts the IR of this expression. The computation of $\boldsymbol{v_i} + t \cdot \boldsymbol{a}$ is postponed until the eval method is invoked on the IR. The eval method compiles the IR under the *DSL-specific* semantics and executes it. For example, the method may generate a CUDA program computing the arithmetic represented by the IR and run it on a GPU. It may interpret the IR on the JVM. Then it *materializes* the resulting value into a host-language Java value to return.

A drawback of deep embedding is that the IR extraction is explicit and DSL users are aware of it. Since the computation is split into two stages, the IR construction/transformation and the IR execution, the DSL users have to explicitly call eval to start the second stage. They also have to be conscious that the return types of add and scale are the type representing the IR, for example, Vec3IR although the type of v2 is a type representing a vector value such as Vec3. If vi or a refers to not the IR (a Vec3IR value) but a vector value (a Vec3 value), it has to be explicitly lifted up to the IR. The DSL user has to write something like cnst(a).add(a.scale(t)). cnst is a method for converting a vector value into the IR representing that value as a constant.

Another drawback of deep embedding is that it has difficulty in supporting DSL-level procedure abstraction. To extract the IR of the code that is related but located apart, the DSL authors and users have to manually implement the extraction. Suppose that $\boldsymbol{v_i}$ in the example above is computed by a procedure getVi defined in the DSL. Then the DSL users might write the following DSL code:

```
v2 = getVi().add(a.scale(t)).eval();
```

However, this does not actually amount to DSL-*internal* procedure abstraction. Here, getVi is not a DSL procedure but a Java method. If getVi returns the IR of the expression computing $\boldsymbol{v_i}$, that IR will be directly expanded in the IR constructed by the code above. To avoid this macro-like expansion, the DSL author would have to provide a Java method constructing the IR representing a procedure call and the DSL users would have to use it and also pass to the eval method the IR of the definition of the procedure. The DSL-user code would be something like this:

```
v2 = call("getVi").add(a.scale(t)).eval(defGetVi);
```

Here, defGetVi is a variable referring to the IR of the procedure definition. This is less simple and natural than the DSL code shown first.

A more user-friendly form of deep embedding is found in Lightweight Modular Staging (LMS) [26], which enables the construction of a new deep embedding library by modularly combining existing ones. LMS relies on Scala's type system and advanced features such as user-defined implicit conversion (and often Scala-Virtualized compiler extension [27]) and thereby it allows deeply embedded code to have more natural representation. Types such as Vec3 and Vec3IR do not seem to play a big role since they are hidden behind type inference. However, it is exactly this type inference that determines which part of program is contained in the extracted IR. The users still may have to be aware of the IR extraction.

Yin-Yang [18] proposed to use Scala's macros for hiding the LMS-style IR extraction and avoiding abstraction leaks in LMS. Programmers do not have to be aware of the type representing the IR. However, these approaches depend on the power of Scala's type system as well as compile-time meta-programming capabilities.

3. DEEP REIFICATION

This section presents our reflection mechanism named *deep reification*. As in deep embedding, DSL authors can use deep reification to extract DSL code expressed with the host-language syntax, transform it into a program implementing the semantics of their DSLs, and execute it. The program after the transformation may be expressed in a different language from the host one. Unlike deep embedding, DSL code can exploit more language constructs of the host language. In case of Java as the host-language choice, the DSL code can call a method and instantiate a class as normal Java code does although the semantics of method calls and object instantiation is still left to the DSL authors.

Deep reification dynamically extracts a self-contained snapshot of the current execution environment for a given code block, which we below call a *root* code block. The snapshot is not a whole environment; it is rather a necessary subset for executing the given root code block. It contains the intermediate representation (IR) of the code, the type definitions, and the objects that will be used/accessed during the execution of that root code block. Here, the code includes what is indirectly invoked from the root code block. This is the origin of "deep" in "deep reification". If the root code block is a method, the extracted code includes all the methods invoked during the execution of that root method.

Since the extracted snapshot contains the IR, deep reification can be used to extract DSL code by giving the DSL code as a root code block. The DSL code can contain method calls. The code of a called method is also extracted although deep reification allows delimiting the trace of a method-call chain. Some method calls should be considered as the invocation of DSL primitives and, if they are primitive invocations, the code of the called methods should not be extracted.

Deep reification is similar to object serialization or marshalling since object serialization also extracts a self-contained object graph for restoring it on a remote 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, or obtainable on demand, which in turn incurs additional overhead and additional complexity in the runtime systems of both the source and target platform.

3.1 Bytespresso

Bytespresso is our prototype of the deep reification mechanism in Java. It is a Java library providing a Reifier class for deep reification. For example, the following code extracts a snapshot for method m under consideration of invocation

with arguments args:

```
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 body[1] used as the root code block of the extraction and args refers to the runtime values of the arguments passed to that method body. The snap method 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 the IR of not only the body of the method m but also other methods' that may be invoked during the execution of the method m. The snapshot also contains a table of classes referred to in the IR. Furthermore, it contains a table of objects accessed through static fields as well as the objects passed as args.

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

```
double[] p = ...;
double[] q = ...;
new GpuDSL().compile(() -> {
    double r = dotProduct(p, q);
    Util.print(r);
});
```

The compile method in the GpuDSL class takes a lambda expression, which is regarded as DSL code, and compiles it. Its implementation would be as follows:

```
void compile(Runnable f) {
    CtMethod m = getStartMethod();
    Object[] args = { f };
    Reifier reifier = new Reifier(m, args);
    Snapshot image = reifier.snap();
    compileAndRun(image);
}
  :
static void start(Runnable f) { f.run(); }
```

It first obtains the method object representing start by calling getStartMethod. Then it performs deep reification by snap with that method object. The respective IRs of the bodies of the start method, the lambda expression f, Util.print, and dotProduct (and others called by dotProduct) are extracted as well as the current values of p and q. They are passed to the DSL compiler compileAndRun, which will generate a CUDA program.

Note that dotProduct is regarded as a DSL method although it is expressed elsewhere in Java (recall that deep embedding does not enable this as mentioned in Section 2). The values of p and q are initialized before the DSL compilation. Since only the result of the initialization is passed to the DSL compiler, the DSL users can naturally describe staged computation: the first stage is computed on the JVM while the second stage is on a GPU.

[1]The variable m does not refer to java.lang.reflect.Method. It refers to a CtMethod object provided by Javassist bytecode engineering library [5]

3.2 Extracting a snapshot

The standard reflection mechanism of Java enables the extraction of data from the current execution environment. Although it cannot be used for extracting code, there are a few techniques for obtaining code. To implement deep reification in Java, Bytespresso obtains bytecode by locating and reading a class file with existing facilities. This approach was chosen to maximize compatibility with current and older versions of Java. A limitation of this approach is that all bytecode has to be stored in class files. So if a program uses a lambda expression, -Djdk.internal.lambda.dumpProxyClasses option has to be given to the JVM running Bytespresso.[2] It forces the JVM to dynamically produce a class file containing the bytecode of a lambda expression.

Bytespresso uses an abstract syntax tree as the IR of extracted code. After reading bytecode, Bytespresso decompiles the bytecode to construct an abstract syntax tree of the method body that the bytecode represents. Since the decompilation may not reconstruct all Java statements, an abstract syntax tree obtained by Bytespresso does not exactly match the original Java source. Bytespresso reconstructs only expressions and control structures. 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.

A snapshot extracted by deep reification includes method bodies that will be indirectly invoked from a root code block given to the mechanism. Since Java supports dynamic method dispatch, Bytespresso conservatively collects all method bodies possibly invoked. The collected method bodies are recorded for each invocation site in the IR. The basic algorithm resembles RTA [2]. A snapshot $\{M, T, Obj\}$ is extracted in the algorithm sketched below. M is a set of method bodies, T is a set of types, and Obj is a set of objects. Suppose that C is a set of classes that may be instantiated during the execution of a root code block given to deep reification. When deep reification is applied to a method body mbody with arguments args, we record mbody in M and run the following algorithm:

1. Add each object v in args to Obj. Add the actual type of v to C and T. If v contains a reference to another object w, also add w and the type of w and recursively do the same on references contained in w.
2. For each expression in mbody,
   - 2.1. If an expression is new D(...) (instantiation of a class D), then add D to C and T.
   - 2.2. If an expression is p.f (reading a field f of an object p), then add the static types of p and f to T. If f is a static field, obtain the current value v of f by reflection and do the same as in 1 on v.
   - 2.3. If an expression is an assignment to p.f (writing to a field f of an object p), then add the static types of p and f to T.
   - 2.4. If an expression is a method call p.m(...) (call a method m on an object p), add the static type s of p is T. For every type t in C, if t is equal to

[2]Other techniques, such as intercepting class loading, could be used alternatively. Also, future versions of Java may support a more direct way of obtaining this information [24].

s or a subtype of s, then record in M a method body of m declared in t. Do the same as in 2 on that method body.

2.5. Otherwise, if an expression contains a type t, then add t to T. For example, if an expression is a type cast, then add the type to T.

When a type t is added to T, all the super types of t are also added to T.

3. Repeat 2 until no new class is added to C.

Due to this algorithm (and inherent limitations) Bytespresso is not able to support invocation of the Java reflection API from DSL code to be reified.

Since Bytespresso performs partial evaluation, a more specific type of p may be statically determined at step 2.4. If such a type is found, then it is used instead of the apparent static type s. Details of the partial evaluation by Bytespresso are mentioned later.

Delimiting an extracted snapshot is a significant issue of deep reification. Some methods might be treated as DSL primitives and thus the IR of their method bodies might not be necessary. Step 2 of the algorithm above is skipped if a method body mbody is abstract or native. Step 2 is also skipped if a method body is annotated with @Native by DSL authors or users.

3.3 IR construction

Bytespresso uses abstract syntax trees (ASTs) as the IR of the extracted code. Their tree nodes are instances of ASTree or its subclasses. Since they support the Visitor pattern [12], DSL compilers can perform code generation by traversing trees.

Some DSL compilers may exploit annotations that their users attach in DSL code. For supporting DSL compilation considering such annotations, Bytespresso recognizes a few annotations and constructs ASTs reflecting them. For example, if an extracted method body has an annotation @Metaclass, Bytespresso changes the class for an AST node representing that method body.

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

This specifies that an AST node for the body of the gpuKernel method is created by a factory object CUDA.Global.instance. DSL users can rely on this feature to control the semantics of DSL code. For example, in some DSL for GPUs, the gpuKernel method would become a __global__ function in CUDA. It is a function running on GPUs.

An @Metaclass annotation is propagated to methods directly or indirectly called from the method with @Metaclass. The AST nodes for those methods are also created by the factory object specified by @Metaclass. This feature is useful to implement a DSL for GPUs since methods have to be classified in CUDA into __host__ functions running on CPUs and __device__ functions running on GPUs. The propagation of @Metaclass automates this classification.

Bytespresso also recognizes the @Native annotation. It is primarily used for delimiting a snapshot but it takes a String argument. A DSL compiler can use this argument for code generation. For example, a DSL compiler may transform the following Java method:

```
@Native("float* p; cudaMalloc(&p,sizeof(float)*(v1+2));"
      + "return p;")
public static float[] mallocFloat(int size) {
   return new float[size];
}
```

into a C function as follows:

```
float* CUDA_mallocFloat_1(int v1) {
   float* p; cudaMalloc(&p,sizeof(float)*(v1+2)); return p;
}
```

The function body is as specified by the argument to @Native. The parameter size in Java is v1 in C. The original body of the Java method is ignored. This feature allows DSL authors to conveniently implement a primitive of their DSLs.

The @Metaclass annotation can be attached to a class declaration. Such an annotation changes the class for an object representing the class definition. For example,

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

Bytespresso creates an instance of ImmutableClass and puts it as the object representing the Vec3 class in the class table of an extracted snapshot. DSL users can specify that the Vec3 class has custom semantics and a DSL compiler can change code generation according to this annotation.

3.4 Optimized extraction

Bytespresso extracts method bodies under the assumption that the semantics of method calls in a DSL is mostly the same as in Java. During snapshot extraction, Bytespresso performs partial evaluation [9, 11]. It basically does constant propagation and folding by using runtime values available at extraction time. Code specialization per method body is also performed so that Bytespresso can have more opportunities of devirtualization [1].

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 lambda expression is implemented as a final field of its object. Bytespresso also recognizes that the values in args passed to the constructor of Reifier are constant values. A reference that a new expression evaluates to is also a constant value. If a variable is initialized with a constant value and does not change its value, the variable is recognized as having a constant value (*i.e.* effectively final in the Java specifications). An AST node representing that variable holds that constant value.

Furthermore, the value of a field with the @Final annotation is recognized as a constant value. Such a field is treated as if it holds a constant value while DSL code is executed. The value of a @Final field may be freely modified in Java. However, if the DSL code extracted by deep reification is found to attempt to update the value of a @Final field, Bytespresso reports an error.

Devirtualization and code specialization

The propagated constant values are used for devirtualization [1]. If a constant value is a reference to an invocation target

of method call, then the method call is devirtualized. The method body that the devirtualized method call invokes is then specialized with the invocation target and the arguments of constant values (so these are regarded as constant values within the specialized body). A new AST for the specialized method body is constructed so that further devirtualization can be performed. The AST node representing the devirtualized method call refers to the constructed new AST. If devirtualization succeeds, only the specialized method body is contained in a snapshot extracted by the deep reification.

Bytespresso also performs dead code elimination. This mechanism can be used to emulate the preprocessor directive `#if` in C by an `if` statement referring to a `static final` field of `boolean` type in Java. If a constant value is a value of branch condition of `if` statement, its not-executed then or else block is eliminated from an extracted AST as dead code. A method call in the eliminated block is not examined to extract other method bodies. 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`.

Code specialization implies a risk of code explosion. To avoid excessive specialization and infinite regression due to recursive calls, if the number of specialized instances of the same Java method body reaches a threshold, Bytespresso stops specialization for that method body. For the same reason, arguments of primitive types such as `int` are not considered for the specialization. Only arguments of reference type are considered.

Inlining

Bytespresso also supports the `@Inline` annotation attached to a method body. If a method body is annotated with `@Inline` and if possible, before constructing an AST, Bytespresso lexically substitutes the body of that method for a method-call expression to that method. Inline code expansion is recursively applied to all method-call expressions within the body of that method with `@Inline`. If a DSL compiler invokes a back-end compiler such as a C compiler, explicit inline expansion by Bytespresso may help the back-end compiler perform more aggressive optimization with deep code analysis. If specified, Bytespresso attempts to specially transform an object passed as an argument to the inlined method. The object is transformed into a set of variables of primitive types. Each variable holds a value of the corresponding field of that object. This may also help optimization by a back-end compiler.

3.5 Other languages

Deep reification can be implemented for other languages than Java if several primitive mechanisms are available. First of all, the language has to provide a reflection (reification) mechanism for obtaining the type and the field values of any given objects at runtime. The definitions of the obtained types have to be also accessible. Method (or function) bodies included in the type definitions need to be extractable. The extracted bodies have to hold sufficient information to reconstruct the IR of the bodies. The ability to faithfully reconstruct the original source code is not required. Deep reification only needs to be able to reconstruct method-call chains. Hence, it may choose low-level IR that is rather similar to machine code.

```
1 final VecDSL dsl = new VecDSL(N);
2 final Vec4Array pos1 = dsl.array();
3 final Vec4Array pos2 = dsl.array();
4 final Vec4Array vel = dsl.array();
5
6 final float softening = 0.01f;
7 final float damping = 1.0f;
8 final float deltaTime = 0.016f;
9 final Func f = (Vec4Array pos, int i, Vec3 pi, float wi) -> {
10  // same as in Listing 1
11 };
12
13 dsl.run(() -> {
14  pos1.tabulate(i -> new Vec4(i, i, i, 2));
15  vel.tabulate(i -> new Vec4(i, i, i, 2));
16  dsl.repeat(R, () -> {
17    pos2.map(f, pos1);
18    pos1.map(f,  pos2);
19  });
20 });
```

Listing 2: N-body simulation in our DSL

4. EXPERIMENTS

To examine the idea of deep reification and Bytespresso, we have developed an embedded DSL for Java. The target domain of this DSL, which we named *Bytespresso-C*, is numerical computing on various platforms including a cluster computer and GPUs. It is a Java class library built on top of Bytespresso. The DSL users write their DSL code in the form of a lambda expression. Bytespresso-C extracts the code and the related data by the deep reification and generates a standalone[3] platform-dependent program in C or CUDA from the extracted code and data. Java objects accessed from the extracted code are translated into global variables of `struct` type in C so that it will be statically allocated for optimization. Updates of these objects in the DSL code are not written back to the original Java objects.

The DSL code is expressed with Java syntax and executed mostly under the Java semantics, as Java objects, methods, and types. So the DSL is fairly compatible with Java. Only a few methods and classes are treated with the DSL-specific semantics so that they can serve as primitive entities abstracting platform-native facilities. Furthermore, array boundary checking and exception handling are not available in the DSL. The multi-threading of Java is not supported as well as the standard Java library. Hence parallel computing has to be described with the primitives provided by the DSL.

4.1 N-body simulation

We first show our vector library in Bytespresso-C. Listing 2 is the N-body simulation written with this library. It is equivalent to Listing 1. Line 14 to 19 is the DSL code. All the classes `VecDSL`, `Vec4Array`, `Vec3`, and `Vec4` are library classes. The `tabulate` method in line 14 and 15 initializes array elements by the given lambda expression. Note that the DSL can naturally call this method regarded as part of the DSL code.

[3]The DSL code can send a resulting value back to Java code but this feature is out of the scope of this paper.

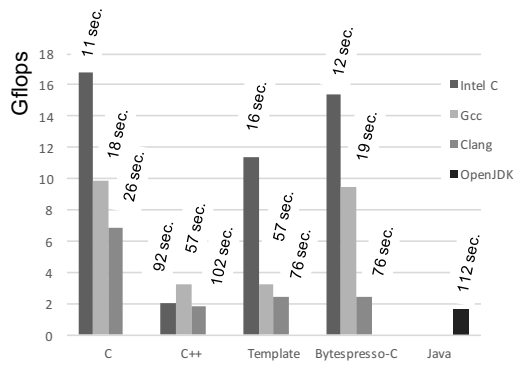

Figure 1: The execution performance of N-body simulation

```
1 final Vec4CudaArrayOnShared shared = dsl.sharedMemory();
2 final Func f = (Vec4Array pos, int i, Vec3 pi, float wi) -> {
3   Vec3 a = shared.fold(pos, acc -> {
4     Vec3 sum=shared.sum((Vec4Array p,int j,Vec3 pj,float wj)->{
5       Vec3 r = pi.sub(pj);
6       float ra = VecDSL.reciprocalSqrt(r.mult(r) + softening);
7       return r.scale(wj * (ra * ra * ra));
8     });
9     return acc.add(sum);
10  });
11  Vec3 v2 = vel.get(i).add(a.scale(deltaTime)).scale(damping);
12  vel.set(i, v2);
13  return pi.add(v2.scale(deltaTime));
14 };
```

Listing 3: N-body simulation targeting CUDA

The arrays pos1 and vel are created beforehand in line 2 and 4, which is out of the DSL code, since Listing 2 implements staged computation. The DSL compiler treats these arrays as constant values exploited by partial evaluation. In this DSL, the more objects are created beforehand in Java code, the better chances of optimization the DSL compiler has.

The repeat method in line 16 repeatedly calls the lambda expression given as the second argument. It calls the lambda expression R-times. Although the for statement can substitute repeat because the DSL borrows most Java syntax, we use repeat so that we can reuse this code in a later example.

A Vec4Array object contains only a final field referring a FloatArray2D object, where array elements are stored. FloatArray2D is a primitive class provided by Bytespresso-C and it is not subject to the Java semantics. Its behavior is customized by Bytespresso-C to emulate a native array in C. A call to its getter or setter method is transformed into a direct access to native array in C. Vec3, Vec4, and Func objects are immutable. The DSL compiler transforms them into stack-allocated values of struct type in C. They are passed not by reference but by copy. Hence no heap memory is allocated for objects and no garbage collection is necessary while running Listing 2. Although all those classes are expressed with Java syntax, the compilation of their code is customized by@Metaclass.

Figure 1 shows the execution performance of Listing 2. We measured the execution time of line 16 to 19. For comparison, we also ran several programs compatible to Listing 2. All the programs are single-threaded and they simulated 30208 bodies for 10 time-steps. The execution time is the average across 5 runs. C denotes a manually implemented C program for reference using bare float arrays without higher-level abstraction such as Vec4Array type. No further optimization techniques like loop tiling are applied to the program. The code size is about 90 lines. C++ denotes a C++ program that is straightforward translation of Listing 2 from Java to C++. All the overridable methods are virtual functions. Vec3 objects are not allocated on the heap but the stack. Template denotes a C++ program optimized by template metaprogramming while preserving the original abstraction. Java denotes the Java program in Listing 2 that was run without Bytespresso-C but with the same DSL classes written in pure Java. The C/C++ programs were compiled by Intel C compiler (version 15.0.0.090) with -fast, GCC 4.8.2 with -Ofast, or Clang 3.4.2 with -Ofast. The Java program was executed by OpenJDK (version 1.8.0_40). All the programs were run on CentOS 6.2 on dual Intel Xeon E5-2687W processors (Sandy Bridge EP, 8 cores, 3.1GHz).

Bytespresso-C achieved good performance comparable to the C program when compiled with the Intel C compiler or GCC. However, the Clang-compiled version did not match up to this. The figure shows that the overall performance significantly depends on a back-end compiler, probably how it effectively applies SIMD vectorization to the programs. The code generated by Bytespresso-C contains struct types for representing vectors and lambda expressions. This poses an obstacle to achieving performance on par with the hand-written C program. Bytespresso-C cannot compile these abstractions away to be bare primitive types.

The DSL compilation was relatively small. It took about 400 msec. (including 300 msec. by the C compiler) with the Intel C compiler.

4.2 GPU computing

Our vector library supports GPU computing in CUDA. To run Listing 2 on a GPU, only the first line is modified to instantiate VecCudaDSL, which is a subclass of VecDSL. The array method in this subclass instantiates Vec4CudaArray to create a vector array (line 2 to 4 in Listing 2) and the map function in Vec4CudaArray runs in parallel by invoking a CUDA kernel function (line 17 and 18). One of the sources of complexity of CUDA programming is memory copying between the host and the GPU. In our vector library, the memory copying is hidden in the repeat method in line 16 in Listing 2, which is overridden in VecCudaDSL.

To support the implementation of these classes, Bytespresso-C provides several Java methods for calling CUDA library functions. In the CUDA code generated by Bytespresso-C, these methods become C functions that call the CUDA library functions when they are invoked. Bytespresso-C also provides the @Metaclass annotation for implementing __global__ or __device__ functions in CUDA. A Java method with that annotation is transformed into such a function in CUDA.

Listing 2 can be modified to use the shared memory (fast local memory) in CUDA. Listing 3 shows the lambda expression f after the modification. The variable shared refers to the object abstracting the shared memory. Bytespresso-C treats this object with special semantics when generating CUDA code. The fold method on this object first divides

```
1 Boundary b = new FixedEndBoundary();
2 b.initializer(new Initializer() { ... });
3 GridFloat2D grid = new CpuGridFloat2D(xsize, ysize, b);
4
5 Initializer init = new Initializer() {
6     public float value(int i, int j) { return 273.15f; }
7 };
8
9 Kernel k = new Kernel() {
10    public float newValue(Float2Array oldValue, Cursor cur,
11                          int t, Reduction r) {
12        float v = c0 * (cur.north(oldValue) + cur.south(oldValue))
13                + c1 * (cur.east(oldValue) + cur.west(oldValue))
14                + c2 * cur.self(oldValue);
15        return v;
16    }
17 };
18
19 grid.initialize(init)
20     .each(Reduction.NO, 1, Predicate.FOREVER, k)
21     .repeat(N, new StdDriver());
```

Listing 4: 2-dimensional 5-point stencil

the elements of the first argument pos into a number of chunks. The chunk is copied onto the shared memory one at a time and the function given as the second argument is invoked with the chunk on the shared memory. Copying onto the shared memory is executed in parallel with neighbor threads. Finally, the fold method accumulates the values returned from the function for every chunk.

We ran the program on NVIDIA Tesla K20m with CUDA 6.5. We compiled it by nvcc 6.5.12. We also compared its performance with the sample code distributed with CUDA Toolkit 6.5. It was written with C++ templates in CUDA to demonstrate the execution performance of CUDA [21] but its program structure was not modular or using rich abstraction. The execution time of the program in Bytespresso-C was 134 msec. and it corresponds to 1.36 tera flops (we counted 20 flops per pair-wise force calculation). The execution time of the sample code from NVIDIA was 122 msec. and it corresponds to 1.5 tera flops. The overhead due to the abstraction of our library was 10%.

4.3 Stencil computation

We wrote a simple framework library for stencil computation and compared its execution speed with equivalent programs written in C, C++, C++ template, and Java. Stencil computation is a popular model for solving a partial differential equation. Listing 4 lists a program written with a two-dimensional version of our framework implemented with Bytespresso-C. It performs five-point stencil computation in single precision. All the classes are provided by the framework. The program creates several objects representing the configuration such as a boundary condition, how to initialize, and a kernel function. Then these objects are assembled in the grid object. This framework regards init and k as the DSL code and starts compilation when the repeat method is called in line 21.

The variable k represents a stencil kernel. It is repeatedly applied to update every element of a regular Cartesian grid until the system reaches a solution. The Cursor object given to the kernel function is used to access the elements adjacent to the current point. It hides implementation details of the

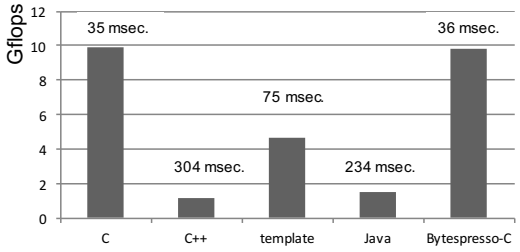

Figure 2: The execution performance of 2D-stencil computation

traversal on the grid and the boundary condition. For example, cur.north(oldValue) reads a point north of the current point. To avoid a conditional branch at runtime, the framework may switch a concrete class of the Cursor object for the points near the border. A Cursor object is a stack-allocated value in the DSL.

Figure 2 shows the execution performance of the stencil computation. Our framework showed comparable performance to the equivalent C program. The grid size was 5000 by 10000. The programs compute till 100 time-steps and we ran the programs five times. The execution time shown in the figure is the average execution time of the single time-step. The programs ran on the same machine in Section 4.1 and we used Intel C compiler. All the programs are single-threaded.

4.4 Distributed computing

The next experiment is on the support to MPI by Bytespresso-C. We also wrote a framework for three-dimensional stencil computing. A program with this framework runs on a supercomputer consisting of distributed nodes communicating via MPI. From the framework users' viewpoint, this framework is not much different from the framework in Section 4.3. The framework users are not necessarily aware of inter-node communication. All the communication via MPI is hidden in the implementation of the framework. We wrote the framework by using MPI primitives provided by Bytespresso-C. They are Java methods that are transformed by the DSL compiler into MPI functions in C.

To examine this framework, we rewrote the Himeno benchmark [15] to use this framework. The resulting benchmark program in Bytespresso-C was similar to Listing 4. This benchmark runs a single three-dimensional Jacobi kernel. The problem size was XL ($512 \times 512 \times 1024$). The original benchmark is written in C with MPI and each MPI process is single-threaded in single precision. The benchmark score mainly reflects the memory bandwidth.

We ran the programs on the TSUBAME 2.5 supercomputer at Tokyo Institute of Technology. The maximum number of nodes we used was 256. Every node had one MPI process whereas it had more than one process when the total number of the processes is more than 256. Each node has dual Intel Xeon X5670 processors (Westmere EP, 6 cores, 2.93GHz). The nodes are connected via QDR InfiniBand network. We used Intel C compiler (version 14.0.2.144) and OpenMPI 1.8.2. The compile option was -Ofast. When the node size is less than 8, the -mcmodel=medium option was also given.

Figure 3 shows the results. We counted 34 floating-point

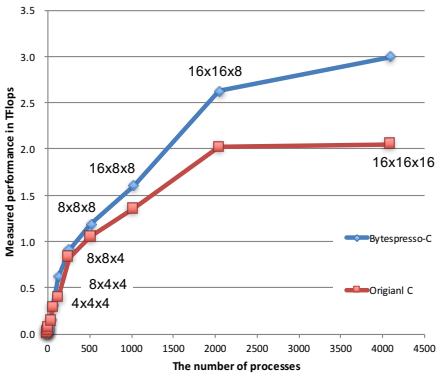

Figure 3: Strong scale performance of the Jacobi kernel of Himeno on TSUBAME2.5

operations per computation on each grid element for all the programs. The labels such as 8x8x4 denote the decomposition of the grid. The performance difference between the original C program and the Bytespresso-C program would be due to the difference in the memory layout of the grid data. The original C program used a four-dimensional array while the Bytespresso-C program used several three-dimensional arrays. Both programs used Cartesian-related MPI functions and the vector types of MPI.

4.5 Comparison to DSL

Since stencil computation is one of typical computation patterns seen in numerical algorithms, there are a number of DSLs for stencil computation. We compared our stencil framework on Bytespresso-C with Physis, an external DSL for stencil computation [20]. Since Physis provides a custom compiler, we expected its potential of better performance. In the following experiments, we used the programs included in the distribution of Physis[4] as the version written in Physis.

Figure 4 shows the execution performance of the Himeno benchmark on a single CPU. For this experiment, we modified our framework to apply the double buffering technique as Physis although the original benchmark program used single buffering. Since the program in Physis accessed grid data via a pointer, the original program and our framework were modified to access via a pointer. For comparison, we also ran the programs where grid data are held in a multi-dimensional array. In the figure, (array) denotes them. The program in Physis did not perform a reduction operation although the others did. The programs were compiled by GCC 4.3.4 with -O3 or Intel C compiler 14.0.2.144 with -Ofast or -fast. They were run on TSUBAME 2.5 supercomputer. The problem size was L ($256 \times 256 \times 512$). The performance numbers were on average across 10 iterations.

Our framework on Bytespresso-C outperformed Physis but this was because the code generation by Physis was not well tuned for execution on CPUs. Our framework also outperformed the original benchmark program in C when the programs were compiled by GCC or Intel C compiler with -Ofast. The reason was that the original program accessed memory to obtain the grid size whenever it computed the address of a grid element. In the program for our framework, the grid size was given at runtime but it was propagated as

[4]https://github.com/naoyam/physis/tree/develop/examples

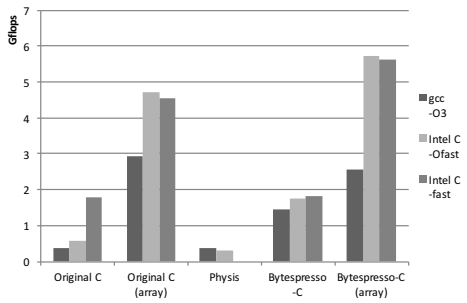

Figure 4: Jacobi kernel of Himeno on single CPU

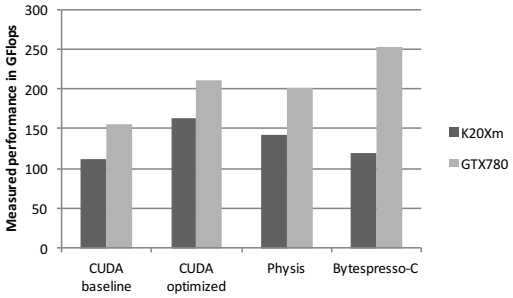

Figure 5: Three-dimensional diffusion

a constant value during the DSL compilation. This is an advantage of the design of Bytespresso-C.

Figure 5 shows the execution performance of the programs solving a three-dimensional diffusion equation on a GPU. Our framework on Bytespresso-C achieved comparable performance to Physis; its performance is sometimes better than Physis. Note that Physis was targeted at NVIDIA's Fermi architecture but we examined with the Kepler architecture. Their performance behavior is notably different [19]. CUDA baseline and CUDA optimized are programs written in CUDA. The grid size was $256 \times 256 \times 256$. We counted 13 floating-point operations per computation on each grid element. The performance numbers were on average across 1000 iterations. All the programs were compiled by nvcc 6.5.12 with -arch=sm_35 and gcc 4.3.4 (4.8.2 for GTX780) with -O3. They were run with CUDA 6.5 on Tesla K20Xm or GeForce GTX780.

5. RELATED WORK

Deep reification and Bytespresso

There have been a few systems providing functionality similar to deep reification or Bytespresso. Lancet [28] allows programmers to directly control just-in-time (JIT) compilation in Scala. Although it does not explicitly provide deep reification exactly as presented in this paper, it is possible to achieve similar feats by tweaking and customizing Lancet. In particular, a customized JIT compiler using Lancet can generate GPU instead of native CPU code as Bytespresso-C does. Lancet's strength lies in customizing JIT compilation similar to systems like OpenJIT [23]. Making extracted snapshots of the environment self-contained for DSLs is not the main focus of that work. 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 can access missing objects and code on demand.

Graal [24], 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 [8] is good for optimization. It is useful, for example, for partial evaluation by Truffle [32]. The accessibility of Graal to bytecode at runtime can be used to implement deep reification. Since Bytespresso performs bytecode decompilation for extracting bytecode, however, it can run on instances of the JVM other than Graal or ones with similar interfaces. Furthermore, bytecode extraction is only a part of deep reification; delimiting necessary code and collecting necessary data are just as, or even more, crucial.

The following illustrates why delimiting necessary code is a significant technical issue. An article [22] on Firepile, which is an offloading system from Scala to CUDA, 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, which refers to runtime constant values, it seems to statically collect methods in all available subclasses of the type of a called object. JaBEE [34] is an offloading system from Java to CUDA and it also supports dynamic method dispatch. It requests programmers to manually delimit necessary code for offloaded computation. They have to store all necessary class files in a specific directory.

Object serialization or marshalling is a simpler form of deep reification. Its snapshots only contain data (and possibly type and code references, but not code itself) extracted from the current execution environment. It has been provided by distributed computing systems since its early days [14, 3]. How to and whether to cut off (or delimit) parts of an object graph for transferal has been a long-standing technical issue.

A few low-level techniques for directly accessing code should be noted as well. They can be used as a primitive for extracting code. Macros in Scala and other languages, can be considered low-level mechanisms for IR extraction. Expression trees in C# extract the IR of a lambda expression typed as Expression. These techniques enable us to transparently extract code/IR without users' awareness. Any kind of host-language code can be extracted. However, these techniques extract the IR of only a lexically delimited code block. They do not allow extracting the IR of a method body invoked within that code block, or DSL authors have to manually extract it.

Bytespresso-C

Delite [4] is an offloading system similar to Bytespresso-C. It also can be regarded as a development system of embedded DSLs, in other words, class libraries providing domain-specific abstraction in the host-platform language. A difference from Bytespresso-C is that Delite exploits LMS as its basis and has drawbacks mentioned in Section 2. Note however that the previously mentioned Lancet may be combined with Delite to address these drawbacks.

A class library on Bytespresso-C such as the ones we showed in Section 4 is transformed and compiled into a language on a destination platform. However, since it is written still in pure Java, it can be regarded as an embedded DSL. It is different from external DSLs such as Physis [20] and ExaSlang [30]. Their code has to be compiled by their own compilers and their development environments including a source code editor have to be dedicated ones. The development costs of these tools are a drawback of external DSLs. However, for fairness sake we would mention that the slightly changed Java semantics of embedded DSLs like Bytespresso-C may threaten the reliability of tool-based refactoring for pure Java.

There have been a number of systems for offloading computation from the JVM to GPUs such as JCUDA [33], Rootbeer [25], JConqurr [13], and JCudaMP [7]. They are similar to Bytespresso-C. 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. Unlike those offloading systems, Bytespresso-C was designed to support not only GPU but also multiple platforms. We enhanced the basic architecture used by those systems to support various platforms and to be an essential component of offloading systems, which is deep reification.

Our previous work

Prior to this work, we have developed a similar system named *WootinJ* [17]. Bytespresso is our latest system of this series of work. In the paper about WootinJ, however, deep reification has not been identified as an underlying support mechanism for DSLs although a simple form of deep reification is internally executed.

Deep reification was first proposed in our workshop paper [6]. This short paper presented only a basic idea of deep reification but it does not mention Bytespresso-C, our DSL using deep reification. This paper is an extended version of that workshop paper.

Our proposal of implicit staging [29] may also be considered a predecessor of this work. Although it also proposed the reification of code, it is limited to expressions. It reifies code at load-time and hence the extracted IR does not contain references to runtime values. The IR extraction did not perform deep call-graph analysis, either.

6. CONCLUSION

This paper presented a reflection mechanism called deep reification for developing embedded DSLs. A technical issue of particular interest is how to delimit extracted code and data. Our prototype of deep reification for Java, Bytespresso, refers to runtime values available at extraction time and performs partial evaluation so that only necessary code and data will be extracted from a program that may contain dynamic method dispatch. This paper also presented Bytespresso-C. It shows that a practical DSL based on deep reification can be constructed. The source code of Bytespresso and Bytespresso-C is available from https://github.com/csg-tokyo/bytespresso.

Acknowledgments

This work was partly supported by JST CREST funding program. We would like to thank Naoya Maruyama for his help with the use of Physis and performance tuning techniques of CUDA code.

References

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

[2] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. of ACM OOPSLA*, pages 324–341. ACM, 1996.

[3] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comp. Syst.*, 2(1):39–59, 1984.

[4] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proc. of the 2011 Int'l Conf. on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100. IEEE Computer Society, 2011.

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

[6] S. Chiba, Y. Zhuang, and M. Scherr. A design of deep reification. In *Companion Proc. of the 15th Int'l Conf. on Modularity (MASS'16 workshop)*, pages 168–171. ACM, 2016.

[7] 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.

[8] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proc. of the 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL '13)*, pages 1–10. ACM, 2013.

[9] A. Ershov. On the essence of compilation. In E. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.

[10] D. P. Friedman and M. Wand. Reification: Reflection without metaphysics. In *Proc. of the 1984 ACM Symp. on LISP and Functional Programming*, pages 348–355. ACM, 1984.

[11] Y. Futamura. Partial computation of programs. In *Proc. of RIMS Symposia on Software Science and Engineering*, number 147 in LNCS, pages 1–35, 1982.

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

[13] 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.

[14] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Trans. Prog. Lang. Syst.*, 4(4):527–551, 1982.

[15] R. Himeno. Himeno benchmark. `http://accc.riken.jp/2444.htm`, 2001.

[16] P. Hudak. Modular domain specific languages and tools. In *Proc. of the 5th International Conference on Software Reuse*, ICSR '98, pages 134–142. IEEE Computer Society, 1998.

[17] M. Ioki and S. Chiba. A framework for multiplatform HPC applications. In *Proc. of Workshop on Programming Models and Applications on Multicores and Manycores (PMAM2014)*, pages 61–69, 2014.

[18] V. Jovanovic, A. Shaikhha, S. Stucki, V. Nikolaev, C. Koch, and M. Odersky. Yin-yang: Concealing the deep embedding of dsls. In *Proc. of the 2014 Int'l Conf. on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 73–82. ACM, 2014.

[19] N. Maruyama and T. Aoki. Optimizing stencil computations for NVIDIA Kepler GPUs. In *Proc. of 1st Int. Workshop on High-Performance Stencil Computations (HiStencils 2014)*, 2014.

[20] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *Proc. of 2011 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 11:1–11:12. ACM, 2011.

[21] L. Nyland, M. Harris, and J. Prins. Fast n-body simulation with CUDA. In H. Nguyen, editor, *GPU Gems 3*, chapter 31, pages 677–695. Addison-Wesley, 2007.

[22] 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.

[23] 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.

[24] Oracle. OpenJDK: Graal project. `http://openjdk.java.net/projects/graal`, 2012.

[25] 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.

[26] T. Rompf and M. Odersky. Lightweight Modular Staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proc. of the Ninth Int'l Conf. on Generative Programming and Component Engineering*, GPCE '10, pages 127–136. ACM, 2010.

[27] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-Virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, 25(1): 165–207, 2012.

[28] 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.

[29] M. Scherr and S. Chiba. Implicit staging of EDSL expressions: A bridge between shallow and deep embedding. In *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *LNCS*, pages 385–410, 2014.

[30] C. Schmitt, S. Kuckuk, F. Hannig, H. Köstler, and J. Teich. Exaslang: A domain-specific language for highly scalable multigrid solvers. In *Proc. of the Fourth Int. Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '14, pages 42–51. IEEE Press, 2014.

[31] B. Smith. Reflection and semantics in a procedural languages. Technical Report MIT-TR-272, M.I.T. Laboratory for Computer Science, 1982.

[32] 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.

[33] 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.

[34] 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.