OpenJIT Frontend System: an implementation of the reflective JIT compiler frontend

Hirotaka Ogawa[1], Kouya Shimura[2], Satoshi Matsuoka[1],
Fuyuhiko Maruyama[1], Yukihiko Sohda[1], and Yasunori Kimura[2]

[1] Tokyo Institute of Technology
[2] Fujitsu Laboratories Limited

Abstract. OpenJIT is an open-ended, reflective JIT compiler framework for Java being researched and developed in a joint project by Tokyo Inst. Tech. and Fujitsu Ltd. Although in general self-descriptive systems have been studied in various contexts such as reflection and interpreter/compiler bootstrapping, OpenJIT is a first system we know to date that offers a stable, full-fledged Java JIT compiler that plugs into existing monolithic JVMs, and offer competitive performance to JITs typically written in C or C++. We propose an architecture for a reflective JIT compiler on a monolithic VM, and describe the details of its frontend system. And we demonstrate how reflective JITs could be useful class- or application specific customization and optimization by providing an important reflective "hook" into a Java system.

1 Introduction

Programming Languages with high-degree of portability, such as Java, typically employ portable intermediate program representations such as bytecodes, and utilize *Just-In-Time compilers (JITs)*, which compile (parts of) programs into native code at runtime. However, all the Java JITs today as well as those for other languages such as Lisp, Smalltalk, and Self, only largely focuses on standard platforms such as Workstations and PCs, merely stress optimizing for speeding up single-threaded execution of general programs, usually at the expense of memory for space-time tradeoff. This is not appropriate, for example, for embedded systems where the tradeoff should be shifted more to memory rather than speed. Moreover, we claim that JITs could be utilized and exploited more opportunely in the following situations:

- **Platform-specific optimizations:** Execution platforms could be from embedded systems and hand-held devices all the way up to large servers and massive parallel processors (MPPs). There, requirements for optimizations differ considerably, not only for space-time tradeoffs, but also for particular class of applications that the platform is targeted to execute. JITs could be made to adapt to different platforms if it could be customized in a flexible way.

- **Platform-specific compilations:** On related terms, some platforms require assistance of compilers to generate platform- specific codes for execution. For example, DSM (Distributed-Shared Memory) systems and persistent object systems require specific compilations to emit code to detect remote or persistent reference operations. Thus, if one were to implement such systems on Java, one not only needs to modify the JVM, but also the JIT compiler. We note that, as far as we know, representative work on Java DSM (cJVM[2] by IBM) and persistent objects (PJama[3] at University of Glasgow) lack JIT compiler support for this very reason.
- **Application-specific optimizations:** One could be more opportunistic by performing optimizations that are specific to a particular application or a data set. This includes techniques such as selection of compilation strategies, runtime partial evaluation, as well as application-specific idiom recognition. By utilizing application-specific as well as run-time information, the compiled code could be made to execute substantially faster, or with less space, etc. compared to traditional, generalized optimizations. Although such techniques have been proposed in the past, it could become a generally-applied scheme and also an exciting research area if efficient and easily customizable JITs were available.
- **Language-extending compilations:** Some work stresses on extending Java for adding new language features and abstractions. Such extensions could be implemented as source-level or byte-code level transformations, but some low-level implementations are very difficult or inefficient to support with such higher-level transformations in Java. The abovementioned DSM is a good example: Some DSMs permit users to add control directives or storage classifiers at a program level to control the memory coherency protocols, and thus such a change must be done at JVM and native code level. One could facilitate this by encoding such extensions in bytecodes or classfile attributes, and customizing the JIT compilers accordingly to understand such extensions.
- **Environment- or Usage-specific compilations and optimizations:** Other environmental or usage factors could be considered during compilation, such as adding profiling code for performance instrumentation, debugging etc. [1]

Moreover, with Java, we would like these customizations to occur within an easy framework of portable, security-checked code downloaded across the network. That is to say, just as applets and libraries are downloadable on-the-fly, we would like the JIT compiler customization to be downloaded on-the-fly as well, depending on the specific platform, application, and environment. For example, if a user wants to instrument his code, he will want to download the (trusted) instrumentation component from the network on-the-fly to customize the generated code accordingly.

[1] In fact we do exactly that in the benchmarking we will show in [21], which for the first time characterizes the behavior of a self-descriptive JIT compiler.

Unfortunately, most JITs today, especially those for Java, are architected to be closed and monolithic, and do not facilitate interfaces, frameworks, nor patterns as a means of customization. Moreover, JIT compilers are usually written in C or C++, and live in a completely separate scope from normal Java programs, without enjoying any of the language/systems benefits that Java provides, such as ease of programming and debugging, code safety, portability and mobility, etc. In other words, current Java JIT compilers are "black boxes", being in a sense against the principle of modular, open-ended, portable design ideals that Java itself represents.

In order to resolve such a situation, the collaborative group between Tokyo Institute of Technology and Fujitsu Limited have been working on a project OpenJIT[19] for almost the past two years. OpenJIT itself is a "reflective" Just-In-Time open compiler framework for Java written almost entirely in Java itself, and plugs into the standard JDK 1.1.x and 1.2 JVMs. All compiler objects coexist in the same heap space as the application objects, and are subject to execution by the same Java machinery, including having to be compiled by itself, and subject to static and dynamic customizations. At the same time, it is a fully-fledged, JCK (Java Compatibility Kit) compliant JIT compiler, able to run production Java code. In fact, as far as we know, it is the ONLY Java JIT compiler whose source code is available in public, and is JCK compliant other than that of Sun's. And, as the benchmarks will show, although being constrained by the limitations of the "classic" JVMs, and still being in development stage lacking sophisticated high-level optimizations, it is nonetheless equal to or superior to the Sun's (classic) JIT compiler on SpecJVM benchmarks, and attains about half the speed of the fastest JIT compilers that are much more complex, closed, and requires a specialized JVM. At the same time, OpenJIT is designed to be a compiler framework in the sense of Stanford SUIF[28], in that it facilitates high-level and low-level program analysis and transformation framework for the users to customize.

OpenJIT is still in active development, and we have just started distributing it for free for non-commercial purposes from `http://www.openjit.org/`. It has shown to be quite portable, thanks in part to being written in Java—the Sparc version of OpenJIT runs on Solaris, and the x86 version runs on different breeds of Unix including Linux, FreeBSD, and Solaris. We are hoping that it will stem and cultivate interesting and new research in the field of compiler development, reflection, portable code, language design, dynamic optimization, and other areas.

The purpose of the paper is to describe our experiences in building OpenJIT, as well as presenting the following technical contributions:

1. We propose an architecture for a reflective JIT compiler framework on a monolithic "classic" JVM, and identify the technical challenges as well as the techniques employed. The challenges exist for several reasons, that the JIT compiler is reflective, and also the characteristics of Java, such as its pointer-safe execution model, built-in multi-threading, etc.

2. We demonstrate how reflective JITs could be useful class- or application specific customization and optimization by providing an important reflective "hook" into a Java system, with the notion of *compilets*. Although the current examples are small, we nevertheless present a possibility of larger-scale deployment of OpenJIT for uses in the abovementioned situations.

2 Overview of the OpenJIT Framework

Although there have been reflective compilers and OO compiler frameworks, OpenJIT has some characteristic requirements and technical challenges that were previously not seen in traditional reflective systems as well as JIT compilers. In order to better describe the technical challenges, we will first overview the OpenJIT framework.

2.1 OpenJIT: The Conceptual Overview

OpenJIT is a JIT compiler written in Java to be executed on "classic" VM systems such as Sun JDK 1.1.x and 1.2. OpenJIT allows a given Java code to be portable and maintainable with compiler customization. With standard Java, the portability of Java is effective insofar as the capabilities and features provided by the JVM (Java Virtual Machine); thus, any new features that has to be transparent from the Java source code, but which JVM does not provide, could only be implemented via non-portable means. For example, if one wishes to write a portable parallel application under multi-threaded, shared memory model, then some form of distributed shared memory (DSM) would be required for execution under MPP and cluster platforms. However, JVM itself does not facilitate any DSM functionalities, nor provide any software 'hooks' for incorporating the necessary read/write barriers for user-level DSM implementation. As a result, one must either modify the JVM, or employ some ad-hoc preprocessor solution, neither of which are satisfactory in terms of portability and/or performance. With OpenJIT, the DSM class library implementor can write a set of compiler metaclasses so that necessary read/write barriers, etc., would be appropriately inserted into critical parts of code.

Also, with OpenJIT, one could incorporate platform-, application-, or usage-specific compilation or optimization. For example, one could perform various numerical optimizations such as loop restructuring, cache blocking, etc. which have been well-studied in Fortran and C, but have not been well adopted into JITs for excessive runtime compilation cost. OpenJIT allows application of such optimizations to critical parts of code in a pinpointed fashion, specified by either the class-library builder, application writer, or the user of the program. Furthermore, it allows optimizations that are too application and/or domain specific to be incorporated as a general optimization technique for standard compilers, as has been reported by [15].

In this manner, OpenJIT allows a new style of programming for optimizations, portability, and maintainability, compared to traditional JIT compilers,

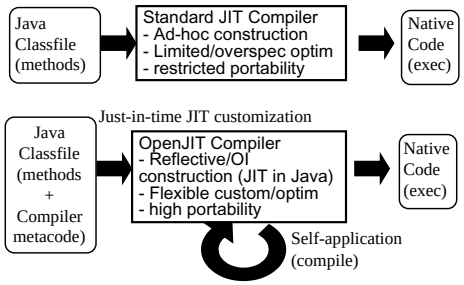

Fig. 1. Comparison of Traditional JITs and OpenJIT

by providing separations of concerns with respect to optimization and code-generation for new features. That is to say, with traditional JIT compilers, we see in the upper half of Figure 1, the JIT compilers would largely be transparent from the user, and users would have to maintain code which might not be tangled to achieve portability and performance. OpenJIT, on the other hand, will allow the users to write clean code describing the base algorithm and features, and by selecting the appropriate compiler metaclasses, or even by writing his own separately, one could achieve optimization while maintaining appropriate separation of concerns. Furthermore, compared to previous open compiler efforts, OpenJIT could achieve better portability and performance, as source code is not necessary, and late binding at run-time allows exploitation of run-time values, as is with run-time code generators.

2.2 Architectural Overview of OpenJIT

The OpenJIT architecture is largely divided into the frontend and the backend processors. The frontend takes the Java bytecodes as input, performs higher-level optimizations involving source-to-source transformations, and passes on the intermediate code to the backend, or outputs the transformed bytecode. The backend is effectively a small JIT compiler in itself, and takes either the bytecode or the intermediate code from the frontend as input, performs lower-level optimizations including transformation to register code, and outputs the native code for direct execution. The reason why there is a separate frontend and the backend is largely due to modularity and ease of development, especially for higher-level transformations, as well as defaulting to the backend when execution speed is not of premium concern. In particular, we strive for the possibility of the two modules being able to run as independent components.

OpenJIT will be invoked just as a standard Java JIT compiler would, using the standard JIT API on each method invocation. A small OpenJIT C runtime is dynamically linked onto the JVM, disguised as a full-fledged C-based JIT compiler. Upon initialization, it will have set the `CompiledCodeLinkVector` within the JVM so that it calls the necessary OpenJIT C stub routines. In particular,

when a class is loaded, JVM calls the `OpenJIT_InitializeForCompiler()` C function, which redirects the invoker functions for each method within the loaded class to to `OpenJIT_invoke()`. `OpenJIT_invoke`, in turn upcalls the appropriate Java `compile()` method in the `org.OpenJIT.Compile` class, transferring the necessary information for compilation of the specific method. It is possible to specify, for each method, exactly which portion of the compiler is to be called; by default, it is the OpenJIT backend compiler, but for sophisticated compilation OpenJIT frontend is called. After compilation, the upcall returns to `OpenJIT_invoke()`, which calls the just compiled code through `mb->invoker` (`mb` = method block). Thus, the heart of OpenJIT compiler is written in Java, and the C runtime routines merely serve to "glue" the JVM and the Java portion of OpenJIT. The details will be presented in [21].

Upon invocation, the *OpenJIT frontend* system processes the bytecode of the method in the following way: The *decompiler* recovers the AST of the original Java source from the bytecode, by recreating the control-flow graph of the source program. At the same time, the *annotation analysis module* will obtain any annotating info on the class file, which will be recorded as attribute info on the AST[2].

Next, the obtained AST will be subject to optimization by the *(higher-level) optimization module*. Based on the AST and control-flow information, we compute the data & control dependency graphs, etc., and perform program transformation in a standard way with modules such as *flowgraph construction module*, *program analysis module*, and *program transformation module* using template matching. The result from the OpenJIT frontend will be a new bytecode stream, which would be output to a file for later usage, or an intermediate representation to be used directly by the OpenJIT backend.

The *OpenJIT backend* system, in turn, performs lower-level optimization over the output from the frontend system, or the bytecodes directly, and generates native code. It is in essence a small JIT compiler in itself.

Firstly, when invoked as an independent JIT compiler bypassing the frontend, the *low-level IL translator* analyzes and translates the bytecode instruction streams to low-level intermediate code representation using stacks. Otherwise the IL from the frontend is utilized. Then, the *RTL Translator* translates the stack-based code to intermediate code using registers (RTL). Here, the bytecode is analyzed to divide the instruction stream into basic blocks, and by calculating the depth of the stack for each bytecode instruction, the operands are generated with assumption that we have infinite number of registers. Then, the *peephole optimizer* would eliminate redundant instructions from the RTL instruction stream, and finally, the *native code generator* would generate the target code of the CPU, allocating physical registers. Currently, OpenJIT supports the SPARC and the x86 processors as the target, but could be easily ported to other machines. The generated native code will be then invoked by the Java VM, as described earlier.

[2] In the current implementation, the existence of annotation is a prerequisite for frontend processing; otherwise, the frontend is bypassed, and the backend is invoked immediately.

3 Oveview of the OpenJIT Backend System

As a JIT compiler, the high-level overview of the workings of OpenJIT backend is standard. The heart of the low-level IL translator is the `parseBytecode()` method of the `ParseBytecode` class, which parses the bytecode and produces an IL stream. The IL we defined is basically an RISC-based, 3-operand instruction set, but is tailored for high affinity with direct translation of Java instructions into IL instruction set with stack manipulations for later optimizations. There are 36 IL instructions, to which each bytecode is translated into possibly a sequence of these instructions. Some complex instructions are translated into calls into run-time routines. We note that the IL translator is only executed when the OpenJIT backend is used in a standalone fashion; when used in conjunction with the frontend, the frontend directly emits IL code of the backend.

Then, RTL converter translates the stack-based IL code to register based RTL code. The same IL is used, but the code is restructured to be register-based rather than encoded stack operations. Here, a dataflow analyzer is then run to determine the type and the offset of the stack operands. We assume that there are infinite number of registers in this process. In practice, we have found that 24–32 registers are sufficient for executing large Java code without spills when no aggressive optimizations are performed[24]. Then, the *peephole optimizer* would eliminate redundant instructions from the RTL instruction stream.

Finally, the *native code generator* would generate the target code of the CPU. It first converts IL restricting the number of registers, inserting appropriate spill code. Then the IL sequence is translated into native code sequence, and ISA-specific peephole optimizations are performed. Currently, OpenJIT supports the SPARC and x86 processors as the target, but could be easily ported to other machines[3]. The generated native code will be then invoked by the Java VM, upon which the *OpenJIT runtime module* will be called in a supplemental way, mostly to handle Java-level exceptions.

The architectural outline of the OpenJIT backend is illustrated in Figure 2. Further details of the backend system can be found in [23].

4 Details of the OpenJIT Frontend System

As described in Section 2, the OpenJIT frontend system provides a Java class framework for higher-level, abstract analysis, transformation, and specialization of Java programs which had already been compiled by `javac`: (1) The decompiler translates the bytecode into augmented AST, (2) analysis, optimizations, and specialization are performed on the tree, and (3) the AST is converted into the low-level IL of the backend system, or optionally, a stream of bytecodes is generated.

[3] Our experience has been that it has not been too difficult to port from SPARC to x86, save for its slight peculiarities and small number of registers, due in part being able to program in Java. We expect that porting amongst RISC processors to be quite easy.

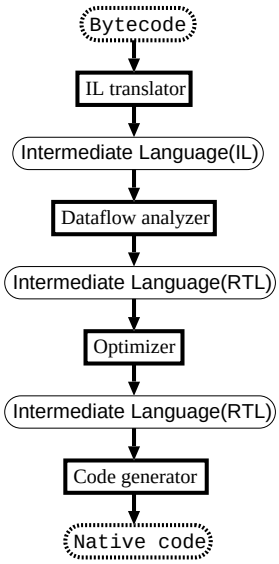

Fig. 2. Overview of the OpenJIT Backend System

Transformation over AST is done in a similar manner to Stanford SUIF, in that there is a method which traverses the tree and performs update on a node or a subtree when necessary. There are a set of abstract methods that are invoked as a hook. The OpenJIT frontend system, in order to utilize such a hook functionality according to user requirements, extends the class file (albeit in a conformable way so that it is compatible with other Java platforms) by adding annotation info to the classfile. Such an info is called "classfile annotation".

The overall architecture of the OpenJIT frontend system is as illustrated in Fig. 3, and consists of the following four modules:

1. **OpenJIT Bytecode Decompiler**
   Translates the bytecode stream into augmented AST. It utilizes a new algorithm for systematic AST reconstruction using dominator trees.
2. **OpenJIT Class Annotation Analyzer**
   Extracts classfile annotation information, and adds the annotation info onto the AST.
3. **OpenJIT High-level Optimizer Toolkit**
   The toolkit to construct "compilets", which are modules to specialize the OpenJIT frontend for performing customized compilation and optimizations.
4. **Abstract Syntax Tree Package**
   Provides construction of the AST as well as rewrite utilities.

We first describe the classfile annotation, which is a special feature of OpenJIT, followed by descriptions of the four modules.

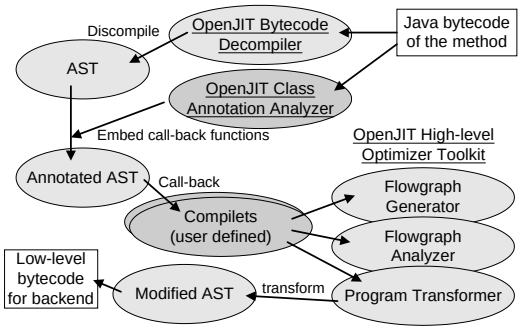

Fig. 3. Overview of OpenJIT Frontend System

4.1 Classfile Annotation

Classfile annotation in OpenJIT is additional info or directive added to the classfile to direct OpenJIT to perform classfile-specific (or application-specific, platform-specific) optimization and customization. Here are examples of directives possible with classfile annotations:

- Support for User-defined Optimizers and Specializers
- Support for Memory Models e.g., DSM
- Optimizing Numerical Code

Support for User-defined Optimizers and Specializers OpenJIT allows user-level definitions and customizations of its optimizer and specializer classes in the frontend. The classfile annotation allows the user to specify which of the classes the user-defined compiler classes to employ, by means of naming the class directly, or encoding the classfile itself as an annotation.

Support for Memory Models e.g., DSM As mentioned in Section 1, the support for various memory models including DSM requires insertion of appropriate Read/Write barriers for access to shared objects. However, there are algorithms to statically determine that objects are immutable or do not escape such as [8, 4, 5, 32], which allow such barriers to be compiled away, eliminating runtime overhead.

Optimizing Numerical Optimizations Numerical performance of Java is known to suffer due to array bounds checks, non-rectangular multidimensional storage allocation, etc. By marking the loops that can be statically determined to use the array in regular ways, we can apply traditional Fortran-style optimizations such as loop transformation, cache blocking, etc.

In order to implement the classfile annotation feature, we employ the attribute region of of each method in the classfile. According to the JVM specs, any attributes that the JVM does not recognize are simply ignored; thus, classfiles with OpenJIT annotations can be executed on platforms without OpenJIT,

achieving high portability (save for the programs that do not work without OpenJIT). One caveat is that there is no simple way to add extra information in the attribute field of classes themselves, due to the lack of appropriate JIT interface in the JVM; thus, one must employ some convention, say, defining a "dummy" null method that is called by the constructor, whose sole purpose is to supply class-wide annotation info that would be cached in the OpenJIT compiler.

In order to create a classfile with annotation information, we either employ an extended version of source-to-bytecode compilers such as `javac`; for classfiles without source, we could use a tool to add such annotation in an automated way; in fact the tool we are currently testing is a modified version of the OpenJIT frontend system.

4.2 OpenJIT Bytecode Decompiler Module

OpenJIT Bytecode Decompiler inputs the bytecode stream from the classfile, and converts it into an augmented AST. The module processes the the bytecode in the following way:

1. Converts the bytecode stream into an internal representation of JVM instruction, and marks the instructions that become the leading instruction of basic blocks.
2. Construct a control flow graph (CFG) with basic block nodes.
3. Construct a dominator tree that corresponds to the CFG.
4. Reconstruct the Java AST by symbolic execution of the instructions within the basic block.
5. Discover the control flow that originated from the short-circuit optimizations of the Java conditional expressions such as `&&` or `||` and `(x ? a * b)`, and recover the expressions.
6. Reconstruct the Java control structure using the algorithm described in [16].
7. Output the result as an AST, augmented with control-flow and dominator information.

All the above steps except (6) are either simple, or could be done with existing techniques, such as that described in [20]. Step (6), is quite difficult; most previous techniques published so far analyzed the CFG directly, and used pattern matching to extract valid Java control structures [20, 26]. Instead, we have proposed an algorithm which walks over the dominator tree, and enumerates over every possible patterns of dominance relation, which has a corresponding Java control structure. Compared to existing techniques such as Krakatoa[26], our method was shown to be faster, and more robust to code obfuscation. Some preliminary details can be found in [16].

4.3 OpenJIT Class Annotation Analyzer Module

The OpenJIT Class Annotation Analyzer module extracts the class annotation from a classfile, and adds the annotation info to the AST. The added annotations are typically *compilets* that modify the compiler more concretely, it processes the annotation in the following way:

1. First, it access the attribute region of the method. This is done by parsing the method block region extracted from the JVM.
2. We process this byte array assuming that the annotation object has been serialized with `writeObject()`, constructing an annotation object.
3. we attach the annotation object to the AST as annotation information.

Because what kind of information is to be embodied in the classfile annotation differs according to its usage, the `OpenJIT_Annotation` is actually an abstract class, and the user is to subclass a concrete annotation class. The abstract superclass embodies the identifier of the annotation, and the AST node where it is to be attached. This is similar in principle to SUIF, except that the annotation must be extracted from the classfile instead of being given a priori by the user.

4.4 OpenJIT High-level Optimizer Toolkit

OpenJIT High-level Optimizer Toolkit is used to construct OpenJIT *compilets*, that are a set of classes that customizes the compiler. The toolkit provides means of utilizing the augmented AST for implementing traditional compiler optimizations, and is largely composed of the following three submodules: [4]

1. Flowgraph Constructor
   Flowgraph Constructor creates various (flow) graphs from the augmented AST, such as dataflow graph, FUD chains, control dependence graph, etc. The Flowgraph class is an abstract class, and Factory Method pattern is employed to construct user-defined flowgraphs.
2. Flowgraph Analyzer
   The Flowgraph Analyzer performs general computation over the flowgraph, i.e., dataflow equation solving, handling merges, fix point calculation, etc. We employ the Command Pattern to subclass the `Analyzer` class for each algorithm, and each subclass triggers its own algorithm with the `execute()` method. The user can subclass the `Analyzer` class to add his own flowgraph algorithm.
3. Program Transformer
   The Program Transformer employs declarative pattern matching and rewrite rules to transform the augmented AST. One registers the rule using the following API:
   - `register_pattern(Expression src, Expression dst)`
   - `register_pattern(Statement src, Statement dst)`
     Registers the transformation rule that transforms the `src` pattern to the `dst` pattern. The pattern can be constructed using the Abstract Syntax Tree Package described next.

[4] In the current version, compilets are not downloadable; this is primarily due to the fact OpenJIT itself is not yet entirely downloadable due to a few restrictions in the JVM. We are currently working to circumvent the restrictions, and a prototype is almost working. Meanwhile, the Toolkit itself is available, and a custom version of OpenJIT can be created with "static" compilets using standard inheritance.

- Node
  - Expression
    - BinaryExpression
      - AddExpression
      - SubtractExpression
      - MultiplyExpression
      - ...
    - UnaryExpression
    - ConstantExpression
    - ...
  - Statement
    - IfStatement
    - ForStatement
    - WhileStatement
    - CaseStatement
    - ...

Fig. 4. Class Hierarchy of The Abstract Syntax Tree Package

- `substitution(Expression root)`
- `substitution(Statement root)`

  Searches the subtree with the designated `root` node depth-first, and if a match is found with the registered patterns, we perform the transformation.

Initial use of the current pattern matching technique proved to be somewhat too low-level; in particular, generation and registration of the transformation rule is still cumbersome. The next version of OpenJIT will have APIs to generate patterns and transformation rules from higher-level specifications, in particular for well-known program transformations (such as code motion, loop transformation, etc.)

4.5 Abstract Syntax Tree Package

The Abstract Syntax Tree Package is a utility package called from other parts of the OpenJIT frontend to implement low-level construction of the augmented AST, patterns for transformation rules, etc. The AST essentially implements the entire syntactic entities of the Java programming language. Each node of the AST corresponds to the expression or a statement in Java. The class hierarchy for the package is organized with appropriate subclassing of over 100 classes: (Fig. 4). We show typical Expression and Statement classes in Fig. 5 and Fig. 6, respectively.

A typical `Expression` subclass for a binary operator (`MultiplyExpression` in the example) consists of the operator ID, left-hand and right-hand expressions, and reference to an annotation object. The `code()` method either generates the low-level IL for the backend, or a Java bytecode stream. The `code()` method walks over the left- and right-hand expressions in a recursive manner, generating

```
public class MultiplyExpression extends BinaryExpression {
   int op;                          // Construct ID
   Expression left;                 // LHS expression
   Expression right;                // RHS expression
   Type type;                       // Type of this expression
   Annotation ann;                  // Embedded Annotation (default: null)

   void code() {                    // Convert AST to backend-IR form
                                    // (or bytecodes)
      if (ann) ann.execute(this); // call-back for metacomputation
      left.code();                  // generate code for LHS
      right.code();                 // generate code for RHS
      add(op);                      // generate code for "operator"
   }

   Expression simplify() {}         // Simplify expression form
                                    // (e.g. convert "a * 1" to "a")
   ...
}
```

Fig. 5. An Expression Class for A Typical Binary Expression (Multiply)

code. When a node has non-null reference to an annotation object, it calls the `execute()` method of the annotation, enabling customized transformations and compilations to occur.

As a typical `Statement` subclass, `IfStatement` recursively generates code for the conditional in a similar manner to `Expression`s.

As such, the current OpenJIT is structured in a similar manner to OpenC++[6], in that syntactic entities are recursively compiled. The difference is that we provide annotation objects that abstracts out the necessary hook to the particular syntax node, in addition to customization of the syntax node themselves. Thus, it is possible to perform similar reflective extensions as OpenC++ in an encapsulated way. On the other hand, experience has shown that some traditional optimizations are better handled using SSA, such as dataflow analysis, constant propagation, CSE, loop transformation, code motion. In the next version of OpenJIT, we plan to support SSA directly by translating the augmented AST into SSA, and providing the necessary support packages.

5 Reflective Programming with OpenJIT—A Preliminary Example

As a preliminary example, we tested loop transformation of the program in Fig. 8 into an equivalent one as shown in Fig. 9 [5]. In this example, we have added

[5] Note that although we are using the Java source to represent the program, in reality the program is in bytecode form, and transformation is done at the AST level.

```
public class IfStatement extends Statement {
   int op;                          // Construct ID
   Expression cond;                 // Condition expression
   Statement ifTrue;                // Statement of Then-part
   Statement ifFalse;               // Statement of Else-part
   Annotation ann;                  // Embedded Annotation (default: null)

   void code() {                    // Convert AST to backend-IR form
                                    // (or bytecodes)
      if (ann) ann.execute(this); // call-back for metacomputation
      codeBranch(cond);             // generate code for Condition
      ifTrue.code();                // generate code for Then-part
      ifFalse.code();               // generate code for Else-part
      addLabel();                   // add label for "Break" statement
   }

   Statement simplify() {}          // Simplify statement form
                                    // (e.g. if (true) S1 S2 => S1)
   ...
}
```

Fig. 6. A Example Statement Class for the "If" Statement

a compilet called `LoopTransformer` using the class annotation mechanism in the attribute region of the `matmul()` method by using a tool mentioned in Section 4.1. The `execute()` method of the `LoopTransformer` class searches the AST of the method it is attached to for the innermost loop of the perfect trinested loop. There, if it finds a a 2-dimensional array whose primary index is only bound to the loop variable of the outermost loop, it performs the necessary transformation. The overview of the `LoopTransformer` is shown in Fig. 7; the real program is actually about 200 lines, and is still not necessarily easy to program due to relatively low level of abstraction that the tree package provides, as mentioned earlier. We are working to provide a higher level API by commonizing some of the operations as a compilet class library.

Also, one caveat is that the IL translator is still incomplete, and as such we have generated the bytecode directly, which is fed into the OpenJIT backend. Thus we are not achieving the best performance, due to the compilation overhead, and the information present in the frontend is not utilized in the backend. Nevertheless, we do demonstrate the effectiveness to some degree.

For OpenJIT, we compared the results of executing Fig. 8 directly, and also transforming at runtime using the OpenJIT frontend into Fig. 9. For sunwjit, we performed the transformation offline at source level, and compiled both programs with javac. The size of the matrices (SIZE) are set to 200×200 and 600×600. Table 1 shows the results, before and after the transformation, and the setup time required for JIT compilation. (The overhead of for sunwjit is zero as it had been done offline.)

```
public class LoopTransformer extends Annotation {
  int loop_nest = 0;
  LocalField index;
  LoopTransformer() {}
  boolean isRegularForm(Statement init, Expression cond, Expression inc) {
    // Check the initializer and the conditions of the For statement
    // to verify that it is in a normal form.
  }
  void execute(Node root) {
    if (root instanceof CompoundStatement) {
      for (int i = 0; i < root.args.length; i++) { execute(root.args[i]); }
    }
    // Test whether the loop is a perfect tri-nested loop
    else if (root instanceof ForStatement &&
             root.body instanceof ForStatement &&
             root.body.body instanceof ForStatement) {
      if (isRegularForm(root.init, root.cond, root.inc) &&
          isRegularForm(root.body.init, root.body.cond, root.body.inc) &&
          isRegularForm(root.body.body.init, root.body.body.cond, root.body.body.inc)) {
          // Record the loop variable of the root
          // Verify that root.body.body does not include a ForStatement
          // If it doesn't then scan the RHS for a 2-dimensional
          //   array of the form ([] ([] index) _)
          // If found then perform the appropriate transformation

    } } }
    else return;
} }
```

Fig. 7. Overview of LoopTransformer

```
public int[][] matmul(int[][] m1, int[][] m2) {
  for (int i = 0; i < SIZE; ++i) {
    for (int j = 0; j < SIZE; j++) {
      for (int k = 0; k < SIZE; k++) {
        T[i][j] += m1[i][k] * m2[k][j];
      }
    }
  }
  return T;
}
```

Fig. 8. Matrix Multiply Method (Original)

```
public int[][] matmul(int[][] m1, int[][] m2) {
  for (int i = 0; i < SIZE; ++i) {
    int tmp[] = m1[i];
    for (int j = 0; j < SIZE; j++) {
      for (int k = 0; k < SIZE; k++) {
        T[i][j] += tmp[k] * m2[k][j];
      }
    }
  }
  return T;
}
```

Fig. 9. Matrix Multiply Method (Transformed)

We see that the execution time of OpenJIT and sunwjit are within 10% of each other. This similar to SpecJVM98 where OpenJIT and sunwjit for SPARCs. So, for the purposes of this benchmark, we can regard both systems to be essentially equivalent, and thus the benefits of reflection can be judged in a straightforward way.

The setup time for OpenJIT without frontend transformation is approximately 1.09 seconds, compared to 0.49 seconds for sunwjit. This verifies our benchmarks in the previous section where the compiler bootstrap overhead was quite small. The 1.59 seconds difference between the original and transformed is the overhead of frontend execution. The overhead consists of the process described in Section 4. We believe we can improve this overhead substantially, as the frontend has not been tuned as much as the backend, especially regarding generation of numerous small objects.

Still we see that, although when the matrix size is small (200×200), the overhead of frontend processing with a compilet exceeds that of the speed gain, for larger problem (600×600) this overhead is amortized for 7% improvement. Moreover, we expect to further amortize this as the transformation is done only once, and as a result, multiple execution of the same method will not pay the overhead allowing us to essentially ignore the setup overhead for 9% gain.

We are in the process of running larger benchmarks, with more interesting compilet examples. Still, we have been able to some preliminary demonstration that run-time reflective customization of OpenJIT frontend with compilets can be beneficial for compute-intensive tasks, by achieving more gains than the overhead added to the JIT compilation process.

6 Related Work

We know of only two other related efforts paralleling our research, namely MetaXa[11] and Jalapeño[1]. Metaxa overall is a comprehensive Java reflective system, constructing a fully reflective system whereby many language features could be reified, including method invocations, variable access, and locking.

Table 1. Results of OpenJIT Frontend Optimization (All times are seconds)

matrix size	200		600	
	before	after	before	after
OpenJIT	2.52	2.26	85.22	77.74
OpenJIT setup-time	1.09	2.68	1.09	2.67
sunwjit	2.34	2.06	80.19	73.55
sunwjit setup-time	0.49	0.49	0.49	0.49

MetaXa has built its own VM and a JIT compiler; as far as we have communicated with the MetaXa group, their JIT compiler is not full-fledged, and is specific to their own reflective JVM. Moreover, their JIT is not robust enough to compile itself[6].

Jalapeño[1] is a major IBM effort in implementing a self-descriptive Java system. In fact, Jalapeño is an aggressive effort in building not only the JIT compiler, but the entire JVM in Java. The fundamental difference stems from the fact that Jalapeño rests on *its own customized JVM with completely shared address space*, much the same way the C-based JIT compilers are with C-based JVMs. Thus, there is little notion of separation of the JIT compiler and the VM for achieving portability, and the required definition of clean APIs, which is mandated for OpenJIT. For example, the JIT compilers in Jalapeño can access the internal objects of the JVM freely, whereas this is not possible with OpenJIT. So, although OpenJIT did not face the challenges of JVM bootstrapping, this gave rise to investigation of an effective and efficient way of interfacing with a monolithic, existing JITs, resulting in very different technical issues as have been described in [21].

OpenJIT is architected to be a compiler framework, supporting features such as decompilation, various frontend libraries, whereas it is not with Jalapeño. No performance benchmarks have been made public for Jalapeño, whereas we present detailed studies of execution performance validating the effectiveness of reflective JITs, in particular memory profiling technique which directly exploits the 'openness' of OpenJIT.

Still, the Jalapeño work is quite impressive, as it has a sophisticated three-level compiler system, and their integrated usage is definitely worth investigating. Moreover, there is a possibility of optimizing the the application together with the runtime system in the VM. This is akin to optimization of reflective systems using the First Futamura projection in object oriented languages, as has been demonstrated by one of the author's older work in [17] and also in [18], but could produce much more practical and interesting results. Such an optimization is more difficult with OpenJIT, although some parts of JVM could be supplanted with Java equivalents, resulting in a hybrid system.

There have been a number of work in practical reflective systems that target Java, such as OpenJava[27], Javassist[7], jContractor[14], EPP[13], Kava[30],

[6] In fact, we are considering collaborative porting of OpenJIT to their system.

just to name a few. Welch and Stroud present a comprehensive survey of Java reflective systems, discussing differences and tradeoffs of where in the Java's execution process reflection should occur[30].

Although a number of work in the context of open compilers have stressed the possibility of optimization using reflection such as OpenC++[6], our work is the first to propose a system and a framework in the context of a dynamic (JIT) compiler, where run-time information could be exploited. A related work is Welsh's Jaguar system[31], where a JIT compiler is employed to optimize VIA-based communication at runtime in a parallel cluster.

From such a perspective, another related area is dynamic code generation and specialization such as [9, 12, 10]. Their intent is to mostly provide a form of run-time partial evaluation and code specialization based on runtime data and environment. They are typically not structured as a generalized compiler, but have specific libraries to manipulate source structure, and generate code in a "quick" fashion. In this sense they have high commonalities with the OpenJIT frontend system, sans decompilation and being able to handle generalized compilation. It is interesting to investigate whether specialization done with a full-fledged JIT compiler such as OpenJIT would be either be more or less beneficial compared to such specific systems. This not only includes execution times, but also ease of programming for customized compilation. Consel et. al. have investigated a hybrid compile-time and run-time specialization techniques with their Tempo/Harrisa system [29, 22], which are source-level Java specialization system written in C; techniques in their systems could be applicable for OpenJIT with some translator to add annotation info for predicated specializations.

7 Conclusion and Future Work

We have described our research and experience of designing and implementing OpenJIT, an open-ended reflective JIT compiler framework for Java. In particular, we proposed an architecture for a reflective JIT compiler framework on a monolithic VM, and demonstrate a small example of how reflective JITs could be useful class- or application specific customization and optimization by defining a compilet which allowed us to achieve 8-9% performance gain without changing the base-level code.

Numerous future work exists for OpenJIT. We are currently redesigning the backend so that it will be substantially extensible, and better performing. We are also investigating the port of OpenJIT to other systems, including more modern VMs such as Sun's research JVM (formerly EVM). In the due process we are investigating the high-level, generic API for portable interface to VMs. The frontend requires substantial work, including speeding up its various parts as well as adding higher-level programming interfaces. Dynamic loading of not only the compilets, but also the entire OpenJIT system, is also a major goal, for live update and live customization of the OpenJIT. We are also working on several projects using OpenJIT, including a portable DSM system[25], numerical optimizer, and a memory profiler whose early prototype we employed in this

work. There are numerous other projects that other people have hinted; we hope to support those projects and keep the development going for the coming years, as open-ended JIT compilers have provided us with more challenges and applications than we had initially foreseen when we started this project two years ago.

Acknowledgments

Many acknowledgments are in order, too many to name here. We especially thank Matt Welsh, who coded parts of OpenJIT during his summer job at Fujitsu. Ole Agesen for discussing various technical issues, etc.

References

1. B. Alpern, D. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, and J. J. Barton. Implementing Jalapeno in Java. In *Proceedings of OOPSLA '99*.
2. Y. Aridor, M. Factor, and A. Teperman. cJVM: a Single System Image of a JVM on a Cluster. In *Proceedings of ICPP '99*, September 1999.
3. M. Atkinson, L. Daynes, M. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *ACM SIGMOD Record*, 25(4), December 1996.
4. B. Blanchet. Escape Analysis for Object-Oriented Languages. Application to Java. In *Proceedings of OOPSLA '99*, pages 20–34, November 1999.
5. J. Bogda and U. Holzle. Removing Unnecessary Synchronization in Java. In *Proceedings of OOPSLA '99*, pages 35–46, November 1999.
6. S. Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA '95*, pages 285–299, 1995.
7. S. Chiba. Javassist — A Reflection-based Programming Wizard for Java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.
8. J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape Analysis for Java. In *Proceedings of OOPSLA '99*, pages 1–19, November 1999.
9. D. R. Engler and T. A. Proebsting. vcode: a retargetable, extensible, very fast dynamic ocde generation system. In *Proceedings of PLDI '96*.
10. N. Fujinami. Automatic and Efficient Run-Time Code Generation Using Object-Oriented Languages. In *Proceedings of ISCOPE '97*, December.
11. M. Golm. metaXa and the Futre of Reflection. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.
12. B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An Evaluation of Staged Run-time Optimization in DyC. In *Proceedings of PLDI '99*, 1999.
13. Y. Ichisugi and Y. Roudier. Extensible Java Preprocessor Kit and Tiny Data-Parallel Java. In *Proceedings of ISCOPE '97*, December 1997.
14. M. Karaorman, U. Holzle, and J. Bruno. iContractor: A Reflective Java Library to Support Design by Contract. In *Proceedings of Reflection '99*, pages 175–196, July 1999.
15. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP '97*, pages 220–242, 1997.

16. F. Maruyama, H. Ogawa, and S. Matsuoka. An Effective Decompilation Algorithm for Java Bytecodes. *IPSJ Journal PRO (written in Japanese)*, 1999.
17. H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa. Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation. In *Proceedings of OOPSLA '95*, pages 57–64, October 1995.
18. H. Masuhara and A. Yonezawa. Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Language. In *Proceedings of ECOOP '98*, pages 418–439, July 1998.
19. S. Matsuoka, H. Ogawa, K. Shimura, Y. Kimura, and K. Hotta. OpenJIT — A Reflective Java JIT Compiler. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.
20. O. Agesen. Design and Implementation of Pep, a Java Just-In-Time Translator. *Theory and Practice of Object Systems*, 3(2):127–155, 1997.
21. H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java. In *Proceedings of ECOOP '2000 (to appear)*.
22. U. P. Schultz, J. L. Lawall, C. Consel, and G. Muller. Towards Automatic Specialization of Java Programs. In *Proceedings of ECOOP '99*, June 1999.
23. K. Shimura. OpenJIT Backend Compiler. http://www.openjit.org/docs/backend-compiler/openjit-shimura-doc-1.pdf, June 1998.
24. K. Shimura and Y. Kimura. Experimental development of java jit compiler. In *IPSJ SIG Notes 96-ARC-120*, pages 37–42, October 1996.
25. Y. Sohda, H. Ogawa, and S. Matsuoka. OMPC++ — A Portable High-Performance Implementation of DSM using OpenC++ Reflection. In *Proceedings of Reflection '99*, pages 215–234, July 1999.
26. T. Proebsting and S. Watterson. Krakatoa: Decompilation in Java. In *Proceedings of COOTS '97*, June 1997.
27. M. Tatsubori and S. Chiba. Programming Support of Design Patterns with Compile-time Reflection. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.
28. Stanford University. SUIF Homepage. http://www-suif.stanford.edu/.
29. E. N. Volanschi, C. Consel, and C. Cowan. Declarative Specialization of Object-Oriented Programs. In *Proceedings of OOPSLA '97*, pages 286–300, October.
30. I. Welch and R. Stroud. From Dalang to Kava - the Evolution of a Reflective Java Extention. In *Proceedings of Reflection '99*, pages 2–21, July 1999.
31. M. Welsh and D. Culler. Jaguar: Enabling Efficient Communication and I/O from Java. *Concurrency: Practice and Experience*, December 1999. Special Issue on Java for High-Performance Applications.
32. J. Whaley and M. Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of OOPSLA '99*, pages 187–206, November 1999.