OpenJIT—A Reflective Java JIT Compiler

— Short Version for the OOPSLA'98 Reflection Workshop —

Satoshi Matsuoka
Tokyo Institute of Technology
and Japan Science and Technology Corporation

Hirotaka Ogawa
Tokyo Institute of Technology

Kouya Shimura, Yasunori Kimura
Fujitsu Laboratories

Koichiro Hotta
Fujitsu Limited

Hiromitsu Takagi
Electrotechnial Laboratory

September 30, 1998

Abstract

The so-called 'Open Compilers' is a technique to incorporate various self-descriptive modules for language customization and optimization based on computational reflection. We apply the open compiler technique to a Java Just-In-Time compiler to develop the *OpenJIT compiler*, which allows class-specific customization and optimization, fostering research of new compilation techniques such as application-specific customization and dynamic optimizations. Benchmarks with the current prototype, which is almost entirely written in Java, have seen comparable benchmark results compared to traditional C-based JIT compilers.

1 Introduction

Programming Languages which support features for high-degree of portability, such as Java, are becoming increasingly important today. Such languages typically employ architecture and operating system independent bytecoded intermediate program representation, and utilized *Just-In-Time compilers (JITs)*, which compile parts of programs which require speedups into native code. However, compared to traditional compilers, JIT compilers lack the overall technical framework for its construction. As a result, portability of the JIT compiler itself, as well as JIT-specific compilation issues, are not well-explored or discussed.

For example, most optimization techniques in traditional compilers execution speed at the expense of larger memory overhead. On the other hand, for most embedded applications, compilers must optimize for *resource efficiency* with moderate speedups. So most commercial JIT compilers today emphasize speed, sacrificing resource efficiency, and there is no way to systematically change this emphasis. More generally, in order to adapt a program to a specific computing environment, a compiler must perform (1) optimizations that are suited to the particular execution environment, and (2) extend code generation to accommodate for application and underlying platform requirements. However, all JIT compilers to date are "black boxes" in that (1) they only perform general, speed-centric optimizations, and (2) have no systematic means for introducing new code generators for language extensions and/or new features of the underlying platforms.

We claim that the fundamental reasons for such shortcomings are that JITs today, although improving with respect to their standard compilation efficiency, are constructed rather ad-hocly solely based on traditional compiler technologies. Most JITs today are either written in C or in assembly language, and have little concern for reuseability, portability (in an OO-framework sense), adaptability, i.e., software engineering properties which a well-constructed software should all embody. Stanford's SUIT compiler[1] is a 'traditional' static compiler constructed as an OO-framework in order to serve as a basis of customization for various compiler research. JITs, on the other hand, are not only boxes, but also their source code, their internal architectural structures, and the compilation and runtime techniques they employ,

are seldom made open, and are thus hindering the progress of new area of exciting compiler research based on dynamic compilation.

In order to resolve such a situation, we propose an open-ended JIT compiler called *OpenJIT*, which is based on reflection and open compiler technologies[2]. OpenJIT itself is mostly written as a Java class framework, and allows on-the-fly extensions to the JIT compilers with compiler extension classes. This allows dynamic adaptation of the compilation process to various computing platforms and applications, according to various cost tradeoffs and the properties of the underlying platform environment.

OpenJIT allows a given Java code to be even more portable with compiler customization; by contrast, although there is a claimed portability story of Java as mentioned above, 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 class-specific optimization which would apply effective but costly optimization techniques in a pinpointed fashion to necessary parts of code. For example, one could perform various numerical optimizations which have been well-studied in Fortran and C but have not been well adopted into JITs for excessive runtime compilation cost, such as loop index analysis and associated transformations. 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 use 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. Open compilers would allow effective construction of such customized compilers, and OpenJIT is no exception. For example, it is reported by Kiczales et. al.[3] that, with open compilers, one could perform extensive loop fusions by exploiting the various semantic equivalence relations between operators of visual image processing, such as commutativity, allowing orders of magnitude speedups. OpenJIT will allow such transformation to take place at the Java bytecode level instead of the source level as in [3], allowing higher-level of portability even when source code is not available.

In this manner, OpenJIT allows a new style of programming for optimization and portability, including portability of performance, compared to traditional JIT compilers, by providing separations of concerns capabilities with respect to optimization and code-generation for new features. That is to say, with traditional JIT compilers, a 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 concern. 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.

Although there have been reflective compilers and OO compiler frameworks, because of being reflective and a run-time compiler, OpenJIT has some characteristic requirements and technical features that were previously not seen in traditional JIT compilers. The latter section introduces the overview of OpenJIT, and report on the first prototype and its performance evaluation.

2 OpenJIT—The Technical Issues

The full paper has more detailed discussions on the technical issues.

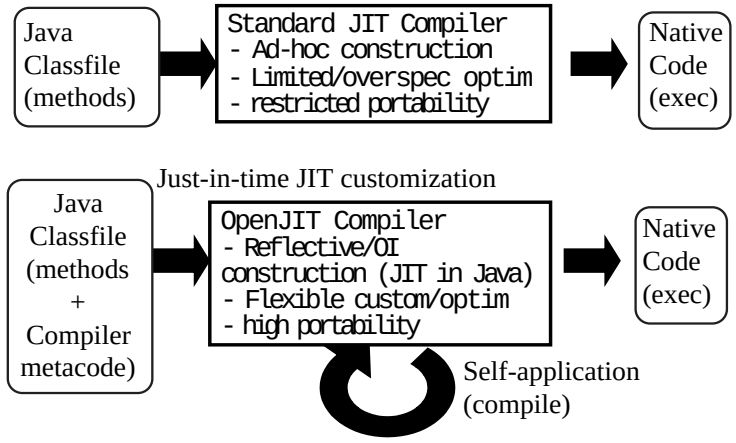

Figure 1: Comparison of Traditional JITs and OpenJIT

As OpenJIT is a reflective system, it is constructed as a class framework written in Java. As a result, it embodies several technical issues that otherwise would not occur for C-based JIT compilers.

1. Self-Compilation of the JIT Compiler

As most of OpenJIT is written in Java, the bytecode of OpenJIT will be initially interpreted by the JVM, and gradually become compiled for faster, optimized execution. Although this allows the JIT compiler itself to adapt the particular execution environment the JIT optimizes for, it could possibly give rise to the following set of problems:

- *Performance of the JIT compiler*: Various speed overhead, including that it is written in Java.
- *Lack of appropriate API for Java-written JIT compilers in JVM*: JVM does not have APIs for reflective JIT compilers.
- *Various Problems with self-modifying code*: Especially since Java is multithreaded.

2.Modification of the Compiler during Execution :

OpenJIT will allow run-time integration of Java components which will extend and/or modify the JIT compiler at runtime. Since these will be classfiles automatically downloaded into JVM, and will be part of a prefabricated optimization metaclasses, an end-user will not have to do his own hacking of the JIT compiler. Still, it could set raise the following set of problems:

- *Restrictions to self-modification*: Appropriate scope control[10] would be necessary.
- *Compiler safety:* Must be controlled via an authentication mechanism
- *Compiler API:* Must be designed to allow dynamic compiler customization

3 Overview of the OpenJIT Architecture

OpenJIT overall is written mostly in Java, and there are also a small JNI stubs for JVM introspection, plus some C-level runtime routines.

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. In particular, we strive for the possibility of the two modules being able to run as independent components.

Below, we describe the OpenJIT frontend and the backend in detail.

3.1 OpenJIT Frontend System

The OpenJIT will be invoked just as a standard Java JIT compiler would, using the standard JIT API on each method invocation. Upon invocation, the OpenJIT frontend processes the bytecode of the method in the following way: The *discompiler* 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.

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*. 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, in turn, performs further low-level optimizations, and outputs the corresponding native code.

3.2 OpenJIT Backend System

The OpenJIT backend system performs lower-level optimization over the output from the frontend system, and generates native code. The *OpenJIT Native Code Transformer* provides the overall abstract framework for backend execution, and defines the class framework APIs for backend modules which could be redefined with compiler metaclasses.

Firstly, the *low-level IL translator* analyzes and translates the bytecode instruction streams to low-level intermediate code representation using stacks. 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. 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. Currently, OpenJIT supports the SPARC processor as the target, but could be easily ported to other machines. The generated native code will be then invoked by the Java VM, upon which the *OpenJIT runtime module* could be called in a supplemental way.

4 OpenJIT Current Status and Preliminary Performance Evaluation

Currently, The Proof-of-concept prototype OpenJIT backend system is functional. It is based on Fujitsu's commercial C-based JIT compiler (FJIT). We ported the program while modifying it so that the program would be more object-oriented. Although the prototype is relatively small (approximately 5000 lines of Java code), it is stable and also fast, being able to run microbenchmarks such as CaffeineMark 3.0, as well as larger tools such as javac and even HotJava. When one wants to use OpenJIT as his JIT compiler, after downloading OpenJIT in the JDK 1.1.X environment, one would set the following environment variables.

```
% setenv JAVA_COMPILER OpenJIT
% setenv CLASSPATH <dir>
```

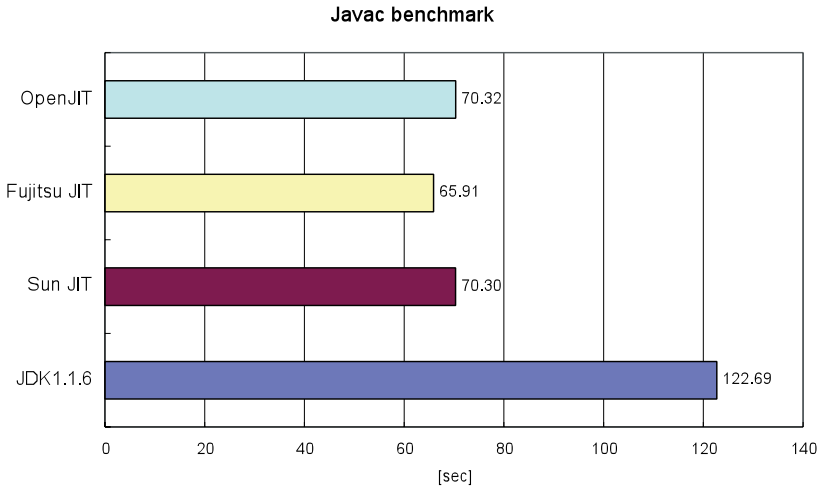

Figure 2: OpenJIT Javac Benchmark

```
% setenv LD_LIBRARY_PATH <dir>
```

This will set the appropriate paths for OpenJIT classfiles, and the C stub routines as well as the runtime routine (`libOpenJIT.so`). Thereon, OpenJIT will be transparent to the user—one would invoke a Java program in a standard way, and JIT compilation would take place as expected. Although there is limited scope control for the current version, the default is for the OpenJIT compilers to be compiled initially, after which the user bytecode will be compiled.

We performed several benchmarks on the prototype OpenJIT backend. Al benchmarks were executed under Sun UltraSparc-II 247Mhz, Solaris 2.6, JDK 1.1.6 environment. Although we have performed several benchmarks, such as CaffeineMark[12] and Nullprogram, we will introduce just one benchmark for brevity. More detailed benchmarks will be found in the full version of the paper.

The Javac benchmark is a more realistic benchmark compared to CaffeineMark. More specifically, we measure the time it takes to compile javac in javac. Because javac is I/O intensive and involve significant number of native method calls, JITs are known to be less effective.

The Graph in Figure 2 show the results of JVM interpreter, Sun's JIT for JDK, FJIT, and OpenJIT. JIT performances are almost all equivalent, and are twice as fast as interpretation. We can see that, for practical programs OpenJIT is comparable in performance against C-based JIT compilers.

The benchmarks we have performed so far have thus shown that, for moderately-sized practical programs, the current OpenJIT prototype exhibits similar performance to C-based JITs, verifying the viability of our approach of self-descriptive dynamic JIT. On the other hand, OpenJIT Frontend is still under design, and the backend also needs to be redesigned from the prototype to allow more fine-grained customizability. In particular, the high-level optimizations in the frontend is fundamentally the same as what are being done with traditional, static compilers; thus, in subsequent releases, the major research issue is how to strike the balance between execution time and memory consumption versus the benefits of sophisticated optimizations.

Then, OpenJIT can enjoy the benefits of both the traditional self-descriptive interpreter-based reflection, where elimination of (true) interpreter overhead is still difficult, and open compilers, where there is no interpretation overhead but lacks dynamic customizability.

5 Conclusion

We introduced OpenJIT, a JIT based on Open Compiler/Reflection techniques. Although OpenJIT is still under development, the current proof-of-concept prototype has shown competitive performance to C-based JIT compilers, and its code is much smaller and easier to understand. We are developing a framework-based APIs using various design patterns so that it could be easily customized, and that multiple customized compilers could co-exist for compilation of a single program. We hope that OpenJIT will serve as a basis for various interesting research on dynamic compilation techniques, and as such it will be made public when completed. Details will be posted on

Acknowledgment

OpenJIT is being developed under the contract with Information Promotion Agency of Japan, titled "Research and Development of Open-ended Just-in-Time Compiler OpenJIT for Java based on Reflection".

References

[1] The Stanford SUIF Compiler Group: SUIF Compiler System, `http://suif.stanford.edu`

[2] John Lamping, Gregor Kiczales, Luis Rodriguez, and Erik Ruf: An architecture for an open compiler, In *Proceedings of IMSA'92: Reflection and Meta-Level Architecture*, pp. 95–106, Tokyo, Nov. 1992.

[3] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin: Aspect-Oriented Programming, In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, Finland. Springer LNCS 1241 June 1997.

[4] Luis Rodriguez, Jr.: A study on the viability of a production-quality metaobject protocol-based statically parallelizing compiler, In *Proceedings of IMSA'92: Reflection and Meta-Level Architecture*, pp. 107–112, Tokyo, Nov. 1992.

[5] Yutaka Ishikawa et. al.: Design and Implementation of Metalevel Architecture in C++ –MPC++ Approach–. In *Proceedings of Reflection'96*, pp.141–154, San Francisco, 1996.

[6] Shigeru Chiba: A Metaobject Protocol for C++. In *Proceedings of ACM OOPSLA'95*, pp. 285–299., Oct. 1995.

[7] Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai, and Akinori Yonezawa: Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages using Partial Evaluation. In *Proceedings of ACM OOPSLA'95*, pp. 300–315., Oct. 1995.

[8] Yuuji Ichisugi and Yves Roudier: The Extensible Java Preprocessor Kit and a Tiny Data Parallel Java. In *Proceedings of ISCOPE'97*, Springer LNCS 1343, pp. 153–160, Marina Del Rey, CA, 1997.

[9] E. Volarschi, C. Consel, and C. Cowan: Declarative Specialization of Object-Oriented Programs. In *Proc. ACM OOPSLA*, pp. 286-300, 1997.

[10] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow. "The Art of the Metaobject Protocol", The MIT Press, Cambridge, MA, 1991.

[11] Kouya Shimua and Yasunori Kimura: Prototyping of a Java JIT Compiler (Java JIT Compiler no Shi-saku). IPSJ–SIGARC Workshop Proceedings, 96-ARC-120, pp.37–42, Dec., 1996 (*In Japanese*).

[12] Pendragon Software: The CaffeineMark 3.0 Benchmark, `http://www.pendragon-software.com/pendragon/cm3/index.html`.