OpenJIT 2: The Design and Implementation of Application Framework for JIT Compilers

[Extended Abstract]

Fuyuhiko Maruyama
Tokyo Institute of Technology
Tokyo, Japan

Satoshi Matsuoka[*]
Hirotaka Ogawa
Naoya Maruyama
Tokyo Institute of Technology
Tokyo, Japan

Kouya Shimura
Fujitsu Laboratories
Kanagawa, Japan

ABSTRACT

We are currently working on a new architecture for Java JIT compilers to allow coexistence of multiple, customized JIT compilers in a single Java VM simultaneously without prohibitive space or programming costs, called OpenJIT2. The project builds on the success of OpenJIT1, the first open-ended JIT compiler for Java written in Java. Not only that our new architecture subsumes the so-called 'bi-level' or 'tri-level' JIT compilers of today, but allows application, environment, or user-specific JIT compilers to coexist and invoked at appropriate moments as decided by the runtime. The compiler fragment modules can either be built-in or even dynamically downloaded from the network on demand to tailor the compiler for specific needs. We believe such customization allows the best performance to be squeezed out of applications in a way not possible with generic optimization strategies. OpenJIT2 will also be publically distributed for free and supported just as was with OpenJIT1 to serve as application framework for general compiler research by others, having a very clean and customizable object-oriented framework structure as opposed to OpenJIT1. We also hope that it will serve as commercial quality replacement JIT compiler for various platforms.

1. INTRODUCTION

Java is now one of the de facto language system running everywhere, from small PDAs to large servers. This is primarily realized by the so called 'write-once, run everywhere' portability characteristics of Java. However, there is no guarantee that the same program will runs at its optimal level everywhere; that is to say Java only realizes platform portability but but not *performance portability*. We believe that technologies to implement performance portability will be very much in demand as Java proliferates to even wider range of execution platforms, such as High-Performance Computing as well as micro-embedded devices with very low power-to-performance budget.

Modern Java VMs employ Just-In-Time(JIT) compiler to speedup execution of Java programs. The only difference between JIT compilers and ordinary static compilers is that JIT compiler compiles parts of program during execution instead of before execution as is with ordinary static compilers. In general, longer compilation time produces better native codes, but there always exists tradeoffs between the compilation time and the quality of resulting code—long compilation time that doesn't benefit overall execution time is not well justified, especially for JIT compilers where the execution time naturally includes the compilation time.

To solve this problem, most modern sophisticated Java VMs employ the so-called adaptive compilation with n-level compilers (n=2,3 typically) with runtime selection techniques to compile only hotspot methods with more expensive compilers (i.e., compilers that produce more efficient code at the expense of longer compilation time) only when benefit would be foreseen. Such compilers are typically completely separate, or only share parts of the code base, and not tailorable from within Java. Moreover, adaptive compilation only attempts to 'recover' performance that would have been achieved with full static compilation in the first place, and does not usually exploit the run-time values or application or environment-specific knowledge that the users may otherwise have in the optimization process. It has been shown that such aggressive compilation strategies speed-up execution of programs in a manner not achievable with general optimization techniques.

OpenJIT2, which is a brand new JIT compiler in Java, attempts to tackle the above problems. In fact OpenJIT2 allows coexistence of multiple compilers in a clean, straightforward fashion, subsuming the multi-level compilation schemes. We allow easy customization of the compiler itself so that aggressive compiler strategies can be built in specifically tailored to the needs of the application, or even downloaded from the network as *compilet* modules. The core parts of OpenJIT2 is already complete, and we expect it to be finished by 2Q 2001. The internal structure is quite different from our OpenJIT1[2], which was also written in Java but was built rather monolithically for various reasons.

2. OVERVIEW OF OPENJIT 2

OpenJIT2 is a JIT compiler for Java based on the construction technique of so-called 'Open Compilers', i.e., a technique to incorporate various self-descriptive modules for language customization and optimization based on computational reflection ideas. To apply the open compiler technique to a Java JIT compiler, OpenJIT2 is almost entirely

[*]Also with Japan Science and Technology Corporation

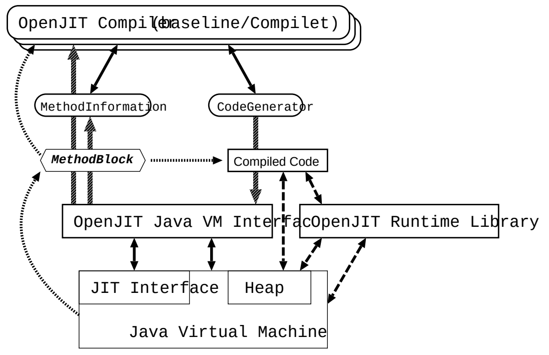

Figure 1: Overview of OpenJIT 2 system

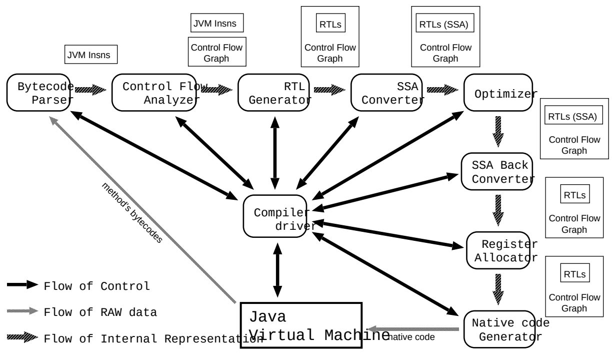

Figure 2: OpenJIT 2 baseline compiler

written in Java as was with OpenJIT1[2], our previous work. With OpenJIT1 we showed that we could construct commercial quality Java JIT compiler in Java itself without significant run-time performance or storage penalty. However, from an software architectural point of view, it was constructed rather monolithically and as such its ability to incorporate self-descriptive modules were poor at best. Even though the later versions allowed the entire JIT compiler to be downloaded as standard Java classfiles from the network, it was difficult to program and download small, customized compiler modules because of the monolithic (i.e., not very Object-Oriented) architecture. In fact, all external projects that utilized OpenJIT as a basis of JIT compiler research we know of patched the methods directly, rather than extending the compiler using object-oriented techniques.

2.1 Architecture

To facilitate the advantage of open compiler technique, OpenJIT2 is completely re-designed and implemented from scratch so that it is not only a JIT compiler but also an application framework for JIT compiler. The framework is similar in spirit to SUIF 2[3], `cmcc`[1], and the RTL system[4], in that it is a design that allows users to write extra compiler modules using object-oriented differential programming, rather than patching the method code themselves. This by all means brings on the full benefits of OO, since new modules need only to contain its unique aspects and common parts of JIT compilers are automatically shared with all compiler modules by OO feature of Java. The challenge then is how to define the framework API so that such customization is possible and easy with JIT compilers.

Figure 1 shows the overview of the entire Java system with OpenJIT2. OpenJIT2 itself consists of two major parts: the main part is the JIT compiler framework and its instantiations as multiple coexisting compilers written in Java (represented as ovals in figure 1) and the other is the native library for communication between the JIT compiler and Java VM (‘OpenJIT Java VM Interface’ in figure 1), and between the compiled code and Java VM (‘OpenJIT Runtime Library’ in figure 1). In this system, each instantiated compiler is a set of Java object which serves as a compiler for (at least) only one method, and as to which set of compiler objects to be employed for compilation of a particular method is selected by the ‘OpenJIT Java VM Interface’ based on the method’s attribute within the Java classfile. If the attribute of a method specifies a particular compiler, OpenJIT2 system will use the specified compiler, which consists of a set of ‘Compilet’ objects, to compile the method. Otherwise, the default ‘baseline’ compiler will be selected. The baseline compiler is actually a set of n-level compilers as seen in other JVM systems as we have described, but they are all instantiations from a common compiler framework sharing the objects in the infrastructure for clean compact representations. From the OpenJIT Java VM Interface’s point of view, the baseline compilers and the compilets are merely JIT compilers, each of which are tailored to compile a particular method, selected at runtime by some runtime selection strategies.

2.2 Design and Implementation

The baseline compiler and compilets are constructed using OpenJIT2 framework and class library. As opposed to OpenJIT1 whose internal structure was quite simple as a compiler, OpenJIT2 facilitates sophisticated compiler modules seen in modern optimizing compilers, so that a variety of analysis, program transformation, and optimization strategies can be implemented easily using standard differential programming. The framework is constructed in a hierarchical manner, where the baseline compiler (`BaselineCompiler` class) is the highest abstraction of OpenJIT2 framework consisting of several components of typical compilers such as the (bytecode) parser, optimizer and the code generator, all being constructed from lower-level OpenJIT2 class library. Figure 2 shows the structure of baseline compiler (we omit the details here for brevity); here the `BaselineCompiler` class corresponds to the ‘compiler driver’, but Compilets can also be constructed in the same manner. For example, it is very easy to construct a compilet which applies only selected optimizations during compilation of a specific method, and it is also not difficult to add new optimization not performed by baseline compiler, using the features of lower level classes that gives full access to not only manipulate the intermediate code but also serving as templates to plug in various analysis and transformation methods. Although the latter is only true only when it uses the same internal representation (IR) of the baseline compiler control flow graph containing RTLs in each node, SSA representation, etc). However, the IR of OpenJIT2 is general enough to perform all textbook optimizations in an easy manner, and moreover, we have carefully designed the classes so that IRs can be extended and used throughout the framework (for example, using appropriate factory methods so that there are no internal dependencies to a particular IR class.)

3. CURRENT STATUS

At this time (March 1, 2001), the implementation of OpenJIT2 is about 80% complete, and we expect to be finish the first version for a release in 2Q, 2001. In fact, most parts of framework have already been designed and implemented, and the remaining parts are merely the register allocator and code generator, plus bug fixes. We have released the current source as a preview to a very limited number of collaborators so that they can familiarize themselves with the structure and the IRs used in OpenJIT2.

We have also preliminarily measured the compilation speed of current implementation using a debug driver that simulate parts of Java VM and OpenJIT Java VM Interface: it reads a classfile and invokes the baseline compiler to compile methods (without register allocation or code generation). Measurements are performed on RedHat Linux 6.2, Pentium III 500MHz, with 384MB memory.

Results we have obtained so far show that our baseline compiler compiles about 3 bytecodes/msec using an interpreter, 8 bytecodes/msec using the OpenJIT1, and 12 bytecodes/msec using Sun HotSpot VM. OpenJIT2 baseline compiler is entirely written in Java, so its compilation speed corresponds to the efficiency of its execution engine, i.e., initially OpenJIT2 compiles methods slowly by using interpretation and the compilation speed gradually becomes faster as compiler self-compiles. From our experiences with OpenJIT1, most parts of JIT compiler should be self-compiled on first encounter for optimal speed, so it is reasonable to consider that the typical compilation speed is at least as fast as the OpenJIT1, and perhaps match or even better than HotSpot by employing Compilet techniques. This is reasonable speed as an optimizing compiler compared with IBM's jitc (10bytecodes/msec) and Jalapeño's optimizing compiler (3-5 bytecodes/msec). We have also found that SSA-based code manipulation is quite expensive, and as such for a fast baseline compiler we can turn SSA off at the cost of higher-level optimization.

4. FUTURE WORKS

We are now working hard to complete the implementation of OpenJIT2. Once implementation is complete, we can evaluate the runtime performance of the OpenJIT2 system. Adding more optimizer modules are also planned, including OO-specific optimizations. Another important work is to add sophisticated adaptive compilation technique to the Java VM Interface, especially on how to effectively select from multitudes of compilers that coexist.

5. CONCLUSIONS

We have briefly overviewed our new framework architecture for JIT compiler that allows multiple coexisting JIT compilers that allows fine-grained, customized compilation of different parts of program. The design and implementation of OpenJIT2 is rather different from the modern JIT compilers that monolithically try to provide 'quick and effective' optimizations, but more akin to frameworks for optimizing static compilers. Thanks to its object-oriented frame design we hope that many people will find uses for OpenJIT2 for numerous interesting projects using JITs and dynamic compilation, pushing the state-of-the-art of language implementation technologies as a whole.

6. REFERENCES

[1] Ali-Reza Adl-Tabatabai, Thomas Gross, and Guei-Yuan Lueh. Code Reuse in an Optimizing Compiler. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 51–68, October 1996.

[2] Hirotaka Ogawa, Kouya Shimura, Satoshi Matsuoka, Fuyuhiko Maruyama, Yukihiko Sohda, and Yasunori Kimura. OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 362–387, June 2000.

[3] Holger Kienle and Urs Hölzle. Introduction to the SUIF 2.0 Compiler System. Technical Report TRCS97-22, UCSB, December 1997.

[4] Ralph E. Johnson, Carl McConnell, and J. Michael Lake. The RTL System: A Framework for Code Optimization. In *Proceedings of the International Workshop on Code Generation*, pages 255–274, 1991.