Optimization Techniques in a Java Just-In-Time Compiler

by

Kazuaki ISHIZAKI*

This thesis describes the design, implementation, and evaluation of optimization techniques in a Java Just-In-Time (JIT) compiler. These optimization techniques avoid the overhead incurred because of type safety and polymorphism for supporting reusability and safety of programs in the Java language. Consequently, Java can achieve the same runtime performance as the familiar C and C++ languages. The techniques described in the thesis were used in the production system 'IBM Developers Kit. Java Technology Edition', and they have contributed to improving the performance of Java since Java was released. In particular, the optimization technique to reduce the overhead of a virtual method call has been adopted by other Java JIT compilers.

Java is an object-oriented language that has features such as platform independency of programs by using a virtual machine, flexibility in programming by using an object-oriented approach and polymorphism, and safety of programs through verification and type safety. However, these feastures also incur runtime overheads. In particular, there are four overheads related to the implementation of the Java language.

An overhead related to the implementation of an object-oriented language with polymorphism is shown as Point 1. Overheads related to the implementation of a type safe language are shown as Points 2, 3, and 4.

1. Virtual method calls in a language with a dynamic class loading.
2. Type inclusion tests for objects.
3. Exception checks in references to array elements or instance variables.
4. Order constraints between instructions to ensure the safety of memory references.

In the rest of this abstract, I will summarize methods to reduce the above four overheads while describing the outline of the thesis.

Chapter 1 describes the motivations and results of our research as described in this thesis.

Chapter 2 describes features of the Java language and their implementation problems as the background of our research. Then it describes representative implementations of the Java language runtime environment. Finally, it explains the detailed implementation of the IBM Java Just-In-Time compiler where I implemented the proposed optimization techniques.

Chapter 3 describes an optimization technique for virtual method calls. A virtual method call with polymorphism provides flexibility in programming. A virtual method at a given call site call may have several callee methods. Since the polymorphism at a call site prevents the whole program analysis and optimizations from being applied, the compiler examines whether there is no overridden method in the class hierarchy. If there is no overridden method, the compiler applies direct devirtualization so that it can extend the optimization scope. For this, class hierarchy analysis can be performed to examine method declarations in the whole class hierarchy. In the Java language, to support dynamic class loading, when a dynamic class loading overrides a method that was previously not overridden, the directly devirtualized code must be invalidated. For this invalidation, although deoptimization had been

proposed involving the recompilation of a running method, this would have required a complex implementation. The chapter describes a method to directly devirtualize a virtual method call with a code patching mechanism. For a given dynamic method call site, the compiler first determines whether the call can be devirtualized, by analyzing the current class hierarchy at runtime. When the call is devirtualized and the target method is enough small, the compiler generates the inlined code for the method, together with the backup code of making the dynamic method call. Only the inlined code is actually executed until the assumption about devirtualization becomes invalid, at which time the compiler performs code patching to make the backup code be executed after that. Since this method is simple, it does not require a complex implementation. Since the technique prevents some code motion across the merge point between inlined code and the backup code, we propose a method using known analysis techniques that eliminate the backup code and move the merge point downward.

Chapter 4 describes an optimization technique for type inclusion tests for objects. A type inclusion test for an object is an important feature to ensure type safety in Java. The type inclusion test is to test whether the type on the right hand side can be converted into a type of class on left hand side in a statement. This conversion can be performed when a given class on the right hand side is a subclass of a class on the left hand side of the statement. Even for the statically typed Java language, all type inclusion tests cannot be evaluated only at compilation time. There are type inclusion tests that can be evaluated at runtime. This chapter describes a method to improve the performance of a type inclusion test by inlining the frequently executed part of a type inclusion test. This is implemented simply than by previous methods, consumes less memory than the previous methods, and can achieve the same performance as the previous methods.

Chapter 5 describes an optimization technique for exception checks. The Java language specification requires doing exception checks when memory is accessed, such as for an access to an instance variable, an access to an array element, or a reference to an object for a virtual method call. These checks in Java incur runtime overhead, while no runtime overhead is incurred since there is no exception check in C and C++. Previous research proposed methods to eliminate redundant exception checks at compilation time. Since there are still exception checks, methods were proposed utilizing the memory protection provided by an OS that traps reads and writes to the memory in the zero page. However, some OSes can trap only for writes to the memory in the zero page. This chapter describes a method to improve the performance of exception checks by encoding a unique condition into a pair of compare & branch instructions corresponding to the cause of an exception, and then decoding the cause of the exception from these instructions in an exception handler. Furthermore, this chapter also describes a method to improve performance using this method when the memory protection cannot be used, but the processor provides a fast compare & branch instruction.

Chapter 6 describes an optimization technique to relax an order constraint between instructions. The assurance of type safety in Java increases the safety and reliability of programs, but it incurs Java exception checks at runtime. These checks are designed to ensure that any faulting instruction causing a hardware exception does not terminate the program abnormally. However, they impose some constraints upon the execution order between an instruction potentially raising a Java exception and a faulting instruction causing a hardware exception. This reduces the effectiveness of instruction reordering optimizations. The chapter describes a new framework to effectively perform speculation for Java using a direct acyclic graph representation based on the static single assignment (SSA) form. Using this framework, we apply a well-known speculation technique to a faulting load instruction to eliminate such constraints. The framework uses edges to represent exception constraints. This allows a compiler to accurately estimate the potential reduction of the critical path length when applying speculation. This chapter also describes two approaches to avoid extra copy instructions and to generate efficient code with minimum register pressure.

Chapter 7 gives the conclusion of the thesis.

*(IBM Research, Tokyo Research Laboratory)