Optimizing Java Native Compiler – Adria/Java

Kenichi Miyata, Hiroyasu Nishiyama, Yuji Chiba, Tomoya Ohta, and Sumio Kikuchi

Systems Development Laboratory, Hitachi, Ltd.

Abstract. Java programs are usually executed by using an interpreter or a JIT-compiler. These methods restrict application of time consuming complex compiler optimizations. To cope with this, we are developing optimizing native compiler for Java that translates a Java bytecode to a native executable of the host computer ahead of its execution. This optimizing native compiler incorporates a translation phase of Java bytecode to intermediate language and a Java oriented optimization phase with existing optimizing compiler. Java oriented optimizations of this optimizing compiler includes static binding of method calls, method inlining, compile-time GC, and exception check elimination in conjunction with optimizations for ordinary procedural languages. Programs compiled with our prototype compiler executes up to 78.7 times faster than our interpreter, and up to 8.6 time faster than our JIT compiler.

1 Introduction

Java[1] is an object–oriented programming language[6, 10] that have following features: (1) Portable execution model using bytecode, (2) Powerful securities model by run–time exception checking, (3) Automatic memory management with garbage collection.

But these advantages cause less performance than traditional programming languages. Currently, JIT (Just–In–Time) compiling is a main execution method for Java. JIT compiler translates bytecode to a native executable of the host computer at run–time in order to reduce overheads of interpreting and executing bytecode. Executing time of bytecode in JIT contains JIT compiling time, so compiler can't spend much time for time consuming complex optimizations.

We are researching a Java native compiler (Adria/Java[2]) that translates Java bytecode to optimized object code. The purpose of our native compiler is efficient execution of Java program.

Generated object files are linked with class libraries and an executable is created, or they are executed in a Java virtual machine with JIT compiler. Since Adria/Java can spend much time to optimize than JIT compiler, it implements various kinds of optimizations.

---

[1] Java and Java related trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

[2] Advanced Compiler for Risc Architecture/Java

```
int add(int x, int y) {
    return (x+y);
}
```

(a) sample method

```
iload_1
iload_2
iadd
ireturn
```

(b) byte-code for method 'add'

```
*++SP = LocalVariable1;
*++SP = LocalVariable2;
temp1 = *SP--;
temp2 = *SP--;
*++SP = temp1+temp2;
return (*SP--);
```

(c) simple conversion result

```
return (LocalVariable1+
        LocalVariable2);
```

(d) Our Method result

Fig. 1. Example of translation

Adria/Java incorporates a translation phase, a Java oriented optimization phase and an existing optimization phase in which apply to loop optimizations, source–level optimizations, instruction–level optimizations, and so on.

Section 2 presents a bytecode translation phase. Section 3 introduces Java oriented optimizations. A result of performance evaluation to our prototype system is shown in Section 4.

2 bytecode Translation Phase

The bytecode translation phase reads Java bytecode and translates it to intermediate language (IL), which is constructed from basic blocks.

2.1 Expressions

The bytecode translation phase changes stack operations on bytecode into representations on an IL. The IL is closely related to procedural language like C or Fortran.

The bytecode is a set of instructions upon virtual stack machine. Changing stack operations into instructions on a normal register architecture is usually inefficient. Figure 1 shows an example of translation. A method Figure 1(a) would be converted to bytecode Figure 1(b). If simply translation was done, Figure 1(c) would be obtained. This program is inefficient because a lot of stack operations occur frequently. In order to prevent such a situation, Adria/Java translates the bytecode instructions into expressions on the IL by emulating Java stack. Figure 2 shows this process and Figure 1(d) indicates a result program.

Adria/Java has a evaluation stack whose element is a pointer to an expression of IL. Firstly, when Adria/Java translates `iload_1`, top of stack points to an expression which refers a local variable 1 (Figure2(a)). Similarly, for `iload_2`, an expression that refer local variable 2 is pushed to evaluation stack (Figure 2(b)). For `iadd`, after two expressions are popped, an adding expression is created

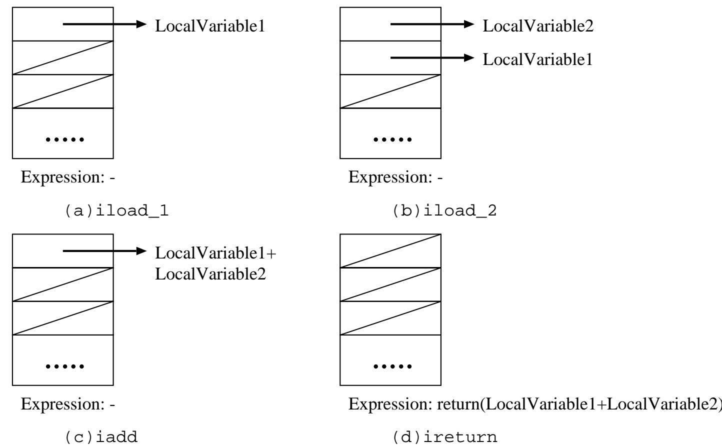

Fig. 2. Conversion process of an expression

and pushed. Finally, for `ireturn`, an expression is popped and a statement `return(LocalVariable1 + LocalVariable2)` is outputted to a basic block.

2.2 Method Calls

Method calls of Java can be classified into three kinds: (1) static method calls, (2) normal method calls and (3) interface method calls.

In Adria/Java, a static method call is converted to a direct function call corresponds to the method.

A normal (instance) method call is converted to an indirect function call using a method dispatch table like C++. The method dispatch table is included in class object. For example, a method call `obj.foo()` would be converted to following statements.

```
clazz = obj.getClass();
func = clazz.dispatch_table[ID number of method 'foo()'];
func();
```

The ID number is a unique in a class hierarchy, which same number is assigned to methods if they have same signatures.

For interface method calls, the possible objects doesn't have any inheritance relations among them, so we can't use the dispatch table. Then an address of the interface method is searched from a method table which has method information declared in its class, traversing the class hierarchy with a key for the method signature. After that, the address is cached in a global table. In next interface method call, its address is searched from the global table. If the address wasn't found, the above searching process is started. Still, the searching key is not a string but an address of a global variable generated from a method signature.

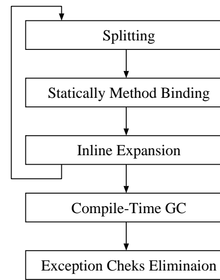

Fig. 3. Phase organization of Java oriented optimizations

3 Java Oriented Optimizations

Adria/Java implements many kinds of optimizations indicated in Figure 3. They will be described in following sections.

3.1 Static Binding of Method Calls

As described above section, Adria/Java translates normal method calls and interface method calls into statements to calculate function addresses and to perform indirect function calls. This process is inefficient because of calculating function addresses at run–time. And generally, indirect function calls often cause a pipeline stall in microprocessor. If possible, Adria/Java apply an optimization that binds a method call with a unique function call statically.

In Java, a class of an object determines a called method at run–time. If only one class could be decided for the object, a callee method can be decided at compile–time. A set of classes for possible objects refered by reference variables is calculated from object generating expressions, method arguments and return values from method calls. This set is propagated along control flow and merge them at merging points of control flow. This is called class flow analysis[5, 12].

As a result of class hierarchy analysis, at some call site, if only one class or only one method ould be decided, the method call is bound statically.

Even if callee method couldn't be decided uniquely, the method is bound statically using one of following sequences of conditional statements.

1. to compare a class of a object with possible classes for the object
2. to compare an address of a method with addresses for possible methods

Since statically bound methods could become targets to inline expansion, if frequently called methods would be bound statically, the execution performance would improve.

For example, suppose some classes or some methods were possible at some call site `obj.foo()`. Taking the approach (1), Adria/Java generates code to check possible classes for the object `obj` and to call function directly matching

with each classes. If the number of possible classes exceeds a certain number, after trying to compare such number times, Adria/Java generates code using method dispatch table. Taking the approach (2), Adria/Java generates method comparing code instead of the class comparing code.

Adria/Java selects a low–cost way considering with the number of classes and methods. Adria/Java can take a profile of objects at method call at run–time. If the profile information was available at compile–time, Adria/Java uses classes distribution at the call site and generates methods comparing code in order of high expectation values.

3.2 Splitting

As described in previous section, unless only one method could be found at the call site where several classes reach there, Adria/Java generates classes or methods comparing code. This code is inefficient because they need comparing operations.

One of the reasons why there are several possible classes at a call site is that several object creations are readched at merge points of control flow through paths from object creation points. Splitting optimization makes a class of an object unique by replicating a instructions from object creating points to method to call sites.

3.3 Inline Expansion

In object oriented programming language, it tends to define a lot of small methods. Therefor overheads of method calls appear remarkably compared with procedural programming language like C even if method calls were bound statically by class hierarchy analysis.

Adria/Java does an method inline expansion to reduce overheads of method calls that are bound statically. As a result of method inline expansion, since there would be more chances to analyze classes at a call site in detail, as showing figure 3, splitting and statically method binding are applied repeatedly.

3.4 Compile–time GC

In C++, users can explicitly control allocation of objectsa to stack and heap and are responsible for freeing heap objects. On the other hand, in Java, all objects are allocated on heap and disused objects are withdrawn from heap by Garbage Collector (GC). Java prescribe all objects are referenced through reference variables. So if many objects were created, overheads of GC couldn't be ignorable.

From evaluations for Java program, it is known that lifetime of objects are synchronized with periods of method execution, which method generates objects[4]. For example, in a following code, an object generated in statement (a) couldn't be refered after statement (b), so lifetime of object is synchronized with the method.

```
int foo() {
  ClassX obj = new ClassX();      // (a)
  return(obj.foo());              // (b)
}
```

In such a situation, allocating objects on stack makes possible automatic withdrawing disused objects at returning from a method call and reduce frequency of GC. Adria/Java implements this optimization called compile–time GC.

The compile–time GC detects objects satisfying following conditions in a method and generates code to allocate objects on stack: (1) Finalyzers hasn't been defined. (2) Generated objects are not assigned to not–private fields. (3) Generated objects don't call methods doing (2).

Objects, which is allocated on stack, are automatically removed when execution process returns to call site.

3.5 Exception Checks Elimination

Java prescribes the execution environment has to check exceptions when it refers objects or subscripts of arrays. The exception handling code causes less performance and disturb another optimizations. Therefor Adria/Java reduces its overheads by eliminating unnecessary exception checks.

Exception Elimination Based on Redundancy of Control Flow If same exception checks were executed through all paths reaching there, the checks can be removed. That is, for two exception check expressions e_1, e_2: (1) e_1 and e_2 are common expressions. (2) There are no definitions of operands in e_2 in all paths from e_1 to e_2. (3) e_1 dominates e_2.

Exception Elimination Based on Redundancy of Data Flow Some exceptions are detected based on operand values of bytecode. For example, a reference of objects is checked whether the value is `null` or not. About value–based checks, Adria/Java calculates reaching definitions of variables from data flow, make sure that all the reaching definitions don't satisfy the exception conditions and eliminate that.

Eliminating Constant Subscript Checks When a subscript expressions and an array length is a compile–time constant, its checks are eliminated or converted to unconditional exception invoking. Although it needs to get the array length, it can't be found from declaration because arrays are generated dynamically in Java. Therefor Adria/Java try to find the array length detecting array generating expressions from reaching definitinos.

Eliminating Subscript Checks Based on Implication Relations Even when an array length is unknown, if a set of subscript checks reaching at subscript checks v covers a range of v, v can be eliminated.

Our method is based on Kolte's method[8]. But while Kolte's method targets FORTRAN, our method targets Java. So checks are eliminated conservatively.

Eliminating Subscript Checks by Loop Length Now think about an array reference $A[a * i + b]$, which has a linear subscript for loop control variable i. This reference is required to satisfy with a condition $0 \leq a * i + b < A.length$ so that it doesn't occur any exceptions. Normalizing it with i, a range of i becomes $-b/a \leq i < (A.length - b)/a$. If this condition is satisfied, the subscript checks in loops can be eliminated.

Eliminating Subscript Checks by Loop Multiplexing If a subscript is a linear expression for loop control variable, Adria/Java try to move subscript checks out of the loop by multiplexing innermost loop.

If a linear subscript reference $A[a*i+b]$ exists in loops, a condition $0 \leq a \times i + b < A.length$ must be satisfied to eliminate subscript checks for it. Let low and high lower and upper limits for i respectively, $-b/a \leq low$ and $high \leq (A.length - b - 1)/a$ are both needed to be true. If there are more than one array references in a loop, Adria/Java generate two kinds of codes; one for that all subscript references don't occur exceptions and another for that some subscript references might occur exceptions. Adria/Java generates code that select these two cases at run–time [3]. That is, if array references $A_0[a_0*i+b_0], A_1[a_1*i+b_1], \ldots, A_n[a_n*i+b_n]$ exists in a loop, and when $-b_0/a_0 \leq low \wedge -b_1/a_1 \leq low \wedge \ldots \wedge -b_n/a_n \leq low \wedge high < (A_0.length - b_0 - 1)/a_0 \wedge high < (A_1.length - b_1 - 1)/a_1 \wedge \ldots \wedge high < (A_n.length - b_n - 1)/a_n$ is true, Adria/Java generates code that it doesn't check for array references $A_0[a_0 * i + b_0], A_1[a_1 * i + b_1], \ldots, A_n[a_n * i + b_n]$.

Eliminating Loop Invariant Checks Applying loop peeling could move loop invariant exception checks outside of the loop[2].

Adria/Java make sure whether operands of each exception check in a loop are loop invariant or not. If there are loop invariant checks, it peels off first time of loop iteration. As a result, the peeled check dominates exception checks in loop, so exception checks in a loop is eliminated by redundancy of control flow.

4 Performance Evaluation

Figure 4 shows a result of performance of Adria/Java. The evaluation is measured on the HI–UX/WE2 operating system. JIT compiler, Java interpreter and

[3] It would be fast to detect possibility of exception occurrence for each array references. But it isn't realistic because of code explosion.

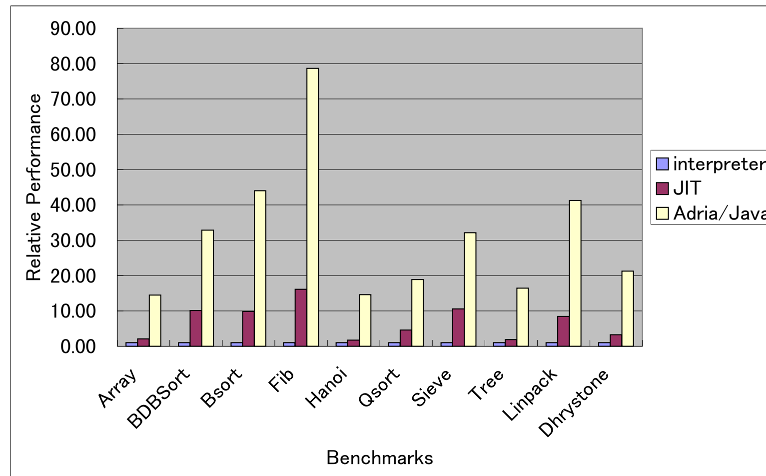

Fig. 4. Performance Evaluation

Adria/Java are implemented on it. Target benchmarks are selected from Symantec Benchmarks[4].

Figure 4 shows relative performance between JIT compiler and Adria/Java compared with performance of interpreter. In this result, Adria/Java achieves perfomance between 14.5 times and 78.7 times than interpreter, and between 3.0 times and 8.6 times than JIT compiler.

5 Related Work

Currently, JIT compiler is a general execution environment. But its optimizations are limited locally[1, 9].

There are two kind of native comiplers, one is translators from Java to C, the other is compilers generating native code directly.

Caffeine compiler[7] and Toba[13] translate bytecode into IL as Adria/Java does. These allocate virtual registers to stack elements, and change stack operations into corresponding operations on virtual registers. This kind of translator is required to apply some optimizations such as copy propagation, dead code elimination, and so on because of many copy statements are generated. On the other hand, our method, which simulates stack, isn't needed unnessesary temporary register, and can reconstruct expressions.

[4] The selected items are Array: array initializing with random number, BDBSort: bidirectional bubble sort, BSort: bubble sort, Fib: Fibonacchi number calculation, Hanoi: the tower of Hanoi, QSort: quick sort, Sieve: sieve calculation, Tree: tree creation, Linpack and Dhrystone.

Vortex[5] and Harissa[12] bind methods statically by class hierarchy analysis. In Vortex, a splitting optimization[3] is implemented to increase chances of statically binding. In Adria/Java, it implements method rearrangement by profile feedback and dynamic method selection by method comparing codes. The method comparing codes is effective to reduce a number of dynamic checking when there are many classes and methods are shared among the classes.

Abount optimizations which run GC statically at compile–time, some researches which target functional programming language have been done[11].

Optimizations which eliminate exception checks exist for FORTRAN[8]. And for Java, some optimizations have been reported that eliminate subscript checks by applying loop restructuring for nested loops[14]. Our apploach make a lot of run–time exception checks possible to eliminate by doing flow analysis and loop nesting.

6 Conclusion

This paper shows an overview and a result of performance about Java native compiler we are developing. This native compiler implements various optimizations for Java, and it gets higher performance than ordinary execution methods.

References

1. Adl-Tabatabai, A.-R., M.Cierniak, G-Y.Lueh, V.M.Parikh and J.M.Stichnoth: Fast, Effective Code Generation in a Just–In–Time Java Compiler. Proceedings of the ACM SIGPLAN'98 COnference on Programming Languages Design and Implementation (1998) 280–290
2. Budimlic, A. and Kennedy, K.: Optimizing Java — Theory and Practice. Proceedings of the ACM 1997 Workshop on Java for Science and Engineering Computation (1997)
3. C.Chambers and D.Ungar: Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically–Typed Object Oriented Programs. Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation (1987) 150–164
4. C.E.McDowell: Reducing garbage in Java. ACM SIGPLAN Notices Vol. 33, No.9. (1998) 84–86
5. Dean, J., Defouw, G., Grove, D., Litvinov, V. and Chambers, C.: Vortex: An Optimizing Compiler for Object–Oriented Languages. OOPSLA'96 Conference Proceedings (1996) 83–100
6. Gosling, J., Joy, B. and Steele, G.: The Java Language Specification. Addison–Wesley (1996)
7. Hsieh, C.-H., Gyllehaal, J. and Hwu, W.: JAVA Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results. Proceedings of the 29th Annual International Symposium on Microarchitecture (1996) 90-99
8. Kolte, P. and WOlfe, M.: Elimination of redundant array subscript range checks. Proceedings of the ACM SIGPLAN'95 Conference on Programming Languages Design and Implementation (1995) 270–278

9. Krall, A. and Grafl, R.: CACAO — A 64 bit Java VM Just–in–Time Compiler. Proceedings of the ACM 1997 Workshop on Java for Science and Engineering Computation (1997)
10. Linkholm, T. and Yellin, F.: The Java Virtual Machine Specification. Addison–Wesley (1997)
11. M.Mohnen: Efficient Compile–Time Garbage Collection for Arbitrary Data Structures. Technical report, Aachen University of Technology (1995)
12. Muller, G., Moura, B., Bellard, F. and Consel, C.: Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. Proceedings of the 3rd Conference on Object–Oriented Technologies and Systems (1997)
13. Proebsting, T., Townsend, G., Bridges, P., Hartman, J., Newsham, T. and Watterson, S.: Toba: Java For Applications a Way Ahead of Time(WAT) Compiler. Proceedings of the 3rd Conference on Object–Oriented Technologies and Systems (1997)
14. S.P.Midkiff, J.E.Moreira and M.snir: Optimizing array reference checking in Java programs. IBM Systems Journal, Vol.37, No.3 (1998) 409-453