Reducing Virtual Call Overheads in a Java VM Just-in-Time Compiler*

Junpyo Lee, Byung-Sun Yang, Suhyun Kim
SeungIl Lee, Yoo C. Chung, Heungbok Lee
Je Hyung Lee, Soo-Mook Moon
walker@altair.snu.ac.kr
School of Electrical Engineering
Seoul National University
Seoul 151-742, Korea

Kemal Ebcioğlu
Erik Altman
kemal@us.ibm.com
IBM T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Abstract

Java, an object-oriented language, uses virtual methods *to support the extension and reuse of classes. Unfortunately, virtual method calls affect performance and thus require an efficient implementation, especially when just-in-time (JIT) compilation is done.* Inline caches *and* type feedback *are solutions used by compilers for dynamically-typed object-oriented languages such as SELF [1, 2, 3], where virtual call overheads are much more critical to performance than in Java. With an inline cache, a virtual call that would otherwise have been translated into an indirect jump with two loads is translated into a simpler direct jump with a single compare. With type feedback combined with adaptive compilation, virtual methods can be inlined using checking code which verifies if the target method is equal to the inlined one.*

This paper evaluates the performance impact of these techniques in an actual Java virtual machine, which is our new open source Java VM JIT compiler called LaTTe *[4]. We also discuss the engineering issues in implementing these techniques.*

Our experimental results with the SPECjvm98 benchhmarks indicate that while monomoprhic inline caches and polymorphic inline caches achieve a speedup as much as a geometric mean of 3% and 9% respectively, type feedback cannot improve further over polymorphic inline caches and even degrades the performance for some programs.

Keywords *virtual method call, Java JIT compilation, inline cache, type feedback, adaptive compilation*

*This work has been supported by IBM T. J. Watson Research Center under a sponsored research agreement.

1 Introduction

Java is a recently created object-oriented programming language [5]. As an object-oriented programming language, it supports *virtual methods*, which allow different code to be executed for objects of different types with the same call.

Virtual method calls in Java incur a performance penalty because the target of these calls can only be determined at run-time based on the actual type of objects, requiring run-time type resolution. For example, extra code needs to be generated by a just-in-time (JIT) compiler such that in many Java JIT compilers like Kaffe [6], CACAO [7], and LaTTe [8], a virtual method call is translated into a sequence of loads followed by an indirect jump rather than a direct jump as for other static method calls.

In dynamically-typed object-oriented languages such as SELF, however, virtual calls cannot be implemented by using simple sequences of loads followed by an indirect jump like in Java [1]. Furthermore, virtual calls are much more frequent than in Java. So, two aggressive techniques have been employed to reduce virtual call overheads: *inline caches* and *type feedback*. With these techniques, a virtual method call can be translated into a simpler sequence of compare then direct jump or can even be inlined with type checking code. Although both techniques are certainly applicable to Java, little is known about their performance impact. Since virtual method calls are less frequent and less costly in Java while both techniques involve additional translation overhead, it is important to evaluate these technques separately since the results from SELF may not apply.

This paper evaluates both techniques in an actual Java JIT compiler. The compiler is included in our open source Java virtual machine called **LaTTe** [8]. Although the implementation of both techniques in LaTTe was straightforward, there were a few tradeoffs and optimization opportunities which we want to discuss in this paper. We also provide detailed analysis of their performance impact on Java programs in the SPECjvm98 benchmark suite.

The rest of the paper is organized as follows. Chapter 2 briefly reviews method calls in Java and summarizes the virtual method call mechanism used by the LaTTe JVM. Chapter 3 describes how we implemented inline caches and type feedback in LaTTe. Chapter 4 shows experimental results. Related work is described in section 5 and the summary follows in section 6.

2 Background

2.1 Method invocation in Java

The Java programming language provides two types of methods: *instance methods* and *class methods* [9]. A class method is invoked based on the class it is declared in via `invokestatic`. Because it is bound statically, the JIT compiler knows which method will be invoked at compile time.

An instance method, on the other hand, is always invoked with respect to an object, which is sometimes called a *receiver*, via `invokevirtual`. Because the actual type of the object is known only at run-time (i.e., bound dynamically), the JIT compiler cannot generally determine its target at compile time. There are some instance methods that can be bound statically, though. Examples are final methods, private methods, all methods in final classes, and instance methods called through the `invokespecial` bytecode (e.g., instance methods for special handling of superclass, private, and instance initialization [9]).

Generally, a method invocation incurs overheads such as creating a new activation record, passing arguments, and so on. In the case of dynamic binding, there is the additional overhead of finding the target method (which is called the *method dispatching* overhead).

2.2 LaTTe JIT Compiler and Virtual Method Tables

LaTTe is a virtual machine which is able to execute Java bytecode. It includes a novel JIT compiler targeted to RISC machines (specifically the UltraSPARC). The JIT compiler generates code with good quality through a clever mapping of Java stack operands to registers with negligible overhead [8]. It also performs some traditional optimizations such as common subexpression elimination or loop invariant code motion. Additionally, the runtime components of LaTTe, including thread synchronization [10], exception handling [11], and garbage collection [12], have been optimized. As a result, the performance of LaTTe is competitive to that of Sun's HotSpot [13] and Sun's JDK 1.2 production release [14].

LaTTe maintains a *virtual method table* (VMT) for each loaded class. The table contains the start address of each method defined in the class or inherited from the superclass. Due to the use of single inheritance in Java, if the start address of a method is placed at offset n in the virtual method table of a class, it can also be placed at offset n in the virtual method tables of all subclasses of the class. Consequently, the offset n is a translation-time constant. Since each object includes a pointer to the method table of its corresponding class, a virtual method invocation can be translated into an indirect function call after two loads: load the virtual table, indexing into the table to obtain the start address, and then the indirect call.

For statically-bound method calls, LaTTe generates a direct jump at the call site, or inlines the target method unless the bytecode size is huge (where the invocation overhead would be negligible) or the inlining depth is large (to prevent recursive calls from being inlined infinitely).

3 Inline Caches and Type Feedback

In this section, we review the techniques of inline caches and type feedback and describe our implementations. We use the example class hierarchy in Figure 1 throughout this section. Both classes `B` and `C` are subclasses of `A` and have an additional field as well as the one inherited from class `A`. Class `B` inherits the method `GetField1()` from class `A`, and class `C` overrides it. All assembly code in this section are in SPARC assembly.

```
class A {
    int field1;
    int GetField1() { return field1; }
}

class B extends A {
    int field2;
    int GetField2() { return field2; }
}
class C extends A {
    int field3;
    int GetField1() { return 0; }
    int GetField3() { return field3; }
}
```

Figure 1. Example class hierachy

3.1 Inline Caches

3.1.1 Monomorphic Inline Caches

When a JIT compiler translates `obj.GetField1()` in Figure 2(a), it cannot know which version of `GetField1()` will actually be called because `obj` can be an object of class `C` as well as class `A` or `B`. Even if class `C` does not exist, the JIT compiler cannot be sure whether if `A`'s `GetField1()` should be called because class `C` can be dynamically loaded later. With a VMT this call is translated into a sequence of load-load-indirect jump, as shown in Figure 2(b), where `#index` means the offset in the VMT (# denotes a translation-time constant).

```
void dummy() {          // o0 contains object
   A obj;               ld    [%o0], %g1
   ...                  // indexing the VMT
   obj = ...;           ld    [%g1+#index], %g1
   obj.GetField();      jmpl %g1
}

       (a)                      (b)
```

Figure 2. Example virtual call in Java and corresponding VMT sequences

The inline cache is a totally different method dispatching mechanism which "inlines" the address of the last dispatched method at a call site. Figure 3(a) shows the translated code using a *monomorphic inline cache* (MIC), where the call site just jumps to a system lookup routine called `method_dispatcher` via a stub. This stub code sets the register `%g1` to `#index`, and thus `method_dispatcher` can determine the called method.

It is called an *empty inline cache* because there is no history for the target method yet. When the call is executed for the first time, `method_dispatcher` finds the target method based on the type of the receiver, translates it if it has not been translated yet, and updates the call site to point to the translated method, which is prepended by the type checking code[1]. Figure 3(b) shows the state of the inline cache when the first encountered receiver is an object of class `A`. Now, our inline cache includes history for one method invocation to the target method. Detailed type checking code is shown in figure 4.

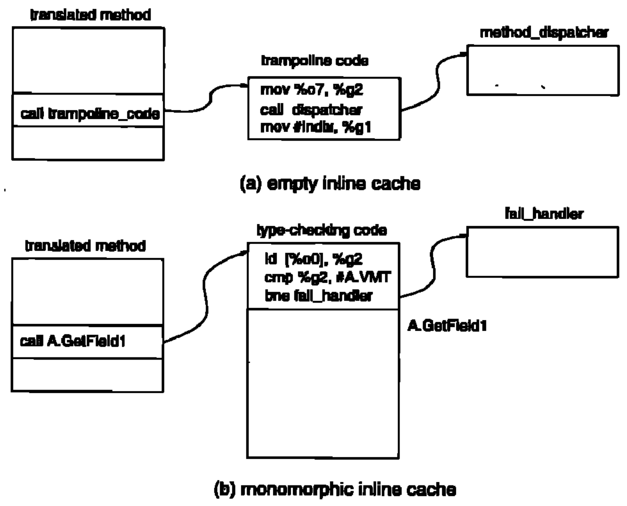


Figure 3. Monomorphic inline caches

Until a receiver with a different type is encountered, the state of the inline cache does not change. If such a receiver is encountered, `fail_handler` operates just like `method_dispatcher`: find the target method, translate it if it has not been translated yet, and update the call site.

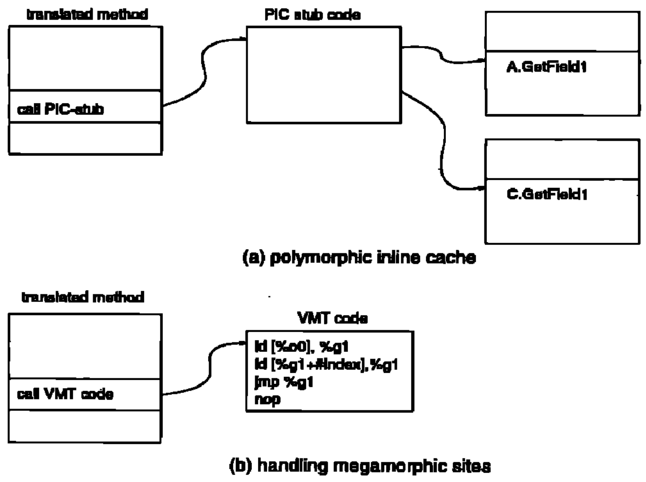
3.1.2 Polymorphic Inline Caches

A *polymorphic inline cache* (PIC) differs from a MIC in dealing with the failure of type checking. Instead of updating the call site repeatedly, it creates a PIC stub code, and makes the call site point to this stub code. The PIC stub code is composed of a sequence of compare, branch, and direct jump instructions where all previously encountered receiver types and corresponding method addresses are inlined. Figure 5(a) shows the status of a call site and the corresponding PIC stub code when the call site encounters objects of class `A` and class `C`. The detailed PIC stub code is as shown in Figure 6.

[1]Since it is possible that a method has multiple type checking codes due to inheritance, a type checking code can be separated from the corresponding method body.

```
/* object pointer is passed via o0 register.
   g1, g2, g3 are scratch registers. */

ld    [%o0], %g2              // load first word of an object
                              // which is VMT pointer of the object

/* A.VMT(VMT pointer of class A) is a translation-time constant. */
sethi %hi(#A.VMT), %g3        // load 32bit constant value
or    %g3, %lo(#A.VMT), %g3   //
cmp   %g2, %g3                // compare two VMT pointers
bne   FAIL_HANDLER            // branch to fail_handler code
                              // if two VMTs are equal
mov   %o7, %g2                // delay slot instruction


/* If type checking code can be located in front of method body,
   this code is not required */
JUMP_TO_TARGET:
call  address of A.GetField1
mov   %g2, %o7                // to prevent from returning back here


/* In our implementation FAIL_HANDLER is located in front of above code */
FAIL_HANDLER:
/* index of the called method in VMT is translation-time constant */
call  fail_handler            // call fail_handler
mov   #index, %g1             // delay slot instructoin
                              // index value is passed to fixup function
                              // via g1 register
```

Figure 4. Detailed type checking code



```
ld [%o0], %g1
ld [%g1+#index],%g1
jmp %g1
nop
```

Figure 5. Polymorphic inline caches

It is not practical for the PIC stub code to grow without limit. If the number of entries in a PIC stub code exceeds a pre-determined value, the corresponding call site is called a *megamorphic site*, and we use VMT-style code instead. Since this code only depends on the index value in the VMT, it can be shared among many call sites. Figure 5(b) explains how megamorphic sites are handled. Although MICs are used in SELF for megamorphic sites, this is only because the VMT-style mechanism cannot be used in SELF, and we think that VMT-style code is more appropriate for megamorphic sites than MICs since the latter may cause frequent updates of the call site, with frequent I-cache flushes as a result.

There are several variations of PICs. If space is tight, the PIC stub can be shared among identical call sites[2]. This type of PIC is called a *shared PIC*, while the former type is called a *non-shared PIC* when the distinction is required.

The PIC stub code can contain counting code for each type test hit and can be reordered based on the frequency of them to reduce the number of type tests needed to find the target. If the reordering is performed only once, and then a PIC stub which had been reordered but without counting code is used, it is called a *counting PIC*. It is also possible that the reordering is performed periodically and PIC stubs always have

[2]If the possible sets of target methods are the same, we call these call sites identical.

```
   /* object pointer is passed via o0 register.
      g1, g2, g3 are scratch registers. */

   ld     [%o0], %g2              // load first word of an object
                                  // which is VMT pointer of the object
   mov    %o7, %g1                // save return address in g1 register

   /* A.VMT(VMT pointer of class A) is a translation-time constant. */
   sethi %hi(#A.VMT), %g3         // load 32bit constant value
   or     %g3, %lo(#A.VMT), %g3
   cmp    %g2, %g3                // compare two VMT pointers
   bne    next1                   //
   nop                            // delay slot instruction
   call   address of A.GetField1 // jump to A.GetField1 if two VMTs are equal
   mov    %g1, %o7                // set correct return address

   /* C.VMT(VMT pointer of class C) is a translation-time constant. */
next1:
   sethi %hi(#C.VMT), %g3
   or     %g3, %lo(#C.VMT), %g3
   cmp    %g2, %g3
   bne    next2
   nop
   call   address of C.GetField1
   mov    %g1, %o7

next2:
   call   fixupFailedCheckFromPIC // call fixup function
   nop
```

Figure 6. Detailed PIC stub code

counting code, and this type of PIC is called a *periodic PIC*.

3.1.3 VMT vs. Inline Caches

Inline caches are favored over VMTs for two reasons. First, the VMT mechanism requires an indirect jump, which is not easily scheduled by modern superscalar microprocessors[3] [15, 16]. Whereas inline caches can be faster in modern microprocessors which do branch prediction. Second, VMTs do not provide any information about call sites. With inline caches, we can get information about the receivers which has been encountered, though MICs can only give the last one. This information can be used for other optimizations, such as method inlining.

3.2 Type Feedback

Although inline caches can reduce the method dispatch overhead at virtual call sites, the call overhead itself still remains. In order to reduce the call overhead, we need to inline the method.

The idea of type feedback [3] is to extract type information of virtual calls from previous runs and feed it back to the compiler for optimization. With type feedback, a virtual call can be inlined with guards which verifies if the target method is equal to the inlined method (we call this *conditional inlining*).

3.2.1 Framework of Type Feedback

In our implementation, type feedback is based on PICs, since it can provide more accurate information for call sites than MICs or VMTs. Type feedback also requires an *adaptive compilation* framework, and has been implemented on an adaptive version of LaTTe, which selects methods to aggressively optimize based on method run counts. When a method is called for the first time, it is translated with register allocation and traditional optimizations[4] while virtual method calls

[3]The cost of an indirect jump is higher in the UltraSparc due to the lack of a BTB (Branch Target Buffer).

[4]We optimize even during the initial translation to isolate the performance impact of inlining for a fair comparison with the other configurations in our experiments; See section 4.1

within it are handled by PICs. If the number of times this method is called exceeds a certain threshold, it is retranslated with conditional inlining also being done.

3.2.2 Conditional Inlining

The compiler decides whether a call site is inlined or not based on the status of inline caches. For example, if the call site in figure 3(b) remains as a monomorphic site, at retranslation time it will be inlined with type checking code as follows:

```
if (obj.VMT == #A.VMT)
   i = obj.field1;
else
   i = obj.GetField1(); //load-load-indirect jump
```

If the call site points to a PIC stub code, but there is only one target method in the stub code, then we can do conditional inlining, except this time the comparison is based on addresses, not on receiver types. For example, if our PIC stub code in Figure 5(a) were composed of type checks for class A and class B (not class A and class C), the addresses of both GetField1s will be identical. So, we can inline the method, but the type check should be replaced by an address check, which includes access to the VMT (two loads), as follows:

```
if (obj.VTM[#index of method GetField1]
     == #address of A.GetField1) // load-load
   i = obj.field1;
else
   i = obj.GetField1(); //load-load-indirect jump
```

If the frequency information of each type or method is available by using a counting PIC, we can improve on the all-or-nothing strategy. Even though there are multiple receivers or multiple target methods, we can inline the call site with a type test or an address test if one case is dominant among the other cases in the PIC stub. Currently, the criteria value to decide whether a case is dominant or not is 80%: If the count of type test hits in a PIC stub exceeds 80% of the total count of PIC stub, it is inlined with type checking code.

3.2.3 Static Type Prediction

For those call sites located at untaken execution paths during initial runs, we do not have any information on the most probable receiver type (but these can be collected even after retranslation). However, if the class of the object on which a virtual call is made has no subclass at translation time, we can easily predict that the receiver type would be the class at runtime. Althouth Java allows dynamic class loading, we found that this prediction is quite accurate for most programs. For the following case, for example, we can inline the call site even if there is no information in the inline cache during retranslation.

```
B obj;
if () { obj = ...;
        obj.GetField2();
}
```

3.2.4 Inlining Heuristic: single vs. multiple

In previous section, we inlined only a method for a virtual call site. It can be possible that a call site has more than two target methods, and neither of them are not dominant. In such a case, we might lose inlining opportunities by restricting the number of inlineable methods for a call site. However, we found that this is not the case for Java programs which we use for testing. In the programs, most (95%) virtual call sites call just a single callee, and enabling to inline multiple methods for a call site does not increase the number of inlined virtual calls significantly.

4 Experimental results

In this section, we evaluate the performance impact of inline caches and type feedback.

4.1 Experimental Environment

Our benchmarks are composed of the SPECjvm98 benchmark suite[5] [17], and table 1 shows the list of programs and a short description for each.

Benchmark	Description	Bytes
_201_compress	Compress Utility	24326
_202_jess	Expert Shell System	45392
_209_db	Database	26425
_213_javac	Java Compiler	92000
_222_mpegaudio	MP3 Decompressor	38930
_227_mtrt	Multi-thread Raytracer	34009
_228_jack	Parser Generator	51380

Table 1. Java Benchmark Description and Translated Bytecode Size

Table 2 lists the configurations used in our experiments. LaTTe-VMT, LaTTe-MIC, and LaTTe-PIC are all the same except in how they handle virtual calls: by using VMTs, MICs and PICs respectively. LaTTe-TF inlines virtual calls using type feedback on an adaptive version of LaTTe, where initial translation is identical to LaTTe-PIC, as described in Section 3.2. Variations in PICs are denoted with each variation surrounded

[5] _200_check is excluded since it is for correctness testing only.

by brackets. For example, a shared PIC is denoted by PIC[S], a counting PIC by PIC[C], and a periodic PIC by PIC[P]. The default version of a PIC is denoted by PIC[] when the distinction is needed.

System	Description
LaTTe-VMT	Virtual calls are handled by VMT.
LaTTe-MIC	Virtual calls are handled by MIC.
LaTTe-PIC	Virtual calls are handled by PIC.
LaTTe-TF	Virtual calls are inlined using type feedback at the retranslation time.

Table 2. Systems used for benchmarking

Our test machine is a Sun Ultra5 270MHz with 256 MB of memory running Solaris 2.6, tested in single-user mode. We ran each benchmark 5 times and took the minimum running time[6], which includes both the JIT compilation overhead and the garbage collection time.

4.2 Characteristics of Virtual Calls

Table 3 shows the characteristics of virtual calls. In the table, V-Call means the total count of virtual calls, M-Call means the total count of monomorphic calls, and S-target means the total count of virtual calls where target method is just one at runtime. About 85% of virtual calls are monomophic calls, and about 90% of virtual calls have only one target method. Some programs like _213_javac, _227_mtrt, and _228_jack have many call sites where the target method is one, even though there are multiple receiver tyeps, and thus we can expect that address check inlining may be effective on these programs. We can also expect that _201_compress will not be much affected by how virtual calls are implemented, since the number of virtual calls is extremely small.

4.3 Analysis of Monomorphic Inline Caches

Table 4 shows the characteristics of MICs. In the table, V-Call means the total count of virtual calls, P-Call means the total count of calls which are called at polymorphic call sites, and Miss means the total count of type check misses. There is no trend in the miss ratios. While some programs such as _209_db and _228_jack have very low miss ratios compared to V-Call, type check misses are very common in _202_jess and _213_javac. Since a type check miss requires invalidating part of the I-cache, the miss ratio can greatly affect the overall performance. So we can expect that the performance of _202_jess and _213_javac may be worse with monomorphic inline caches.

4.4 Analysis of Polymorphic Inline Caches

Tables 5, 6, 7, and 8 shows the average number of type checks in a PIC stub for each configuration of PICs. Each column of the tables is divided by the maximum number of possible entries in a PIC stub and the threshold value which determines when reordering takes place. This value is applied to both PIC[C] and PIC[P], although the reordering takes place just once in the former while it takes place periodically in the latter. If the threshold value is zero, it means no counting.

At first glance, we find that the numbers of average type checkings are very small, even though monomorphic sites are not included in the numbers. If the average number is calculated for every inline cache, including monomorphic sites, the number will be even more closer to 1. However, only if counting is enabled are the numbers are less than 2 in every case.

From table 5 and 6, it is clear that counting PICs are effective in reducing the number of type checks in a PIC stub, except for _228_jack. Although the numbers are generally reduced as the threshold value is increased, they are unchanged or even increased for some programs and seem to saturate at certain values. So simply increasing the threshold does not guarantee improvements. _228_jack has very different characteristics from other programs, and these come from a single polymorphic site[7] which accounts for about a half of the total polymorphic calls, and exhibit very strange behavior: after the call site is switched to a polymorphic inline cache from a monomorphic inline cache, the newly encountered type is received repetitively for about a thousand times, and thereafter the former type is used repetitively for over 1 million times. So the default PIC scheme without counting is better than that with counting in this case.

We can also see the effect of inaccuracies caused by sharing PIC stubs from table 5 and 6. Although it seems natural that a non-shared version would be more accurate than a shared version, the difference is not apparent in non-counting versions. For other configurations, the numbers in table 5 are better than those in table 6, except for _213_javac, where some PIC stubs are changed into VMT-style code because sharing increases the number of entries. However, the difference is lower than 0.3 for most cases.

[6]For the SPECjvm98 benchmarks, our total elapsed running time is not comparable with a SPECjvm98 metric.

[7]A call site in "indexOf" method in java.util.Vector class which calls "equals" method.

Benchmark	V-Call (×1000)	M-Call (×1000)	S-target (×1000)	M-Call/V-Call	S-target/V-Call
_201_compress	12.9	11.6	11.8	0.897	0.912
_202_jess	34,306	27,718	28,435	0.808	0.829
_209_db	16,492	16,479	16,480	0.999	0.999
_213_javac	53,130	37,840	42,552	0.712	0.801
_222_mpegaudio	10,025	8,781	8,841	0.876	0.882
_227_mtrt	264,612	240,130	261,775	0.907	0.989
_228_jack	17,247	14,094	16,959	0.817	0.983
GEOMEAN				0.854	0.904

Table 3. Characteristics of virtual calls

Benchmark	V-Call(×1000)	P-Call(×1000)	Miss(×1000)	Miss/V-Call	Miss/P-Call
_201_compress	12.9	1.3	0.58	0.045	0.433
_202_jess	34,306	6,587	3201	0.093	0.486
_209_db	16,492	12	2	0.000	0.203
_213_javac	53,130	15,289	5487	0.103	0.359
_222_mpegaudio	10,025	1,244	54	0.005	0.044
_227_mtrt	264,612	24,482	583	0.002	0.024
_228_jack	17,247	3,152	1	0.000	0.001

Table 4. Characteristics of monomorphic inline caches

Tables 7 and 8 are shown for comparison with tables 5 and 6. Although a periodically reordered PIC is hard to use in real implementations because it always incurs counting overhead which involves load-add-store sequences, it can be seen as a somewhat ideal configuration in terms of the number of type checks. The difference between the periodic version and the non-periodic version is lower than 0.2 in most programs, and thus the quality of counting PICs are quite acceptable.

Table 9 shows the space overhead of PICs for both non-shared and shared versions. In the table, N means the number of PIC stubs, and max means the possible maximum number of entries in each PIC stub. The overhead seems to be small in most programs except for _213_javac, where the shared version can greatly reduce the overhead. Since the sharing of PIC stubs does not degrade the performance severely for most programs, shared PICs can be useful when space is tight.

4.5 Analysis of Type Feedback

Tables 10, 11, and 12 show the effect of type feedback in terms of the number of inlined virtual calls. As a base system, four different PIC variations (PIC[S], PIC[SC], PIC[], and PIC[C]) are used. The main purpose of PICs is providing profile information. So a larger number of maximum entries in PIC (10) is used, and the threshold for counting PICs is set to 1000 in order not to affect the accuracy too much[8]

Generally, the number of inlined virtual calls is reduced as the retranslation threshold is increased. Although more accurate profile information is available with a high retranslation threshold than with a low retranslation threshold, the opportunities missed by delaying retranslation seems to be high. In addition, the number of inlined calls for many programs is constant regardless of the type of PICs. There can be two reasons: One possibility is that the method is too big to be inlined. In this case, inlining such a method is related to the inlining heuristic and is beyond the scope of this paper. The other possiblility is that a single method is not dominant for a call site. However, this is not the case for SPECjvm98 benchmarks, and it will be shown in following section.

For some programs like _213_javac and _227_mtrt, which are affected by the type of PICs used, the counting version is preferable to the non-counting version. A call site which has multiple receiver types and thus can be inlined only with an address check, can be inlined with a type check if counting information is available and only one receiver type is dominant. And a call site which involves more than two target methods and thus cannot be inlined in our implementation, can be inlined if one method is dominant. While _213_javac is influenced by both effects, i.e., the amount of type check inlining and address check inlining are increased,

[8]Since a PIC is reused for a call site where method inlining is not done, the value should not be too large.

Benchmark	PIC max entry = 5					PIC max entry = 10				
	0	10	50	100	200	0	10	50	100	200
_201_compress	1.540	1.480	1.498	1.526	1.540	1.540	1.480	1.498	1.526	1.540
_202_jess	1.517	1.488	1.507	1.486	1.486	1.517	1.488	1.507	1.486	1.486
_209_db	1.541	1.461	1.467	1.475	1.488	1.541	1.461	1.467	1.475	1.488
_213_javac	1.500	1.259	1.227	1.226	1.216	1.949	1.810	1.771	1.777	1.674
_222_mpegaudio	2.120	1.359	1.267	1.266	1.269	2.120	1.359	1.267	1.266	1.269
_227_mtrt	1.305	1.162	1.108	1.092	1.092	1.305	1.162	1.108	1.092	1.092
_228_jack	1.209	1.800	1.800	1.800	1.800	1.209	1.800	1.800	1.800	1.800

Table 5. Average number of type checks with non-shared counting PICs (LaTTe-PIC[C])

Benchmark	PIC max entry = 5					PIC max entry = 10				
	0	10	50	100	200	0	10	50	100	200
_201_compress	1.790	1.693	1.711	1.728	1.769	1.790	1.693	1.711	1.728	1.769
_202_jess	1.579	1.570	1.578	1.576	1.576	1.579	1.570	1.578	1.576	1.577
_209_db	1.700	1.653	1.594	1.601	1.614	1.700	1.653	1.594	1.601	1.614
_213_javac	1.508	1.185	1.176	1.175	1.180	1.722	1.686	1.577	1.444	1.450
_222_mpegaudio	2.161	1.358	1.267	1.267	1.269	2.161	1.358	1.267	1.267	1.269
_227_mtrt	1.220	1.099	1.099	1.100	1.100	1.220	1.099	1.099	1.100	1.100
_228_jack	1.051	1.980	1.980	1.981	1.981	1.398	1.895	1.895	1.895	1.895

Table 6. Average number of type checks with shared counting PICs (LaTTe-PIC[SC])

the former effect is dominant in _227_mtrt, where the increase in the amount of type checking inlining is almost the same as that of decrease in address checking inlining.

4.6 Analysis of Inlining Heuristic

Table 13, 14, and 15 show the total number of inlined virtuals call under different inlining heuristics: inlining single method for a call site and inlining all the possible methods for a call site. The numbers in the column of `Single method inlining` are the sum of type check inlining and address check inlining in the tables of previous section. The numbers in the column of `All method inlining` are obtained by inlining all the methods which have been encountered during initial run and can be inlined by our inlining rule (size and depth)[9]. As we have expected from the fact that about 90% of virtual call sites have only one target method, there is little improvements in terms of the number of inlined virtual calls, even though all possible methods are permitted to be inlined. Only _213_javac has opportunites to be improved by inlining multiple methods for a call site. So, we can think that inlining only a method for a call site is sufficient for most programs. If many call sites still remain as not inlined, this is due to other factors like method size or inlining depth.

[9]Counting version is excluded since there is no difference from non-counting version when all the possible methods are inlined. And shared version is also excluded since it causes code explosion in _213_javac.

4.7 Performance Impact of Inline Caches and Type Feedback

Table 16 shows the total running time (tot) of each program for 4 configurations of **LaTTe**. Translation overhead (tr) is also included in the total running time. Since there is little difference in running time between the different configurations of PIC and TF, only one instance each from both are listed here. The exact configurations are like this:

1. **PIC** non-shared counting PICs, maximum number of entries = 5, reordering threshold = 100

2. **TF** based on non-shared counting PICs, maximum number of entries = 10, reordering threshold = 1000, retranslation threshold = 10

On the whole, MICs improve the performance of **LaTTe** by a geometric mean of 3.0%, PICs by 9.0%, and type feedback by 7.4%, compared with LaTTe-VMT. As pointed out in the previous section, MICs exhibit poor performance in _202_jess and _213_javac, which have high ratio of type check misses. PICs, as we have expected, solved the problem experienced by MICs which is exposed in the above programs, without severe degradation in other programs, and improves the performance of almost all programs compared with VMTs.

However, type feedback seems to be effective only for _227_mtrt, where the number of inlined virtual calls are much larger than other programs. The number of

Benchmark	PIC max entry = 5				PIC max entry = 10			
	10	50	100	200	10	50	100	200
_201_compress	1.480	1.498	1.526	1.540	1.480	1.498	1.526	1.540
_202_jess	1.366	1.366	1.367	1.367	1.366	1.367	1.367	1.367
_209_db	1.461	1.467	1.475	1.492	1.461	1.467	1.475	1.492
_213_javac	1.196	1.197	1.197	1.198	1.620	1.620	1.623	1.623
_222_mpegaudio	1.264	1.265	1.267	1.270	1.264	1.265	1.267	1.270
_227_mtrt	1.033	1.033	1.033	1.033	1.033	1.033	1.033	1.033
_228_jack	1.019	1.019	1.019	1.020	1.019	1.019	1.019	1.020

Table 7. Average number of type checks with non-shared periodic PICs (LaTTe-PIC[P])

Benchmark	PIC max entry = 5				PIC max entry = 10			
	10	50	100	200	10	50	100	200
_201_compress	1.693	1.711	1.728	1.769	1.693	1.711	1.728	1.769
_202_jess	1.429	1.429	1.430	1.430	1.429	1.430	1.430	1.430
_209_db	1.590	1.594	1.601	1.618	1.590	1.594	1.601	1.618
_213_javac	1.163	1.163	1.165	1.164	1.385	1.386	1.385	1.386
_222_mpegaudio	1.265	1.266	1.267	1.269	1.265	1.266	1.267	1.269
_227_mtrt	1.035	1.035	1.035	1.035	1.035	1.035	1.035	1.035
_228_jack	1.030	1.030	1.030	1.031	1.114	1.114	1.114	1.114

Table 8. Average number of type checks with shared periodic PICs (LaTTe-PIC[SP])

inlined virtual calls in other programs seems to be low to compensate for both the retranslation overhead (increase in translation time) and inlining overhead (increase in code size, register pressure, and so on). Since the performance of type feedback depends on the inlining heuristic as well as the retranslation framework, both have to be carefully implemented to measure the effect of type feedback correctly. Our implementation could be improved in both of these points.

However, the result from _227_mtrt gives us some expectation about the effect of type feedback. In _227_mtrt, some getter methods such as GetX, GetY, and GetZ are very frequent, and the performance of the benchmark is greatly improved by inlining such methods. So the more common a coding style using accessor methods are, the more effective type feedback could be.

5 Related work

Our work is based on polymorphic inline caches and type feedback. Polymorphic inline caches were studied by Urs Hölzle et al. [2] in the SELF compiler and achieved a median speedup of 11% over monomorphic inline caches. Type feedback was proposed by Urs Hölzle and David Ungar [3]. They implemented type feedback in the SELF compiler using PICs and improved performance by a factor of 1.7 compared with non-feedback compiler. Since virtual calls are more frequent in SELF, and also since the default dispatching overhead is much larger than that of the VMTs which can be used in Java, they achieved larger speedup than ours. Furthermore, their measurements compare execution time while excluding translation time overhead.

The most relevant study was done by David Detlefs and Ole Agesen [18]. They also targetted Java, used conditional inlining, and proposed a method test which is identical to an address test. However, they mainly concentrated on inlining rather than on inline caches, and they did not use profile information to inline virtual calls.

Gerald Aigner and Urs Hölzle [19] implemented optimizaing source-to-source C++ compiler. They used static profile information to inline virtual calls, and improves the performance by a median of 18% and reduces the number of virtual function calls by a median factor of five.

Karel Driesen et al. [16] extensively studied various dynamic dispatching mechanisms on several modern architectures. They mainly compared inline cache mechanisms and table-based mechanisms which employ indirect branches, and showed that the latter does not perform well on current hardware. They also expected that table-based approaches may not perform well on future hardware.

Olver Zendra et al. [20] have implemented polymorphism in the SmallEiffel compiler. They also eliminated use of VMTs by using a static variation of PICs and inlined monomorphic call sites. However, they relied on static type inference and did not use runtime feedback.

Benchmark	non-shared			shared		
	N	max = 5	max = 10	N	max = 5	max = 10
_201_compress	6	536	536	5	512	512
_202_jess	24	2,368	2,648	14	1,708	1,988
_209_db	6	536	536	5	512	512
_213_javac	396	54,584	65,560	86	15,476	18,136
_222_mpegaudio	25	3,008	3,008	18	2,308	2,308
_227_mtrt	62	5,604	5,604	23	2,120	2,120
_228_jack	19	1,856	1,856	12	1,380	1,380

Table 9. Size of PIC stub code

Benchmark	Type-check inlining (× 1000)				Address-check inlining (× 1000)			
	PIC[S]	PIC[SC]	PIC[]	PIC[C]	PIC[S]	PIC[SC]	PIC[]	PIC[C]
_201_compress	0	0	0	0	0	0	0	0
_202_jess	12,036	12,036	12,036	12,036	0	0	0	0
_209_db	6,547	6,547	6,547	6,547	0	0	0	0
_213_javac	7,665	8,205	7,654	8,215	427	679	489	718
_222_mpegaudio	3,337	3,337	3,337	3,337	0	0	0	0
_227_mtrt	208,238	209,811	208,238	209,113	2,230	560	2,230	1,348
_228_jack	2,396	2,396	2,396	2,396	0	0	0	0

Table 10. Inlined calls by type feedback: retranslation threshold = 10

Based on the experiences of C++ programs, Brad Caler and Dirk Grunwald [21] proposed using "if conversion", which is similiar to type feedback except that it uses static profile information.

6 Conclusion and Future work

We have implemented inline caches and type feedback in the **LaTTe** JIT compiler and evaluated these techniques.

Although some programs suffer from frequent cache misses, MICs achieve a speedup of 3% by geometric mean over VMTs. Polymorphic inline caches solve the problem experienced by MICs without incurring overheads elsewhere and achieve a speedup of 9% by geometric mean over VMTs using counting PICs. We have also tested several variations of PICs and shown the characteristics of PICs in Java programs. Counting PICs reduce the average number of type checks in a PIC stub compared with a non-counting version, and achieve an average number of type checks close to that of a periodic version, within 0.2 for most programs. If memory is a matter of concern, then shared PICs can save space with only a reasonable degradation in performance.

The effect of type feedback is not fully shown in this study. The overall performance is even worse than that of counting PICs. Although it is true that some programs have little opportunity to improve in terms of virtual calls, the result is partly because we cannot apply optimizations selectively only when it is beneficial. However, the performance of _227_mtrt, which does many virtual calls to small methods, is greatly improved by type feedback, and gives us insight about the performance impact of type feedback. If a coding style which uses more abstraction and makes more calls becomes dominant in Java programs, type feedback will be more effective.

The study of type feedback also exposed other problems: adaptive compilation and method inlining. To avoid degradation due to type feedback, it is very important to estimate the costs incurred by retranslation and inlining, and to apply conditional inlining only to hot-spots.

References

[1] Urs Hölzle. *Adaptive Optimization For SELF: Reconciling High Performance With Exploratory Programming.* PhD thesis, Stanford University, August 1994.

[2] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the 5th European Conference on Object-Oriented Programming (ECOOP'91)*, 1991.

[3] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1994.

[4] LaTTe: A fast and efficient Java VM just-in-time compiler. http://latte.snu.ac.kr/, 1999.

Benchmark	Type-check inlining (× 1000)				Address-check inlining (× 1000)			
	PIC[S]	PIC[SC]	PIC[]	PIC[C]	PIC[S]	PIC[SC]	PIC[]	PIC[C]
_201_compress	0	0	0	0	0	0	0	0
_202_jess	12,031	12,031	12,031	12,031	0	0	0	0
_209_db	5,964	5,964	5,964	5,964	0	0	0	0
_213_javac	7,694	7,808	7,565	7,759	97	291	525	739
_222_mpegaudio	3,335	3,335	3,335	3,335	0	0	0	0
_227_mtrt	208,148	209,748	208,148	209,050	2,263	560	2,263	1,348
_228_jack	2,391	2,391	2,391	2,391	0	0	0	0

Table 11. Inlined calls by type feedback: retranslation threshold = 50

Benchmark	Type-check inlining (× 1000)				Address-check inlining (× 1000)			
	PIC[S]	PIC[SC]	PIC[]	PIC[C]	PIC[S]	PIC[SC]	PIC[]	PIC[C]
_201_compress	0	0	0	0	0	0	0	0
_202_jess	12,031	12,031	12,031	12,031	0	0	0	0
_209_db	5,964	5,964	5,964	5,964	0	0	0	0
_213_javac	7,691	7,946	7,621	8,561	97	36	526	461
_222_mpegaudio	3,335	3,335	3,335	3,335	0	0	0	0
_227_mtrt	208,148	209,748	208,148	208,516	2,263	560	2,263	1,886
_228_jack	2,391	2,391	2,391	2391	0	0	0	0

Table 12. Inlined calls by type feedback: retranslation threshold = 100

[5] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[6] T. Wilkinson. Kaffe: A JIT and interpreting virtual machine to run Java code. http://www.transvirtual.com/, 1998.

[7] A. Krall and R. Grafl. Cacao - A 64-bit Java VM just-in-time compiler. In *Proceedings of the PPoPP '97 Workshop on Java for Science and Engineering Compuation*, 1997.

[8] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioğlu, and Erik Altman. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[9] F. Yellin and T. Lindholm. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[10] Byung-Sun Yang, Junpyo Lee, Jinpyo Park, Soo-Mook Moon, and Kemal Ebcioğlu. Lightweight Monitor in Java Virtual Machine. In *Proceedings of the 3rd Workshop on Interaction between Compilers and Computer Architectures*, Oct 1998.

[11] SeungIl Lee, Byung-Sun Yang, Suhyun Kim, Seongbae Park, Soo-Moon Moon, Kemal Ebcioğlu, and Erik Altman. On-demand translation of Java exception handlers in the LaTTe JVM just-in-time compiler. In *Proceedings of the 1999 Workshop on Binary Translation (Binary99)*, 1999.

[12] Yoo C. Chung, Soo-Mook Moon, Kemal Ebcioğlu, and Dan Sahlin. Reducing sweep time for a nearly empty heap. To appear in the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00).

[13] Java Hotspot performance engine. http://java.sun.com/products/hotspot/, April 1999.

[14] Java 2 SDK (1.2.1_03) production release for Solaris. http://www.sun.com/solaris/java, 1999.

[15] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in C++. In *Proceedings of the 1996 ACM Conference on Object Oriented Programming Systems, Languages, and Applicatrions (OOPSLA)*, 1996.

[16] Karel Driesen, Urs Hölzle, and Jan Vitek. Message dispatch on pipelined processors. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, 1995.

[17] SPEC JVM98 benchmarks. http://www.spec.org/osg/jvm98/, 1998.

[18] David Detlefs and Old Agesen. Inlining of virtual methods. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, 1999.

[19] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP'96)*, 1996.

[20] Oliver Zendra, Dominique Colnet, and Suzanne Collin. Efficient dynamic dispatch without virtual function tables. the SmallEiffel compiler. In *Proceedings of the 1997 ACM Conference on Object Oriented Programming Systems, Languages, and Applicatrions (OOPSLA)*, 1997.

[21] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of the 1994 ACM Symposium on Principles of Programming Languages (POPL)*, 1994.

Benchmark	Single method inlining (× 1000)				All method inlining (× 1000)
	PIC[S]	PIC[SC]	PIC[]	PIC[C]	PIC[]
_201_compress	0	0	0	0	0
_202_jess	12,036	12,036	12,036	12,036	12,167
_209_db	6,547	6,547	6,547	6,547	6,547
_213_javac	8,092	8,884	8,143	8,933	11,365
_222_mpegaudio	3,337	3,337	3,337	3,337	3,473
_227_mtrt	210,468	210,371	210,468	210,461	210,463
_228_jack	2,396	2,396	2,396	2,396	2,396

Table 13. Comparison of inlined calls: retranslation threshold = 10

Benchmark	Single method inlining (× 1000)				All method inlining (× 1000)
	PIC[S]	PIC[SC]	PIC[]	PIC[C]	PIC[]
_201_compress	0	0	0	0	0
_202_jess	12,031	12,031	12,031	12,031	12,162
_209_db	5,964	5,964	5,964	5,964	5,964
_213_javac	7,792	8,099	8,090	8,499	11,610
_222_mpegaudio	3,335	3,335	3,335	3,335	3,471
_227_mtrt	210,411	210,308	210,411	210,398	210,405
_228_jack	2,391	2,391	2,391	2,391	2,391

Table 14. Comparison of inlined calls: retranslation threshold = 50

Benchmark	Single method inlining (× 1000)				All method inlining (× 1000)
	PIC[S]	PIC[SC]	PIC[]	PIC[C]	PIC[]
_201_compress	0	0	0	0	0
_202_jess	12,031	12,031	12,031	12,031	12,162
_209_db	5,964	5,964	5,964	5,964	5,964
_213_javac	7,789	7,982	8,147	9,022	11,654
_222_mpegaudio	3,335	3,335	3,335	3,335	3,471
_227_mtrt	210,411	210,308	210,411	210,402	210,405
_228_jack	2,391	2,391	2,391	2,391	2,391

Table 15. Comparison of inlined calls: retranslation threshold = 100

Benchmark	VMT		MIC		PIC		TF		Speedup		
	tot[1]	tr	tot[2]	tr	tot[3]	tr	tot[4]	tr	[1]/[2]	[1]/[3]	[1]/[4]
_201_compress	69.23	2.72	69.10	2.74	69.68	2.71	66.18	2.97	1.00	0.99	1.05
_202_jess	41.48	4.39	45.88	4.32	38.40	4.31	41.73	5.63	0.90	1.08	0.99
_209_db	74.27	2.92	66.78	2.90	67.60	2.88	66.99	3.10	1.11	1.10	1.11
_213_javac	63.94	8.16	71.76	7.99	59.01	7.98	66.22	13.88	0.89	1.08	0.97
_222_mpegaudio	47.49	4.44	47.36	4.40	47.08	4.37	51.01	4.83	1.00	1.01	0.93
_227_mtrt	58.97	3.64	49.56	3.52	47.70	3.51	42.11	5.18	1.19	1.24	1.40
_228_jack	51.42	5.50	44.66	5.40	44.95	5.40	45.39	7.26	1.15	1.14	1.13
GEOMEAN									1.030	1.090	1.074

Table 16. Total running time