Virtual Method Resolution and Automatic Method Inlining in Java

by
Dhairya Jhunjhunwala - 200010021
Kumar Satyam - 200050064

Indian Institute of Technology, Bombay
May 2024

Guide :
Manas Thakur, IIT Bombay

Github Repo :
https://github.com/dhairyaj9/CS60004-PA4-MONO-POLY.git

Abstract

Virtual method calls are a fundamental feature offered by Java, an object-oriented programming language. However, they are also a source of degradation of performance at run time and imprecision in interprocedural analysis. In a virtual method invocation, the actual method to be called depends on the runtime type of the object being invoked upon. This may require a dynamic dispatch mechanism (e.g., a virtual method table, or vtable) to determine the correct method to call based on the object's type at runtime. This adds overhead.
We present a technique to perform method resolution statically using a Points-to Type Propagation analysis, where we identify a monomorphic virtual method invocation, and replace it with a static method invocation, thus eliminating any dispatch overhead and reducing imprecision.

Method inlining is a compiler optimization in which the method call is replaced by the body of the method being called. Method inlining is very effective in improving performance of programs that have many small methods and in which a large proportion of instructions executed are virtual calls. The opportunity for our optimization arises only at those call sites for which the set of target methods is a singleton.
We have implemented a multi-level automatic method inlining at the Java bytecode level using the Soot framework based on a set of criteria.

We demonstrate our method implementation and performance analysis results using test cases in the next sections.

Table Of Contents

1. Virtual Method Resolution

1.1 Motivation

In a virtual method invocation, the actual method to be called depends on the runtime type of the object being invoked upon. This may require a dynamic dispatch mechanism (e.g., a virtual method table, or vtable) to determine the correct method to call based on the object's type at runtime. This adds overhead.

Virtual methods are instance methods and require an object reference (this pointer) to be passed to the method. Additionally, the runtime type of the object may need to be determined to handle polymorphism, which can add overhead.

Hence, it should be clear that virtual method calls are expensive at run time and replacing them wherever possible might improve performance. Possible ways of avoiding the overhead associated with virtual method calls are either eliminating them completely or by replacing them by less expensive instructions. Eliminating a method call completely is possible if the target of the method call is known statically and we are allowed to inline the code of the called method into the caller and remove the calI instruction.

Another method to do so is to replace it with other bytecode instructions (namely invokestatic) for method invocations at call sites where the target method is known statically. These are less expensive to execute and can be used instead of the virtual call instructions if the virtual call has been resolved.

In a static method invocation, the method being called is known at compile time because static methods belong to the class rather than an instance of the class. Therefore, the method can be invoked directly without any additional lookup or dispatch overhead.

1.2 Technique

We are resolving virtual methods by modifying the bytecode instructions. We are replacing each monomorphic virtual method call with a static invoke.

Since the conversion is to the invokestatic bytecode, then a new static method has to be created and added to the class. This static method would be similar in functionality to the method being invoked by the virtual call instruction, but would differ from it in that it would have one extra explicit parameter corresponding to the implicit “this” parameter.

We are achieving a higher precision of method resolution by identifying possible types of a caller using a Points-To Type Propagation Analysis instead of the regular Class Hierarchy Analysis. This helps identify the possible types of the receiver for each callsite, not just based on the Class Hierarchy, but instead based on the Points-To information of the possible objects that are receivers of the method called. In assignment statements, the type of the assigned value is propagated to the target variable. In function calls, the types of arguments are propagated to corresponding parameters. In pointer

dereferences, the type information associated with the pointer is used to determine the type of the referenced object.

We are adopting a bottom-up method resolution structure, where each method that is resolved and converted into a monomorphic static invoke, will have all of its monomorphic virtual invokes already resolved into static invokes. A static version of a virtual method is created only once, and subsequently used in further monomorphic invocations of that virtual method.

1.3 Results

1.3.1 Static Performance Analysis

We analyse the static performance of our optimization on various call graph metrics namely different types of edges and number of monomorphic call sites.

	Without Method Resolution	**With Method Resolution**
CHA # virtual_edges	20	4
CHA #static edges	**6**	21
PTA #virtual edges	13	**2**
PTA #static edges	**6**	19
CHA #polymorphic call sites	4	**1**

CHA #monomorphic call sites	14	**21**
PTA #polymorphic call sites	1	**1**
PTA #monomorphic call sites	17	**19**

Testcase 1 Static Results

	Without Method Resolution	**With Method Resolution**
CHA # virtual_edges	18	**0**
CHA #static edges	0	**18**
PTA #virtual edges	18	**0**
PTA #static edges	0	**18**
CHA #polymorphic call sites	0	**0**
CHA #monomorphic call sites	18	**18**
PTA #polymorphic call sites	0	**0**
PTA #monomorphic call sites	18	**18**

Testcase 2 Static Results

1.3.2 Dynamic Performance Analysis

We analyse the dynamic performance of our optimization on the basis of 2 metrics. First we compare the actual number of calls made for each type of invoke statement - static and virtual. Next, we compare the time taken by the program to run.

	Without Method Resolution	**With Method Resolution**
Total #actual virtual calls	3050842	**50983**
Total #actual static calls	**3007897**	6007914
Total #actual invokes	**6058739**	6058897

	Without Method Resolution	**With Method Resolution**
Time taken to run	0.256s	**0.216s**

Testcase 1 Dynamic Results

	Without Method Resolution	**With Method Resolution**
Total #actual virtual calls	32047802	**47922**
Total #actual static calls	**7613**	32007619
Total #actual invokes	**32055415**	32055541

	Without Method Resolution	**With Method Resolution**
Time taken to run	2.502s	**2.057s**

Testcase 2 Dynamic Results

2. Automatic Method Inlining

2.1 Motivation

The basic idea in method inlining is to statically replace a method invocation instruction by the code representing the body of the method that is the target of the calI. By performing this transformation, the over-head associated with executing the method invocation instruction can be avoided.

In programs developed in an object oriented manner, the frequency of invocation instructions is expected to be considerably greater. Further, in object oriented languages the overhead of method lookup associated with these instructions make them expensive at run time.

Another factor that makes method inlining a useful optimization is that it eliminates the control flow edges because of the invocation instruction from the Control Flow Graph (CFG). Frequent branches in the code mean that some of the techniques used for optimization at the architecture level, like pipelining, cannot be performed optimally. Instruction scheduling and pipelining are techniques that are very effective if the program has a relatively simple flow of control.

The effect on performance is even more significant in object oriented languages because typically programs have methods with small bodies. These small methods are expected to contain mostly instructions to manipulate fields within the declaring

class of the method, and not very many method invocation instructions.

2.2 Criteria for Method Inlining

Once we encounter a method invocation site while traversing through our reachable methods, the next step is to decide whether it should be inlined or not. This decision is broken down into 2 parts.

The first part being **CAN** we inline the method that is called into the caller? We are determining this using the first part of our analysis where each monomorphic virtual call site is converted to a static invoke callsite. Thus each monomorphic call CAN be inlined into the caller, as we can statically determine the method that it is being resolved to.

The second part is **SHOULD** we inline the method that is called into the caller? If the invocation instruction is not an important factor in the overall execution time of the application, then the potential benefit of inlining it might not be worth the cost of actually inlining it. We are determining whether we SHOULD inline a function based on a set of static characteristics of the program.

i) Number of statements in the callee method : If the callee method is very small then it is expected to be beneficial to inline calls to it. This is because the time to execute the method invocation instruction would be a significant overhead in the overall time required to execute the method call. Thus eliminating the invocation instruction is likely to lead to significant benefits if the method call was executed frequently.

Conversely, if the callee method had a large number of statements, then it is expected to be relatively complex and the invocation instruction itself is unlikely to be the main overhead in the method call.

ii) Presence of a loop in which the invoke lies : We are speculating statically that a loop present in a program might be where a program is spending most of its time. Hence we are categorising it as a 'hot spot'. We are identifying whether a method callsite is inside a loop of the Control Flow Graph and subsequently relaxing our conditions on the callee method size inside loops. Even a larger callee method will reap benefits if inlined inside a for loop running a large number of times.

2.3 Technique

There were several steps involved in the process of inlining a method.

Multi-Level Inlining : We only inline at the call sites that we determined to be important using the criteria mentioned above. We are performing a multi-level inlining approach which we have implemented using a bottom-up strategy. This allows us to maximally inline functions, as the smallest functions are usually the ones called at the bottommost part of the call graph. These are the functions that have the most overhead too. This method is advantageous if the method has many small virtual method calls. This might result in some methods at the top of the call chain not being inlined as they would grow large due to inlining earlier. But since every control flow path must terminate in a

method at the end of some call chain, bottom up order is a good scheme.

Duplicate and Add Locals : We are creating a new local in the caller method for each of the locals declared in the callee method that is being inlined. This essentially involves cloning each local in the callee method, adding the cloned local (which has the same type as the local in the callee method) to the method body as the caller method, and storing the mapping from the locals in the callee method to the corresponding locals that have been created in the caller method. The mapping is used when statements are to be duplicated for inlining. Name conflicts must be avoided between locals present in the method originally and the cloned locals that are added to the method. We are resolving name conflicts using the context at the point where the method is being inlined.

Handling Method Returns : We are adopting a unique way to handle returns from methods. Note that a method might return from many points and these must be handled carefully. We are identifying the local used to receive the value in the caller, and using it to receive the returned variable at every return point in the inlined callee code. We are then adding a 'goto' statement from each of these return points to the next unit in the caller, essentially mimicking the way a return works in a method call. This requires us to keep track of the next unit of a method for each unit that we are iterating over.

Handling Parameters passed to the Callee : Parameters called to the callee are handled in a similar way, where we obtain a mapping of the parameters passed to the callee, and the arguments of the function call of the callee. We then replace the parameter initialising statements in the callee code identified

using an identity statement with an assignment statement that has the local that was mapped to this parameter on the right hand side instead of the parameter number.

Duplicate, Modify and Add Statements : Each statement in the method body of the callee method is cloned and the locals accessed in the statement are adjusted using the mapping between locals in the callee method and the new locals created for inlining in the caller method using the context. We are capturing the flow of callee statements within themselves by cloning the whole body of the callee method first. This preserves redirection relationships between the units of the cloned body. Now when we modify or replace any unit of the callee that we are going to copy in the caller body, we modify the redirection relationships using the redirectJumpsToThisTo() and swapWith() functions present in Soot. Each different type of statement in each callee unit has different characteristics, and must be modified in a separate way to maintain a correspondence with the original callee method. These statements include but are not limited to the AssignStmt, InvokeExpr, BinopExpr, UnopExpr, ArrayRef, InstanceOfExpr, NewArrayExpr and others. Each had to be handled separately to ensure sound and precise transfer from callee to caller.

Handling Concurrent Modification of the Method Body : One important step we had to remain careful about is modifying the unit chain of the caller concurrently while iterating through it which throws the ConcurrentModification exception. This seems extremely tricky to handle as there is no way to perform method inlining without modifying the caller method body units. The way to go about this is to use Snapshot Iterator provided by the Soot, which disables throwing a concurrent modification

error and hence allows you to modify the unit body while iterating through it.

2.4 Results

2.4.1 Static Performance Analysis

We analyse the static performance of our method inlining optimization on the number of call sites present in our program. This includes the virtual and static function calls

	Without Method Inlining	**With Method Inlining**
Total #virtual invoke sites	12	**1**
Total #static invoke sites	6	**2**
Total #invoke sites	18	**3**

Testcase 1 Static Results

	Without Method Inlining	**With Method Inlining**
Total #virtual invoke sites	18	**0**
Total #static invoke sites	**0**	**0**
Total #invoke sites	18	**0**

Testcase 2 Static Results

2.4.2 Dynamic Performance Analysis

We analyse the dynamic performance of our optimization on the basis of 2 metrics. First we compare the actual number of calls made for each type of invoke statement - static and virtual. Next, we compare the time taken by the program to run.

	Without Method Inlining	**With Method Inlining**
Total #actual virtual calls	3050842	**51227**
Total #actual static calls	3007897	**7946**
Total #actual invokes	6058739	**59173**

	Without Method Inlining	**With Method Inlining**
Time taken to run	0.256s	**0.165s**

Testcase 1 Dynamic Results

	Without Method Inlining	**With Method Inlining**
Total #actual virtual calls	32047802	**48120**
Total #actual static calls	**7613**	7650
Total #actual invokes	32055415	**55770**

	Without Method Inlining	**With Method Inlining**

Time taken to run	2.502s	**1.495s**

Testcase 2 Dynamic Results

Conclusion

In this assignment, we have focused on reducing the overhead associated with polymorphic virtual method calls in Java bytecode.
The first part of this assignment was the resolution of monomorphic virtual calls to static calls to reduce the dispatch and lookup overhead. This was done using a Points-To Type Analysis that is more precise than a CHA.
The second part of this assignment was the implementation of method inlining, and the study of its impact in improving the performance of programs compiled at Java bytecode. This was done using multiple inlining criteria and transformation of Bytecode using Soot.

Analysis of Virtual Method Resolution

Based on the **static results** obtained for Virtual Method Resolution, we clearly see the following -
i) The number of edges to the possible function calls is much lesser using the Points-To Type Propagation method as compared to the CHA method.

ii) The number of virtual edges reduces significantly after we apply our optimization to resolve monomorphic virtual method calls.
iii) The number of static edges increases significantly after we apply our optimization to convert monomorphic virtual method calls to static method calls.

Based on the **dynamic results** obtained for Virtual Method Resolution, we clearly see the following -
i) The total number of static invoke made during runtime is significantly higher in our optimised code as compared to the original code.
ii) The total number of invokes made during runtime is about the same before and after the optimization since we are only replacing the method calls and not adding or subtracting any.
iii) The time taken to run our optimised code with the JIT compiler disabled is much significantly lesser than the time taken to run the original code, hence we see an improvement.

Analysis of Automatic Method Inlining

Based on the **static results** obtained for Automatic Method Inlining, we clearly see the following -
i) The number of static and virtual edges both reduce significantly in both the CHA call graph and the Points-To Type Propagation Analysis due to method inlining
ii) The number of monomorphic call sites should ideally reduce to only the number which is not ideal to inline due to our criteria.

Based on the **dynamic results** obtained for Virtual Method Resolution, we clearly see the following -
i) The total number of static invoke made during runtime is almost negligible as we are inlining almost all static methods unless they are too big.
ii) The total number of virtual invokes made during runtime is similar to number of virtual invokes after the Poly-to-Mono optimization as those were the resolved virtual invokes.
iii) The time taken to run our optimised code with the JIT compiler disabled is much significantly lesser than the time taken to run the original code, hence we see an improvement. The improvement was performing even better than only performing the first optimization as our second optimization adds on top of the first optimization.