A Study of Type Analysis for Speculative Method Inlining in a JIT Environment

Feng Qian and Laurie Hendren
{fqian, hendren}@sable.mcgill.ca

School of Computer Science, McGill University

Abstract. Method inlining is one of most important optimizations to achieve a high performance JIT compiler in Java virtual machines. A type analysis allows the compiler directly inline monomorphic calls. At runtime, the compiler and type analysis have to handle dynamic class loading properly because the analysis result is only correct at compile time. Loading of new classes could invalidate previous analysis results and optimizations. Class hierarchy analysis (CHA) has been used successfully in JIT compilers for speculative inlining with various invalidation techniques as backup.

In this paper, we present the results of a limit study of method inlining using dynamic type analysis on a set of standard Java benchmarks. We developed a general type analysis framework for measure the effectiveness of several well-known type analysis, including CHA, RTA, XTA and VTA. Surprisingly, the simple dynamic CHA is nearly as good as an ideal type analysis for inlining virtual method calls. It leaves no room for other type analysis to improve. On the other hand, only reachability-based interprocedural type analysis (VTA) is able to capture the majority of monomorphic interface calls. We measured the runtime overhead of interprocedural type analysis in the JIT environment. To overcome the memory overhead of dynamic whole-program analysis, we outlined the design of a demand-driven inter-procedural type analysis for inlining hot interface calls.

1 Introduction

Object-oriented programming languages encourage programmers to write small methods and compact classes. The code is easy to read, modify, and maintain. Java programs exemplify such design idea: many tiny methods have only one line code to access a field, return a hash value, or invoke another method. Design patterns [?] use class inheritance and virtual calls extensively to obtain great engineering benefits. The downside of using small methods is that a program has to make frequent method calls. Object-oriented programs heavily rely on compilers to reduce calling overhead. Method inlining is one of most important optimizations to achieve a high performance Java virtual machine.

Polymorphic calls have been studied extensively in the context of C++ in Driesen [?]. The Java programming language only allows single inheritance on normal classes, but allows multiple inheritances on interfaces. Virtual calls in Java can be categorized into two kinds: *virtual calls* on normal class types and *interface calls* on interfaces. Virtual calls can be implemented very efficiently by modern JIT compilers. Various techniques reduce the overhead of interface calls as well.

Although the direct overhead of virtual calls is low, further performance improvement is often obtained from method inlining and optimizations on inlined code. Inlining creates larger code blocks for program analysis and removes the side-effect of original calls. Thus, optimization opportunities are increased after inlining.

A key step of method inlining is to decide which method(s) to be inlined at a call site. This can be achieved by using language constructors such as *final*, *private* declaratives, or language constraints such as type analysis results. An alternate is to profile targets of call sites. Inlining using language constraints is conservative and it supports direct inlining that maximizes optimization opportunities. In this paper, we study the method inlining using type analysis results.

Static type analysis for Java programs [?,?,?,?] may not be directly applicable in a JIT compiler because of dynamic features of Java virtual machines. The type set of a variable might have new members as new classes loaded. Optimizations based on old results could be invalidated. Various techniques have been devised to use dynamic class hierarchy analysis for directly inlining in the presence of dynamic class loading and JIT compilation.

In this paper we evaluate the effectiveness of several dynamic type analysis for method inlining in a Java virtual machine (Jikes RVM [?]). We built a common type analysis framework for supporting speculative inlining with invalidations. We use an efficient call graph profiling mechanism [?] to log call targets of each virtual call site. The logged information is used as an ideal type analysis for re-executing the benchmark. We compare the inlining results of other type analysis to the ideal one. In order to measure the maximum inlining potential of type analysis, we also relaxed the size limit on inlining targets. Surprisingly, the simple CHA is nearly as good as the ideal type analysis for inlining virtual calls. It leaves little room for improvement. On the other hand, CHA is less effective for inlining interface calls. We found majority of interface invocations are from a small number of hot call sites which are used in a very simple pattern. A whole-program type analysis, dynamic VTA, has been developed to capture monomorphic interface calls. We analyze the effectiveness and costs of the whole-program approach. The main difficulty of dynamic whole-program analysis is that it requires large heap space for maintaining analysis data which co-exist with application data in the heap. From our experience, we believe demand-driven is the right approach to make dynamic inter-procedural analysis be practical in Java virtual machines.

Our objective is to understand how well a type analysis can do at runtime, and what opportunities there are for improvement. In this study, we made following contributions:

– A limit study of method inlining using dynamic type analysis on a set of standard Java benchmarks;
– Experience of inter-procedural reachability-based type analysis in a JIT environment;
– Outline of a demand-driven type analysis for inlining interface calls.

We introduce necessary background and related work in Section 2. In Section 3, we describe the design of a common type analysis framework for speculative inlining. The limit study results are also presented in this section. The whole-program VTA type analysis is described in Section 4 with experimental results. Finally, we discuss our experience from this study and future work in Section 5.

2 Background and Related Work

Methods are important semantic abstraction boundaries of object-oriented programs. A method call requires setting up a stack frame, passing parameters and returning the result. A virtual call has extra costs of looking up call target at runtime. Compiler writers work hard to remove these boundaries to improve the performance. Method inlining is an important optimization of object-oriented programs. Inlining reduces the direct cost of calls and creates new opportunities for further optimizations on inlined code.

2.1 Background

The Java virtual machine specification [**?**] defines four different invoke instructions:

invokestatic is used to call a static method, whose target method can be binded statically (at compile time);
invokespecial is used to call instance initialization, private, and superclass methods;
invokevirtual is used to make virtual calls, where the callee method is declared in a class (vs. interface);
invokeinterface is used to make virtual calls, where the callee method must be declared in an interface.

Java methods can be categorized into two types: *non-virtual* and *virtual*. *Static* and *instance initialization* [1] methods are non-virtual, and the rest of methods are virtual. A Java-to-bytecode compiler (e.g., *javac* and *jikes*) chooses the right instruction for each call in the source code according to static type information and class hierarchy. A Java virtual machine checks constraints at runtime, for instance, the callee method of an *invokestatic* must be a static method. A JIT compiler can bind a non-virtual call and its target at compile

[1] Methods with a special name <*init*>.

time, if the callee method has been resolved. Static binding enables efficient implementation of non-virtual calls. For example, a resolved non-virtual call takes two instructions in Jikes RVM.

The beauty of object-oriented programming languages comes from supporting of virtual methods. In the Java programming language, a virtual call has the form of <`x, A.m()`> where `x` is a variable pointing to objects of type `A` or its subtypes, and `m()` is the method signature. The real target of each invocation depends on the type of object pointed to by the variable `x` at runtime. We call the object pointed to by `x` the *receiver* of the call, and `m()` is a *callee method signature*.

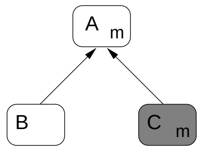
In the Java virtual machine specification, virtual methods are invoked by *invokevirtual* or *invokeinterface* instructions, depending on whether the declaring class of the callee method is a class or interface. A virtual call is slightly more expensive than a non-virtual one since it requires method lookup. By taking advantage of single inheritance, the *invokevirtual* bytecode is implemented by three instructions in Jikes RVM. The implementation of *invokeinterface* is much more expensive than *invokevirtual* due to multiple inheritance. Alpern et. al. [?] did excellent introduction and provided an efficient solution of the problem.

Our focus in this study is on the inlining of virtual methods rather than the implementation. Due to polymorphism, a virtual call may invoke several different methods in the course of program execution. Compilers can use static type analysis or profiling information to devirtualize the call to a set of possible target methods and inline one or several of them into the caller. A static type analysis requires the whole program available. The Java execution model allows dynamic loading and lazy compilation only when classes or methods are used. A new challenge of type analysis in a JIT environment is to handle these dynamic features. We discuss existing work on this topic in the rest of this section.

2.2 Guarded inlining using class and method tests

An easy solution of dynamic class loading is to guard inlining code with runtime checks. Detlefs and Agesen [?] pioneered this problem in Sun's JVM. Figure 1 is a similar example from the original paper. It shows a class hierarchy and a piece of Java code. A class `A` declares a virtual method `m()` and a class `B` extends `A` without overriding the method `m()`. Another class `C` extends `A` but overrides the method `m()`. If the compiler chooses to inline `A.m()` based on some profiling information or heuristics, it generates a class test to ensure that only objects of type `A` can reach inlined code of `A.m()`:

```
a_class_obj = a->class;
if (a_class_obj == class_A) {
  inlined A.m()
} else {
  normal virtual call
}
```




```
void foo(A a) {
  ......
  a.m();
  ......
}
```

Fig. 1. A simple example: a hierarchy and a method

The drawback of *class test* is that it only covers the case when an object type is `A`. If the object type is `B`, the control falls to the normal virtual call, although the target is still `A.m()`.

Detlefs and Agesen [?] provided a solution to the dilemma of class test. It uses *method tests* to guard inlining of virtual method invocations with the assumption that the target can be obtained from the class information cheaply. In most Java virtual machines, obtaining a method address from a virtual method table takes only one load instruction. Using *method tests*, the compiler generates inlined code:

```
a_class = a->class;
m_addr = a_class->m;
if (m_addr == class_A->m)
  inlined A.m()
else
  normal virtual call
```

A single method test can cover more classes than a class test with the cost of one load instruction in the fast path.

Two tests can be selectively used in practice. For example, if the inlined method is in a `final` class, the class test can be used since there is no other receiver types due to language constraints.

2.3 Direct inlining using preexistence analysis

The method and class tests are mechanisms guarding inlined code safely even in the presence of dynamic class loading. For monomorphic virtual calls, these tests seem redundant except as assurance. Detlefs and Agesen [?] pointed out that, in a Java virtual machine, a compiler can remove the tests for currently monomorphic calls induced by CHA, and register the compiled code with the assumption that the inlined method is not overridden (the call is monomorphic). Right after a class loader loads a new class, it checks all methods of superclasses overridden by methods in the newly loaded class. For compiled methods whose assumptions were invalidated, their entries in virtual method tables are conveniently reset to lazy compilation code stubs. The next invocation of an invalidated method triggers recompilation using the new, correct CHA results.

However, there is a drawback in this approach. Only the next invocation of an invalidated method can trigger recompilation and execute on the correct code. If a thread is executing a compiled method while dynamic class loading invalidates it, resetting the virtual method table entry cannot correct the execution of the thread. Detlefs and Agesen [?] pointed out that, if the receiver object of a call is *preexisting* before the method, resetting its entry in virtual method table is enough to ensure safety because these method activations running on threads would not create objects of newly loading classes. They proposed a simple *invariant argument analysis* for proving the preexistence of receivers. The analysis tracks simple data-flow between reference type variables. If a variable is only assigned values from parameters, then it is preexistent before the method call.

2.4 Code patching

Preexistence analysis may not always succeed on removing method and class tests for inlined monomorphic calls. Ishizaki et. al. presented a code patching technique [?] to remove the direct overhead of all method and class tests for monomorphic calls in the presence of dynamic class loading.

Code patching uses dynamic CHA to identify monomorphic calls based on the class hierarchy at compile time. Each inlined call site has a backup path which does normal virtual invocation. The compiler has to record the addresses at the beginning of inlined code and the start of backup path. It also registers a dependency of the inlined site on the assumption that the call is monomorphic. When dynamic class loading happens and invalidates the assumption, the compiler patches the code at the beginning of inlined code by a direct jump instruction to the address of the backup path. The virtual machine can optionally reset the invalidated method's entry in the virtual method table.

The advantage of code patching is that it removes memory loads and conditional branch instructions required by class and method tests. Also recompilation is not a necessity. The disadvantage is that it needs to track detailed dependency and synchronize caches explicitly on some architectures. Optimization opportunities on inlined code are still limited because the compiler has to use a conditional branch instruction in IRs to match the beginnings of inlined code and the backup path. The control flows are merged after inlined code and the backup path. Preexistence based inlining can still be used together with code patching.

2.5 Other related work

Ishizaki et. al. [?] also did extensive study of dynamic devirtualization techniques for Java programs. Our study answers the question of what is the limit of method inlining using type analysis. In their experiments, size limits were posed on inline targets, and techniques using dynamic CHA can inline about 46% of virtual calls (execution counts). Our limit study shows that CHA is nearly as perfect as an ideal type analysis. Our second focus is on dynamic IPAs. In the set of benchmarks, interface calls were used in a simple pattern and a context insensitive interprocedural type analysis could be as good as the ideal type analysis. Demand-driven is necessary to make IPA to be practical in JIT compilers. We also investigate requirement of speculative optimizations in JIT compilers.

For monomorphic calls whose receivers could not be proved to be preexisting, *thin guards* [?] can be used to combine several tests into one. Although guarded inlining removes the overhead of building call stacks and passing parameters, the control flow created by guards limits the effectiveness of further optimizations on inlined code. The merge of inlined code and backup path essentially removes the data-flow benefits from inlining. Thin guards are able to create large regions of inlined code for optimizations. Control-flow splitting [?] breaks the merge by duplicating the path and improves the effectiveness of optimizations.

3 A Type Analysis Framework Supporting Speculative Inlining

A static type analysis for Java programs answers a basic question: what is the set of possible runtime types of a variable *at a program point*? A *dynamic* type analysis is also *time-sensitive*: results may change over program's execution. In order to use the type analysis results for optimizations in a JIT environment, there are a few requirements we set for the analysis:

`dynamic:` it has to handle dynamic features of Java programs, such as dynamic class loading, reference resolution, and JIT compilation;
`conservative:` analysis results are correct at any point during the program execution;
`just-in-time:` the analysis should be able to notify clients when previous analysis results are about to change during execution.

A dynamic type analysis fits into a Java virtual machine without changing the lazy strategy of handling class loading and compilation. Eagerly loading a class can increase the resource usage and reduce the effect of other optimizations. The *conservativeness* ensures optimizations based on analysis results are correct at the analysis time (it might be invalidated in the future). If the analysis can update its results *just-in-time* and invalidate any optimized code based on old results, clients can use the analysis for speculative optimizations. Our objective is to design a type analysis framework supporting speculative inlining in a JIT compiler.

3.1 Framework structure

We designed a type analysis interface shown in Figure 2. In a Java method, a call site is uniquely identified by the method and a bytecode index. Given the method and bytecode index, the `getNodeId` method returns a node ID for further queries. The node ID allocation decides the granularity of different type analysis. For example, CHA and RTA uses a single ID for all call sites, XTA allocates a node ID for all call sites in the same method, and VTA assigns different IDs to different call sites. The `lookupTargets` method returns an array of targets resolved by using reaching types of the node with a given callee method signature. The detailed lookup procedure is same as virtual method lookup, defined by the

JVM specification [?]. An inline oracle makes inline decisions according to the lookup results.

If the type analysis finds a monomorphic call site (with only one target), then the oracle decides to perform speculative inlining (preexistence or code patching). It must register a dependency via the `checkAndRegisterDependency` method. A dependency says that, given a node and a callee method signature, a compiled method (*cm*) is valid only when the lookup results have one target that is same as the parameter *target*.

After registering the dependency successfully, any change in the type set of the node causes verification of dependencies on this node. The `verifyDependency` method is called by the type analysis when the node has a new reaching type. For each dependency of the node, the verification procedure performs method lookup using the new reaching type and the callee method signature. If the lookup result is different from the target method of the dependency, the compiled method must be invalidated immediately.

```
public interface TypeAnalysis {
  public int getNodeId(VM_Method caller, int bcindex);
  public VM_Method[] lookupTargets(int nodeid, VM_Method callee);
  public boolean checkAndRegisterDependency(int nodeid,
                                            VM_Method callee,
                                            VM_CompiledMethod cm,
                                            VM_Method target);

  protected void verifyDependency(int nodeid, VM_Class newKls);
}
```

Fig. 2. TypeAnalaysis interface

A *TypeAnalysis* implementation has to monitor system events such as class loading, method compilation, etc. We have implemented several type analysis depicted in Figure 3. CHA and RTA only differentiate classes that participate in the reaching type sets. We made a new variation of CHA, called ITA, to only allow classes with instances participate in reaching types. XTA and VTA share many components. A special class, `IdealTypeAnalysis`, uses profiled results for the purpose of limit study. All implementations satisfy the requirements defined at the beginning of this section. A client, `StaticInlineOracle`, uses analysis results for speculative inlining.

3.2 A limit study of method inlining using dynamic type analysis

An ideal type analysis To measure how precise a type analysis could be, we need an ideal type analysis for comparison. If a benchmark runs deterministically, we could profile targets in the first run, and use the profiled targets as faked analysis results for the second run. We use an inexpensive call graph profiling

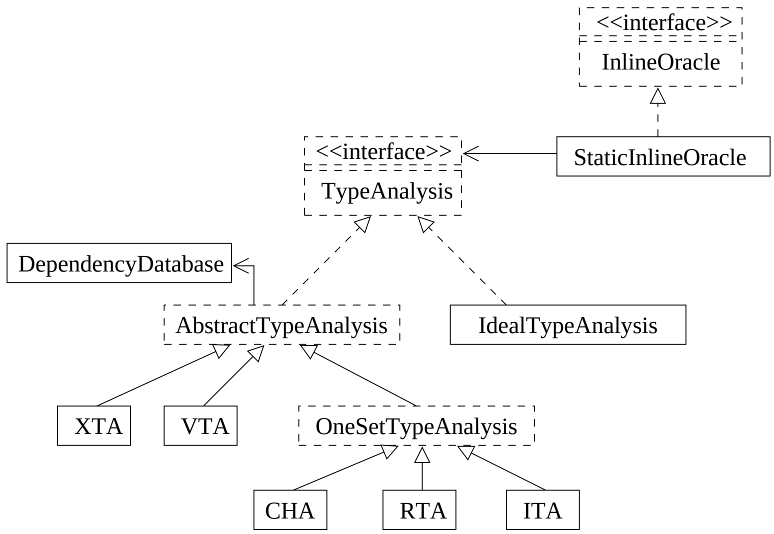


Fig. 3. Type analysis framework diagram

mechanism [?] to gather call targets. An *IdealTypeAnalysis* parses the profiled targets for call sites, and the *lookupTargets* method returns target(s) for a call site.

Experimental approach An inline oracle has to balance the benefits and costs of inlining. Excessive inlining may blow up code sizes and slow down the execution. Therefore, a JIT compiler usually sets a size limit on inlined targets by some heuristics. Paper [?] details a study of different heuristics in Jikes RVM. For our purpose of study, we would like to measure the maximum potential of a type analysis for method inlining without a size limit. However, inlining all call sites is not feasible. Instead, we only inline the hottest call sites without a size limit.

Benchmarks Our benchmark set includes SpecJVM98 suite [?], SpecJBB2000 [?], a CFS subset evaluator from a data mining package Weka [?], a simulator of certificate revocation schemes [?], and a variation of simulator interwoven with AspectJ code for detecting calls that return `null` on error conditions.

Table 1 shows dynamic characteristics of benchmark executions. We ignored call sites in the RVM code and Java libraries compiled into the boot image. Virtual and interface calls are measured separately. Column 2 and 5 (*total*) report the total counts of invocations in each category. Column 3 and 6, labeled as *#hottest*, are numbers of hottest call sites, ranked in top 100, whose invocations are more than 1% of *total* in Column 2 and 5. Column 4 and 7 are percentages of invocations contributed by these hottest call sites.

benchmark	invokevirtual			invokeinterface		
	total	#hottest	coverage	total	#hottest	coverage
_201_compress	2,191M	7	89%	0	N/A	N/A
_202_jess	964M	25	71%	0	N/A	N/A
_205_raytrace	2,837M	16	29%	0	N/A	N/A
_209_db	762M	8	99%	149M	5	99%
_213_javac	688M	10	20%	34M	5	92%
_222_mpegaudio	846M	25	80%	2M	11	98%
_228_jack	264M	22	74%	46M	11	93%
SpecJBB2000	8,162M	9	34%	146M	7	99%
CFS	639M	15	92%	0	N/A	N/A
simulator(orig)	44M	5	71%	0	N/A	N/A
simulator(aspects)	162M	13	72%	0	N/A	N/A

Table 1. Coverage of the hottest call sites

It is interesting to point out that, for most of benchmarks, majority of invocations are from a small number of hot call sites. Less than 25 call sites contribute more than 1% invocations. Only about half of benchmarks have more than 1M interface invocations. These benchmarks have less than 10 hot interface call sites that contribute to more than 92% of invocations.

The *_213_javac* benchmark has a large code base generated by scanner and parser. Invocation counts spread in many call sites. *SpecJBB2000* has a large code base as well, and it runs much longer than other benchmarks. Hot call sites selected by 1% threshold contribute only about 34% of total invocations.

A list of hottest call sites are provided to the inline oracle. The size limit is removed when inlining call sites in the list. Thus, the inline oracle can exploit the potential of a type analysis as much as possible.

As we discussed in Section 2, a virtual call site can be inlined using different techniques:

- *direct* direct inlining if the callee method is *private* or *final*;
- *preex* direct inlining with invalidation checks if the receiver can be proved to be preexistent prior method calls;
- *cp* guarded inlining with code patching;
- *mt* or *ct* guarded inlining with method or class tests.

If a call site is currently monomorphic according to the type analysis results, guards are chosen as a command line option. It can be code patching, method or class tests. We use code patching since it has less runtime overhead (column *cp*);

Monomophic interface calls can be directly inlined if the receiver is preexistent, or inlined with guards. We found that, in our benchmark set, nearly all receivers of hot interface calls cannot be proved to be preexistent by invariant-parameter analysis. In our results, we omit the *preex* category of interface calls. In our another experiment, the inline oracle inlines polymorphic call sites (guarded by method or class tests) that have 1 or 2 targets resolved using type analysis results. However, this does not make big increases of inlined calls (only _213_javac has 2% increases). We choose not to inline polymorphic calls in our experiment.

Limit study results Table 2 compares the results of dynamic *CHA* and *IdealTypeAnalysis*. Each benchmark has two rows: `ideal` and `cha`, showing dynamic counts of inlined calls using different type analysis. As usual, virtual and interface calls are presented separately. Dynamic CHA does a nearly perfect job for resolving virtual calls for most benchmarks except *_213_javac* and *simulator(aspects)*. On these benchmarks, majority of dynamic invocations are contributed by monomorphic call sites. The sum of *direct*, *preex* and *cp* is close to the coverage in Table 1. `_213_javac` leaves a small gap between *cha* and *ideal*. Column 8 shows that a large portion of interface invocations are from monomorphic call sites too. Dynamic CHA is ineffective on inlining interface calls. Other two simple type analysis, RTA and ITA, did not improve the results of inlining interface calls because common interfaces are implemented by different classes that are likely to be instantiated.

		invokevirtual				invokeinterface		
		total	direct	preex	cp	total	cp	mt
_201_compress	ideal	2,191M	99%	0	0	0	0	0
	cha	2,191M	99%	0	0	0	0	0
_202_jess	ideal	994M	58%	6%	21%	7M	0	0
	cha	994M	58%	6%	21%	7M	0	0
_205_raytrace	ideal	2,837M	0	50%	41%	0	0	0
	cha	2,837M	0	50%	41%	0	0	0
_209_db	ideal	762M	31%	0	67%	150M	99%	0
	cha	762M	31%	0	67%	150M	0	0
_213_javac	ideal	701M	28%	7%	15%	35M	95%	0
	cha	701M	27%	7%	10%	35M	0	0
_222_mpegaudio	ideal	846M	73%	3%	0	2M	57%	0
	cha	846M	73%	3%	0	2M	57%	0
_228_jack	ideal	258M	13%	16%	39%	46M	86%	0
	cha	258M	12%	15%	39%	46M	25%	8%
SpecJBB2000	ideal	8,250M	32%	29%	11%	148M	99%	0
	cha	8,119M	32%	29%	12%	146M	0	0
CFS	ideal	639M	38%	6%	52%	0	0	0
	cha	639M	38%	6%	52%	0	0	0
simulator	ideal	44M	99%	0	0	0	0	0
(original)	cha	44M	99%	0	0	0	0	0
simulator	ideal	162M	11%	19%	53%	0	0	0
(aspects)	cha	162M	11%	0	53%	0	0	0

Table 2. Limit study of method inlining using type analysis

Discussion Simulator(aspects) is an interesting benchmark. Injecting code for checking aspects increases the number of invocations and changes inlining behaviors dramatically. Nearly all virtual calls are monomorphic and can be directly inlined in the original benchmark. With aspects, dynamic CHA misses monomorphic calls in the *preex* category. After looking at the benchmark closely, we found this is due to the generic implementation of pointcuts.

The pointcut implementation wraps primitive values in objects and passes them to AspectJ libraries. The value is then unwrapped after the library call. The code for unwrapping in the original AspectJ library looks like

```
int intValue(Object v) {
  if (v instanceof Number)
    return ((Number)v).intValue();
  ......
}
```

The single call site of ((Number)v).intValue() in the Conversions.intValue method contributes 19% *preex* invocations. Dynamic CHA failed to inline this call site because the Number class has several subclasses, Integer, Double, and Long, and the call site is identified as polymorphic.

This particular problem can be solved in two ways: 1) use a context-sensitive reachability-based type analysis, or 2) change the implementation of Conversions.intValue method to facilitate the type analysis. We changed the method to use a tighter type, *Integer*, in the type cast expression, then the call site becomes directly inlineable.

Since the number of hot interface call sites is small, we investigated them one by one. It turns out these hot interface calls are used in a similar pattern:

```
// <TYPE> is java.util.Vector, java.util.Hashtable, etc.
Enumeration e = <TYPE>.elements();
......
while (e.hasMoreElements())
  index[i++] = (Entry)e.nextElement();
```

Enumeration is an interface in java.util package. The *while* loop makes two or more interface calls for enumerating elements of underlying data structures. Dynamic CHA assumes all implementations of the interface are in the runtime type set of e, although each <TYPE> class returns a specific implementation. Without interprocedural information or inlining the <TYPE>.elements() method, a type analysis cannot produce precise type information of e. Therefore, these interface call sites cannot be inlined by using dynamic CHA.

From this limit study, we had a few conclusions:

- most virtual calls in standard Java benchmarks are monomorphic;
- dynamic CHA is nearly perfect for inlining virtual calls;
- dynamic CHA is ineffective on inlining interface calls;
- to assist compiler optimizations, a programmer should use more precise types when it does not sacrifice other engineering benefits;
- a large percentage of interface calls are monomorphic and used in a simple pattern, it requires interprocedural analysis to discover the precise type of the receiver.

4 Dynamic Interprocedural Type Analysis

In Section 3, we built a type analysis framework for supporting speculative inlining. We also did a limit study of method inlining on a set of standard Java benchmarks. Dynamic CHA is unable to get useful type information for variables whose objects come from other methods. In this section, we study interprocedural, reachability-based, type analysis for inlining interface calls.

There are two different approaches to perform interprocedural analysis in a Java virtual machine. A whole-program analysis analyzes all classes and methods that can participate the program execution. A demand-driven analysis only analyzes part of code related to a request. We start with the while-program approach.

We designed and developed a dynamic version XTA [**?**] type analysis in our previous work [**?**]. XTA uses separate sets for methods and fields. A type reaching a caller can reach a callee if it is a subclass of the callee's parameter types. Types can be passed between methods by field accesses as well. We first implemented a dynamic XTA type analysis as an example of how to deal with dynamic class loading and reference resolution. However, due to lack of intraprocedural dataflow information, XTA results are very coarse. Although the computed type sets are smaller than ones from CHA, it still could not recognize important monomorphic interface call sites. For method inlining study, XTA results were no better than dynamic CHA.

4.1 Dynamic variable type analysis (VTA)

Design Variable type analysis (VTA) is a fast reachability-based interprocedural type analysis proposed by Sundaresan et. al. [**?**]. Given a Java program (all application and library classes), static VTA constructs a directed type flow graph $G = (V, E, \tau)$ where:

- V is a set of nodes, representing local variables, method formals and returns, static and instance fields, and array elements;
- E is a set of directed edges between nodes, an edge $a \rightarrow b$ represents an assignment of a's value to b;
- $\tau : V \rightarrow T$ is a map from a node to a set of types (classes).

The analysis constructs a VTA graph by a one-pass scan of the bytecode of classes. Local assignments, field reads and writes are modeled by edges straightforwardly. To deal with method calls, it relies on a coarse type analysis, such as CHA or RTA, to construct a conservative call graph first. Each call edge is modeled by edges from parameter variables to callee's formals, and returns are modeled properly. An allocation site adds the allocated type to the type set of the node representing left-hand side variable.

We use the same approach outlined in [**?**] to adopt the static VTA in a JIT compiler. In the whole-program approach, the constraint collector monitors method compilation events at runtime. Before a method is compiled, the constraint collector parses the bytecode of the compiled method and collects constraints. The collector uses the front-end of the optimizing compiler in Jikes RVM, which converts bytecode to a three address intermediate representation, HIR. Several optimizations are performed during translation, and each variable in HIR has a declaring type recovered from the bytecode.

Dynamic VTA analysis is driven by events from JIT compilers and class loaders. Figure 4 shows the flow of events. In the dotted box are the three modules of dynamic VTA analysis: VTA graphs, the analysis (include constraint collector), and dependency databases.

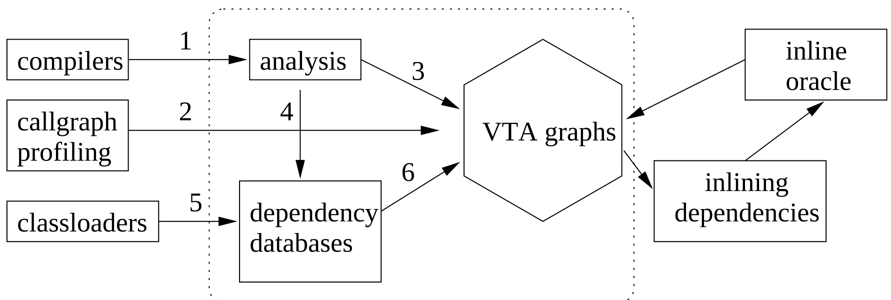


Fig. 4. Model of VTA events

Many system events can change the VTA graph. Whenever the graph is changed (either the graph has an new edge, or a node has a new reaching type), a propagator propagates type sets of nodes (related to changes) until no further change occurs. Whenever the reaching type set of a node gets changed, it verifies dependencies on this node registered by clients (see Section 3), such as inlining oracles. The client has a chance to perform invalidation if the assumption of optimizations is violated.

Propagations To support speculative optimizations, the analysis must keep the results up-to-date whenever the client makes queries. One approach is to trigger type propagations *eagerly* whenever the VTA graph is changed, e.g, when constraint collector adds a new edge or a new reaching type. The second approach is to cache graph changes when collecting constraints of a method, and *batch* propagations at the end of constraint collection. This is a safe solution since the analysis is always performed before the compilation which is certainly before the method execution. During propagation, the propagator uses a work-list for related nodes and caches recent changes to reduce the amount of propagation work. Other events, such as class loading, reference resolution, and call edges, however, trigger propagation immediately. The third approach, as suggested in [**?**], propagates types only when a client makes queries. Propagating types of nodes subject to queries reduces propagation steps. However, a major drawback of on-demand propagation is that the analysis results might be outdated and it cannot invalidate wrongly speculated optimizations in time.

We implemented eager, cached, and on-demand propagation approaches. On-demand propagation is only used with guarded inlining using method/class tests. Table 3 compares the characteristics of VTA graphs when using *eager* and *batch* propagation strategies. The 2nd column shows the number of propagation requests. The 3rd and 6th columns are the constrain collection time in seconds. The 4th and 7th columns are the number of propagation steps. Since *batch* propagator caches changes, it has less propagation requests than *eager* propagation. Columns 5 and 8 are time spent on propagation. Eager propagation carries one type only and batch propagation needs to perform set operations. Eager propagation spends less time than batch approach.

		eager			batch		
benchmark	requests	c time(s)	steps	time(s)	c time(s)	steps	time(s)
_209_db (124s)	1,188	0.32	4,889	0.02	0.32	3,258	0.03
_213_javac (87s)	2,261	3.71	405,170	1.37	2.57	132,595	1.72
_228_jack (37s)	1,515	0.73	58,761	0.11	0.72	23,909	0.13

Table 3. Dynamic characteristics of VTA graphs using different propagation approaches

Effectiveness of dynamic VTA Not surprisingly, VTA is able to handle the simple pattern of interface calls in our benchmarks set. Table 4 compares dynamic counts of inlined interface calls. Dynamic VTA is able to catch all monomorphic interface calls and allows them to be inlined by using code patching.

benchmark	Ideal(cp)	VTA(cp)
_209_db	99%	99%
_213_javac	95%	95%
_228_jack	86%	86%
SPECjbb2000	99%	99%

Table 4. Comparison of VTA and Ideal type analysis for inlining interface calls

Memory overhead of whole-program VTA Although dynamic VTA allows the JIT compiler to utilize maximum inlining opportunities, the cost of whole-program VTA is also high in terms of memory overhead. Figure 5(a) compares the size of live data of *IdeaTypeAnalysis*, *CHA*, and *VTA*, using *_213_javac* as the example. Heap occupation has been increased by one-third for VTA graphs. Figure 5(b) depicts time spent on each GC using a copying collector. It is clear that the whole-program interprocedural analysis has a very high memory overhead.

4.2 Demand-driven analysis

The whole-program interprocedural analysis analyzes all compiled methods. Although the analysis itself is reasonably fast, the VTA graph representation takes a lot of memory space. Because Jikes RVM shares the same heap with applications, more live data triggers more frequent garbage collections and increases GC time in each collection. Section 3 shows that only a few call sites are hot in each benchmark. If the analysis only analyzes code related to these hot call sites, the time and space overhead could be reduced.

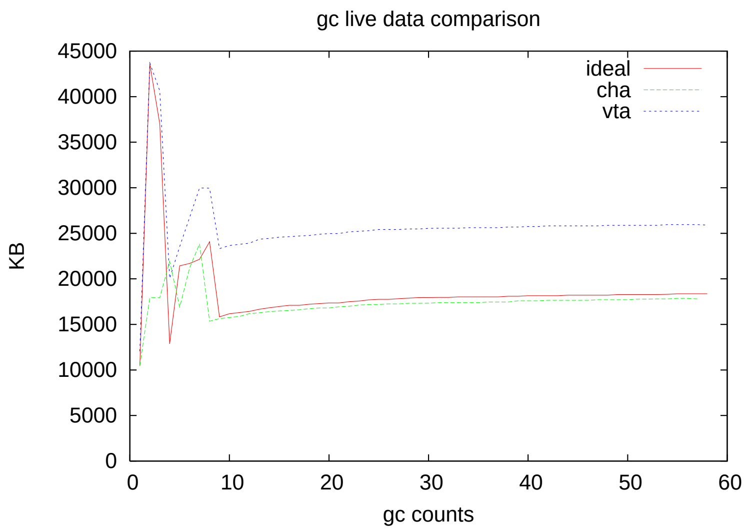
Demain-driven analysis [**?**,**?**] itself is an interesting research topic. It has been explored in the context of static points-to analysis. It is attractive to perform incremental, demand-driven IPA at runtime in a JIT environment.

We are developing a demand-driven VTA type analysis. For speculative inlining of interface calls, we outline the requirement and design of a demand-driven analysis. Given a receiver variable `e`, the compiler would decide if it is interesting to perform type analysis on `e`. A simple heuristic is that `e`'s declaring type is a common interface, such as *Enumeration* and *Iteration*, often used in loops, and there are interface invocations in loops made on the variable. Once the compiler decides to analyze `e`, the demand-driven analysis would look at the source of `e`. If there is a single definition of `e` in the method, it continues on look at where the value of *e* comes from. There are several possible cases, we describe each case and solution:

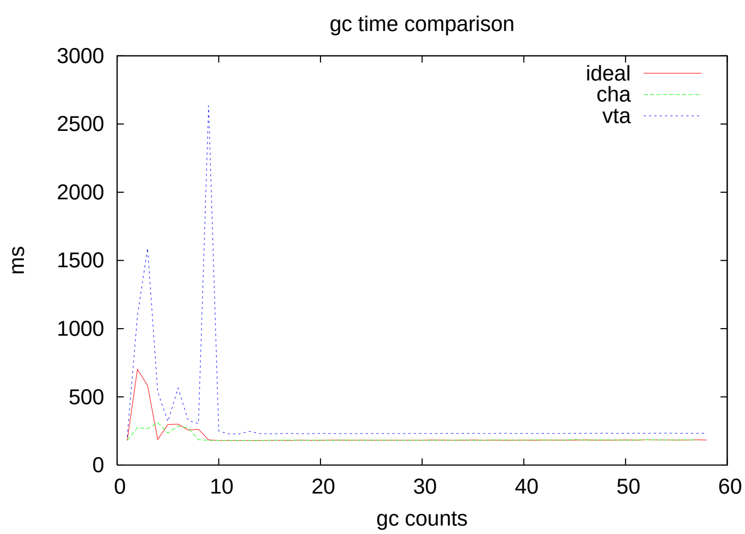
- `e` is passed in as a parameter. In this case, the analysis gives up since it is too expensive to analyze all callers of the method;
- `e` gets its value from a field or an array element. It requires a whole-program analysis in order to know all writes to the field or array element. Field analysis [**?**]could be used if it is available, otherwise, the analysis gives up in this case;
- `e` gets its value from a method call. This is the case the analysis tries to analyze further, and it performs following analysis:
  - is there only one target of the call site? if it is, then
  - can the type analysis get precise types of return objects in the target method?

  If the analysis successfully find the precise type of `e`, it then registers a dependency that assumes the analyzed method is the only target of the call site.

Above approach could successfully inline the common interface call pattens we found in our benchmarks. However, it is fairly limited to these patterns. We believe the right approach should look at patterns used in the real applications and analyze import ones case by case.



(a) live data after each GC



(b) time spent on each GC

Fig. 5. (_213_javac) GC behaviors affected by VTA

5 Discussion and Future Work

In this study, we made a few interesting observations about speculative method inlining.

Observation 1 A common wisdom is that inlining increases optimization opportunities. However, in the presence of speculative optimizations, inlining may reduce optimization opportunities as well. Figure 6 shows such an example. In Figure 6(a), the method `Foo.m()` is declared as *virtual*, but not overridden. Thus the call site in the `child` method is a candidate of direct inlining based on the receiver's preexistence prior method call (Figure 6(b)). However, if a compiler inlines `child()` into `parent()`, and the receiver of `foo.m()` is not preexistent prior the `parent()`. The call site can only be inlined with a guard as in Figure 6(c). Since the frequency of calling `foo.m()` is much more than calling `child()`, the performance of `parent()` might not be maximized. This pattern did happen in the *_213_javac* benchmark.

```
parent() {
  Foo f = getfield
  this.son(f);
}

child(Foo foo) {
  while(cond)
    foo.m()
}
```

(a) source

```
parent() {
  Foo f = getfield
  this.son(f);
}

child(Foo foo) {
  while (cond)
   inlined Foo.m(foo)
}
```

(b) not inline 'child'

```
parent(){
  Foo f = getfield
  while(cond)
    if (Foo.m is currently final)
     inlined Foo.m(f)
    else
     Foo.m(f)
}
```

(c) inline 'child'

Fig. 6. An example where inlining can reduce optimization opportunities

Observation 2 Our second observation is that inlining decisions may be affected by library implementations. A Java virtual machine is bundled with a specific implementation of Java class library. For example, the GNU classpath [?] is an open-source implementation of Java libraries and used by many open source Java virtual machines, including Jikes RVM. The implementation of `Hashtable.elements()` in the GNU classpath (version 0.07) returns objects of a single type `Hashtable$Enumerator`. The implementation in Sun's JDK 1.4.2_04, however, may return objects of `Hashtable$EmptyEnumerator` and `Hashtable$Enumerator`. Several hot interface call sites in our benchmark set would not be directly inlined if using Sun's JDK.

Future work We are currently working on the demand-driven type analysis outlined in Section 4 for interface calls. we are also looking at more applications of dynamic interprocedural analysis for optimizations in JIT compilers. A new research topic is to investigate the effectiveness of compiler optimizations on different design patterns.

References

1. B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000.
2. B. Alpern, A. Cocchi, S. J. Fink, D. Grove, and D. Lieber. Efficient implementation of java interfaces: Invokeinterface considered harmless. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*, pages 108–124, 2001.
3. M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney. A Comparative Study of Static and Profile-Based Heuristics for Inlining. In *Unknown yet*, pages 1 – 100, 1999.
4. M. Arnold and B. G. Ryder. Thin Guards: A Simple and Effective Technique for Reducing the Penalty of Dynamic Class Loading. In *ECOOP'02—16th European Conference for Object-Oriented Programming*, pages 498 – 524, 2002.
5. D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '96)*, pages 324 – 341, Oct. 1996.
6. Certrevsim. http://www.pvv.ntnu.no/ andrearn/certrev/sim.html.
7. C. Chambers and D. Ungar. Making Pure Object-Oriented Languages Practical. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 1 – 15, 1991.
8. Gnu classpath. `http://www.gnu.org/classpath"`.
9. J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In W. G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Åarhus, Denmark, Aug. 7–11 1995. Springer.
10. D. Detlefs and O. Agesen. Inlining of Virtual Methods. In *13th European Conference on Object-Oriented Programming (ECOOP'96)*, pages 258 – 278, June 1999.
11. K. Driesen. *Efficient Polymorphic Calls*. The Kluwer International Series in Engineering and Computer Science, 2001.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
13. S. Ghemawat, K. Randall, and D. Scales. Field Analysis: Getting Useful and Low-Cost Interprocedural Information. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 334–344, Vancouver, BC, Canada, June 2000.
14. N. Heintze and O. Tardieu. Demand-Driven Pointer Analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 24 – 34, 2001.
15. M. Hirzel, A. Diwan, and M. Hind. Pointer Analysis in the Presence of Dynamic Class Loading. In *ECOOP'04—18th European Conference for Object-Oriented Programming*, June 2004.

16. K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 294–310, 2000.
17. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
18. F. Qian and L. Hendren. Towards Dynamic Interprocedural Analysis in JVMs. In *3rd Virtual Machine Research and Technology Symposium (VM'04)*, pages 139 – 150, May 2004.
19. Spec JBB2000 benchmark. `http://www.spec.org/jbb2000/`.
20. Spec JVM98 benchmarks. `http://www.spec.org/osg/jvm98/index.html`.
21. V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 264–280, 2000.
22. F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings OOPSLA 2000 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 281–293, Oct. 2000.
23. F. Vivien and M. C. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 35 – 46, 2001.
24. Weka 3: Data Mining Software in Java. `http://www.cs.waikato.ac.nz/ml/weka/`.