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 the most important optimizations for JIT compilers in Java virtual machines. In order to increase the number of inlining opportunities, a type analysis can be used to identify monomorphic virtual calls. In a JIT environment, the compiler and type analysis must also handle dynamic class loading properly because class loading can invalidate previous analysis results and invalidate some speculative inlining decisions. To date, a very simple type analysis, class hierarchy analysis (CHA), has been used successfully in JIT compilers for speculative inlining with invalidation techniques as backup.
This paper seeks to determine if more powerful dynamic type analyses could further improve inlining opportunities in a JIT compiler. To achieve this goal we developed a general *dynamic* type analysis framework which we have used for designing and implementing dynamic versions of several well-known static type analyses, 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. There is little room for further improvement. On the other hand, only a reachability-based interprocedural type analysis (VTA) is able to capture the majority of monomorphic **interface** calls.

1 Introduction

The Java programming language [3] encourages programmers to write compact classes and small methods to obtain great engineering benefits. However, using small methods requires frequent method calls. A high performance Java virtual machine heavily relies on JIT compilers to reduce calling overhead.

Even though 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 analyses and improves the accuracy of intraprocedural analyses which must often handle method calls conservatively. Thus, method inlining is a very important part of a Java optimizer because it further reduces method call overhead and also increases other opportunities for optimizations.

* The author is currently affiliated with Goolge Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043, USA. This research was done while the author was at McGill University.

R. Bodik (Ed.): CC 2005, LNCS 3443, pp. 255–270, 2005.

A key step of method inlining is to decide which method(s) can be inlined at a call site. This can be achieved by using information conveyed via language constructs such as *final* and *private* declarations (which provide restrictions on which methods could be called), or the information can be gathered using a type analysis which determines which runtime types may be associated with a receiver, and hence which methods may be called. Another alternative is to profile targets of call sites. Inlining based on language constructs and type analyses results is conservative at analysis time and it supports direct inlining that maximizes optimization opportunities. In this paper, we study method inlining using type analysis results.

Static type analyses for Java programs [8,5,19,20] are not directly applicable to JIT compilers because of dynamic features of Java virtual machines. The type set of a variable might have new members as new classes are loaded and thus 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 analyses for method inlining in a Java virtual machine (Jikes RVM [1]). We built a common type analysis framework for expressing dynamic type analyses and used the results of these analyses for speculative inlining with invalidations. We then used this framework to perform a study of how many method calls can be inlined for the different varieties of type analyses.

We were also interested in finding the upper bound on how many calls that can be inlined, to determine if more accurate type analyses are required. To gather this information we used an efficient call graph profiling mechanism [16] 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 analyses to the ideal one. In order to measure the maximum inlining potential of a type analysis, we also relaxed the size limit on inlining targets.

Our results were quite surprising. The simple CHA is nearly as good as the ideal type analysis for inlining virtual method calls and leaves little room for improvement. On the other hand, CHA is less effective for inlining interface calls. Further, we found that the majority of interface invocations are from a small number of hot call sites which are used in a very simple pattern.

In order to capture the monomorphic interface calls we developed *dynamic VTA*, which is a whole-program analysis. We analyzed the effectiveness and costs of this whole-program approach. We found that the main difficulty of such a dynamic whole-program analysis is that it requires large data structures which must co-exist with application data in the heap.

Our objective is to understand how well a dynamic type analysis can perform with respect to method inlining in a JIT compiler, and what opportunities there are for improvement. In this study, we made following contributions:

- a limit study of method inlining using dynamic type analyses on a set of standard Java benchmarks;

– development and experience of an interprocedural reachability-based type analysis in a JIT environment; and
– interesting observations in speculative inlining.

We introduce the necessary background 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. Related work is discussed in Section 5. Finally, in Section 6, we conclude with some observations and plans for future work.

2 Background

Java methods can be categorized into two types: *non-virtual* and *virtual*. *Static* and *instance initialization* methods are non-virtual, and the rest of the methods are virtual. A JIT compiler can bind a non-virtual call and its target at compile 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 [13].

The beauty of object-oriented programming languages comes from supporting 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. The object pointed to by `x` is called 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. [2] give an excellent introduction and provide an efficient solution to the problem.

Due to polymorphism, a virtual call may invoke several different methods in the course of program execution. Compilers can use a 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. The Java execution model allows dynamic class loading which loads classes only on first use, and lazy compilation which compiles methods only on first execution. A new challenge of type analyses in a JIT environment is to handle these dynamic features properly.

An easy solution to dynamic class loading is to guard inlined code with runtime checks. Detlefs and Agesen [9] pioneered this technique in Sun's JVM. Given a virtual call site <`x, A.m()`>, a type analysis (e.g. CHA) might be able to prove the call site is monomorphic at compile time and the target is `A.m()`. If a JIT compiler chooses to inline the call site, it generates a *class test* instruction

comparing `x`'s type to `A`. The inlined `A.m()` is executed only if the runtime check succeeds, otherwise a normal virtual call is made.

The drawback of the *class test* is that it only covers the case when an object type is `A`. If the object type is a subclass `B` which does not override `A.m()`, the control falls to the normal virtual call, even though the target is still `A.m()`. *Method tests* fix the problem by testing the target method address instead of the receiver's type. After instructions loading `x`'s type information (`A` or `B`), the compiler generates one more instruction to obtain the target address of `m()` from the type information (`A.m()` even if the type is `B`). The inlined code is protected by a test of the target address to the address of `A.m()`. A single method test can cover more classes than a class test with the cost of one load instruction in the fast path.

Method and class tests have direct runtime overhead and optimizations on inlined code are limited by conditional test instructions. Detlefs and Agesen [9] pointed out that, in a Java virtual machine, a compiler can directly inline currently *monomorphic* call sites induced by CHA. When dynamic class loading makes an inlined monomorphic call site be polymorphic, the class loader must invalidate the compiled method which directly inlined the call site. The next invocation of an invalidated method triggers recompilation using the new, correct CHA results. To ensure the approach is safe for compiled methods running on threads, it requires a static analysis (*invariant argument analysis* [9]) to prove the preexistence of receiver variables of inlined calls.

The invariant argument analysis may not always succeed in removing method and class tests for inlined monomorphic calls. Ishizaki et al. [12] presented a code patching technique to remove the direct overhead of all method and class tests for *monomorphic* call sites in the presence of dynamic class loading. Code patching uses CHA to identify monomorphic calls at compile time. Each inlined call site has a backup path which does normal virtual invocation. The compiler records 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.

A static type analysis calculates conservative type sets for variables in object-oriented programs. Class hierarchy analysis (CHA) [8] assumes all subtypes of a variable's declaring type are in the runtime type set of the variable. Rapid type analysis (RTA) [5] performs a one pass scan of the program and prunes the CHA results by removing types that do not have an allocation site in the program. XTA [20] is a simple interprocedural type analysis. It uses one set to represent all variables in a method, and propagates type sets along call edges. Variable type analysis (VTA) [19] is a fast reachability-based interprocedural type analysis. Each variable has a type set and the analysis uses intraprocedural data-flow. More advanced analyses have higher costs. One focus in our study is

to show how to adopt these static type analyses to a JIT environment and to study their effectiveness on speculative method inlining.

3 A Type Analysis Framework Supporting Speculative Inlining

A *static* analysis is performed at compile-time and must make conservative assumptions that include all possible runtime executions. A static type analysis answers a basic question: what is the set of all possible runtime types of variable v at program point P. A *dynamic* type analysis is performed in a JIT environment, and therefore it is *time-sensitive*. It answers a query similar to a static one, except the answer is not for *all executions*, but for execution prior the time of answering the query. The results may change over program's execution. In order to use type analysis results for optimizations in a JIT environment, there are a few requirements we set for the analysis:

`dynamic:` it has to handle Java's dynamic features seamlessly, such as dynamic class loading, reference resolution, and JIT compilation;
`conservative:` analysis results must be correct at analysis time with respect to the executed part of the program;
`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. 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*, it can be used for speculative optimizations with some invalidation mechanisms. 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 1. 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 analyses. For example, CHA and RTA use 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 the same as virtual method lookup, defined by the JVM specification [14]. 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 (using 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 the 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. 1. TypeAnalaysis interface

A *TypeAnalysis* implementation has to monitor system events such as class loading, method compilation, etc. We have implemented several type analyses as depicted in Figure 2. We used JikesRVM v2.3.0 with the *FastAdaptiveCopyMS* configuration. JikesRVM implemented dynamic CHA, and we re-implemented it in our framework. 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 to participate in reaching types. XTA and VTA share many components. A special class, `IdealTypeAnalysis`, uses profiled results for the purpose of our limit study. All implementations satisfy the requirements defined at the beginning of this section. A client, `StaticInlineOracle`, uses the analysis results for speculative inlining.

3.2 A Limit Study of Method Inlining Using Dynamic Type Analyses

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 can profile targets in the first run, and then use the profiled targets as faked analysis results for the second run. We use an inexpensive call graph profiling mechanism [16] to gather call targets. An *IdealTypeAnalysis* parses the profiled targets for call sites, and the *lookupTargets* method returns profiled 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 size and slow down the execution. Therefore, a JIT compiler usually sets a size limit on inlined targets

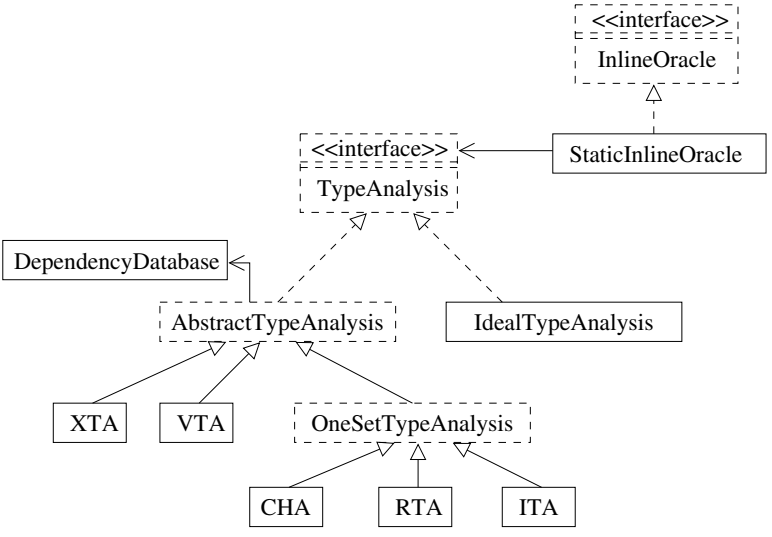

Fig. 2. Type analysis framework diagram

using some heuristics. Hazelwood and Grove [11] described the size heuristic used in Jikes RVM. For the purpose of our 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 most frequently invoked call sites, without a size limit.

Benchmarks. Our benchmark set includes the SpecJVM98 suite [18], SpecJBB2000 [17], a CFS subset evaluator from a data mining package Weka [22], a simulator of certificate revocation schemes [6], and a variation of the simulator interwoven with AspectJ code for detecting calls that return `null` on error conditions.

Table 1 summarizes 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. Columns labeled *total* report the total counts of invocations in each category. Columns labeled *#hottest* are numbers of hottest call sites, ranked in the top 100, whose invocations are more than 1% of *total* in Columns 2 and 5. Columns labeled *coverage* are percentages of invocations contributed by these hottest call sites.

It is interesting to point out that, for most of the benchmarks, the majority of invocations are from a small number of hot call sites. Fewer than 25 call sites exceed the 1% threshold. Only about half of the benchmarks have more than 1M interface invocations. These benchmarks have fewer than 11 hot interface call sites that contribute to more than 92% of invocations.

The `_213_javac` benchmark includes a large amount of auto-generated code. Invocation counts are spread over many call sites. `SpecJBB2000` has a large code base as well, and it runs much longer than other benchmarks. Hot call sites selected by our 1% threshold contribute only about 34% of total invocations.

Table 1. Coverage of the 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

A list of hottest call sites are provided to the inline oracle. The size limit is removed for 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 5, a virtual call site can be inlined using different techniques:

- *direct*: direct inlining if the called 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 analysis results, guards are chosen as a command line option. It can be code patching or method/-class tests. For our experiment we used code patching since it has less runtime overhead.

Monomorphic interface calls can be directly inlined if the receiver is preexistent, or inlined with guards. We found that, in our benchmark set, receivers of nearly all hot interface calls cannot be proven to be preexistent by an invariant argument analysis. Thus, in our results, we omit the *preex* category for interface calls. We also performed another experiment where the inline oracle inlined polymorphic call sites (guarded by method or class tests) that had 1 or 2 targets resolved using type analysis results. However, this did not lead to significantly more inlined calls (only `_213_javac` has a 2% increase). Thus, we do not inline polymorphic calls in our experiment reported here.

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 analyses. Virtual and interface calls are presented separately. Column *total* is the count of invocations in each category. In the *virtual* category, dynamic CHA did nearly as perfect a job as the ideal

type analysis in most benchmarks, except _213_javac and simulator(aspects). On these benchmarks, the 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* (5% of virtual calls could not be solved by CHA), and CHA does not resolve 19% monomorphic virtual calls of simulator(aspects) that were proven to be preexistent by the *ideal* type analysis. In the *interface* category, column 8 shows that a large portion of interface invocations are from monomorphic call sites as well. Dynamic CHA is ineffective on inlining interface calls. Furthermore, the other two simple type analyses, 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.

Table 2. Limit study of method inlining using type analyses

		virtual				interface		
		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

Discussion. Simulator(aspects) is an interesting benchmark. Injecting AspectJ advice code increases the number of invocations and changes inlining behaviors dramatically. In the original benchmark, nearly all virtual calls are monomorphic and can be directly inlined. With aspects, dynamic CHA misses all 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 boxes primitive values in objects and passes them to AspectJ libraries. The value is then unboxed after the library call. The original code for unboxing looks like

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

The single call site of `((Number)v).intValue()` 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 unboxing 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 conclude that:

- 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 precise types when it does not sacrifice other engineering benefits;
- a large percentage of interface calls are monomorphic and used in a simple pattern, but it requires an interprocedural analysis to discover the precise type of the receiver.

4 Dynamic Interprocedural Type Analysis

In Section 3, we presented a type analysis framework for supporting speculative inlining. We also presented the results of our limit study of method inlining which

showed that dynamic CHA is not strong enough for inlining interface calls. In this section, we present an interprocedural, reachability-based, type analysis that is suitable for inlining interface calls.

There are two different approaches to performing a dynamic 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 the part of code related to a request. In this paper we focus on the whole-program approach.

We designed and implemented a dynamic version XTA in our previous work [16] as an example of how to deal with dynamic class loading and reference resolution. However, due to lack of intraprocedural data-flow 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. From the method inlining study, we found XTA results were no better than dynamic CHA.

4.1 Dynamic Variable Type Analysis (VTA)

Design. VTA [19] uses intraprocedural data flow information to propagate type sets. 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).

We use the same approach outlined in [16] to adopt the static VTA to 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 and creates VTA edges. 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. An HIR operand has a declaring type.

Dynamic VTA analysis is driven by events from JIT compilers and class loaders. Figure 3 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.

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 has a new member, the analysis verifies dependencies on this node registered by inlining oracles (see Section 3). The oracle has a chance to perform invalidation if inlining assumptions are violated.

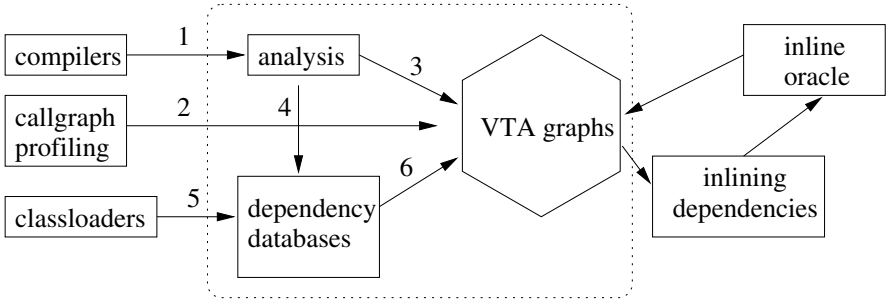

Fig. 3. Model of VTA events

Propagations. To support speculative optimizations, the analysis must keep the results up-to-date whenever a client makes queries. An *eager* approach propagates new types whenever the VTA graph is changed. The second approach is to cache graph changes when collecting constraints of a method, and *batch* propagations at the end of constraint collection. The third approach, as suggested in [15], performs depth-first search (DFS) on nodes that a client makes queries on. Whenever the graph changes, it has to perform DFS on all nodes whose types were used for speculative optimizations to verify that the optimizations are not invalidated by new changes. In our study, we found both eager and batch propagations are efficient, with respect to the total execution time of each benchmark. `_213_javac` takes up to 1.7 seconds and other benchmarks take less than 1 second.

Effectiveness of dynamic VTA. Not surprisingly, VTA is able to handle the simple pattern of interface calls in our benchmarks set. Table 3 compares dynamic counts of inlined interface calls. We omitted benchmarks with few interface calls. Dynamic VTA is able to catch all monomorphic interface calls and allows them to be inlined by using code patching.

Table 3. Comparison of VTA and IdealTypeAnalysis for inlining interface calls

benchmark	Ideal(cp)	VTA(cp)
_209_db	99%	99%
_213_javac	95%	95%
_228_jack	86%	86%
SpecJBB2000	99%	99%

Our preliminary performance measurement shows *Ideal* type analysis yields a small performance improvement over *CHA*. Due to heavier GC workload introduced by VTA graphs, `_213_javac` and `SpecJBB2000` slowed down when using the *copying mark-sweep* collector. Recently we switched to a *generational mark-sweep* collector, which promotes most of VTA graph objects to old generations. The impact of GC has been reduced. Table 4 compares the best run of 10 runs of two benchmarks. Unfortunately, both `_209_db` and `SpecJBB2000` trigger bugs

in the *generational mark-sweep* collector in the version of JikesRVM that we are using for our implementation.[1]

Table 4. Performance comparison using VTA and CHA (GenMS)

benchmark	CHA	VTA	speedups
_213_javac	3.924s	3.900s	0.6%
_228_jack	2.514s	2.460s	2.1%

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. Using `_213_javac` as an example, VTA analysis increases the live data by 60%. It is clear that the whole-program interprocedural analysis has a very high memory overhead.

5 Related Work

We discussed some related work of method inlining while introducing the background in Section 2. The section discusses additional related work on the topic.

Ishizaki et al. [12] conducted an extensive study of dynamic devirtualization techniques for Java programs. In their experiments, size limits were put on inlined targets, and techniques using dynamic CHA were shown to inline about 46% of virtual calls (execution counts). Our study answers the question of what is the limit of method inlining using different type analyses. By lifting the size limit on hottest call sites, we were able to understand the maximum inlining potential using a type analysis. Our limit study shows that CHA performs nearly as well as an ideal type analysis for inlining.

Pechtchanski and Sarkar [15] presented a framework for dynamic optimistic interprocedural analysis (DOIT) in a JIT environment. For each method, the DOIT analysis builds a value graph similar to a VTA graph. However, due to lack of a complete dynamic call graph, DOIT does not track type flow between method calls (parameters and returns). Instead, it uses conservative subtypes of declaring types of method parameters and returns. DOIT is good at obtaining precise type information for fields whose values are assigned in one method and used by another method. Our work focused on limit study of method inlining using type analyses, including online interprocedural analyses based on dynamic call graphs. Our results independently confirms that dynamic CHA is effective for inlining virtual calls in Java programs.

Profiled-directed inlining is effective to identify profitable inlining targets at polymorphic call sites. However, profile-directed inlining requires runtime tests to guard the inlining target. Our focus is on exploiting unguarded inlining opportunities exposed by type analyses.

[1] It is possible that this issue will be resolved when we upgrade our implementation to the lastest version of JikesRVM.

6 Conclusions, Observations and Future Work

In this paper we have presented a study on the limits of speculative inlining. Somewhat to our surprise we found that using dynamic CHA for speculative inlining is almost as good as using an "ideal" analysis, for inlining virtual method calls. However, for even simple uses of interface calls, none of dynamic CHA, RTA, ITA or XTA gives enough information for determining that interface calls are monomorphic. Rather, to detect these opportunities, one requires a stronger type analysis and we presented a dynamic version of VTA for this purpose.

Our experiments with our dynamic VTA do show that it provides detailed enough type information to identify inlining opportunities for interface calls in our benchmark set. However, we also note that the memory overhead of our whole program approach to dynamic VTA is quite large, and we plan to investigate an alternative demand-driven approach.

In addition to these main contributions of the paper, we also made several other general observations about speculative method inlining.

Observation 1. The conventional wisdom is that inlining increases optimization opportunities. However, in the presence of speculative optimizations, inlining may reduce optimization opportunities as well. Figure 4 shows such an example. In Figure 4(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 4(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 4(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.child(f);
}

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

(a) source

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

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

(b) inline *Foo.m* only

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

(c) inline *child*, then *Foo.m*

Fig. 4. An example where inlining can reduce optimization opportunities

The above dilemma could be resolved by using on-stack replacement technology [10,21] or thin guards [4]. Indeed, method invalidation performs on-stack replacement at method entries. A compiler can insert a general on-stack replace-

ment point after the statement *Foo f = getfield* with a condition that *Foo.m* is currently final. The compiler can directly inline the body of *Foo.m* into the loop.

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 [7] 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. Based on this study we have concluded that a type analysis for invokeinterfaces is an important area of research, and we are currently working on a demand-driven analysis and compact graph representation to reduce the costs of dynamic VTA. We are also looking at more applications of dynamic interprocedural analysis in JIT compilers. A new research topic is to investigate the effectiveness of compiler optimizations on different design patterns.

Acknowledgments. This work was supported, in part, by NSERC. We would like to thank Navindra Umanee for his proofreading of this paper. We also appreciated anonymous reviewers' constructive comments.

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. K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language (Third Edition)*. Addison-Wesley, 2000.
4. M. Arnold and B. G. Ryder. Thin Guards: A Simple and Effective Technique for Reducing the Penalty of Dynamic Class Loading. In *16th European Conference for Object-Oriented Programming (ECOOP'02)*, 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. Gnu classpath. `http://www.gnu.org/classpath`.
8. J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *9th European Conference on Object-Oriented Programming (ECOOP'95)*, pages 77 – 101, Aug. 1995.
9. D. Detlefs and O. Agesen. Inlining of Virtual Methods. In *13th European Conference on Object-Oriented Programming (ECOOP'99)*, pages 258 – 278, June 1999.
10. S. J. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *International Symposium on Code Generation and Optimization (CGO'03)*, pages 241 – 252, March 2003.
11. K. Hazelwood and D. Grove. Adaptive Online Context-Sentitive Inlining. In *International Symposium on Code Generation and Optimization (CGO'03)*, pages 253 – 264, March 2003.
12. 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 (OOPSLA'00)*, pages 294–310, 2000.
13. Jikes^{TM} Research Virtual Machine. `http://www-124.ibm.com/developerworks/-oss/jikesrvm/`.
14. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
15. I. Pechtchanski and V. Sarkar. Dynamic optimistic interprocedural analysis: A framework and an application. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 195 – 210, 2001.
16. 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.
17. Spec JBB2000 benchmark. `http://www.spec.org/jbb2000/`.
18. Spec JVM98 benchmarks. `http://www.spec.org/osg/jvm98/index.html`.
19. 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 *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, pages 264–280, 2000.
20. F. Tip and J. Palsberg. Scalable Propagation-based Call Graph Construction Algorithms. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, pages 281–293, Oct. 2000.
21. Urs Hölzle and Craig Chambers and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 32 – 43, 1992.
22. Weka 3: Data Mining Software in Java. `http://www.cs.waikato.ac.nz/ml/weka/`.