Towards Dynamic Interprocedural Analysis in JVMs *

Feng Qian Laurie Hendren

School of Computer Science, McGill University
3480 University Street, Montreal, Quebec
Canada H3A 2A7
{fqian,hendren}@cs.mcgill.ca

Abstract

This paper presents a new, inexpensive, mechanism for constructing a complete call graph for Java programs at runtime, and provides an example of using the mechanism for implementing a dynamic reachability-based interprocedural analysis (IPA), namely dynamic XTA.

Reachability-based IPAs, such as points-to analysis and escape analysis, require a context-insensitive call graph of the analyzed program. Computing a call graph at runtime presents several challenges. First, the overhead must be low. Second, when implementing the mechanism for languages such as Java, both polymorphism and lazy class loading must be dealt with correctly and efficiently. We propose a new, low-cost, mechanism for constructing runtime call graphs in a JIT environment. The mechanism uses a profiling code stub to capture the first execution of a call edge, and adds at most one more instruction to repeated call edge invocations. Polymorphism and lazy class loading are handled transparently. The call graph is constructed incrementally, and it supports optimistic analysis and speculative optimizations with invalidations.

We also developed a dynamic, reachability-based type analysis, dynamic XTA, as an application of runtime call graphs. It also serves as an example of handling lazy class loading in dynamic IPAs.

The dynamic call graph construction algorithm and dynamic version of XTA have been implemented in Jikes RVM. We present empirical measurements of the overhead of call graph profiling and compare the characteristics of call graphs built using our profiling code stubs with conservative ones constructed by using dynamic class hierarchy analysis (CHA).

*The work was supported, in part, by NSERC.

1 Introduction

*Inter*procedural analyses (IPAs) derive more precise program information than *intra*procedural ones. Static IPAs provide a conservative approximation of runtime information to clients for optimizations. A foundation of IPA is the call graph of the analyzed program. IPAs for Object-Oriented (OO) programs share some common challenges. Virtual calls (polymorphism) make call graph construction difficult. Further, since the code base tends to be large, the complexity and precision of the analysis must be carefully balanced.

One difficulty of call graph construction for OO languages lies in how to approximate the targets of polymorphic calls. In addition to polymorphism, call graph construction for Java is further complicated by the presence of dynamic class loading. Static IPAs assume that the whole program is available at analysis time. However, this may not be the case for Java. A Java program can download a class file from the network or other unknown resources. Even when all programs exist on local disks, a VM may choose to load classes lazily, on demand, to reduce resource usage and improve responsiveness [21]. When a JIT compiler encounters an unresolved symbolic reference, it may choose to delay resolution until the instruction is executed at runtime. A dynamic analysis has to deal with these unresolved references. A more subtle problem, usually ignored by static IPAs for Java, is that a runtime type is defined by both the class name and its initial class loader. Therefore, a correct dynamic IPA has to be incremental (dealing with dynamic class loading), efficient, and type safe.

Although Java's dynamic features pose difficulties for program analyses, there are many opportunities at runtime that can only be enjoyed by dynamic analyses. For example, a dynamic analysis only needs to analyze loaded classes and invoked methods. Therefore, the analyzed code base can be much smaller than in a conservative static analysis. Further, dynamic class loading can improve the precision of type analyses. The

set of runtime types can be limited to loaded classes. Thus, a dynamic analysis has more precise type information than its static counterpart. Further, in contrast to the conservative (pessimistic) nature of static analysis, a dynamic one can be optimistic about future execution, if used in conjunction with runtime invalidation mechanisms [12, 18, 23, 30].

Over the last 10 years, VM technology has greatly advanced. JIT compilers now implement most *intra*procedural data-flow analyses that can be found in static compilers [1, 22]. Further performance improvements have been achieved using adaptive and feedback-directed compilation [4, 5].

Dynamic *inter*procedural analysis, however, has not yet been widely adopted. Some type-based IPAs [18, 23] have gained ground in JIT compilation environments. However, work relating to more complicated, reachability-based IPAs, such as dynamic points-to analysis and escape analysis, is only just starting to emerge [15].

In this paper, we present a call graph construction mechanism for reachability-based interprocedural analyses at runtime. Instead of approximating a call graph as in static IPAs, our mechanism uses a profiling code stub to capture invoked call edges. The mechanism overcomes difficulties caused by dynamic class loading and polymorphism. Most overhead happens at JIT compilation and class loading time. It has only small overhead on the performance of applications in a JIT environment. A very desirable feature of the mechanism is that call graphs can be built incrementally while execution proceeds. This enables speculative optimizations using runtime invalidations for safety.

Dynamic IPAs seem more suitable for long-running applications in adaptive recompilation systems. Pechtchanski and Sarkar [23] described a general approach of using dynamic IPAs. A virtual machine gathers information about compiled methods and loaded classes in the initial state, and performs recompilation and optimizations only on selected hot methods. When the application reaches a "stable state", information changes should be rare.

Based on our new runtime call graph mechanism, we describe the design and implementation of an online version of an example IPA, XTA type analysis [32]. Dynamic XTA uses dependency databases to handle unresolved types and field references. The analysis is driven by VM events such as compilation, class loading, or the discovery of new call edges.

The rest of paper is organized as follows. Section 2 introduces our new call graph construction mechanism which serves as the basis for dynamic IPAs. In the following section, Section 3, we describe the design of a specific dynamic IPA, dynamic XTA type analysis, in the presence of lazy class loading. The call graph mechanism and dynamic XTA have been implemented in Jikes RVM. Section 4 analyzes the cost of call graph profiling and compares the characteristics of profiled call graphs to conservative ones built by dynamic CHA on a set of standard benchmarks. Section 5 discusses related work and conclusions are presented in Section 6.

2 Online Call Graph Construction

Context-insensitive call graphs are commonly used by IPAs, where a method is represented as one node in the call graph. There exists a directed edge from a method *A* to a method *B* if *A* calls *B*.

Dynamic class hierarchy information can be used to build a conservative call graph at runtime. However, it is desirable to have a more precise call graph for most interprocedural analyses. We propose a new mechanism for profiling and constructing context-insensitive call graphs at runtime. The mechanism initializes call edges using a profiling code stub. When the code stub gets executed, it generates a new call edge event, then it triggers method compilation if the method is not compiled yet, and finally patches the address of the real target. The mechanism captures the first execution event of each call edge, and the first execution has some profiling overhead. The repeated calls only need to execute at most one more instruction. Clients, such as call graph builders, can register callback routines called by a profiling code stub when new call edges are discovered. Callbacks can perform necessary actions before the callee is invoked.

The remainder of this section is structured as follows. First, in Section 2.1, we briefly introduce a conservative approach for building call graphs using runtime class hierarchy information. In Section 2.2 we give the necessary background, describing the existing implementation of virtual method tables in Jikes RVM. In Section 2.3 we describe the basic mechanism we propose for building call graphs at runtime, and in Section 2.4 we show how this basic mechanism can be optimized to reduce overheads.

2.1 Conservative call graph construction using dynamic CHA

Due to polymorphism, the exact types of the receiver of a virtual call site may not be known at analysis time. Class hierarchy analysis (CHA) [9] makes the conservative assumption that all subtypes of a receiver's declaring type are possible types at runtime.

CHA was originally suggested as a static analysis, where all classes and the complete class hierarchy are known at compile time. However, when adapting CHA

to be a dynamic analysis one must consider that the class hierarchy can grow as classes are dynamically loaded. Thus a dynamic CHA must record all virtual call sites that have already been resolved. When a new class is loaded, it must be included in the the type set of any recorded call site whose receiver's declaring class is a super type of the newly loaded class. If the newly added type, of a call site, declares a method with the same signature as the callee, a new call edge to the method must be generated at this call site. Hirzel et. al. [15] have given a detailed description of this approach. In our study, we implemented a call graph builder using dynamic CHA to compare with our proposed profiler-based mechanism.

2.2 Background: virtual method table

We propose a profiling mechanism for constructing a more precise dynamic call graph than a conservative one constructed using dynamic CHA. To understand how the mechanism works, we first revisit the virtual method dispatch table in Jikes RVM [1], which is a standard implementation in modern Java virtual machines. Figure 1(a) depicts the object layout in Jikes RVM. Each object has a pointer, in its header, to the Type Information Block (TIB) of its type (class). A TIB is an array of objects that encodes the type information of a class. At a fixed offset from the TIB header is the Virtual Method Table (VMT) which is embedded in the TIB array. A resolved method has an entry in the VMT of its declaring class, and the entry offset to the TIB header is a constant, say `method_offset`, assigned during class resolution. A VMT entry records the instruction address of the method that owns it. Figure 1(b) shows that, if a class, say `A`, inherits a method from its superclass, `java.lang.Object`, the entry at the method offset in the subclass' TIB has the inherited method's instruction address. If a method in the subclass, say `D`, overrides a method from the superclass, the two methods still have the same offset, but the entries in two TIBs point to different method instructions.

Given an object pointer at runtime, an *invokevirtual* bytecode is implemented by three basic operations:

```
TIB = * (ptr + TIB_OFFSET);
INSTR = TIB[method_offset];
JMP INSTR
```

The first instruction obtains the TIB address from the object header. The address of the real target is loaded at the `method_offset` offset in the TIB. Finally the execution is transferred to the target address.

Lazy method compilation works by first initializing TIB entries with the address of a lazy compilation code stub. When a method is invoked for the first time, the code stub gets executed. The code stub triggers the compilation of the target method and patches the address of the compiled method into the TIB entry (where the code stub resided before).

2.3 Call graph construction by profiling

In normal lazy method compilation, the code stub captures the first invocation of a method without distinguishing callers. In order to capture call edges, we extended the TIB structure to store information per caller. Figure 1(c) shows the extended TIB structure. The TIB entry of a method is replaced by an array of instruction addresses. We call the array a Caller-Target Block (CTB). The indices of CTB slots (*caller_index*) are dynamically assigned to callers of the method by the JIT compilers. Note that now an *invokevirtual* bytecode takes one extra load to get the target address.

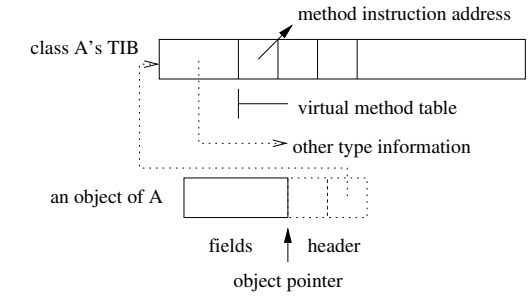
```
TIB = * (ptr + TIB_OFFSET);
/* load method's CTB array from TIB */
CTB = TIB[method_offset];
/* load method's code address */
INSTR = CTB[caller_index];
JMP INSTR
```

The lazy method compilation code stub is extended to a profiling code stub which, in addition to triggering the lazy compilation of the callee, also generates a new call edge event from the caller to the callee. Initially all of the CTB entries have the address of the profiling code stub. When the code stub at a CTB entry gets executed, it notifies clients monitoring new call edge events, and compiles the callee method if necessary. Finally the code stub patches the callee's instruction address into the CTB entry. Clearly the profiling code stub at each entry of the CTB array will execute at most once, and the rest of the invocations from the same caller will execute the callee's machine instruction directly.

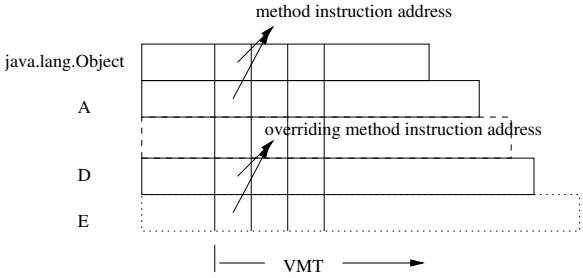
There remain four problems to address. First, one needs a convenient way of indexing into the CTBs which works even in the presence of lazy class loading. Second, the implementation of interface calls should be aware of the CTB array. Third, object initializers and static methods can be handled specially. Fourth, we must handle the case where an optimizing compiler inlines one method into another. Our solution to these four problems is given below.

2.3.1 Allocating slots in the CTB

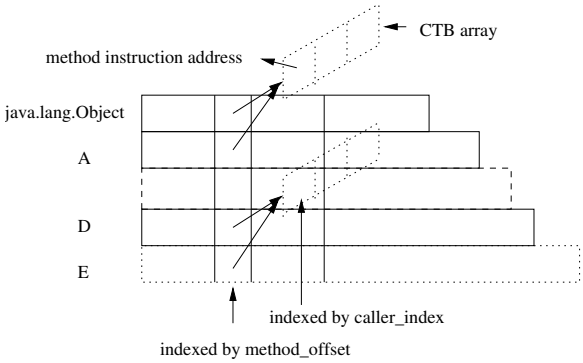
To index callers of a callee, our modified JIT compiler maintains a table of *(callee, caller)* pairs. In Java bytecode, the target of *invokevirtual* is only a symbolic reference to the name and descriptor of the method as well as a symbolic reference to the class where the method can be found. Resolving the method reference requires the class to be loaded first. A VM can delay method resolution until the call instruction is executed at runtime.



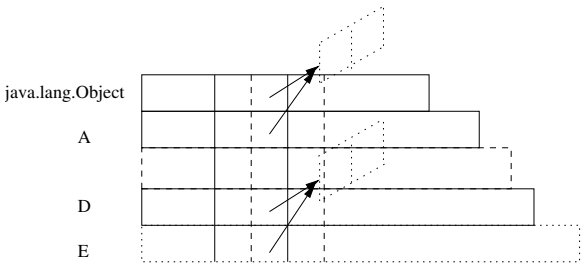
(a) TIB in Jikes RVM



(b) VMT in Jikes RVM



(c) Extended VMT for profiling call graph



(d) Inlining 1 element of CTB

Figure 1: Virtual Method Dispatching Table in Jikes RVM

Therefore, the real target may not be known at JIT compilation time.

To deal with lazy class resolution and polymorphism, our approach uses the callee's method name and descriptor in the table. For example, if both methods `X.x()` and `Y.y()` have virtual calls of a symbolic reference `A.m()`, and another method `Z.z()` has a virtual call of `B.m()`, our approach assumes that all three methods are possible callers of any method with the signature `m()`[1], and allocates slots in the TIB for all of them. At runtime, only two CTB entries of `A.m()` may be filled, and only one entry of `B.m()` may get filled. With this solution no accuracy is lost, but some space may be wasted due to unfilled CTB entries. Although some space is sacrificed, our approach simplifies the task of handling symbolic references and polymorphism. In real applications we observed that only a few common method signatures, such as `equals(java.lang.Object)`, and `hashCode()`, have large caller sets where space is unused.

2.3.2 Approximating interface calls

Interface calls are considered to be more expensive than virtual calls in Java programs because a normal class can only have a single super class, but could implement multiple interfaces. Jikes RVM has an efficient implementation of interface calls using a interface method table with conflict resolution stubs [2].

We tried two approaches to handling interface calls in the presence of CTB arrays. Our first approach profiles interface calls by allocating a *caller_index* for a call site in the JIT compiler and generating an instruction before the call to save the index value in a known memory location. After a conflict resolution stub has found its target method, it loads the index value from the known memory location. The CTB array of the target method is loaded from the TIB array of receiver object's declaring class. The target address is read out from the CTB at the index, and finally the resolution stub jumps to the target address. This approach uses two more instructions to store and load the *caller_index* than *invokevirtual* calls. After introducing one of our optimizations in Section 2.4, inlining CTB elements into TIBs, the conflict resolution stub requires more instructions to check the range of the index value to determine if the indexed CTB element is inlined in the TIB or not.

Our second approach was to simply use dynamic CHA to build call edges for *invokeinterface* call sites, without introducing profiling instructions.

Since our profiling results showed that the number of

[1] A full method descriptor should include the name of the method, parameter types, and the return type. In this example, we use the name and parameter types only for simplicity.

call edges from *invokeinterface* call sites is only a small portion of all edges, we chose to use the second approach for the remaining experiments in this paper.

2.3.3 Handling object initializers and static methods

Because there are many object initializers that share a common name `<init>` and descriptor, their CTB arrays may grow too large if we allocate CTB slots using the name and descriptor as index. Since calls of object initializers and static methods are monomorphic, the allocation of CTB slots for each method is independent of other methods with the same name and descriptor. For example, static methods `A.m()` and `B.m()` both can use the same CTB index for different callers. Therefore, there is no superfluous space in CTB arrays of object initializers and static methods. For unresolved static or object initializer method references, a dependency on the reference from the caller is registered in a database. When the method reference gets resolved (this happens due to a class loading event), the dependency is converted to a call edge conservatively. Using the *_213_javac* benchmark as example, we found this conservativeness only adds 1.5% more edges.

2.3.4 Dealing with Inlining

In an adaptive system, inlining might be applied on a few hot methods. We capture these events as follows. When a callee is inlined into a caller by an optimizing JIT compiler, the call edge from the caller to callee is added to the call graph unconditionally. This is a conservative solution without runtime overhead. Since an inlined call site is likely executed before its caller becomes hot, the number of added superfluous edges is modest.

2.4 Optimizations

Since Jikes RVM is written in Java, our runtime call graph construction mechanism may incur two kinds of overhead. First, adding one instruction per call can potentially consume many CPU cycles because Jikes RVM itself is compiled using the same compilers used for compiling the applications, and it also inserts many system calls into applications for runtime checks, locks and object allocations. Second, a CTB array is a normal Java array with a three-word header; thus CTB arrays can increase memory usage and create extra work for garbage collectors.

Table 1 shows the distribution of the CTB sizes for the SpecJVM98 benchmarks [27] profiled in a *FastAdaptiveSemiSpace* boot image. The boot image contains mostly RVM classes and a few Java utility classes. We only profiled methods from Java libraries and benchmarks. A small number of methods of classes in the boot image may have CTB arrays allocated at runtime because there is no clear cut mechanism for distinguishing between Jikes RVM code and application code. The first column shows the range of the number of callers. The second and third columns list the distributions of methods belonging to Java libraries and SpecJVM application code.[2] To demonstrate that most methods have few callers, we calculated the cumulative percentages of methods that have no caller, $\leq$ 1, $\leq$ 3 and $\leq$ 7 callers in the first to fourth rows. We found that 89% of methods from (loaded classes in) Java libraries and 84% of methods from SpecJVM98 have no more than 7 callers. In these cases, it is not wise to create short CTB arrays because each array header takes 3 words. The last data row labelled "`TOTAL`" gives the total number of methods of all classes and the number of methods in each of two subcategories.

#callers	Java Libraries		SpecJVM App	
0	2214	69.08%	507	19.32%
1	291	78.16%	815	50.38%
2-3	172	83.53%	608	73.55%
4-7	170	88.83%	283	84.34%
8-	358		411	
TOTAL	3205		2624	

Table 1: Distribution of CTB sizes

To avoid the overhead of array headers for CTBs, and to eliminate the extra instruction to load the CTB array from a TIB in the code for *invokevirtual* instructions, a local optimization is to inline the first few elements of the CTB into the TIB. Since caller indices are assigned at compile time, a compiler knows which part of the CTB will be accessed in the generated code. To accommodate the inlined part of the CTB, a class' TIB entry is expanded to allow a method to have several entries. Figure 1(d) shows the layout of TIBs with one inlined CTB element. When generating instructions for a virtual call, the value of the caller's CTB index, `caller_index`, is examined: if the index falls into the inlined part of the CTB, then invocation is done by three instructions:

```
TIB = * (ptr + TIB_OFFSET);
INSTR = TIB[method_offset + caller_index];
JMP INSTR
```

Whenever a CTB index is greater than or equal to the inlined CTB size, *INLINED_CTB_SIZE*, then four instructions must be used for the call:

```
TIB = * (ptr + TIB_OFFSET);
CTB = TIB[method_offset + CTB_ARRAY_OFFSET];
INSTR = CTB[caller_index - INLINED_CTB_SIZE];
JMP INSTR
```

[2] We used package names to distinguish classes.

Note that in addition to saving the extra instruction for inlined CTB entries, the space overhead of the CTB header is eliminated in the common cases where all CTB entries are inlined.

Another source of optimization is to avoid the overhead of handling system code, such as runtime checks and locks, inserted by compilers, because this code is frequently called and ignoring them does not affect the semantics of applications. To achieve this, the first CTB entry is reserved for the purpose of system inserted calls. Instead of being initialized with the address of a call graph profiling stub, the first entry has the address of a lazy method compilation code stub or method instructions. When the compiler generates code for a system call, it always assigns the *zero* `caller_index` to the caller. To avoid the extra load instruction, the first entry of a CTB array is always inlined into the TIB.

3 Dynamic XTA Type Analysis

A runtime call graph is constructed incrementally while a program runs. Dynamic IPAs using call graphs also have to perform analysis incrementally. A dynamic IPA has to overcome the difficulties of dynamic class loading and lazy resolution of references. As one example application of profiled call graphs, we developed a dynamic XTA type analysis which can serve as a general model of other dynamic IPAs.

Tip and Palsberg [32] proposed a set of propagation-based call graph construction algorithms for Java with different granularities ranging from RTA to 0-CFA. 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. To approximate the targets of a virtual call, XTA uses the reachable types of the caller to perform method lookups statically. When new targets are discovered, new edges are added into the graph. The analysis performs propagation until reaching the fixed point. XTA has the same complexity as subset-based points-to analysis, $O(n^3)$, but with fewer nodes in the graph.

XTA analysis is a good candidate as a dynamic IPA: it requires reasonably small resources to represent the graph since it ignores the dataflow inside a method. The results might be less precise than an analysis using a full dataflow approach [23, 31]. On the other hand, rich runtime type information may improve the precision of dynamic XTA. The results of the analysis can be used for method inlining and the elimination of type checks.

Like other static IPAs for Java, static XTA assumes the whole programs are available at analysis time. Dynamically-loaded classes must be supplied to the analysis manually. The burden on static XTA is to approximate targets of polymorphic calls while propagating types along the call graph. However, dynamic XTA does not have this difficulty because it uses the call graph constructed at runtime. The call graph used by dynamic XTA is significantly smaller than the one constructed by static XTA. However, a new challenge for dynamic XTA comes from lazy class loading. In a Java class file, a call instruction has only the name and descriptor of a callee as well as a symbolic reference to a class where the callee can be found. Similarly, field access instructions have symbolic references only. At runtime, a type reference is resolved to a class type and a method/field reference is resolved to a method/field before any use. [3]

```
class A { Object f; }

class B extends A {
}

 A a;
 a.f = ...;

 B b;
 o = b.f;
```

(a) Java source

```
......
putfield A.f Ljava/lang/Object;

getfield B.f Ljava/lang/Object;
......
```

(b) compiled bytecode

Figure 2: Field reference example

Figure 2 shows a simple example to help understand the problem caused by symbolic references. Class *B* extends class *A*, which declares a field *f*. Field accesses of *a.f* and *b.f* were compiled to *putfield* and *getfield* instructions with different symbolic field references *A.f* and *B.f*. At runtime, before the *getfield* instruction gets executed, the reference *B.f* is resolved to field *f* of class *A*. However, a dynamic analysis or compiler cannot determine that *B.f* will be resolved to *f* of *A* without loading both classes *B* and *A*.

Resolution of method/field references requires the classes to be loaded and resolved first. However, a JVM may choose to delay such resolution as late as possible to reduce resource usage and improve responsiveness [21]. To port a static IPA for Java to a dynamic IPA, the analysis must be modified to handle unresolved references.

[3] In following presentation, we use type(s) as a short name for resolved class type(s), and use references for symbolic references, e.g., type references, method/field references.

In this section, we demonstrate a solution for the problem for dynamic XTA; our solution is also applicable to general IPAs for Java.

Our dynamic XTA analysis constructs a directed XTA graph $G = \{V, E, TypeFilters, ReachableTypes\}$:

- $V \subseteq M \cup F \cup \{\alpha\}$, where M is a set of resolved methods, F is a set of resolved fields, and α is an abstract name representing array elements;
- $E \subseteq V \times V$, is the set of directed edges;
- $TypeFilters \subseteq E \rightarrow S$, is a map from an edge to a set of types, S;
- $ReachableTypes \subseteq V \rightarrow T$, is a map from a node to a set of resolved types T.

The XTA graph combines call graphs and field/array accesses. A call from a method *A* to a method *B* is modelled by an edge from node *A* to node *B*. The filter set includes parameter types of method *B*. If *B*'s return type is a reference type, it is added in the filter set of the edge from *B* to *A*. Field reads and writes are modelled by edges between methods and fields, with the fields' declaring classes in the filter. Each node has a set of reachable (resolved) types.

Basic graph operations include adding new edges, new reachable types, and propagations:

- *addEdge(a, b, T)*, creates an edge from a node *a* to a *b* if it does not exist yet; then adds types in set T to the filter set associated with the edge;
- *addType(a, t)*, adds a type *t* to the reachable type set of a node *a*;
- *propagate(t, a)*, propagates a type *t* to successors of a node *a* if *t* is a subtype of a type in the filter set associated with the edge from *a* to its successor. If *t* is not in a successor's reachable type set, it is recursively propagated to all of that successor's descendants until there are no further changes.

Since the call graph is constructed at runtime using code stubs, there are no new edges created during propagation. The complexity of the propagation operation is linear.

Dynamic XTA 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 XTA analysis: XTA graphs, the analysis, and dependency databases. The JIT compilers notify the analysis by channel 1 that a method is about to be compiled. The analysis scans the bytecode of the method body and, for each *new* instruction with a resolved type, the analysis adds the type into the reachable type set of the method via channel 3; otherwise it registers a dependency on the unresolved type reference for the method via channel 4. Similarly for field accesses, if the field reference can be resolved without triggering class loading, the analysis adds a directed edge into the graph via channel 3; otherwise, it registers a dependency on unresolved field reference for the method. Since we use call graph profiling code stubs to discover new call edges, the code stubs can add new edges to the graph by channel 2. Whenever a type reference or field reference gets resolved, the dependency databases are notified (by channel 5), and registered dependencies on resolved references are resolved to new reachable types or new edges of the graph. Whenever the graph is changed (either an edge is changed, or a new reachable type is added), a propagator propagates type sets of related nodes until no further change occurs.

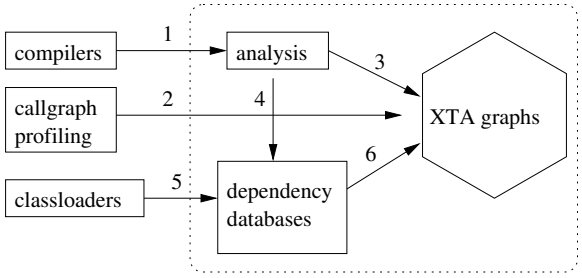


Figure 3: Model of XTA events

Compared to static IPAs such as points-to analysis, the problem set of dynamic XTA analysis is much smaller because the graph contains only compiled methods and resolved fields at runtime. Although optimizations such as off-line variable substitution [24] and online cycle elimination [11] can help reduce the graph size further, the complexity and runtime overhead of the algorithms may prevent them from being useful in a dynamic analysis. Efficient representations for sets and graphs, such as hybrid integer sets [14, 20] and BDDs [7], are more important since the dynamic analysis has bounded resources. In our implementation, graphs, sets, and dependency databases were implemented using hybrid integer sets and integer hash maps.

Graph changes are driven by runtime events such as newly compiled methods, newly discovered call edges, or dynamically loaded classes. Similar to the DOIT framework [23], clients using XTA analysis for optimizations should register properties to be verified when the graph changes. Since the analysis can notify the client when a change occurs, the clients can perform an invalidation of compiled code or recover execution states to a safe point. The exact design and implementation details for verifying properties and performing invalidations are beyond the scope of this paper. Readers can

find more about dependency management and invalidation techniques in [12, 16, 18].

4 Evaluation

We have implemented our proposed call graph construction mechanism in Jikes RVM [19] v2.3.0. Our benchmark set includes the SpecJVM98 suite [27], SpecJBB2000 [26], and a CFS subset evaluator from a data mining package Weka [34]. We made a variation of the *FastAdaptiveCopyMS* boot image for evaluating our mechanism. In our experiment, classes whose names start with `com.ibm.JikesRVM` are not presented in the dynamic call graphs because (1) the number of RVM classes is much larger than the number of classes of applications and libraries, and (2) the classes in the boot image were statically compiled and optimized. Static IPAs such as extant analysis [28] may be applied on the boot image classes. We report the experimental results for application classes and Java library classes.

In our initial experiments we found that the default adaptive configuration gave significantly different behaviour when we introduced dynamic call graph construction because the compilation rates and speedup rates of compilers were affected by our call graph profiling mechanism. It was possible to retrain the adaptive system to work well with our call graph construction enabled, but it was difficult to distinguish performance differences due to changes in the adaptive behaviour from differences due to overhead from our call graph constructor. In order to provide comparable runs in our experiments, we used a counter-based recompilation strategy and disabled background recompilation. We also disabled adaptive inlining. This configuration is more deterministic between runs as compared to the default adaptive configuration. This behavior is confirmed by our observation that, between different runs, the number of methods compiled by each compiler is very stable. The experiment was conducted on a PC with a 1.8G Hz Pentium 4 CPU and 1G memory. The heap size of RVM was set to 400M. Note that Jikes RVM and applications share the same heap space at runtime.

The first column of Table 2 gives four configurations of different inlined CTB sizes and the default *FastAdaptiveCopyMS* configuration without the dynamic call graph builder. The boot image size was increased about 10%, as shown in column 2, when including all compiled code for call graph construction. Inlining CTB elements increases the size of TIBs. However, changes are relatively small (the difference between inlined CTB sizes 1 and 2 is about 153 kilobytes), as shown in the second column.

The third column shows the memory overhead, in bytes, of allocated CTB arrays for methods of classes in Java libraries and benchmarks when running the *_213_javac* benchmark with an input size 100. The time for creating, expanding and updating CTB array is negligible.

Inlined CTB sizes	bootimage size (bytes)		CTB space (bytes)
default	24,477,688	N/A	N/A
1	26,915,236	9.96%	678,344
2	27,068,340	10.58%	660,960
4	27,327,760	11.64%	637,312
8	27,838,796	13.73%	607,712

Table 2: Bootimage sizes and allocated CTB sizes of _213_javac

A Jikes RVM-specific problem is that the RVM system and applications share the same heap space. Expanding TIBs and creating CTBs consumes heap space, leaving less space for the applications, and also adding more work for the garbage collectors. We examine the impact of CTB arrays on the GC. Since CTB arrays are likely to live for a long time, garbage collection can be directly affected. Using the *_213_javac* benchmark as example with the same experimental setting mentioned before, GC time was profiled and plotted in Figure 4 for the default system and configurations with different inlined CTB sizes. The x-axis is the garbage collection number during the benchmark run, and the y-axis is the time spent on each collection. We found that, with these CTB arrays, the GC is slightly slower than the default system, but not significantly. When inlining more CTB elements, the GC time is slightly increased. This might be because the increased size of TIBs exceeds the savings on CTB array headers when the inlining size gets larger. We expect a VM with a specific system heap would solve this problem.

The problem mentioned above also poses a challenge for measuring the overhead of call graph profiling. Furthermore, the call graph profiler and data structures are written in Java, which implies execution overhead and memory consumption, affecting benchmark execution times. To only measure just the overhead of executing profiling code stubs, we used a compiler option to replace the allocated caller index by the *zero* index. When this option is enabled, calls do not execute the extra load instruction and profiling code stub, but still allocate CTB arrays for methods. For CFS and SpecJVM98 benchmarks, we found that usually the first run has some performance degradation when executing profiling

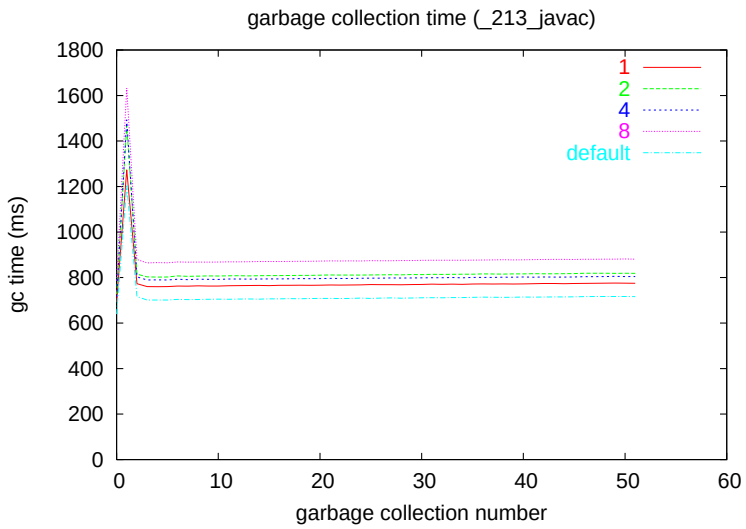


Figure 4: GC time when running _213_javac

code stubs (up to 9% except for _201_compress[4]), but the degradation is not significant upon reaching a stable state (between -2% to 3%). The performance of SpecJBB2000 is largely unaffected. Compared to not allocating CTB arrays at all (TIBs, however, are still expanded), the performance change is also very small. For our set of benchmarks, it seems that inlining more CTB array elements does not result in further performance improvements.

Table 3 shows the size of the profiled call graph compared to the one constructed by using dynamic CHA. The size of the call graph generated by CHA is shown in the second and third columns where, in the second column, the total number of call edges is followed by those from *invokevirtual* call sites only. The number of methods is given in the third column. From the second column, we can see that for all benchmarks, the major part of call edges come from *invokevirtual* instructions. The fourth and fifth columns show the size of profiled call graph and the percentages comparing to the sizes of the CHA call graphs. Call graphs constructed using our profiling code stubs have 20% to 50% fewer call edges than the CHA-based ones. More call edges from *invokevirtual* sites were reduced than for the other types of call instructions because we took the conservative CHA approach on other types of call instructions to reduce runtime overhead. The reduction for the number of methods is not as significant as for the number of call edges.

[4]The first run of _201_compress does not promote enough methods to higher optimization levels.

5 Related Work

Static call graph construction for OO programming languages focuses on approximating a set of types that a receiver of a polymorphic call site may have at runtime. Static class hierarchy analysis (CHA) [9] treats all subclasses of a receiver's declaring class as possible types at runtime. Rapid type analysis (RTA) [6] prunes the type set of CHA by eliminating types that do not have an allocation site in the whole program. Static CHA and RTA examine the entire set of classes. Propagation-based algorithms propagate types from allocation sites to receivers of polymorphic call sites along a program's control flow. Assignments, method calls, field and array accesses may pass types from one variable to another. Context-insensitive algorithms can be modelled as unification-based [29] or subset-based [3] propagation as points-to analysis. The complexity varies from $O(N\alpha(N,N))$ for unification-based analysis to $O(N^3)$ for subset-based analysis. Context-sensitive algorithms [10] might yield more precise results but are difficult to scale to large programs. Since CHA and RTA do not use control flow information, both are considered to be fast algorithms when compared with propagation-based algorithms. Both VTA [31] and XTA analysis [32] are simple propagation-based type analyses for Java. The analyses can either use a call graph built by CHA/RTA, then refine it, or build the call graph on the fly [25].

Ishizaki et. al. [18] published a new method of utilizing class hierarchy analysis for devirtualization at runtime. If the target of a virtual call is not overridden in the current class hierarchy, a compiler may choose to inline the target directly with a backup code of normal vir-

benchmark	CHA			Profiling		
	#edges		#methods	#edges		#methods
compress	733	458	365	516 (70%)	241 (53%)	303 (83%)
jess	2549	1364	1130	1986 (78%)	801 (59%)	802 (71%)
db	961	578	413	711 (74%)	328 (57%)	350 (84%)
javac	9427	8137	1662	4315 (46%)	3025 (37%)	1169 (70%)
mpegaudio	1228	849	645	853 (69%)	475 (56%)	474 (73%)
mtrt	1192	833	563	950 (80%)	591 (71%)	446 (79%)
jack	1746	1131	703	1413 (81%)	799 (71%)	572 (81%)
jbb	4166	2802	1394	3221 (77%)	1757 (63%)	1160 (83%)
CFS	2101	1552	843	1259 (60%)	712 (46%)	557 (66%)

Table 3: The number of call edges and methods discovered by CHA and Profiling

tual call. To cope with dynamic class loading, the runtime system monitors class loading events. If a newly loaded class overrides a method that has been directly inlined in some callers, the code of callers has to be patched with the backup path before class loading proceeds. Pechtchanski and Sarkar [23] presented a framework for performing dynamic optimistic interprocedural analysis in a Java virtual machine. Similar to dynamic CHA, their framework builds detailed dependencies between optimistic assumptions for optimizations and runtime events such as method compilation. Invalidation is a necessary technique for correctness when the assumption is invalidated. Neither of these approaches explored reachability-based analysis which requires a call graph as the base. Our work inherits the merits of their work, supporting optimistic optimizations and invalidations. Bogda and Singh [8] experimented an online interprocedural shape analysis, which uses an inlining cache to construct the call graph at runtime. However, their implementation was based on bytecode instrumentation, which incurs a large overhead. Our work aims to build an accurate call graph with little overhead to enable reachability-based IPAs at runtime.

In parallel to our work, Hirzel et. al. [15] adapted a static subset-based pointer analysis to a runtime analysis in Jikes RVM. In their work, an approach similar to ours is used to handle lazy class loading and unresolved method references. However, they used a conservative call graph constructed by dynamic CHA, and the analysis also considers the dataflow in a method. It would be interesting to see how much the smaller call graph produced by our mechanism could improve their pointer analysis results.

Code-patching [18] and stack-rewriting [12, 17] are necessary invalidation techniques for optimistic optimizations. Those operations might be expensive at runtime. An optimization client should use these techniques wisely. For example, if an optimistic optimization has rare invalidations, these techniques can be applied. In situations of frequent invalidations or incomplete IPA information, an optimization may choose runtime checks to guard optimized code.

Static IPAs for Java programs assume whole classes are available at analysis time. Dynamically loaded classes should be supplied to the analysis manually. Sreedhar et.al. [28] proposed an *extant analysis framework* which performs unconditional static optimizations on references that can only have types in the closed world (known classes by analysis), and guided optimizations on references with possible dynamically loaded types. However, the effectiveness of online *extant analysis* may be compromised by the laziness of class loading at runtime. Java poses access restrictions on fields by modifiers. Field analysis [13] uses access modifiers to derive useful properties of fields for optimizations.

A new wave of VM technology is adaptive feedback-directed optimizations [4, 16, 22]. Sampling is a technique for collecting runtime information with low costs. Profiling information provides advice to compilers to allocate resources for optimizing important code areas. Compared with feedback-directed optimizations, optimizations based on dynamic IPAs can be optimistic using invalidation techniques instead of using runtime checks. Dynamic IPAs also provide a complete picture of an executing program. The new proposed mechanism is capable of finding all invoked call edges in executed code. In many cases, profiling information can be aggregated with IPAs. For example, Jikes RVM's adaptive system samples call stacks periodically and builds a weighted, partial call graph for adaptive inlining. A complete call graph constructed by our mechanism could be annotated with sampled weights on edges and clients could perform probabilistic analysis using the call graph.

6 Conclusions

In this paper we have proposed a new runtime call graph construction mechanism for dynamic IPAs in a JIT environment. Our approach uses code stubs to capture the first-time execution of a call edge. The new mechanism avoids iterative propagation which is costly at runtime. We also addressed another important problem faced by dynamic IPAs: lazy class loading. Our approach handles the problem transparently. An important characteristic of our mechanism is that it supports speculative optimizations with invalidation backups. Our preliminary results showed that the overhead of online call graph construction is very small, and the call graph is much smaller than the one built by dynamic CHA.

Based on runtime call graphs, we outlined the design of a dynamic XTA type analysis. The model of handling unresolved references is applicable to other dynamic IPAs.

Based on the encouraging results so far, we are working on combining call graph profiling and dynamic CHA to deal with boot images and JNI calls. We also plan to use the results of dynamic XTA to expose more opportunities for method inlining. We are also planning to use the runtime call graphs, and the fundamental approach already used for dynamic XTA, for developing other dynamic reachability-based IPAs, e.g. escape analysis [33].

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*, pages 108–124, 2001.

[3] L. O. Andersen. Program Analysis and Specialization for the C Programming Language, May 1994. Ph.D thesis, DIKU, University of Copenhagen.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adapative Optimization in the Jalapeño JVM. In *Proceedings the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, Oct 2000.

[5] M. Arnold, M. Hind, and B. Ryder. Online Feedback-Directed Optimization of Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 111 – 129, October 2002.

[6] 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*, pages 324 – 341, Oct 1996.

[7] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to Analysis Using BDDs. In *Proceedings of the Conference on Programming Language Design and Inplementation*, pages 103–114, June 2003.

[8] J. Bogda and A. Singh. Can a Shape Analysis Work at Run-time? In *USENIX Java Virtual Machine and Technology Symposium*, pages 13 – 26, April 2001.

[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—9th European Conference for Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Åarhus, Denmark, August 1995. Springer.

[10] M. Emami, R. Ghiya, and L. J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[11] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 85–96, June 1998.

[12] 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*, pages 241 – 252, March 2003.

[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, June 2000.

[14] N. Heintze. Analysis of Large Code Bases: The Compile-Link-Analyze Model, 1999. http://cm.bell-labs.com/cm/cs/who/nch/cla.ps.

[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] U. Hölzle. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming, 1994. Ph.D Thesis, Standford University.

[17] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the Conference on Programming Language Design and Implementations*, pages 32 – 43, July 1992.

[18] 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, October 2000.

[19] Jikes^{TM} Research Virtual Machine. http://www-124.ibm.com/developerworks/oss/jikesrvm/.

[20] O. Lhoták and L. Hendren. Scaling Java Points-to Analysis Using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[21] S. Liang and G. Bracha. Dynamic Class Loading in the Java(TM) Virtual Machine. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 36 – 44, October 1998.

[22] M. Paleczny, C. Vick, and C. Click. The Java HotSpot(TM) Server Compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, pages 1 – 12, April 2001.

[23] 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, October 2001.

[24] A. Rountev and S. Chandra. Off-line Variable Substitution for Scaling Points-to Analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 47 – 56, June 2000.

[25] A. Rountev, A. Milanova, and B. Ryder. Points-to Analysis for Java Using Annotated Constraints. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 43 – 55, October 2001.

[26] Spec JBB2000 benchmark. http://www.spec.org/jbb2000/.

[27] Spec JVM98 benchmarks. http://www.spec.org/jvm98/.

[28] V. C. Sreedhar, M. G. Burke, and J.-D. Choi. A Framework for Interprocedural Optimization in the Presence of Dynamic Class Loading. In *Proceedings of the Conference on Programming Language Design and Implementations*, pages 196 – 207, June 2000.

[29] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[30] T. Suganuma, T. Yasue, and T. Nakatani. A Region-Based Compilation Technique for a Java Just-In-Time Compiler. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 312 – 323, June 2003.

[31] 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*, pages 264–280, October 2000.

[32] 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*, pages 281–293, October 2000.

[33] 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, May 2001.

[34] Weka 3: Data Mining Software in Java. http://www.cs.waikato.ac.nz/ml/weka/.