McGill University
School of Computer Science
Sable Research Group

Towards Dynamic Interprocedural Analysis in JVMs

Sable Technical Report No. 2003-5

Feng Qian and Laurie Hendren
[fqian, hendren]@cs.mcgill.ca
Sable Research Group, School of Computer Science, McGill University

October 21, 2003

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, type 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 mechanism with low costs for constructing runtime call graphs in a JIT environment. The mechanism uses a profiling code stub to capture the fist-time execution of a call edge, and adds at most one more instruction to the 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, and we present empirical measurements of the overhead of call graph construction and the characteristics of dynamic XTA.

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-orient (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 the 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, Java has its own language-specific feature, 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 the resource usage and improve responsiveness [16]. When a JIT compiler encounters an unresolved symbolic reference, it may choose to delay the 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 nature (pessimistic) of static analysis, a dynamic one can be optimistic about future execution, if used in conjunction with runtime invalidation mechanisms [9, 13, 20, 25].

Over the last 10 years, VM technology has advanced to a new stage. JIT compilers already implement most *intra*procedural data-flow analyses that can be found in static compilers [1, 19]. Further performance improvements have been achieved using adaptive and feedback-directed compilation [3, 18].

Dynamic *inter*procedural analysis, however, has not yet been widely adopted. Some type-based IPAs [13, 20] have gained ground in JIT compilation environments. However, more complicated, reachability-based IPAs, such as points-to analysis and escape analysis, have not been successfully applied in JIT compilers.

In this paper, we present an online call graph construction mechanism for reachability-based interprocedural analyses at runtime. Instead of approximating a call graph, 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 negligible effect on the performance of applications in an adaptive 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 [20] 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 [27]. 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 IPA. In the following section, Section 3, we describe the design of a specific dynamic IPA, dynamic XTA analysis, in the presence of lazy class loading. The call graph mechanism and dynamic XTA have been implemented in Jikes RVM, and the cost of call graph construction and runtime characteristics of dynamic XTA analysis is presented in Section 4. 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.

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 back the address of the real target. The mechanism captures the first execution event of each call edge, and only the first execution has some profiling overheads. 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. In Section 2.1 we give the necessary background, describing the existing implementation of virtual method tables in Jikes RVM. In Section 2.2 we describe the basic mechanism we propose for building call graphs at runtime, and in Section 2.3 we show how this basic mechanism can be optimized to reduce overheads.

2.1 Background: virtual method table

To understand how the mechanism works, we first revisit the virtual method dispatching 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 its owner method. 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 back the address of the compiled method into the TIB entry (where the code stub resided before).

2.2 Call graph construction at runtime

In the normal lazy method compilation, the code stub captures the first invocation of a method without distinguishing callers. However, in order to capture call edges we need to store information per caller. In order to achieve this, an array of instruction addresses can be put in the TIB entry of the callee method. We call the array a Caller-Target Block (CTB). The callers of the method have different indices into the method's CTB array. Now all of the CTB entries have the address of a profiling code stub and the code stub itself is modified so that in addition to triggering the lazy compilation of the callee, it generates a new edge event. Clients monitoring new edge events get notified. Finally the code stub patches back the callee's instruction address into the entry of the CTB array.

Figure 1(c) shows the concept of extended arrays. Note that now an *invokevirtual* bytecode takes one extra load to get the target address.

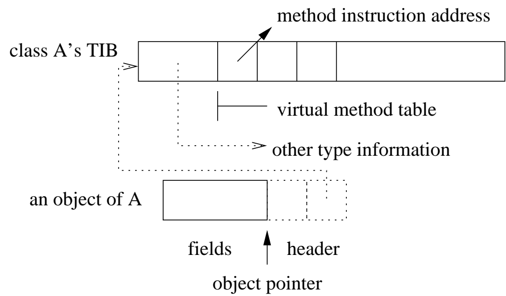
```
TIB = * (ptr + TIB_OFFSET);
/* load method's CTB array from TIB */
CTB = TIB[method_offset];
/* load method's code address */
INSTR = CTB[caller_index];
JMP INSTR
```

There remain three problems to address. First, one needs a convenient way of indexing into the TIBs which works even in the presence of lazy class loading. Second, object initializers must be handled specially. Third, we must handle the case where an adaptive system inlines one method into another. Our solution to these three problems is given below.

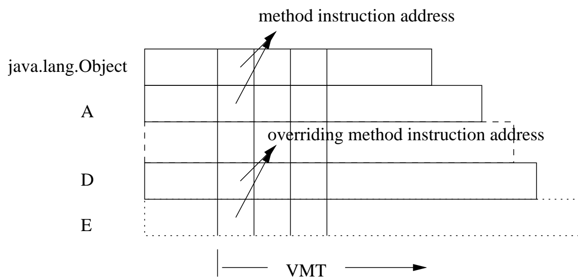
2.2.1 Allocating slots in the TIB

To index callers of a callee, our modified JIT compiler maintains a table of *(callee, caller)* pairs. In Java bytecodes, 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. Therefore, the real target may not be known at JIT compilation time.

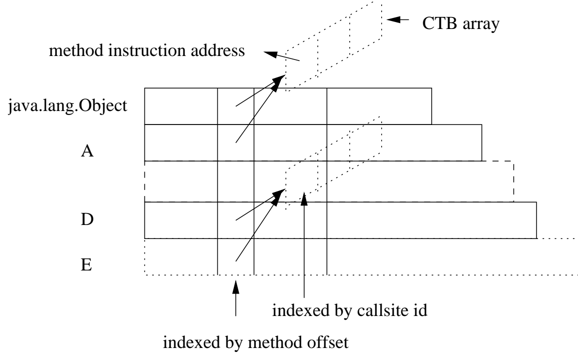
To deal with lazy class resolution and polymorphism, our approach uses the callee's method name and descriptor in the table. For example, if both methods `X.x()` and `Y.y()` have virtual calls of a symbolic reference `A.m()`, and another method `Z.z()` has a virtual call of `B.m()`, our approach assumes that all three methods are possible callers of any method with the signature `m()`, 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. This conservative solution uses some superfluous space, but does not sacrifice accuracy. Further, it 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.



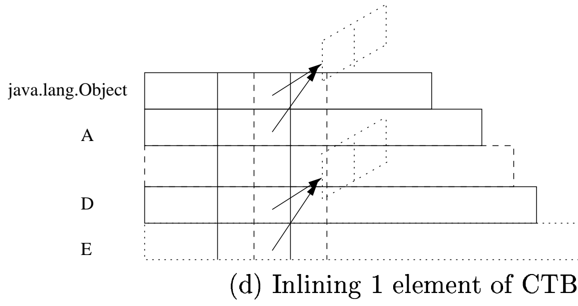
(a) TIB in Jikes RVM



(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

2.2.2 Handling object initializers

Because there are many object initializers that share a common name `<init>`, their CTB arrays may grow too large. Since calls of static methods and object initializers are monomorphic, we use a conservative approach to avoid CTB arrays and profiling code stubs. The only problem to overcome is how to handle unresolved method references (for lazy class loading). If the target method reference of an *invokespecial* instruction denotes an object initializer and the reference can be resolved at JIT compilation time, our approach conservatively assumes there is a call edge from the compiled method to the resolved initializer method. For unresolved `<init>` 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.

2.2.3 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 overheads.

2.3 Optimizations

Since Jikes RVM is written in Java, our runtime call graph construction mechanism may incur two kinds of overheads. First, adding one instruction per call can potentially consume many CPU cycles because Jikes RVM 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.

#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

Table 1 shows the distributions of the CTB sizes for the SpecJVM98 benchmarks [23] in a *FastAdaptive-SemiSpace* 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.[1] To demonstrate that most methods have few callers, we calculated the cumulative percentages of methods that have no caller, ≤ 1, ≤ 3 and ≤ 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 CTB arrays with a short length because each array header takes 3 words. The last data row labelled "`TOTAL`" gives total number of methods of all classes and numbers of methods in the two sub-categories.

To avoid the overhead of array headers for TIBs, 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

[1] We used package names to distinguish classes.

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, *CTB_INLINED_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 - CTB_INLINED_SIZE];
JMP INSTR
```

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

Tip and Palsberg [27] proposed a set of propagation-based call graph construction algorithms for Java with different granularities between RTA and 0-CFA. XTA analysis 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 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 because 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 [20, 26]. 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 elimination of type checks.

Like other static IPAs for Java, static XTA assumes 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 dynamic call graph constructed at runtime. The call graph used by dynamic XTA is significantly smaller than the one constructed by the static XTA. However, a new challenge of 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. [2]

[2] 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.

```
class A { Object f; }          A a;
                               a.f = ...;
class B extends A {
}                              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 [16]. To port a static IPA for Java to a dynamic IPA, the analysis must be modified to handle unresolved references. In this section, we demonstrate a solution for the problem for dynamic XTA, which is also applicable to general IPAs for Java.

Our Dynamic XTA analysis constructs a 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 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 together. 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 proper edges between methods and fields, with 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 not in a successor's reachable type set, it is recursively propagated to all descendants until no changes.

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

Dynamic XTA analysis is driven by events from JIT compilers and class loaders. Figure 3 shows event flows. 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 bytecode. When seeing a *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 is 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 change occurs.

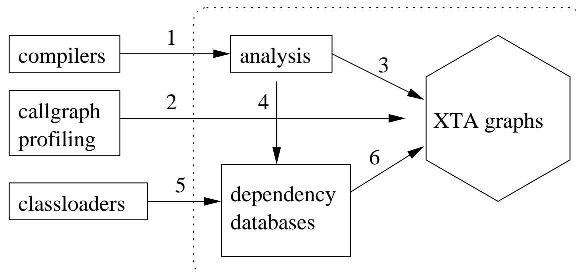


Figure 3: Models of XTA events

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 [21] and online cycle elimination [8] 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 [11,15] and BDDs [5], 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 [20], 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 [9,12,13].

4 Evaluation

We have implemented proposed call graph construction mechanism in Jikes RVM [14] v2.2.1. The SpecJVM98 suite [23] was chosen as our benchmark set. In our experiments, classes in the boot image are not represented in the dynamic call graphs because (1) the number of RVM classes is much larger than the number of classes of applications and libraries, (2) the classes in the boot image were statically compiled and optimized. Static IPAs such as extant analysis [30] may be applied on the boot image classes. We report the experimental results for application classes and Java library classes.

We have measured space overhead for CTB arrays at runtime with inlined size 1, 2, 4, and 8. For our experiment the virtual machine was started once without restarting between different benchmarks. The benchmarks are run in the order of `_201_compress`, `_202_jess` `_205_raytrace`, `_209_db`, `_213_javac`, `_222_jess`, `_227_mtrt`, and `_228_jack`, and each benchmark runs 10 times with input size 100. This concatenation of all benchmarks simulates a long running application.

Inlined CTB sizes	bootimage size (bytes)		CTB space (bytes)	#methods with non-null CTB	
default	34,284,100		N/A	N/A	
1	37,741,640	10.08%	783,392	3,108	53.32%
2	37,977,940	10.77%	754,756	2,003	34.36%
4	38,227,708	11.50%	725,104	1,223	20.98%
8	38,721,688	12.94%	691,992	769	13.19%

Table 2: Memory overhead

The first column of Table 2 gives four configurations of different inlined CTB sizes and the default 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 and online XTA analysis. Inlining CTB elements increases the size of TIBs. However, changes are relatively small (the difference between inlined CTB sizes 1 and 2 is about 236 kilo bytes), 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. The last column gives the number of methods with non-null CTB arrays and its percentage in total methods of loaded library and benchmark classes. As more CTB elements get inlined, fewer methods need CTB arrays. The time for creating, expanding and updating CTB array are negligible.

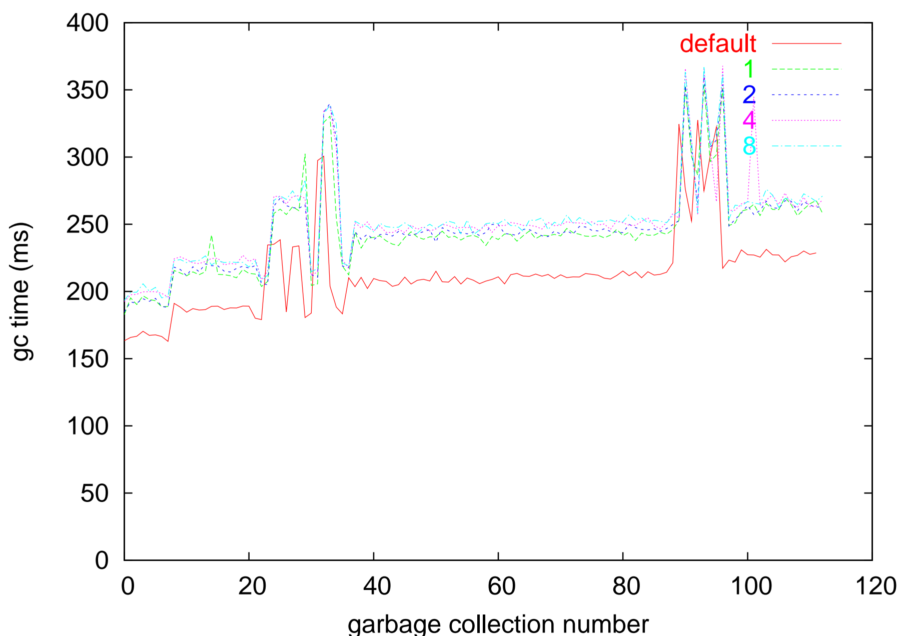


Figure 4: GC time when running SpecJVM98

Now we look at the performance impact after introducing CTB arrays. Since CTB arrays are likely living for a long time, garbage collection is likely to be directly affected. Using the same experimental setting mentioned before, GC time was profiled and plotted in Figure 4 for the default system and the 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 collection is slightly slower than the default system, but not significantly.

The call graph building code stub itself has very little performance overhead. It is only executed when a call edge is invoked for the first time. Some invocation bytecodes may have one more extra load instruction,

if their CTB entry was not inlined, as described in Section 2.3. For benchmarks in the SpecJVM98 suite, we found the performance changes are negligible. Further, ror benchmarks from the SpecJVM98 suite, it seems that inlining more CTB array elements does not result in performance improvement.

An interesting side note concerns the impact of adding new analyses to baseline compiled code when using the Jikes RVM adaptive compilation decision system. Our first experiments showed an unexpected slowdown when we enabled the dynamic XTA. After detailed investigations we discovered that this was due to perturbing the adaptive decision system, as follows. An IPA client may register callbacks when a new edge is discovered. Dynamic XTA callback adds new edges into the graph and may perform type propagation if necessary. The callback may incur overheads during the initial phase of applications. This has an implicit impact on Jikes RVM's adaptive recompilation decision system [3] because the baseline compiled code is slower than the adaptive system expects. This results in fewer methods being promoted to higher optimization levels, and thus slower code. To compensate for the effect of the additional work in code stubs, we re-trained the adaptive compilation system to optimize about the same number of methods as the default system. We found that usually the first run has small performance degradation (up to 7.62%) when performing dynamic XTA, but it is not significant when reaching a stable state.

A dynamic XTA only analyzes compiled methods, and as a result the XTA graph is very small. Table 3 shows the size of final XTA graphs. The second column gives the number of nodes including methods and fields, and the number of method nodes is given in braces. The third column shows the number of graph edges, and the number of call edges is in braces. Tip and Palsberg [27] reported that the call graph of `javac` benchmark constructed by static XTA analysis (not including Java libraries) has 1,366 methods and 13,113 call edges. The dynamic XTA graph has 1,115 methods and 4,500 call edges for both Java libraries and benchmarks. We can see that the size of graph is reasonably small, which allows efficient type propagation at runtime.

benchmark	#nodes		#edges	
compress	404	(315)	802	(577)
jess	932	(778)	2347	(1826)
db	439	(349)	963	(699)
javac	1396	(1115)	6140	(4500)
mpegaudio	638	(484)	1406	(893)
mtrt	568	(454)	1469	(1141)
jack	729	(574)	1925	(1433)

Table 3: Characteristics of dynamic XTA graphs

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) [7] treats all subclasses of a receiver's declaring class as possible types at runtime. Rapid type analysis (RTA) [4] prunes the type set of CHA by eliminating types that do not have an allocation site in 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 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 [24] or subset-based [2] 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 [17] 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 as fast algorithms when comparing with propagation-based algorithms. Both VTA [26] and XTA analysis [27] 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 [22].

Ishizaki et.al. [13] published a new method of utilizing class hierarchy analysis for devirtulization at runtime. If the target of a virtual call is not overridden in the current class hierarchy, the compiler may choose to inline the target directly with a backup code of normal virtual 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. Igor and Sarkar [20] 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 [6] 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.

Code-patching [13] and stack-rewriting [9, 28] 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. [30] 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 [10] uses access modifiers to derive useful properties of fields for optimizations.

A new wave of VM technology is adaptive feedback-directed optimizations [3, 12, 19]. Sampling is a technique for collecting runtime information with cheap cost. 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, a runtime call graph could be annotated with weights of samples on edges and clients could perform analysis using probabilities.

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 negligible.

Based on runtime call graphs, we outlined the design of a dynamic XTA type analysis and we demonstrated that the size of graphs built by dynamic XTA were quite small. Further, the model of handling unresolved references is applicable to other dynamic IPAs.

Based on the encouraging results so far, we are working on extending the current implementation of dynamic XTA 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 [29].

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In W. G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Åarhus, Denmark, August 1995. Springer.

[8] 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.

[9] 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.

[10] 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.

[11] N. Heintze. Analysis of large code bases: The compile-link-analyze model, 1999. `http://cm.bell-labs.com/cm/cs/who/nch/cla.ps`.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] Maryam Emami and Rakesh Ghiya and Laurie 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.

[18] Mattew Arnold and Michael Hind and Babara 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] Spec JVM98 benchmarks. `http://www.spec.org/osg/jvm98/index.html`.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] Urs Hölzle and Craig Chambers and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the Conference on Programming Language Design and Implementations*, pages 32 – 43, July 1992.

[29] 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.

[30] Vugranam C. Sreedhar and Michael G. Burke and Jong-Deok 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.