Towards Dynamic Interprocedural Analysis in JVMs

Feng Qian and Laurie Hendren

{fqian,hendren}@cs.mcgill.ca.

School of Computer Science, McGill University

`http://www.sable.mcgill.ca`

Motivation

Goal:

- do *interprocedural* analysis supporting *speculative* optimizations in a JIT compiler

Motivation

Goal:

- do *interprocedural* analysis supporting *speculative* optimizations in a JIT compiler

Problems:

- construct a high quality call graph efficiently
- deal with dynamic class loading
- handle unresolved symbolic references
- . . . . . .

Dynamic call graphs

A call graph is a representation of call relations between methods.

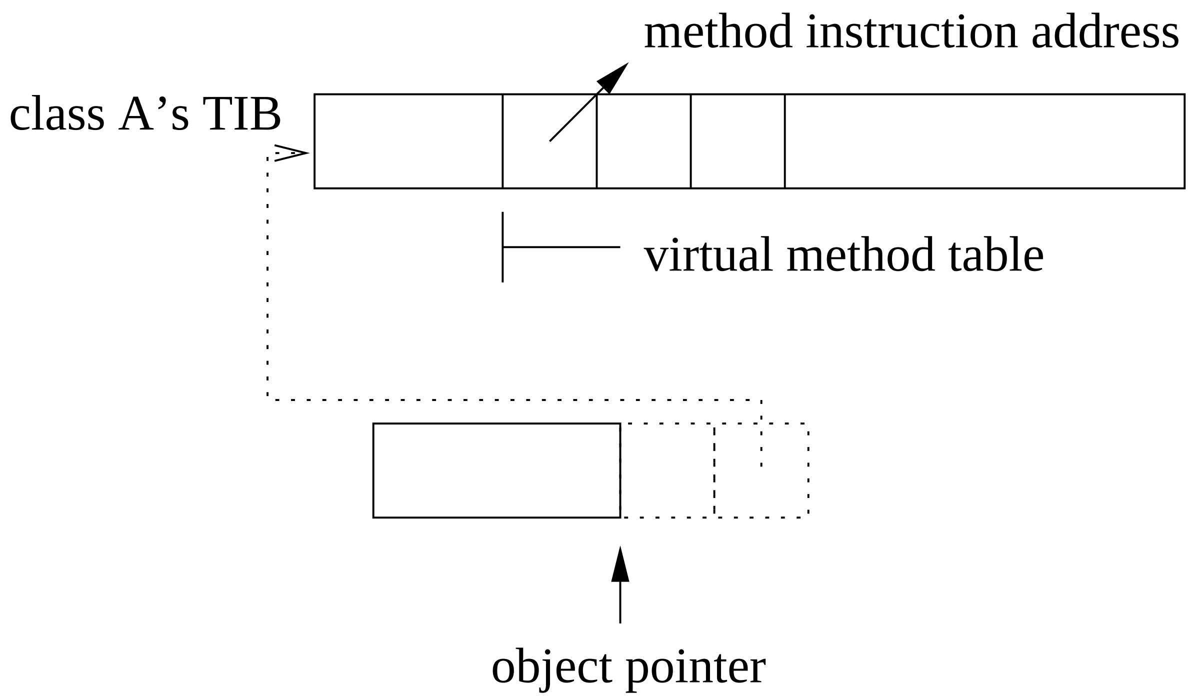
A dynamic call graph

- is constructed incrementally
- is conservative w.r.t. executed code
- supports speculative optimizations

Road map

- Motivation
- Constructing call graphs using profiling stubs
- Constructing call graphs using type analysis
- Evaluation
- Dynamic XTA
- Related work and conclusion

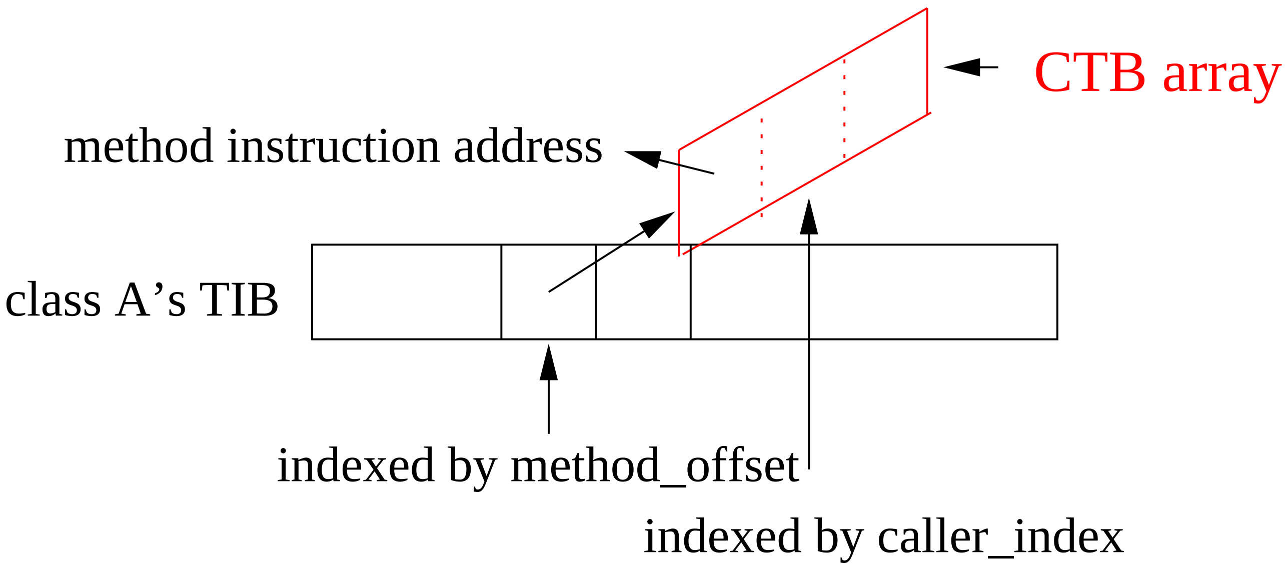
Background: virtual method calls in JikesRVM




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

*The *method_offset* is a runtime constant.

Incorporating call graph profiling stubs




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

*The *caller_index* is a runtime constant.

CTB arrays

- an entry of a CTB array
  - is initialized to the address of a profi ling code stub
  - contains real method code address after executing the code stub
- *caller_index* assignment
  - handles polymorphism and symbolic references properly
  - may waste some space in CTBs

Call graph profiling code stubs

A call graph profiling code stub

- generates a call edge event
- triggers the compilation of the method if necessary
- patches the instruction address into the CTB entry

Note: a call edge only triggers the profiling stub once (at its first invocation).

Optimizations

- Majority of methods have a small number of callers
- Inlining first few CTB elements into TIBs eliminates the extra load

	2	4	8
compress	97.26%	99.99%	99.99%
javac	21.62%	64.25%	83.53%
jack	48.51%	77.82%	86.01%

- Type analysis can be used for non-virtual and interface calls
- Runtime overhead ranges from -2% to 3% for our set of benchmarks

Road map

- Motivation
- Constructing call graphs using profiling stubs
- Constructing call graphs using type analysis
  - Evaluation
  - Dynamic XTA
  - Related work and conclusion

Dynamic type analysis

During the execution of a program *P*, define

initialized_types(P) the set of initialized classes
(built by class loaders)

rapid_types(P) the set of initialized classes having
allocation sites
(built by JIT compilers)

instantiated_types(P) the set of classes having instances
(built by allocators)

Sets are dynamically expanded as program runs.

Dynamic CHA, RTA, and ITA

Let *hierarchy_types(C)* be the set of types including *C* and its subclasses.

When compiling a resolved call *C.m()*, the following type set is used for computing call targets:

Class hierarchy analysis :

hierarchy_types(C) $\cap$ *initialized_types(P)*

Rapid type analysis :

hierarchy_types(C) $\cap$ *rapid_types(P)*

Instantiation-based type analysis :

hierarchy_types(C) $\cap$ *instantiated_types(P)*

Handle dynamic expansion of type sets

Maintain a database of *RESOLVED_CALLSITES*

resolved (callee) method $\Rightarrow$ { call sites }

Let C be a new member of the type set,

```
for each virtual method m of C
  for each m′ overridden by m
    for each resolved call site s calling m′
      generate a call edge from s to m
```

A similar approach is used to handle unresolved method calls.

Road map

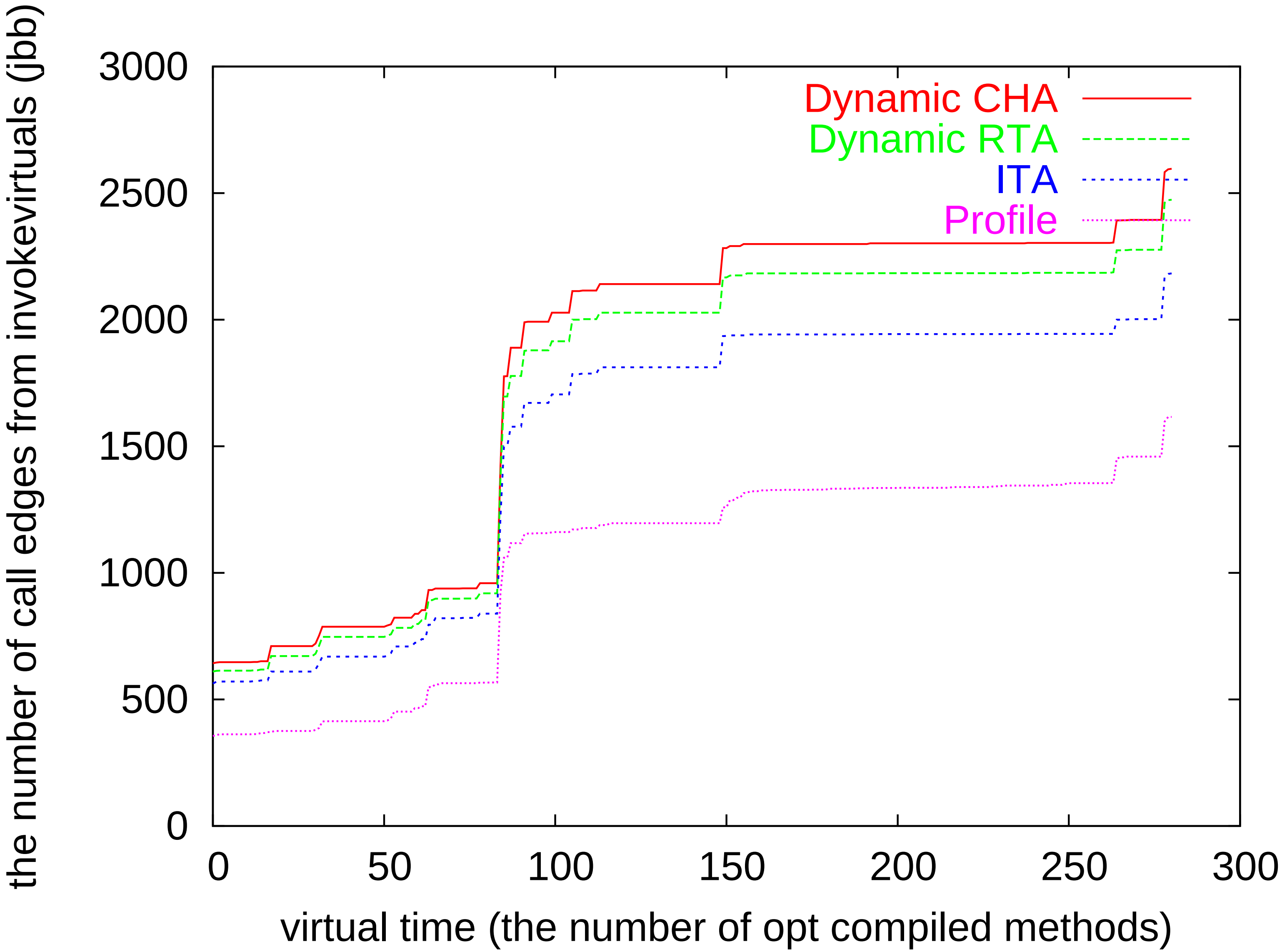
- Motivation
- Constructing call graphs using profiling stubs
- Constructing call graphs using type analysis
- Evaluation
- ◦ Dynamic XTA
- ◦ Related work and conclusion

Call graph sizes

benchmarks	CHA	RTA	ITA	Prof
compress	458	432 (94%)	380 (83%)	240 (52%)
javac	8141	7706 (95%)	6376 (78%)	2775 (34%)
jack	1131	1062 (94%)	997 (88%)	785 (69%)
jbb	2802	2663 (95%)	2379 (85%)	1734 (64%)

Table 0: the number of call edges from *invokevirtual* calls at the end of benchmark runs.

Call graph sizes at runtime (jbb)



Road map

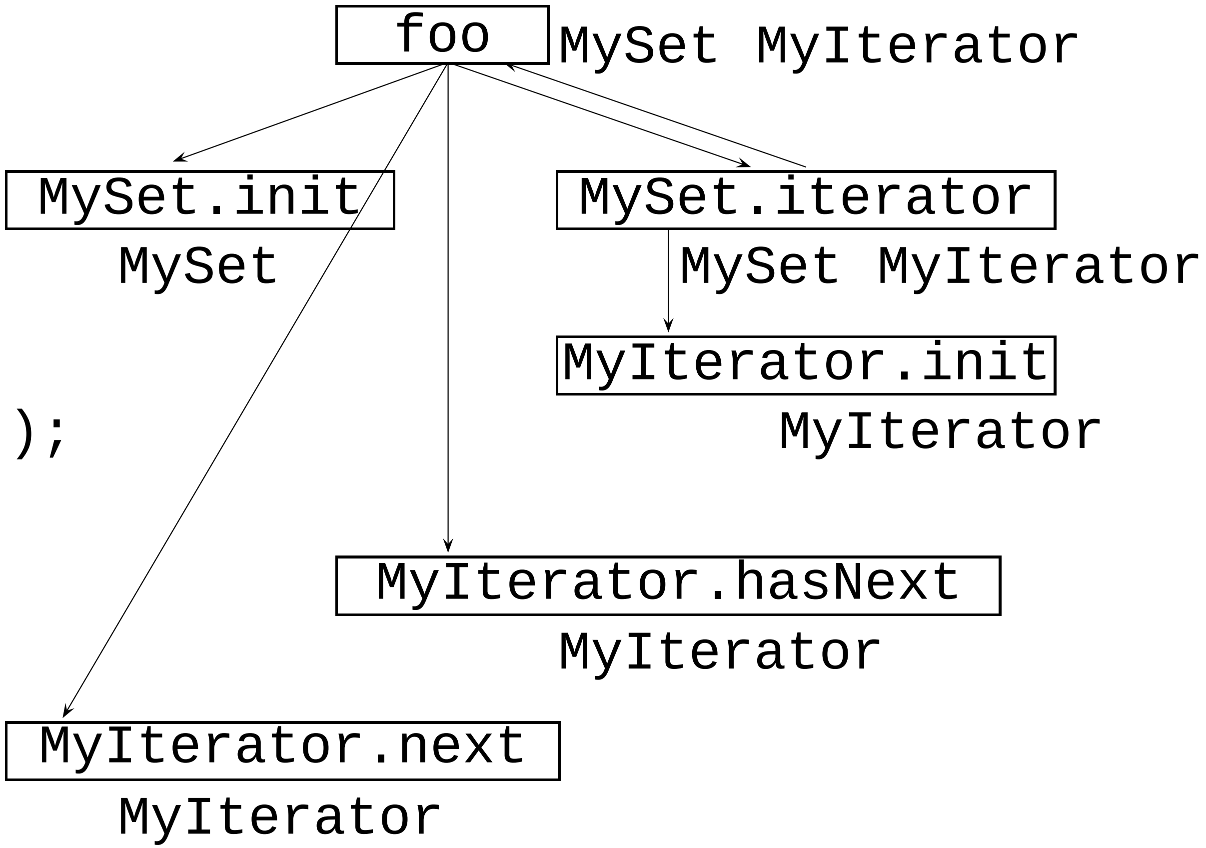
- Motivation
- Constructing call graphs using profiling stubs
- Constructing call graphs using type analysis
- Evaluation
- Dynamic XTA
- ◦ Related work and conclusion

Static XTA (Tip & Palsberg 2000)

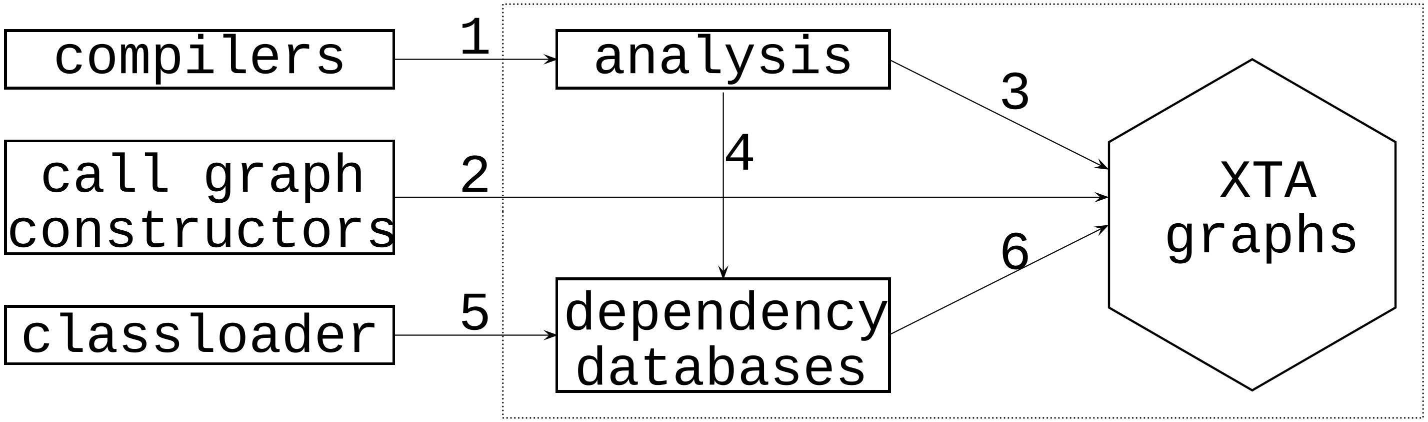
- models method calls, field and array accesses
- ignores intraprocedural data-flows

```
foo():
  Set s = new MySet();
  it = s.iterator();
  it.hasNext()
  it.next()

MySet.iterator():
  return new MyIterator();
```



Dynamic XTA



- the dynamic XTA is event-driven
- unresolved field/array references are handled by dependency databases
- results are optimistic

Related Work

- Static call graph construction algorithms (CHA, RTA, VTA, etc.)
- Dynamic optimistic interprocedural analysis (DOIT by Perchtchanski & Sarkar OOPSLA 2001)
- Pointer analysis in the presence of dynamic class loading (Hirzel et.al. ECOOP 2004)
- Online shape analysis (Bogda et. al. JVM 2001)

Conclusion

- Proposed a new, inexpensive, call graph profiling mechanism
- Studied several dynamic type analysis for call graph construction
- Presented a model of dynamic interprocedural analysis
- Working on more advanced interprocedural analysis
- How to use the analysis results? and what kind of speculative optimizations can we do?

Questions

?