Inlining Java Native Calls at Runtime

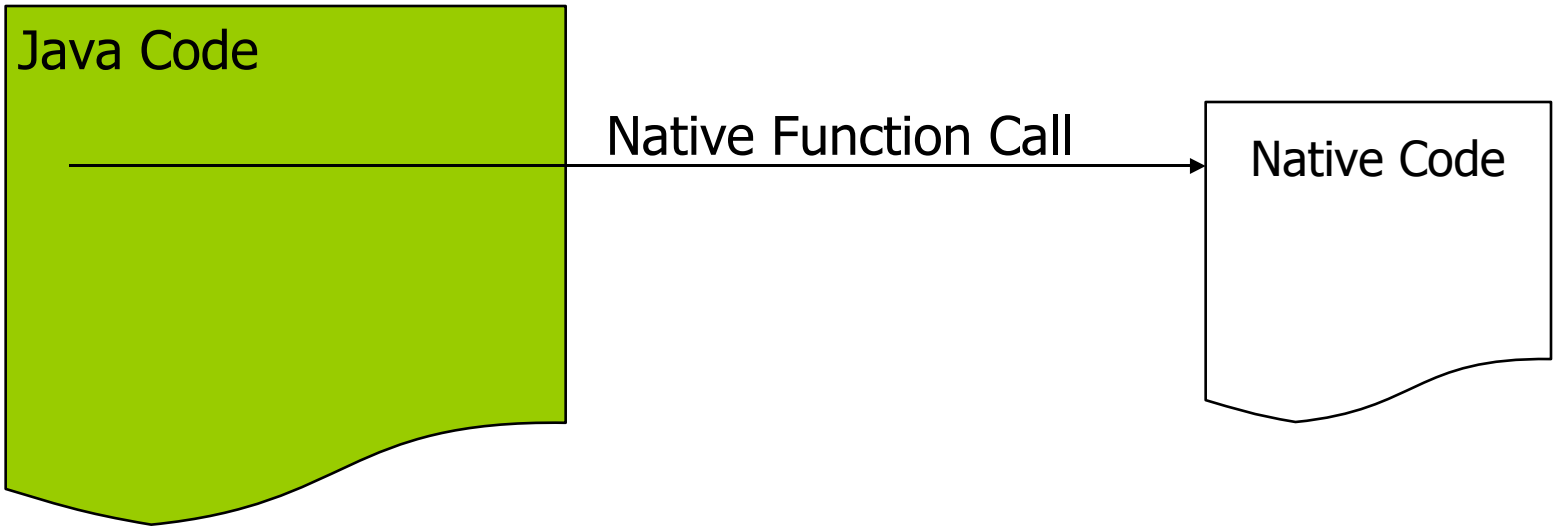
(CASCON 2005 – 4th Workshop on Compiler Driven Performance)

Levon Stepanian, Angela Demke Brown
Computer Systems Group
Department of Computer Science, University of Toronto

Allan Kielstra, Gita Koblents, Kevin Stoodley
IBM Toronto Software Lab

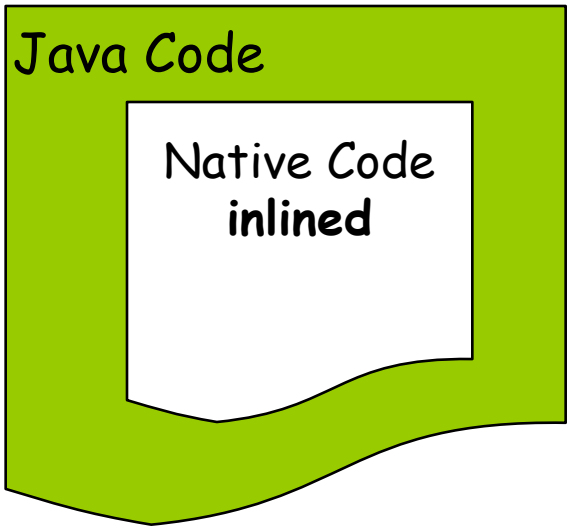
In a nutshell

- Runtime native function inlining into Java
  - Optimizing transformations on inlined JNI calls
  - Opaque and binary-compatible while boosting performance



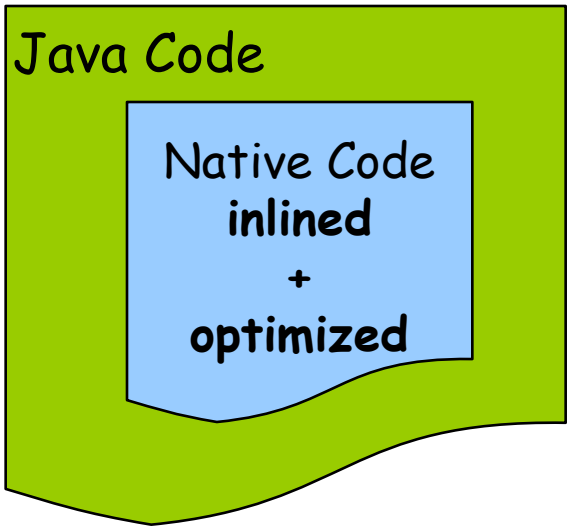
In a nutshell

- Runtime native function inlining into Java
  - Optimizing transformations on inlined JNI calls
  - Opaque and binary-compatible while boosting performance



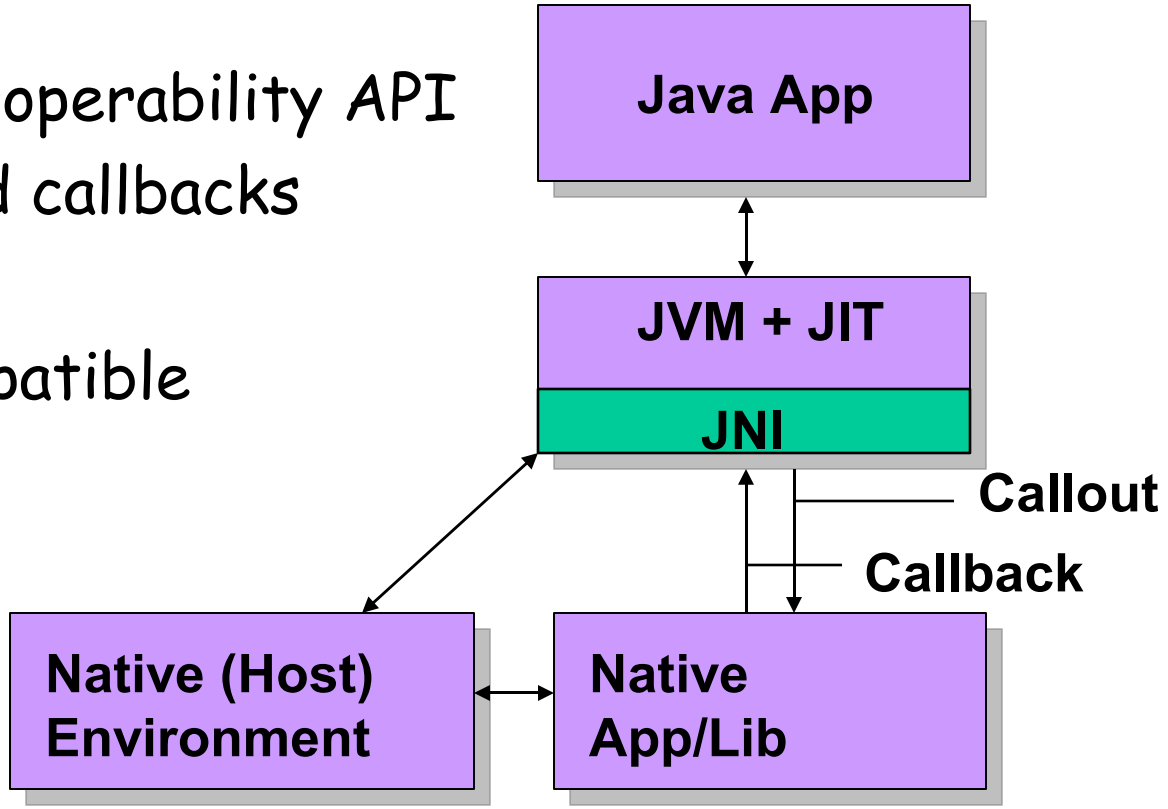
In a nutshell

- Runtime native function inlining into Java
  - Optimizing transformations on inlined JNI calls
  - Opaque and binary-compatible while boosting performance



Motivation

- The JNI
  - Java's interoperability API
  - Callouts and callbacks
  - Opaque
  - Binary-compatible



Motivation

- The JNI
  - Pervasive
    - Legacy codes
    - Performance-critical, architecture-dependent
    - Features unavailable in Java (files, sockets etc.)

Motivation

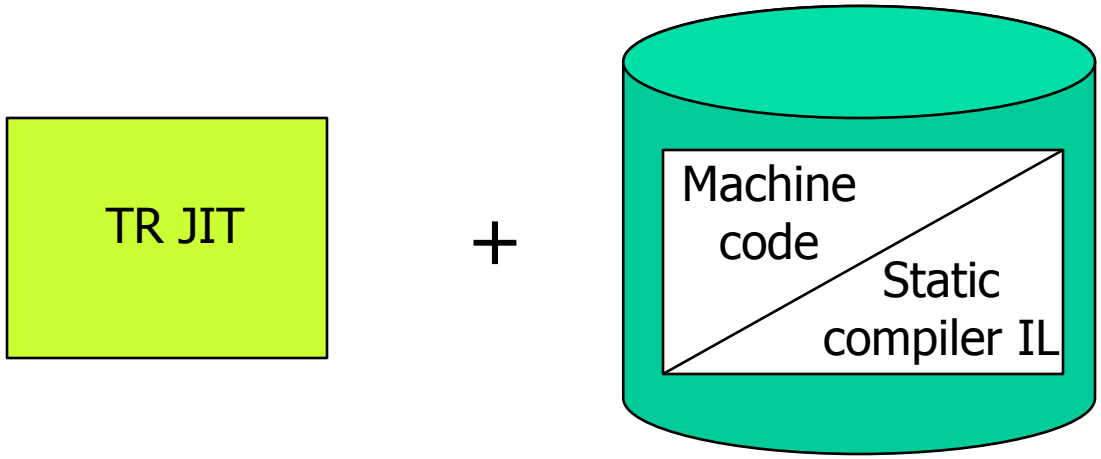
- Callouts run to 2 to 3x slower than Java calls
- Callback overheads are an order of magnitude larger
  - JVM handshaking requirements for threads **leaving** and **re-entering** JVM context
  - i.e. stack switching, reference collection, exception handling
- JIT compiler can't predict side-effects of native function call

Our Solution

- JIT compiler based optimization that inlines native code into Java
- JIT compiler transforms inlined JNI function calls to constants, cheaper operations
- Inlined code exposed to JIT compiler optimizations

Infrastructure

- IBM TR JIT Compiler + IBM J9 VM
- Native IL to JIT IL conversion mechanism
  - Exploit Native IL stored in native libraries
  - W-Code to TR-IL at runtime



Outline

- Background Information
- Method
- Results
- Future Work

Sample Java Class

```
class SetFieldXToFive{

    public int x;
    public native foo();

    static{
        System.loadLibrary(…);
    }
}
```

Sample Java Class

```
class SetFieldXToFive{

    public int x;
    public native foo();

    static{
        System.loadLibrary(…);
    }
}
```

Sample Native Code

GOAL: obj.x = 5

```
JNIEXPORT void JNICALL Java_SetFieldXToFive_foo
  (JNIEnv * env, jobject obj){

   jclass cls = (*env)->GetObjectClass(env,obj);
   jfieldID fid =
      (*env)->GetFieldID(env,cls,"x","I");
   if (fid == NULL)
      return;

   (*env)->SetIntField(env,obj,fid,5);
}
```

Sample Native Code

GOAL: obj.x = 5

```
JNIEXPORT void JNICALL Java_SetFieldXToFive_foo
  (JNIEnv * env, jobject obj){

   jclass cls = (*env)->GetObjectClass(env,obj);
   jfieldID fid =
      (*env)->GetFieldID(env,cls,"x","I");
   if (fid == NULL)
      return;

   (*env)->SetIntField(env,obj,fid,5);
}
```

SetFieldXToFive

Sample Native Code

GOAL: obj.x = 5

```
JNIEXPORT void JNICALL Java_SetFieldXToFive_foo
  (JNIEnv * env, jobject obj){

    jclass cls = (*env)->GetObjectClass(env,obj);
    jfieldID fid =
       (*env)->GetFieldID(env,cls,"x","I");
    if (fid == NULL)
       return;

    (*env)->SetIntField(env,obj,fid,5);
}
```

Sample Native Code

GOAL: obj.x = 5

```
JNIEXPORT void JNICALL Java_SetFieldXToFive_foo
  (JNIEnv * env, jobject obj){

   jclass cls = (*env)->GetObjectClass(env,obj);
   jfieldID fid =
      (*env)->GetFieldID(env,cls,"x","I");
   if (fid == NULL)
      return;

   (*env)->SetIntField(env,obj,fid,5);
}
```

Sample Native Code

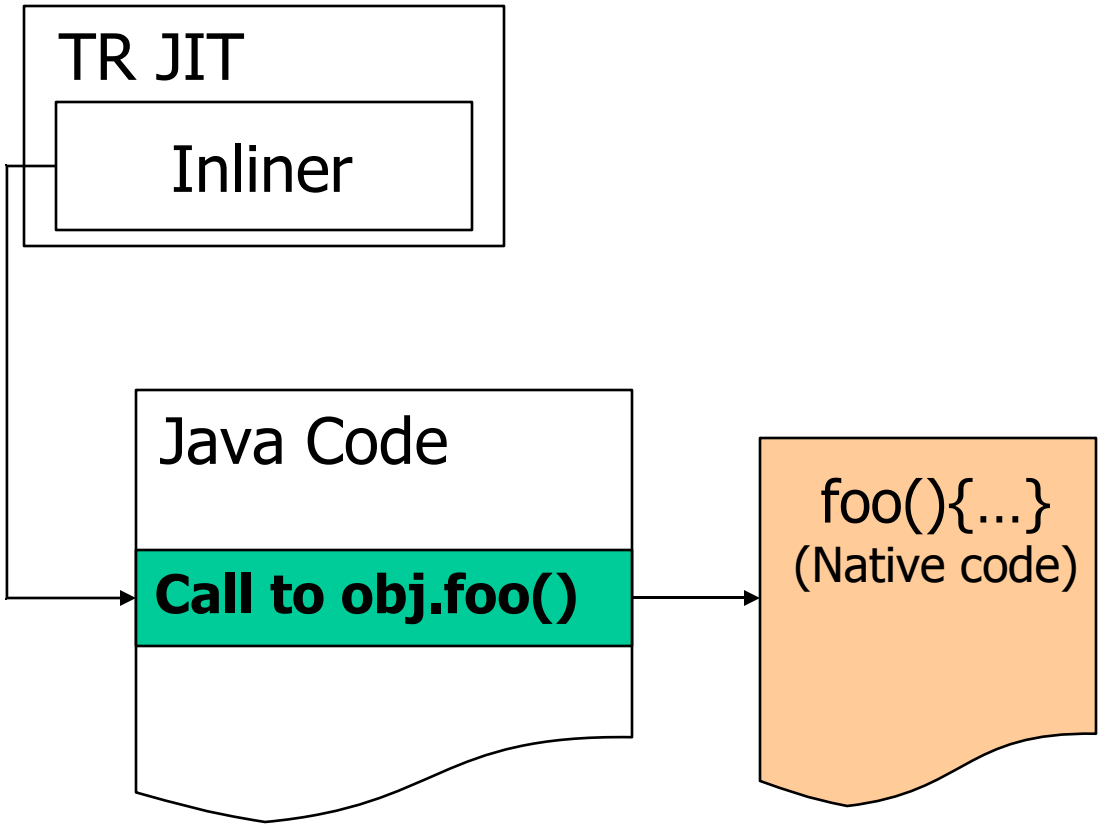
GOAL: obj.x = 5

```
JNIEXPORT void JNICALL Java_SetFieldXToFive_foo
  (JNIEnv * env, jobject obj){

   jclass cls = (*env)->GetObjectClass(env,obj);
   jfieldID fid =
      (*env)->GetFieldID(env,cls,"x","I");
   if (fid == NULL)
      return;

   (*env)->SetIntField(env,obj,fid,5);
}
```

Native Inlining Overview

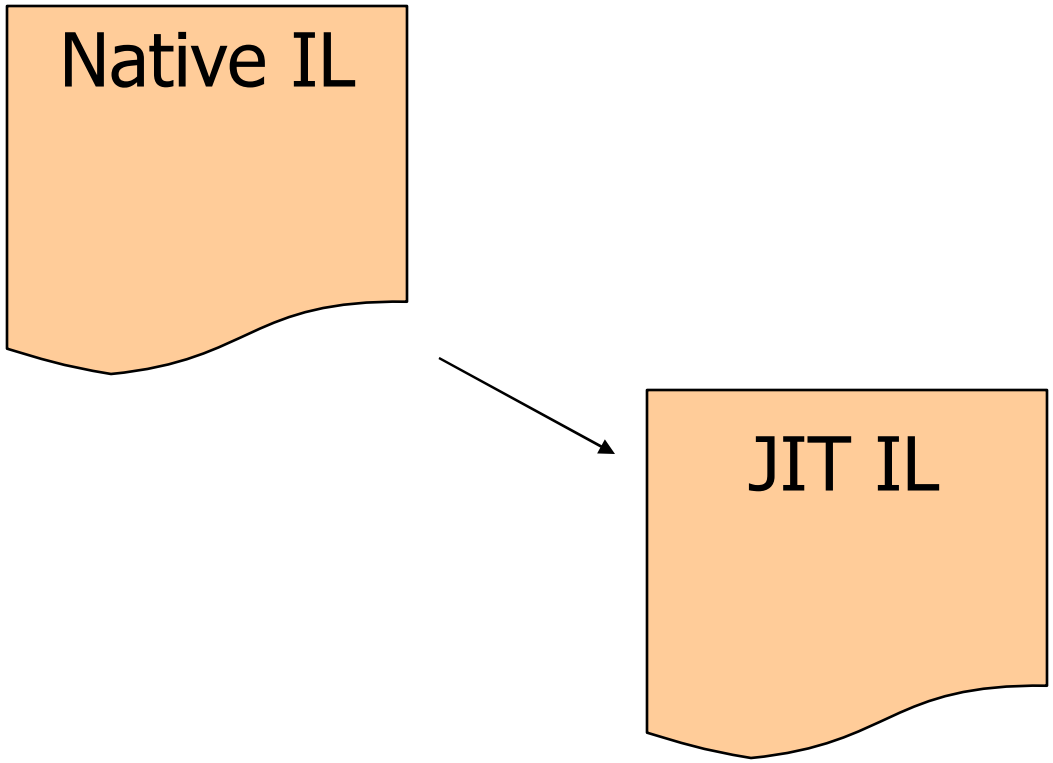
1. Inliner detects a native callsite
2. Extracts and converts Native IL to JIT IL
3. Identifies inlined JNI calls
4. Transforms inlined JNI calls
5. Finishes inlining

Method - Step 1



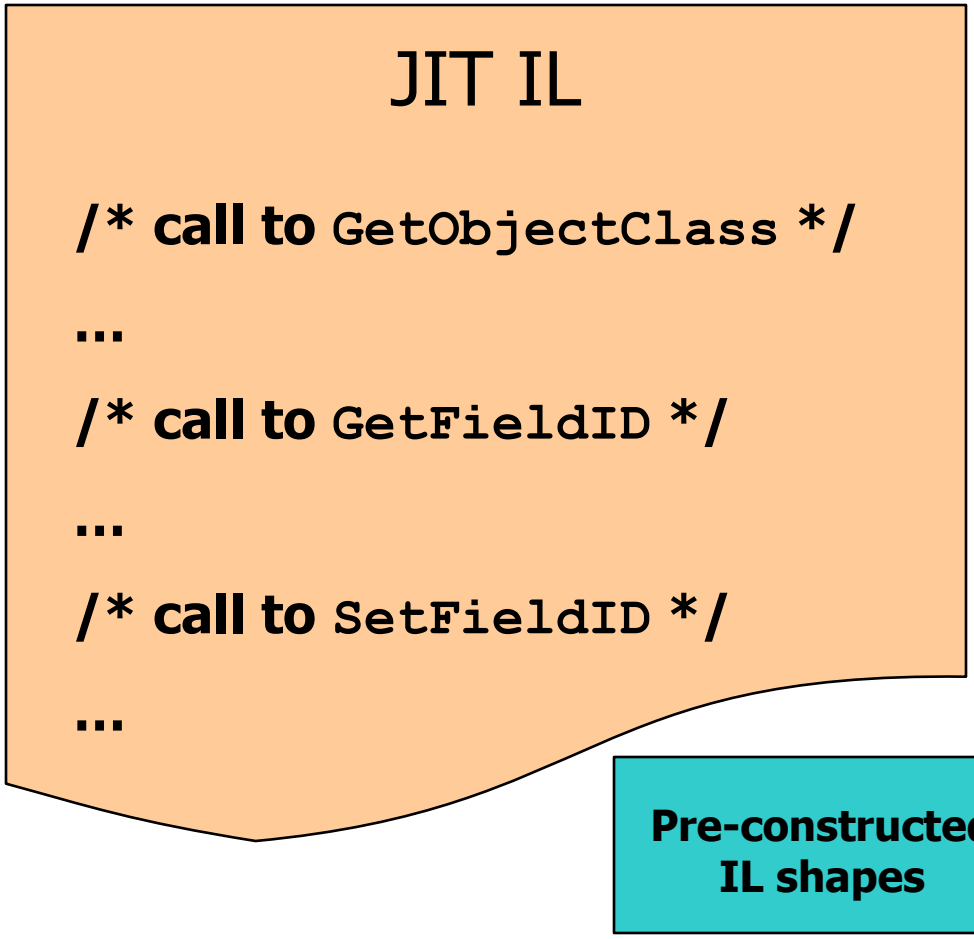
1. Inliner detects a native callsite

Method – Step 2



1. Inliner detects a native callsite
2. Extracts and converts Native IL to JIT IL

Method – Step 3



1. Inliner detects a native callsite
2. Extracts and converts Native IL to JIT IL
3. Identifies inlined JNI calls ←

Method – Step 4

```
jclass cls =
   (*env)->GetObjectClass(env,obj);

jfieldID fid =
   (*env)->GetFieldID(env,cls,"x","I");
if (fid == NULL)
   return;

(*env)->SetIntField(env,obj,fid,5);
```

1. Inliner detects a native callsite
2. Extracts and converts Native IL to JIT IL
3. Identifies inlined JNI calls
4. Transforms inlined JNI calls

Method – Step 4

Constant: SetFieldXToFive class data structure

```
jfieldID fid =
  (*env)->GetFieldID(env,cls,"x","I");
if (fid == NULL)
  return;
```

```
(*env)->SetIntField(env,obj,fid,5);
```

1. Inliner detects a native callsite
2. Extracts and converts Native IL to JIT IL
3. Identifies inlined JNI calls
4. Transforms inlined JNI calls

Method – Step 4

Constant: SetFieldXToFive class data structure

Constant: Offset of field "x"

(*env)->SetIntField(env,obj,fid,5);

1. Inliner detects a native callsite
2. Extracts and converts Native IL to JIT IL
3. Identifies inlined JNI calls
4. Transforms inlined JNI calls

Method – Step 4

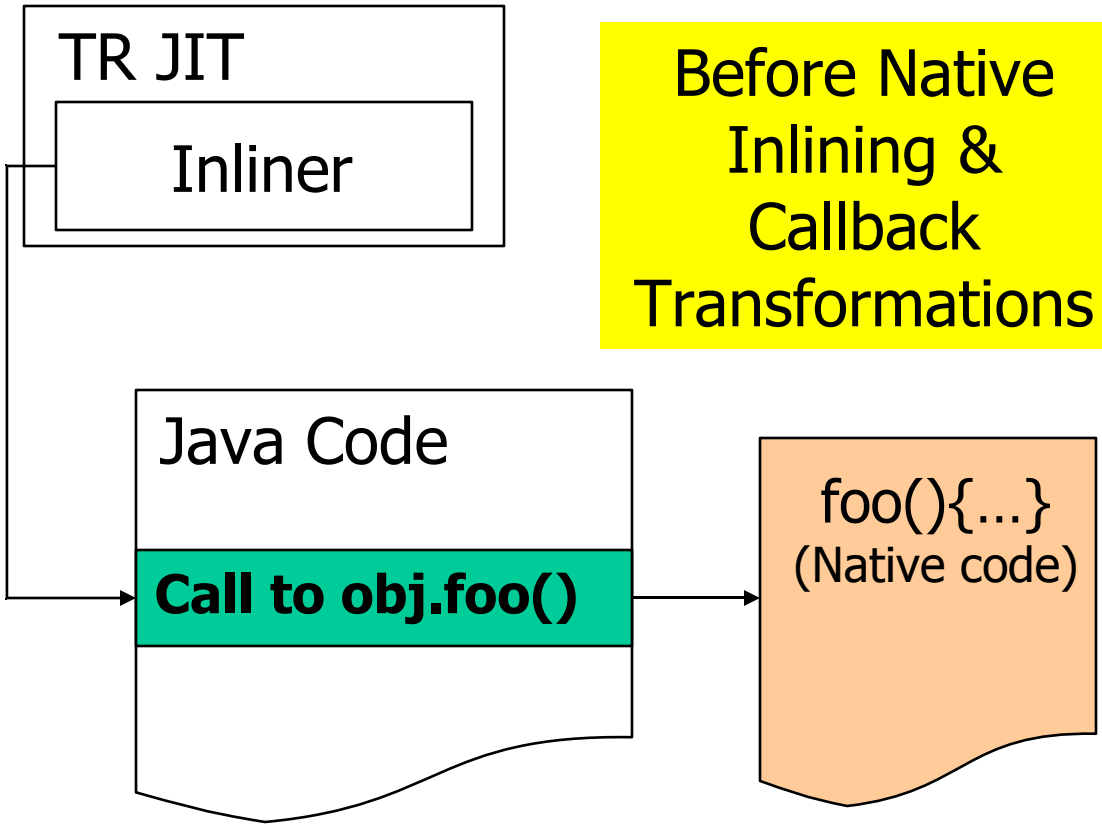
Constant: SetFieldXToFive class data structure

Constant: Offset of field "x"

JIT IL: obj.x = 5

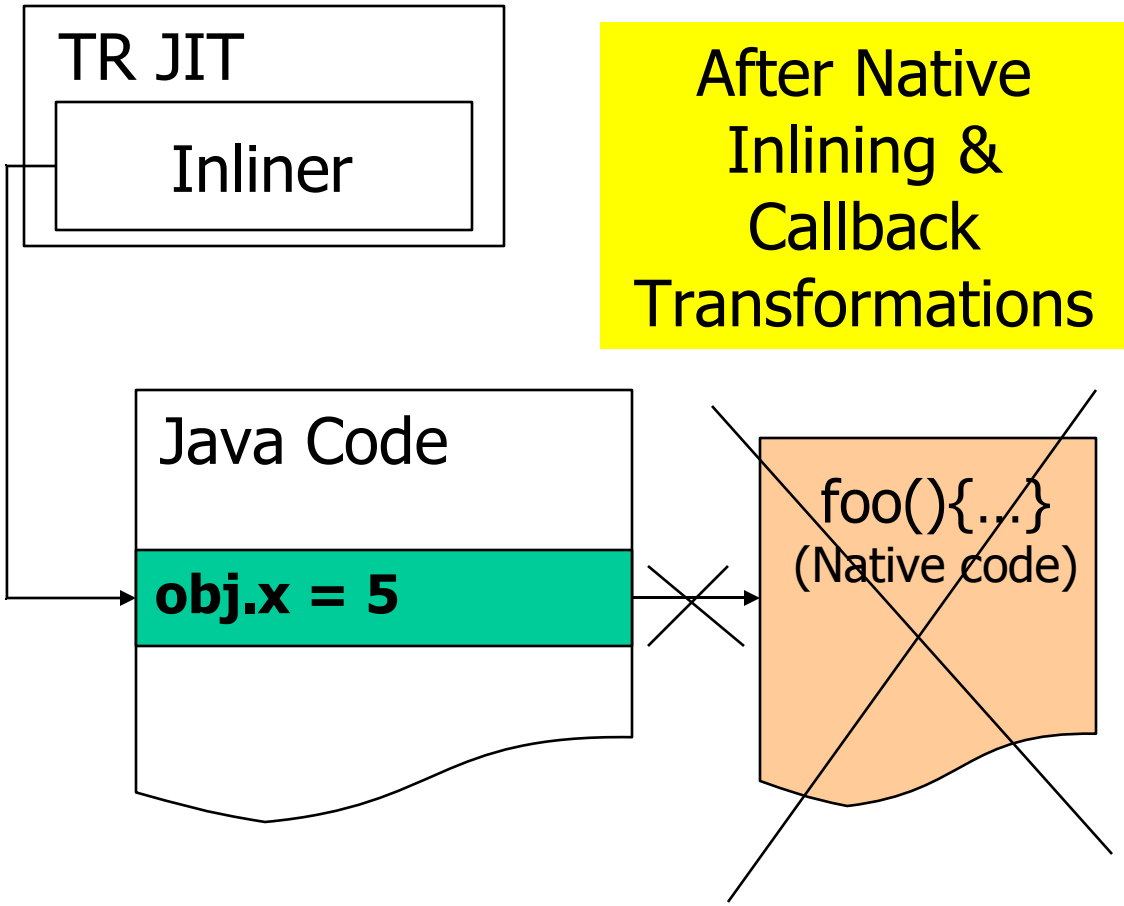
1. Inliner detects a native callsite
2. Extracts and converts Native IL to JIT IL
3. Identifies inlined JNI calls
4. Transforms inlined JNI calls

The Big Picture



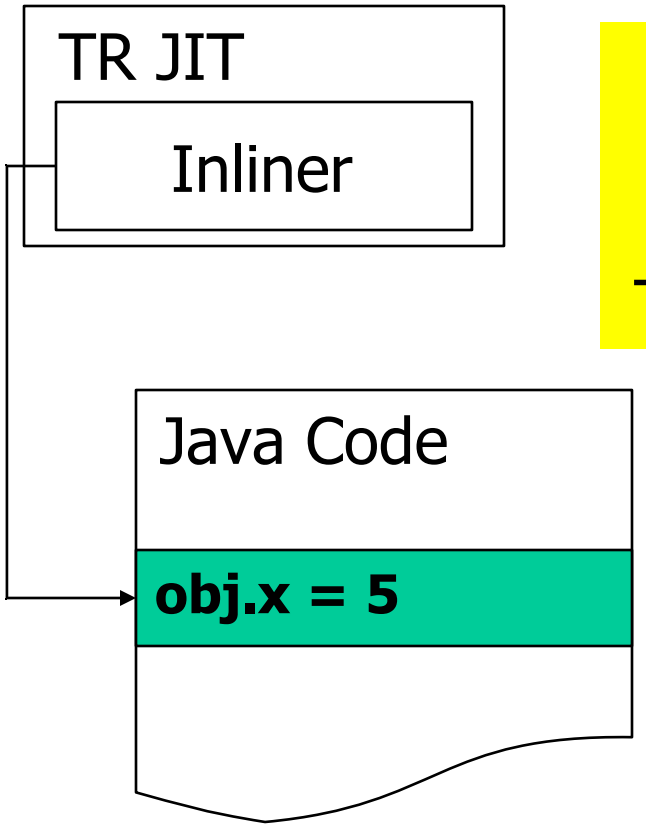
1. Inliner detects a native callsite
2. Extracts and converts Native IL to JIT IL
3. Identifies inlined JNI calls
4. Transforms inlined JNI calls
5. Finishes inlining

The Big Picture



1. Inliner detects a native callsite
2. Extracts and converts Native IL to JIT IL
3. Identifies inlined JNI calls
4. Transforms inlined JNI calls
5. Finishes inlining

The Big Picture



After Native Inlining & Callback Transformations

1. Inliner detects a native callsite
2. Extracts and converts Native IL to JIT IL
3. Identifies inlined JNI calls
4. Transforms inlined JNI calls
5. Finishes inlining

Outline

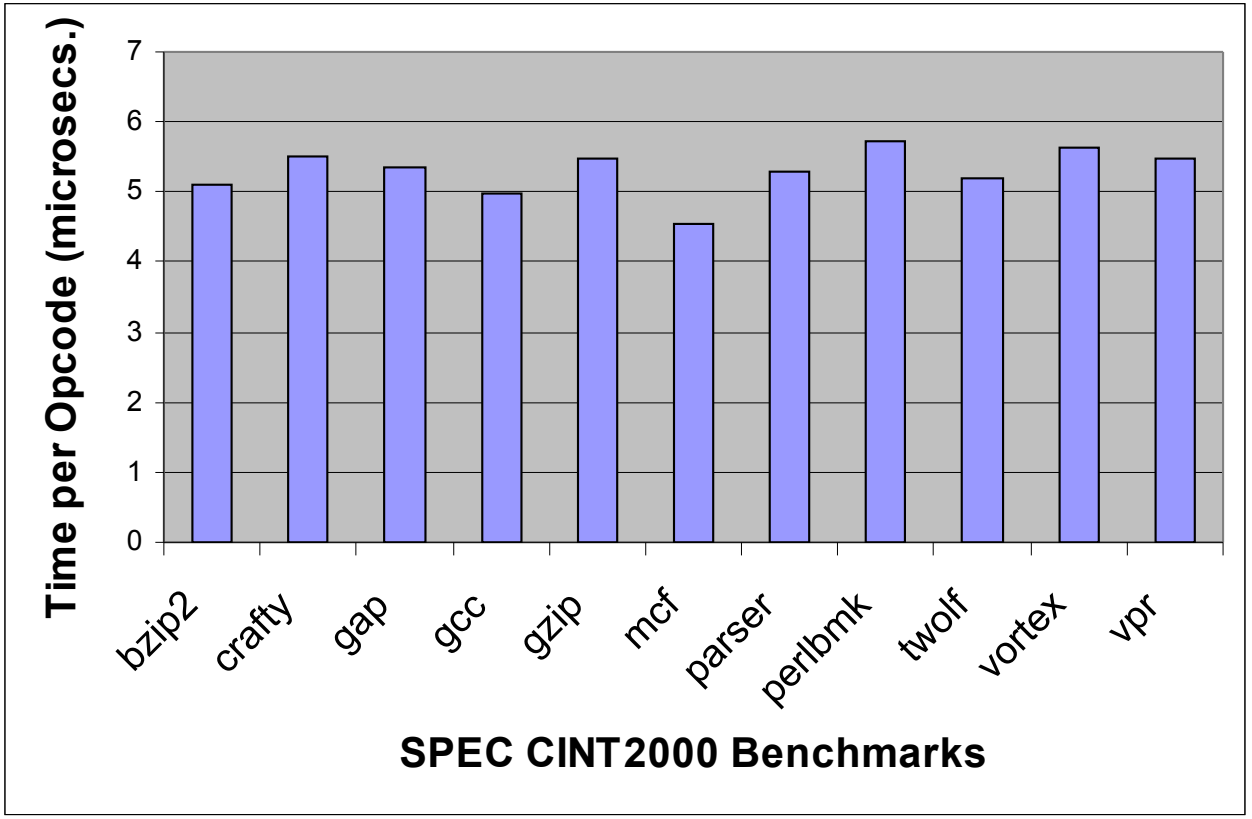
- Background Information ➨
- Method ➨
- Results
- Future Work

Experimental Setup

- Native function microbenchmarks
  - Average of 300 million runs
- 1.4 GHz Power4 setup
- Prototype implementation

Cost of IL Conversion

- 5.3 microseconds per W-Code



Inlining Null Callouts

- Null native method microbenchmarks
- Varying numbers of args (0, 1, 3, 5)
  - Complete removal of call/return overhead
    - Gain back 2 to 3x slowdown
  - confirmed our expectations

Inlining Non-Null Callouts

	Speedup (X)	
Microbenchmark Test	**Instance**	**Static**
hash	5.5	1.8

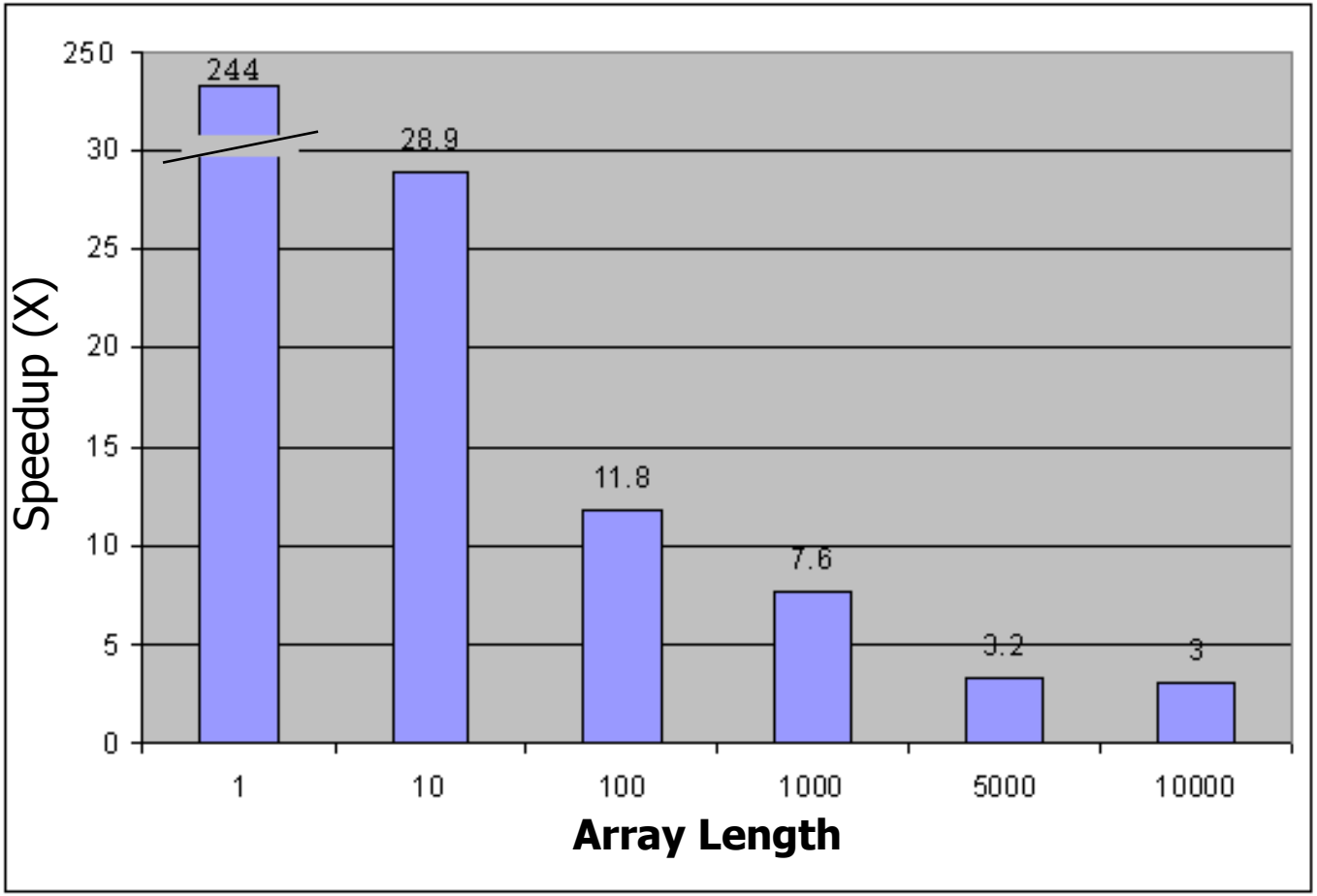
- smaller speedups for natives performing work
- instance vs. static speedup

Inlining & Transforming Callbacks

	Speedup (X)	
Microbenchmark Test	**Instance**	**Static**
CallVoidMethod	12.9	11.8

- Reclaim order of magnitude overhead

Data-Copy Speedups



- **Transformed** `GetIntArrayRegion`

Exposing Inlined Code To JIT Optimizations

Microbenchmark Test	Speedup (X)
GetArrayLength	93.4

```
FindClass

GetMethodID

NewCharArray

GetArrayLength
```

Conclusion

- Runtime native function inlining into Java code
- Optimizing transformations on inlined Java Native Interface (JNI) calls
- JIT optimize inlined native code
- Opaque and binary-compatible while boosting performance

- Future Work
  - Engineering issues
  - Heuristics
  - Larger interoperability framework

Fin