Enabling Constant-Time Interface Method Dispatch in Embedded Java Processors

Thomas B. Preußer  Martin Zabel  Rainer G. Spallek

Vienna, JTRES’07
Itinerary

- Method Dispatch Overview
- Interface Type Coercion
- Employment Issues
- SHAP Integration
- Discussion
Method Dispatch – Foundations

**Method Dispatch**
- is the *dynamic* binding of method implementations to invocations.
- incurs overhead over a static compiler-resolved call.

**Specification**
Search for first matching method from *runtime* type up the type hierarchy:
- Most specific implementation is invoked.
- Search order is relevant for multiple inheritance:

![Diagram](attachment:image.png)

Diamond Problem: Which A?
Which f()?
Method Dispatch – Java Specifics

- *Statically verifiable* type system.
  vs. *dynamic* typing à la Perl, Python, JavaScript (duck typing)
- *Single* dispatch based solely on the type of *this* argument.
  vs. *Multiple* dispatch à la Common Lisp
- *Single* inheritance of method implementations.
  vs. *Multiple* inheritance à la Eiffel, C++, Python, Perl
- *Multiple* inheritance of interfaces.

Concept also found in Objective-C (*protocols*) and C#.
Method Dispatch – Single Inheritance

Straightforward memory layout of instance state and VMT:

This is interpretation-independent.

State and VMT interpretation is valid for all superclasses.
Method Dispatch – Multiple Inheritance

**Approaches:**

- Reflective Search
- Sparse Method Tables
- Hashed Lookup with Conflict Resolution

On statically-typed platforms:

- class-specific data structures
- problem size reduction by indirection through *itable
  set of interface methods* → *set of interfaces*

**JVM Implementations:**

- Jalapeño (Jikes RVM): hashed lookup in fixed-size IMTs
- CACAO: sparse arrays of *itable references*
- JOP/SableVM: sparse interface method tables
- SHAP: *itable attachment through type coercion*
Interface Type Coercion

Coloring of reference values to identify class-specific itable for desired interface.

Goals:

- Automatic coercion by compiler or classfile loader.
- Allow use of standard/legacy classfiles.
- Coloring must be transparent to all but interface-related operations.
Interface Type Coercion

Coloring of *reference values* to identify class-specific *itable* for desired interface.

**Goals:**
- Automatic coercion by compiler or classfile loader.
- Allow use of standard/legacy classfiles.
- Coloring must be transparent to all but interface-related operations.

**Enabled by:**
- DFA based on type meta-data found in classfiles using ASM.
- Injection of constant-time coercion bytecodes.
- Reference value representation as *(Object, Color)* tuple.
Data Flow Analysis

Construction of data flow graph by abstract interpretation.

**Source Vertices**
- method arguments
- values returned by a method call
- successful checkcasts
- caught exceptions

**Inner Vertices**
- stack values
- values in local variables (including arguments)

**Sink Vertices**
- arguments passed to a method
- values stored into an array

Edges reflect data movement.
Interest limited to connected components with interface type sinks!
Data Flow Graph – Example Component

```
new Thread(flag? new A() : new B());
```
Legacy Issue: Multitype

```java
ifxxx @0
new A
dup
invokespecial A.<init>()V
goto @1
```

```
@0:
new B
dup
invokespecial B.<init>()V
```

```
@1:
dup
invokeinterface I1.f()V
invokeinterface I2.g()V
```

– Branch
– Create Instance of A
– Create Instance of B
– Merging Control Flows
– Invoke Interface Method of I1
– Invoke Interface Method of I2
Legacy Issue: Late Runtime Verification

```java
ifxxx @0
new A
dup
invokespecial A.<init>()V
goto @1

@0:
new Object
dup
invokespecial Object.<init>()V

@1:
invokeinterface I1.f()V
```

- Branch
- Create Instance of A
- Create Instance of Object
- Merge Control Flows
- Invoke Interface Method
Disarming Runtime Casts

Runtime casts to interface types require expensive `itable` search. Fast interface method dispatch is rendered absurd if preceded by a cast.

**Measures**

- Never uncolor references.
- Simply check the color before performing a searching cast (fast!).
- Defensive coloring for pre-Tiger legacy code (after SableVM evaluation).

**Result**

References stored in generic data structures will already have the correct color if:

- having been passed as typed method argument such as through `addActionListener(ActionListener)`, or
- having been stored in a appropriately parametrized generic collection such as `ArrayList<ActionListener>`.
SHAP Integration

Embedded JVM with restricted runtime type information:
- CLDC 1.0 runtime implementation
- no support of reflection, and
- no itable searching interface typecasts (hit or fail).

Use of indirect itables:
- cb uses statically resolved itable references.
- Indirection through VMT at invocation rather than at coloring time.
- itable can be reused for all subclasses once created.
- cc resolved through itable.
Indirect itables

IMT I[B] valid for instances of class B and subclass C.

VMT: A
0: @A.g()  IMT: I[A]
   0: #0   g()

VMT: B
0: @B.f()  IMT: I[B]
1: @B.g()  0: #1   g()

VMT: C
0: @B.f()  1: @C.g()  2: @C.h()

By runtime class

B b = ...;
...           
I i = b;
...           
i.g();

Attach I[B]
Discussion

- Coercion impact on code size highly dependable on application.
- Well below 0.1% for all the Embedded Caffeine benchmarks.
- Optimizing compilers supplying tight `StackMapTables` would be favorable for:
  - virtualization of interface method calls,
  - devirtualization of virtual method calls, and
  - abandoning defensive coloring.
- Tupled interface objects hinders great adoption on standard JVMs.
Thank you!