Multi-Dispatch in the Java™ Virtual Machine: Design and Implementation

Computing Science

University of Saskatchewan

Chris Dutchyn (dutchyn@cs.ualberta.ca)

Recognize this code?

```
class Component ... {
 void processEvent(AWTEvent e) {
  if (e instanceof FocusEvent)    processEvent((FocusEvent e));
  else if (e instanceof MouseEvent) {
   switch (e.getID()) {
    case MouseEvent.MOUSE_PRESSED:
    case MouseEvent.MOUSE_EXITED: processEvent((MouseEvent e));
                                  break;
    case MouseEvent.MOUSE_DRAGGED:
    case MouseEvent.MOUSE_MOVED: processEvent((MouseMotionEvent e));
                                 break;
    ...}
  } else if (e instanceof KeyEvent) processEvent((KeyEvent e));
  } else if (e instanceof ...)) ...}
}
```

The *Visitor* Pattern

- **The need for method selection based on the dynamic type of more than one argument is so strong that**
  - OOPSLA included a 3-page paper in their first conference describing double dispatch, already a well-known technique at that time
- **Since then GoF raised this technique to the level of a design pattern**

Multi-Argument Dispatch

- Graphical User Interfaces
  - especially *Drag & Drop*, *Cut & Paste*
- Printing
  - result depends on printer capabilities and document type
- Binary methods
  - key example is equality testing: `Comparable`

Multiple Dispatch

- **Method selection based on multiple arguments**
- **It can be implemented as**
  - **a sequence of dispatches on one argument**
    - each of these is a **uni-dispatch**
  - **a single operation applying all arguments**
    - this is a **multi-dispatch**

Overview

- How does Multi-Dispatch work in the JVM?
  - Modify JVM without changing Java language
  - Multi-dispatching at a callsite
  - Denoting multi-dispatchable methods
- How fast is it?
  - Event Dispatch
  - Multi-Swing

Dispatch Taxonomy

		Dynamic		
		None	Uni-	Multi-
Static	None	Pascal	Smalltalk	CLOS
	Uni-			
	Multi-	Modula 2	C++ Java	Cecil Dylan?

Anatomy of Java Dispatch

Java Source

```
…
Component c =
  new Button;
…
AWTEvent e = new
  FocusEvent;
…
c.processEvent(e);
…
```

Java Binary

```
1 CLASS       "Component"
2 NAME&TYPE   "processEvent"
              "(LAWTEvent;)V"
3 METHOD      #1, #2
…
apush c
apush e
invokevirtual #3
…
```

Method Overloading = Static Multi-Dispatch

- `javac` selects
  - from the method dictionary of the receiver's static class
  - the method that is *most specific applicable* to the declared types of the arguments

```
class Component … {
 void processEvent(AWTEvent e)
    { … }
 void processEvent(FocusEvent f)
    { … }
 void processEvent(KeyEvent k)
    { … }
 … }
```

Method Overriding = Dynamic Uni-Dispatch

- JVM selects
  - from the list of methods understood by the receiving object's class
  - the method that lexically matches the name & type signature

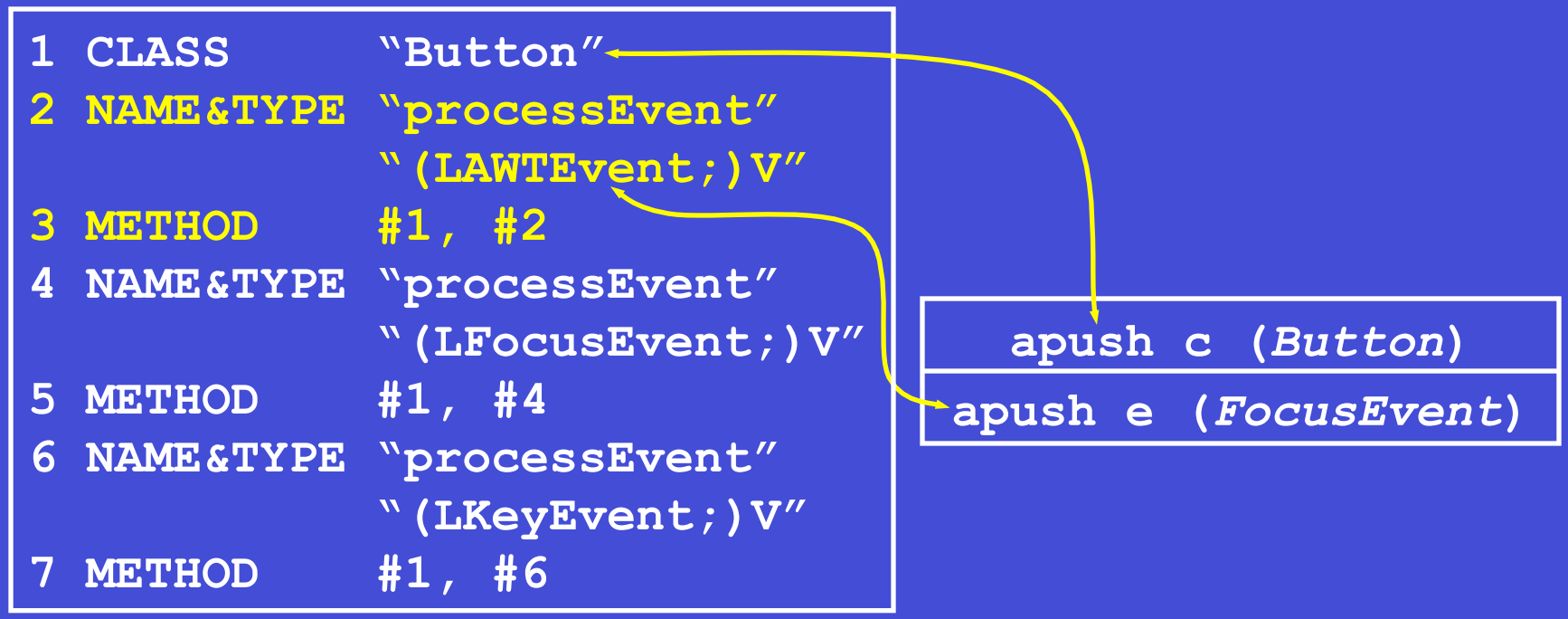
```
1 CLASS       "Button"
2 NAME&TYPE   "processEvent"
              "(LAWTEvent;)V"
3 METHOD      #1, #2
4 NAME&TYPE   "processEvent"
              "(LFocusEvent;)V"
5 METHOD      #1, #4
6 NAME&TYPE   "processEvent"
              "(LKeyEvent;)V"
7 METHOD      #1, #6
```

Invoking the Method

- The JVM must allocate a new operand stack and prepare to execute the bytecode for the method
  - in the Sun classic VM, this process is performed by calling an *invoker* subroutine that is specialized on a method-by-method basis for the number and kind of arguments
- Dispatch is complete
  - in this case, into a method that checks the event type, and dispatches again

At a Callsite

- The signature describes the formal parameters
  - the number, type, and size
- The operand stack holds the matching arguments



Multi-Dispatch a Callsite

- More precise information about argument types
  - ⇒ More precise method with same arity can be selected by a replacement invoker we call the *multi-invoker*

```
1 CLASS       "Button"
2 NAME&TYPE   "processEvent"
              "(LAWTEvent;)V"
3 METHOD      #1, #2
4 NAME&TYPE   "processEvent"
              "(LFocusEvent;)V"
5 METHOD      #1, #4
6 NAME&TYPE   "processEvent"
              "(LKeyEvent;)V"
7 METHOD      #1, #6
```

```
apush c (Button)
apush e (FocusEvent)
```

Multi-Dispatch Compatibility

- We select the most-specific applicable method that matches the original method:
  - the same number of arguments
  - each multimethod parameter has the same type as (or be a subtype of) the corresponding argument
- We also verify that the method return type is the same as (or a subtype of) the overridden method
  - otherwise we throw `IllegalReturnTypeChange`

Denoting Multi-Dispatch

- Implement interface `MultiDispatchable`
  - signals that all method calls to objects of this class (and subclass) are multi-dispatched
  - eliminates user-written dispatch code
- Do not install multi-invoker when inappropriate

```
class Button extends Component
  implements MultiDispatchable {
  …
 void processEvent(AWTEvent e) {
    /*** code removed ***/ }
  …
 void processEvent(FocusEvent f) {
    /* unchanged */ }
 void processEvent(MouseEvent f) {
    /* unchanged */ }
 void processEvent(KeyEvent k) {
    /* unchanged */ }
  …
 void setX(int x) { … } … }
```

Alternate Denotation

- **The mJVM only applies the empty interface at class load time**
  - **if we "leaked" the method-by-method flag out into the .class file, then we could provide method-granularity multi-dispatch**
    - but any method that was eligible as an overriding multimethod would be multi-dispatched as well
- **What should the language notation be?**

Java Syntax Unchanged

- Using an empty interface is not novel
  - `Cloneable` operates in the same way
- Unlike other efforts, we neither extend the language syntax nor modify the compiler
  - we use an unchanged `javac` to compile tests
- We can extend existing libraries to support multi-dispatch without source and without additional layers of dispatch

Static Checking

- `MDLint` - a code-testing tool that checks binary files for the previous error and for:
  - overriding throws clauses are a subset of overridden method
  - overriding method has equal or greater visibility
    - note that visibility security checks are still in effect
  - Lapses in these are *Java* errors not *JVM* errors
- Also warns about argument combinations that lead to ambiguous multi-dispatches
  - only when encountered does the multi-JVM throw `AmbiguousMethodException`

Class Numbering

- **Initially, each class has the same index as it's parent**
- **As classes are loaded**
  - **Any multimethods that distinguish**
    - two classes that have the same index
    - force the subclass to be renumbered
      - Recursively down the inheritance hierarchy

Class Loader Constraints

- **Classes are not uniquely identified by their names**
  - They are identified by class-loader and name
- **class loader constraints (Bracha)**
  - **ensure that object instances don't leak across class loader boundaries**
    - my `private` field is `public` in your class

Constraints Not Needed

- **If all dispatch were multi-dispatch**
  - **No class loader constraints needed**
    - (Saraswat)
  - **Because my version of your class is always treated as distinct**
    - Tweaking the hole causes a method-not-found exception

Always-On Multi-Dispatch?

- **Yes!? (work in progress)**
  - **a hand-full of places need to be rewritten (javac)**
    - Recognized by static cast of method argument
  - **Solution alternate method names**
    - Otherwise the most precise method will be invoked

3. Implementation Details

Implementation Overview

- Any class that wishes its methods to be multi-dispatched needs to implement an empty interface `MultiDispatchable`

```
class Component implements MultiDispatchable { … }
```

- Our virtual machine recognizes this special interface (just like it does `Cloneable`) and sets a flag bit
- At interpretation time, the VM checks the flag and multi-dispatches as appropriate

Invoking a Method

- A `Component::processEvent()` clause

```
0x43          ; aload_1          (load e)
0xc1 0x0004   ; instanceof #4    (class FocusEvent)
0x99 0x0018   ; ifeq             (check for 0 result)
0x42          ; aload_0          (load this)
0x43          ; aload_1          (load e)
0xc0 0x0004   ; checkcast #4     (class FocusEvent)
0xb6 0x0013   ; invokevirtual #19 (call-site
                Component::processFocusEvent)
0xa7 0x????   ; goto ????        (jump to end of method)
. . .         ; next instanceof test
```

Call-Site Information

- Note that at the call-site, we have all of the information needed to perform a dispatch:
  - method name
  - number of arguments
  - location of uni-dispatch receiver
    - `aComponent`
  - types of other arguments
    - `FocusEvent`

Locating Methods Given an Instance

- **Sun's JVM is a handle-based interpreter**
- **Each object handle is a pointer to a structure containing**
  - **a pointer to its instance variable values**
  - **a pointer to table of methods for its class**
    - equivalent to a C++ *virtual function* table

Performing a Uni-Dispatch

① Locate constant pool entry and compute the number of arguments

② Find method name and signature from constant pool entry

③ Locate receiver on stack and dereference handle to locate method table

④ Scan through method table looking for exact match for
  - » name
  - » signature

⑤ Push arguments onto new activation record

⑥ Begin interpreting new bytecodes

Performing a Multi-Dispatch

① **Locate constant pool entry and compute the number of arguments**

② **Get method name from constant pool entry**

③ **Locate receiver on stack and dereference handle to locate method table**

➔ **If multi-dispatch flag set in receiver:**

- ➔ Technique-specific selection of more precise method

➔ **Else uni-dispatch:**

- ④ Scan through method table looking for exact match for
  - » name
  - » signature

⑤ **Push arguments onto new activation record**

⑥ **Begin interpreting new bytecodes**

The MSA Algorithm (1)

- Most Specific Applicable is the algorithm specified in JLS for the Java compiler.
- Consider `processEvent(e)` where `e` is a `MouseMotionEvent`

The MSA Algorithm (1)

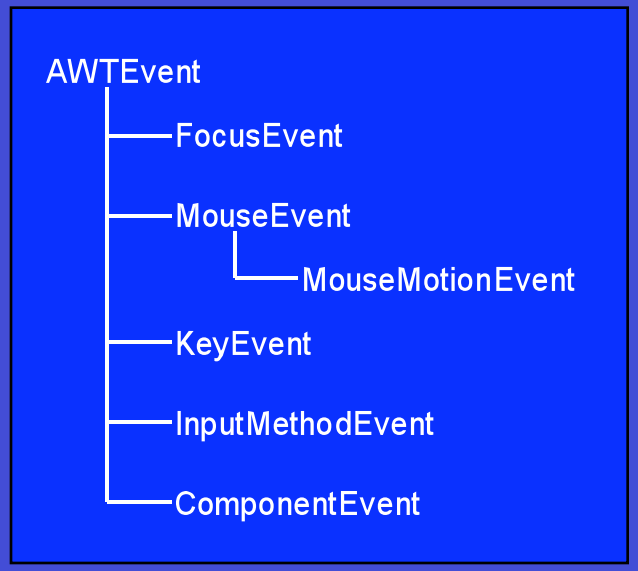
- Build list of potential methods with the same name and arity from the receiver's class

`processEvent(AWTEvent)`
`processEvent(FocusEvent)`
`processEvent(MouseEvent)`
`processEvent(MouseMotionEvent)`
`processEvent(KeyEvent)`
`processEvent(InputMethodEvent)`
`processEvent(ComponentEvent)`

The MSA Algorithm (2)

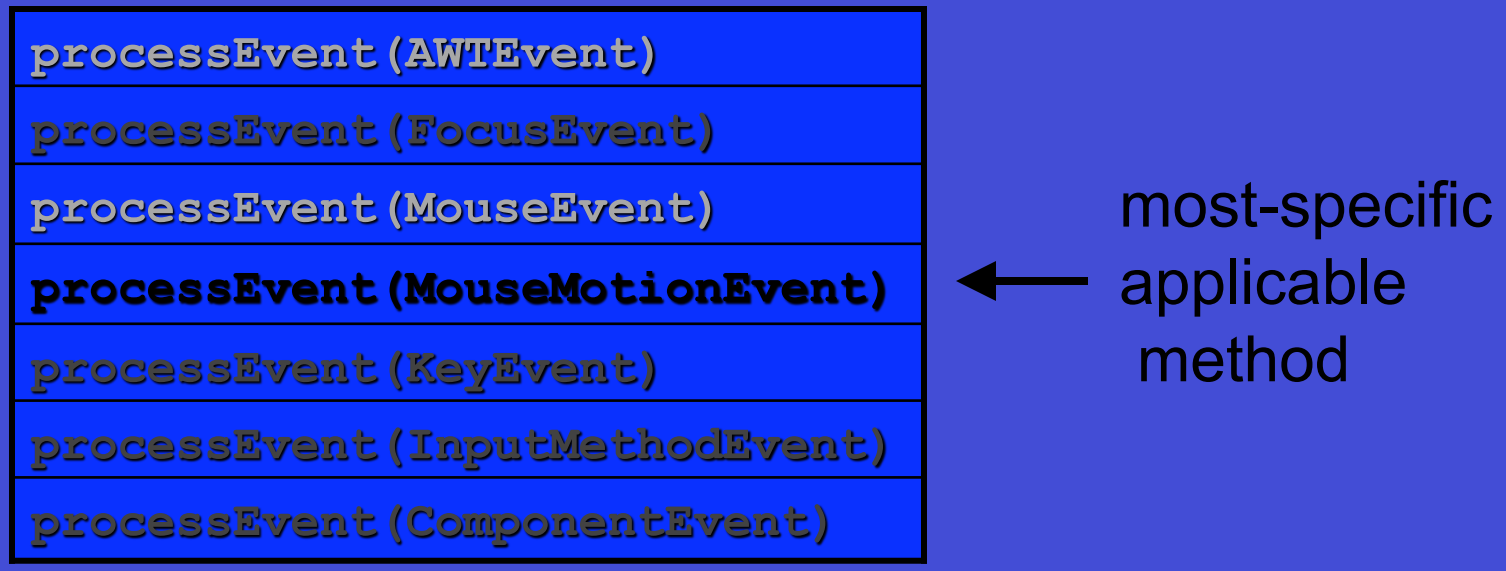
- Filter out inapplicable methods
  - e.g. a `MouseMotionEvent` is not a subclass of `FocusEvent`

`processEvent(AWTEvent)`
`processEvent(FocusEvent)`
`processEvent(MouseEvent)`
`processEvent(MouseMotionEvent)`
`processEvent(KeyEvent)`
`processEvent(InputMethodEvent)`
`processEvent(ComponentEvent)`



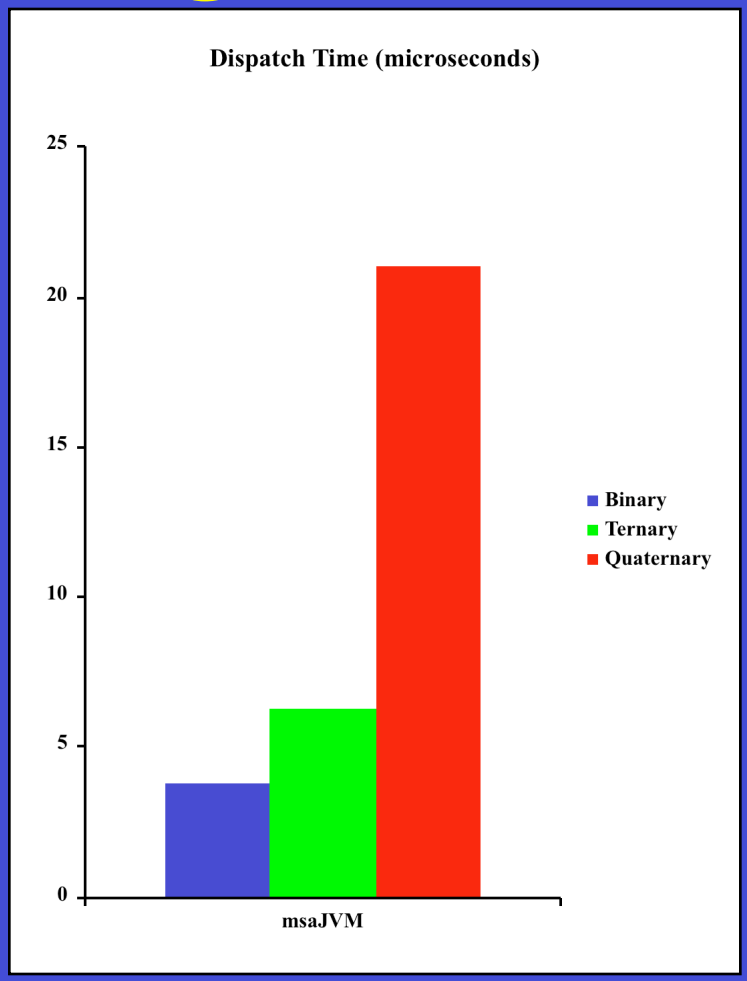
The MSA Algorithm (3)

- **Determine most specific by comparing argument types for remaining methods**
  - **e.g. a `MouseMotionEvent` can be widened to a `MouseEvent`, but not vice versa**



Efficiency of MSA

- The MSA algorithm described above matches the algorithm used by the `javac` compiler in its static multi-dispatch
- But, it is expensive to perform so many subclass tests
- In the worst case, the algorithm is $O(m^2n)$
  - m = number of methods
  - n = arity of methods



Dispatch Techniques

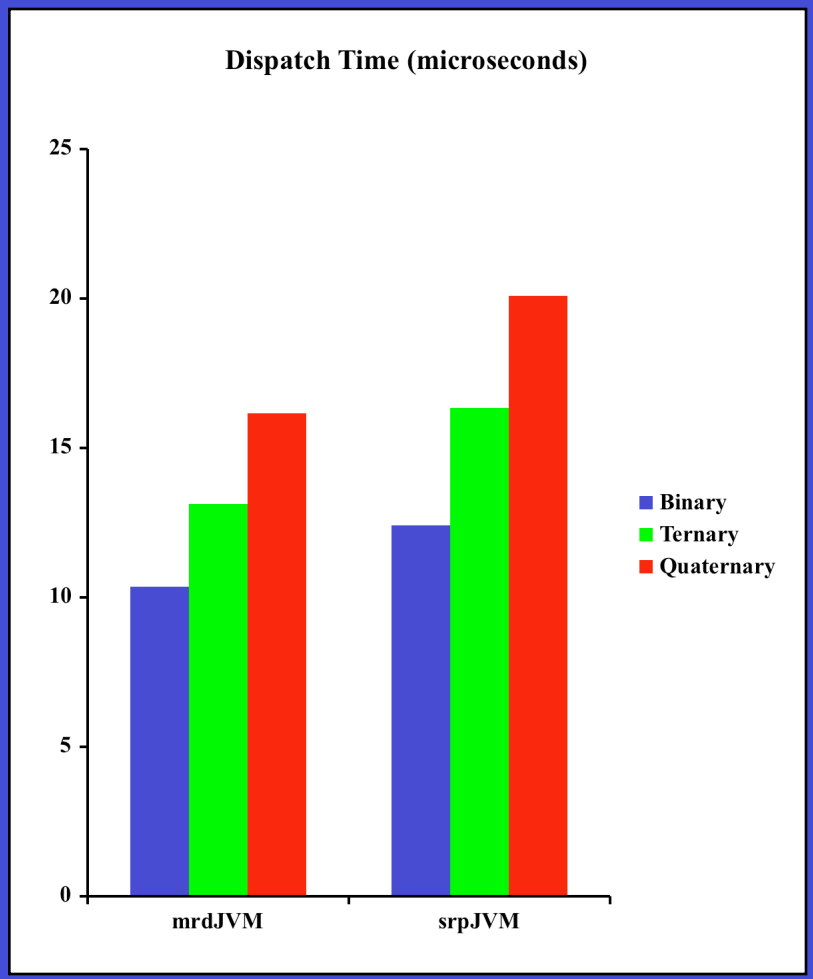
- MSA - dynamic version of static multi-dispatch implemented by `javac`
- Two table-based techniques; *consider table:*

	Button	CheckBox	List	ScrollBar
processEvent()	*method1*	*method4*	*method7*	*method10*
paint()	*method2*	*method5*	*method8*	*method11*
resize()	*method3*	*method6*	*method9*	*method12*

  - MRD – multiple row displacement
  - SRP – single row displacement

Efficiency of MRD, SRP

- Table-based techniques build a compressed dispatch table which can quickly dispatch to the correct method in *O(n)* time
- These techniques show a much slower time penalty as arity increases



Compatibility

- Total run time (user + system) for two tests:
  1. 1 million double dispatches
  2. Compile of `sun.tools`

Total Time	Event Dispatch	`Javac`
SunJVM	6.28 s	174.3 s
MDJVM	6.31 s	175.1 s

- uni-dispatch program overhead is almost undetectable
- `javac` runs properly with our multi-dispatch VM

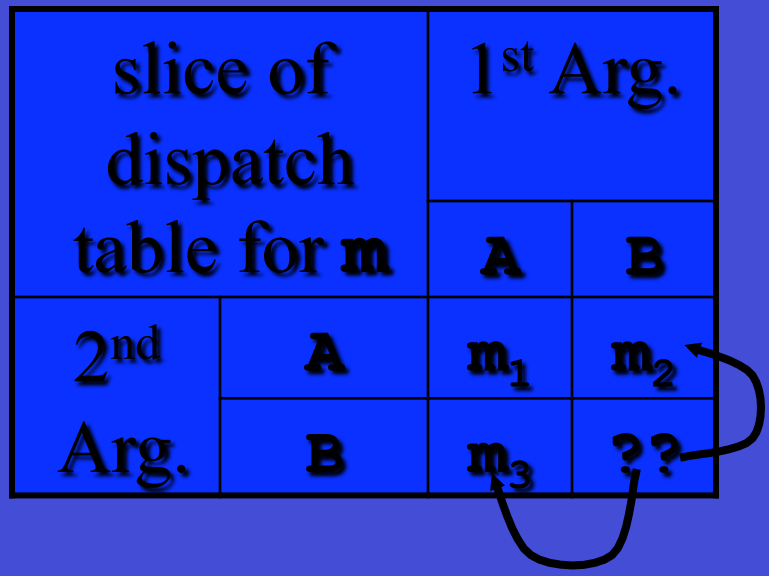
4. Issues

- Method Conflicts
- Incompatible Return Types
- Null Arguments
- Thread Safety/Efficiency
- Primitive Widening

Issues: Method Conflicts

- We treat all arguments as the same priority, hence, we can encounter ambiguous method conflicts which the compiler does not detect:

```
class A {}
class B extends A {}
class X {
    void m1(A, A) { . . . }
    void m2(A, B) { . . . }
    void m3(B, A) { . . . }
    void main() {
        A aB = new B();
        m(aB, aB);  // ??
    }
}
```



Issues: Incompatible Return Types

- Java insists that overriding methods must have the same return type

- But, `javac` does not recognize multi-dispatch methods as overriding

```
class A {
        int m() { . . . }
}

class B extends A {
        String m() { . . . } // ILLEGAL
}

class X {
    int m(A) { . . . }
    String m(B) { . . . }
    void main() {
        A aB = new B();
        int i = m(aB);       // ??
    }
}
```

Issues: `null` arguments

- Java treats null arguments inconsistently:
  - if the null is typed, then the method corresponding to the type of the null will be invoked
  - if the null is untyped, then uni-dispatch Java invokes the same method that multi-dispatch

```
class A {}
class B extends A {}
class X {
        void m(A) { . . . }
        void m(B) { . . . }
        void main {
                A a = null;

                m(a);    // uni-dispatch invokes m(A)
                         // multi-dispatch invokes m(B)

                m(null);          // both invoke m(B)
        }
}
```

Issues: Thread Safety/ Efficiency

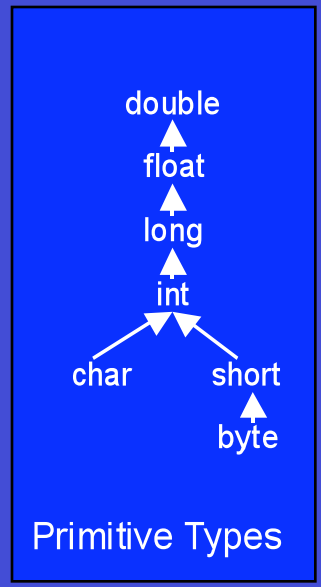
- Table-based techniques also need to be thread-safe and thread-efficient
  - we don't want dispatch on all threads to stop while we rebuild our dispatch table
  - we have designed an approach that uses two tables, and permits dispatch through one while the other is being extended with new classes and methods

Issues: Primitive Widening

- The compiler automatically widens primitive arguments based on the static multi-dispatch method

```
class A {
        void m(int I) { . . . }
}
class B extends A {
        void m(byte b) { . . . }
}
class X {
    main(){
        byte b = 8;
        A aB = new B();
        B bB = new B();

        aB.m(b);          // javac encodes as m(int)
        bB.m((int) b);       // programmer does same
    }
}
```



Performance Results

Hot Spot JVM

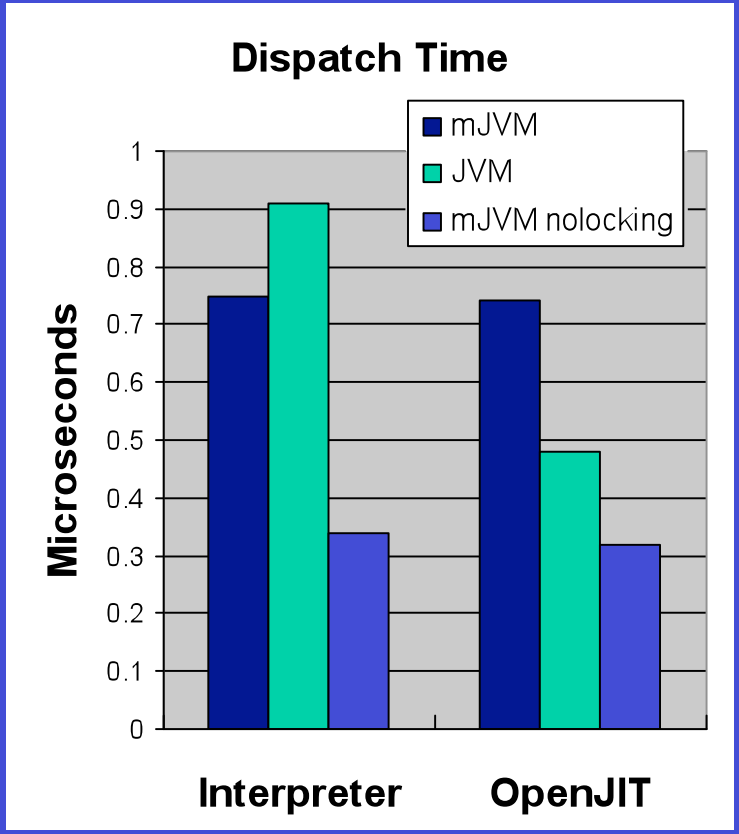
- Work in progress
  - I'm here to learn more
- Reporting on our work with
  - JVM 1.2
  - OpenJIT

Event Dispatch

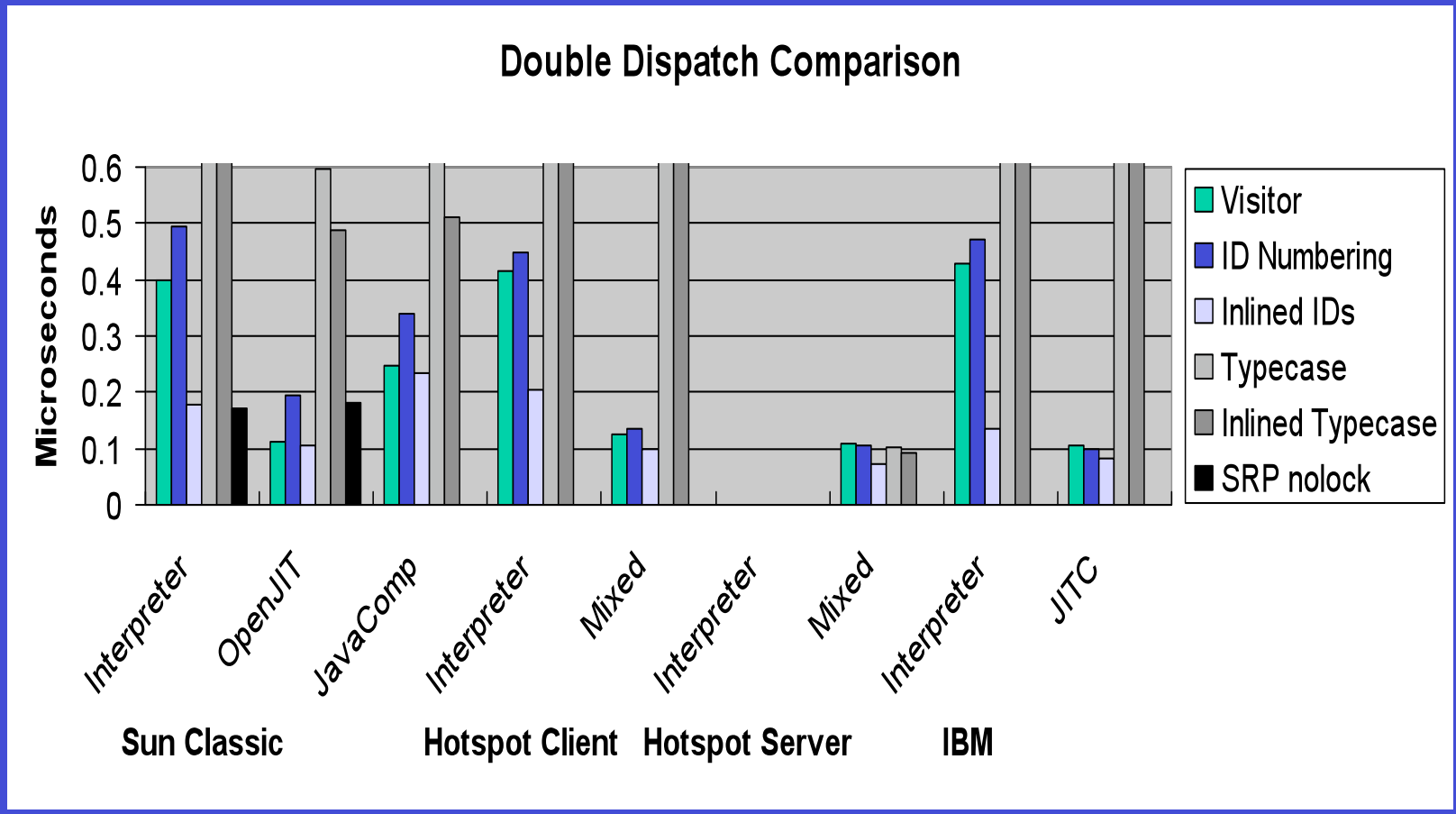
- **Dispatch**

  `processEvent()`

  across
  - **seven event types**
  - **seven components**
- **VMs:**
  - JVM = classic VM
  - mJVM = multi-JVM with table-based dispatcher



Double Dispatch Compared

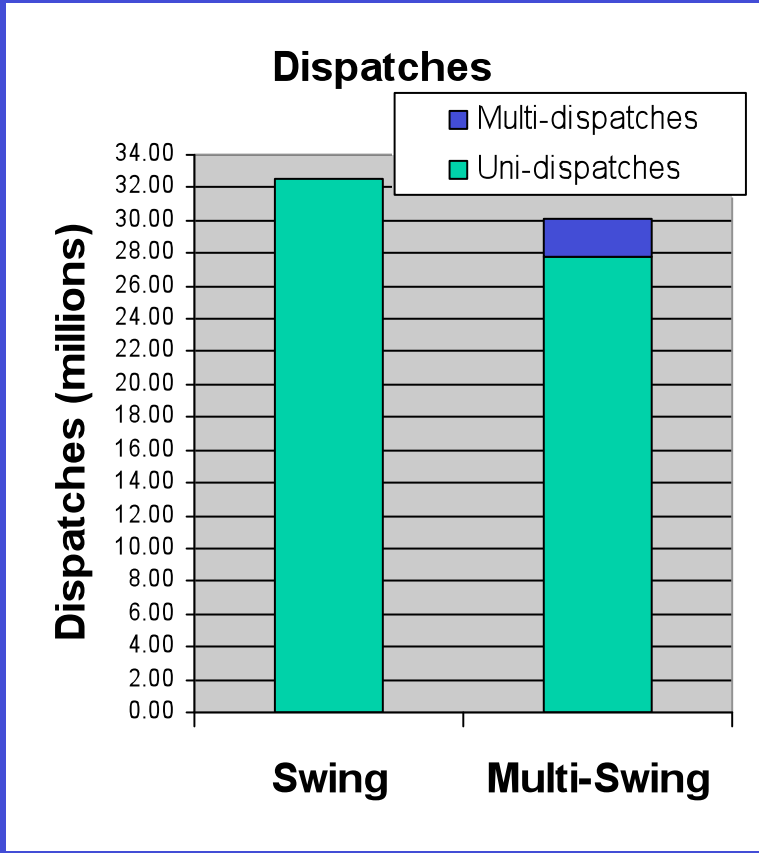


Multi-Swing (and AWT)

1. Modified 92 of 846 classes (11%)
2. Replaced 171 conditionals (5%)
3. Mean number of decision points reduced from 3.8 to 2.0 per method

4. Added 57 new event subclasses
5. Added 123 new multimethods

Multi-Swing Results



Multi-Dispatch and OpenJIT

- Disabled inlining of private and final methods because they may still be dispatched based on arguments:

```
class LoggingButton extends Button {
 void processEvent(AWTEvent e) {
  this.logEvent(e); super.processEvent(e);
 }
private void logEvent(FocusEvent f) { … }
private void logEvent(MouseEvent m) { … }
… }
```

OpenJIT 1.1.15 Support

- Callsites compile to an invoker that dispatches the target method from a virtual function table
- For multimethods we replace the virtual function table target to a routine which
  - selects a more precise multimethod
  - jumps to the actual compiled code
- It calls the same multimethod dispatcher that operates in the interpreter

Summary

- **Added dynamic multi-dispatch to Java**
  - without changing syntax or compiler,
  - allowing programmer to select classes supporting multi-dispatch,
  - without penalizing existing uni-dispatch,
  - and maintaining reflection and existing APIs.
- **Provide initial performance results**
  - table-based dispatcher surpasses interpreter
  - table-based dispatcher compatibile with JIT.

Future Work

1. Complete HotSpot implementation

2. Support table reduction when unloading classes

Questions?

Supplemental Material

- Source Availability
- Double Dispatch Comparisons
- Supported Invoke Bytecodes
- Alternate Denotation

Source Availability

- We're packaging it up
  - we expect to have it available in a couple of weeks.
- But ...
  - We use the Sun *classic VM* under the research provisions of their Community Source License.
  - Hence, you must agree to that same license;
  - And we need to ensure that you accept that license, so only e-mail requests accepted.

Supported Invoke Bytecodes

- **Invokevirtual** `VirtualMultiDispatchable`
  - **invokevirtual_quick / wide**
  - **invokenonvirtual**
- **invokestatic** `StaticMultiDispatchable`
  - **invokestatic_quick / wide**
- **invokespecial**

  `SpecialMultiDispatchable`
  - **invokespecial_quick / wide**
- **invokeinterface - no interface defined because invoker routines already redefined**

The Key Question

~~Do we need dynamic multiple dispatch?~~

How do we implement it?

Multi-Dispatch in the Java VM