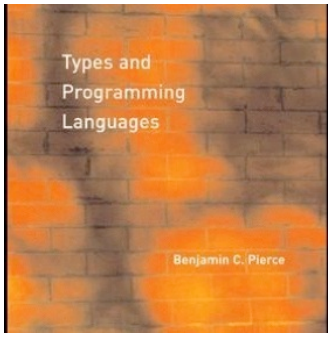
Virtual and Interface Methods



Method Invocation

- Method invocation is one of the key features of OOPLs
- The process of determining which code to run is call "method dispatch" and depends on the runtime type of the receiver (late binding).
- We will take Java as a practical example and study how compile-time resolution and runtime dispatch interact.

Methods

- Methods implement the behavior associated with classes and objects

```
[public|private|protected|abstract|final|synchronized|native]*
[<Type>]  <MethodName> ( [[final] <Type> <Name>]+ )
[throws <Type>+]
[{<Block>}|;]
```

- `abstract` methods have "`;`" as body. All other methods must have a `{<block>}` body
- `final` methods can not be overridden
- constructors have no return types
- methods must not fall of the end of the body without returning the proper type. But the following is allowed

```
int m() throw E { throw new E(); }
```

Method signatures

- The signature of a method consists of the name of the method and types of formal parameters to the method.
  - Example:

    ```
    public int  foo( char c, HashTbl b)
    ```

    has signature

    ```
    foo(C;HashTbl)
    ```

- A class may not declare two methods with the same signature.

Overriding & Hiding

[override] A method A.m overrides a method B.n if

1. A <:$_s$ B,
2. signature(m) = signature(n)
3. return-type(m) = return-type(n)
4. throws(m) ≤ throws(n)
5. if n is **`public`** then m is **`public`**, if n is **`protected`** then m is (**`protected`**|**`public`**), if n is default-access then m is not **`private`**
6. n is not **`static`**

[hide] A static method A.m hides a static method B.n if

1. A <:$_s$ B,
2. signature(m) = signature(n)
3. return-type(m) = return-type(n)
4. throws(m) ≤ throws(n)
5. if n is **`public`** then m is **`public`**, if n is **`protected`** then m is (**`protected`**|**`public`**), if n is default-access then m is not **`private`**

Overloading

- If two methods of a class have the same name but different signatures they are said to be overloaded.

  Example:

```
class C {
   int  foo( C bar ) {}
   int  foo( B bore) {}
   void foo( int i) throws Stuff {}
   char foo( B bore) throws IllegalValue {} // error
```

Method invocation

- We consider the process of binding a source code method invocation expression to an actual method
  - partly done at compile time
  - partly done at run time
- Syntax:

```
<MethodName>( <args> ) |
<Primary> . Identifier ( <args> ) |
super . Identifier ( <args> )
```

Examples:

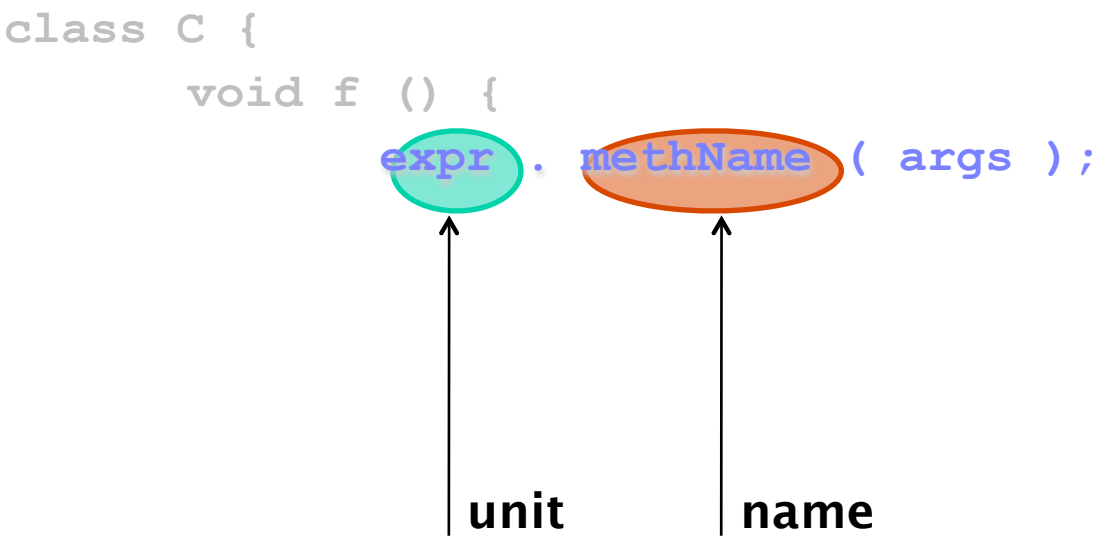
```
foo(arg)
obj.foo(arg)
HashTbl.foo(arg)
obj.foo(arg).bar()
```

Determine Unit and Name (step 1)

- Determine the name of the method and the unit (class or interface) in which the method is located. Consider the cases:

1. `<MethodName>(<args>)`
   if `MethodName` is
   - … an identifier then the unit is the one containing the method invocation and `MethodName` is the name of the method
   - … a qualified name `TypeName.Identifier` then the unit is `TypeName` and `Identifier` is the method's name
   - … a qualified name `FieldName.Identifier` then the unit is type-of (`FieldName`) and `Identifier` is the method's name
2. `<Primary>.Identifier(<args>)`
   unit is type of `Primary` expr, `Identifier` is method's name
3. `super.Identifier(<args>)`
   unit is the superclass, `Identifier` is method's name

Determine Unit and Name (step 1)



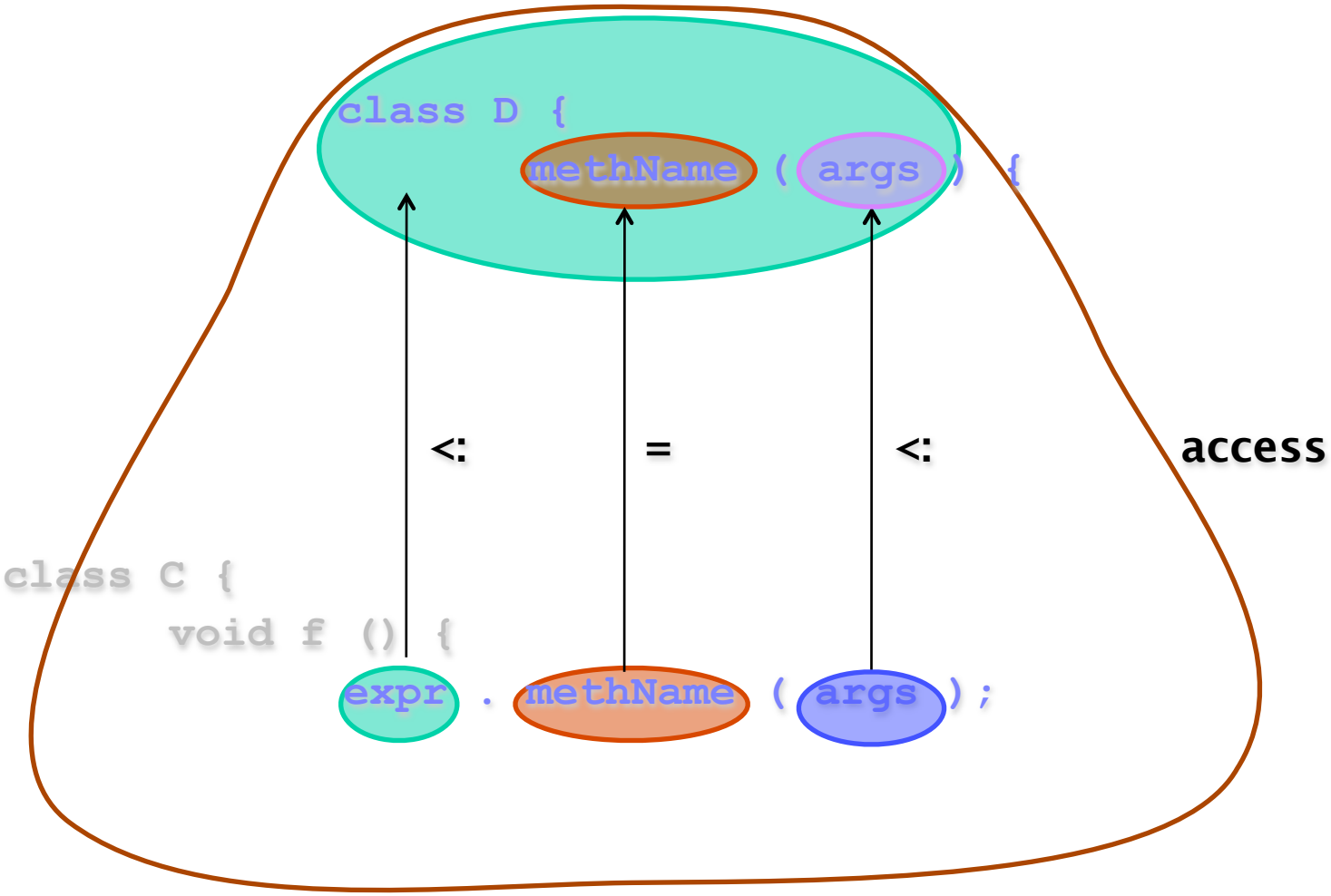
Determine Method Signature (step 2)

- Search the unit for method that are applicable and accessible.

1. A method is applicable if
   - number of parameters = number of arguments
   - type of each argument can be converted by method invocation conversion to the corresponding parameter
   - the name of the method is the one determined in step 1

   All inherited methods are included in this search.
2. Whether a method is accessible to an invocation depends on the access modifier in the declaration and on where the method appears.

Determine Method Signature (step 2)



Determine Method Signature (step 3)

- Then choose the most specific method.

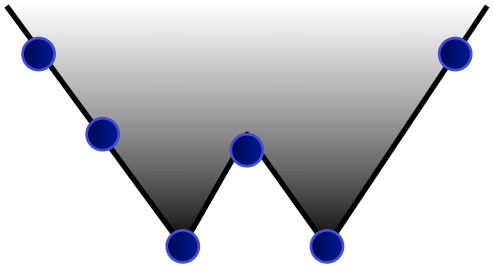
  Given methods `T m( T1, ... Tn)` and `U m′( U1, ... Un)`, `m` is more specific than `m′` if

  - `T` can be converted to `U` by method invocation conversion
  - for all i from 1 to n, `Ti` can be converted to `Ui` by method invocation conversion

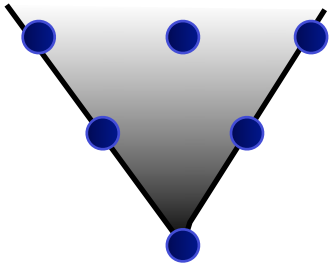
  A method is said to be maximally specific for an invocation if it is applicable and accessible and there is no method more specific

  A method is said to be the most specific if it is the only maximally specific method for an invocation.

Determine Method Signature (step 2)



Ambiguous



Correct

Determine Appropriateness (step 3)

- If there is a most specific declaration for a method invocation, it is called the compile-time declaration.

  Check:

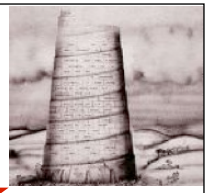
1. `<MethodName>(<args>)`
   if the invocation occurs within a `static` method or an initializer then the compile-time declaration must be `static`.
2. `<TypeName>.<MethodName>(<args>)`
   then the compile-time declaration must be `static`.
3. If the return type is `void`, then the invocation must be a top-level expression.

- Output:
  - method name,
  - class or interface that contains the compile-time declaration
  - number and type of parameters, result type,
  - invocation mode: static, nonvirtual (private), super, interface, virtual

Method Invocation (step 4)

- At run-time, method invocation requires five steps:
  1. Compute the target reference
     static methods have no target, otherwise it is either this or the value of the primary
  2. Evaluate arguments in order from left to right.
  3. Check accessibility of type and method
     Throw **`NoSuchMethodError`** if method not found, **`IncompatibleClassChangeError`** if the class does not implement the interface in which the method was found, **`IllegalAccessError`** if the class or method is not accessible due to access modifier changes.
  4. Locate method to invoke
     Throw **`NullPointerException`** if target is null.
  5. Create frame, synchronize and transfer control

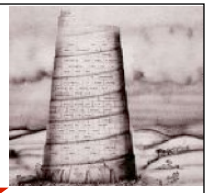
Implementing Method Invocation



- Four types of invocation bytecodes:
  - invokespecial -- direct function call (compile/load-time binding)
  - invokestatic -- direct function call (compile/load-time binding)
  - invokevirtual -- indirect call (runtime binding)
  - invokeinterface -- indirect call (runtime binding)

The difference between `invokevirtual` and `invokeinterface` boils down to the difference between single and multiple inheritance. (A class may have a single parent, but can implement multiple interfaces)

Invokevirtual

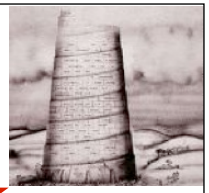


- Given a call:

```
x.foo(c, "hello")
```

- At compile-time the type of `x` is some class `T`, the signature of the method is `foo(C,String);`
- At run-time, type preservation ensures that `x` will refer to an object of type `T`. If `T` is a class, the object referred to from `x` can be either a `T` or a subclass of `T`.
- Implementation:
  - Every method signature defined in class `T` is given a unique offset which must be larger than any offset of inherited methods
  - A table of code pointers is associated to every object.
  - The compiler generates a call to `x->vtable [FOO_C_STRING]`

Invokeinterface



- Given a call:

```
x.foo(c, "hello")
```

- At compile-time the type of `x` is some interface `T`, the signature of the method is `foo(C,String);`

- Fun fact one:

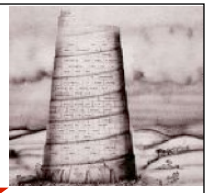
  the JVM verifier does not check that `x` has a type that implement `T`!

- Fun fact two:

  `T` needs only be loaded when the call is executed.

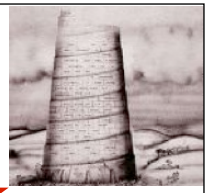
- How do we implement this efficiently?

Class Object Search



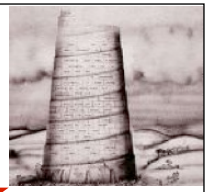
- Naive strategy: every class maintains a list of directly implemented interfaces.
- Implement `invokeinterface` by searching the hierarchy and performing all dynamic checks required by the specification.

Searched ITables



- Every class has a list of interface tables.
- Every interface has unique IID
- Every method defined by an interface has unique MID.
- Implementation of invokeinterface given an oref, IID, MID:
  - get the interface table from the oref
  - scan it for the IID
  - if found
    - move-to-front the IID in the table
    - use the MID to index in the itable and jump
  - if not found, do a dynamic type check,
    - load the interface and add the itable or throw an exception

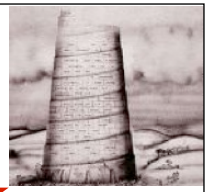
Directly Index ITables



- Every class has an array of interface tables.
- Every interface has an IID (not necessarily unique)
- Every method defined by an interface has unique MID.
- Implementation of invokeinterface given an oref, IID, MID:
  - do a subtype test to ensure that oref implements the interface
  - get the interface table from the oref and index it at IID,
  - use the MID to index in the itable and jump

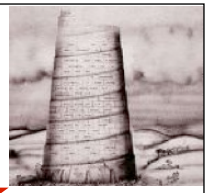
  - As an optimization: initialize the ITable with thunks that do the subtype test and load the interfaces

Jikes IMTs [OOPSLA01]



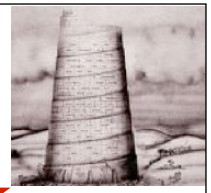
- Every method defined in an interface is assigned an offset (not unique)
- Every class has an Interface Method Table (IMT)
- Invokeinterface uses the offset to index the table and when there is a conflict, compiles a stub for resolving which method to call.

Jikes IMTs [OOPSLA01]



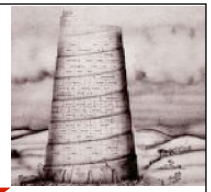
```
//   virtual invocation sequence
//   t0 contains a reference to the receiver object
L    s2, tibOffset(t0)  // s2 := TIB of the receiver
L    s2, vmtOffset(s2)  // s2 := VMT entry for method being dispatched
MTLR s2                 // move target address to the link register
BLRL                    // branch to it (setting LR to return address)


//   IMT-based interface invocation sequence
//   t0 contains a reference to the receiver object
L    s2, tibOffset(t0)  // s2 := TIB of the receiver
L    s2, imtOffset(s2)  // s2 := IMT entry for signature being dispatched
L    s1, signatureId    // put signature id in hidden parameter register
MTLR s2                 // move target address to the link register
BLRL                    // branch to it (setting LR to return address)
```

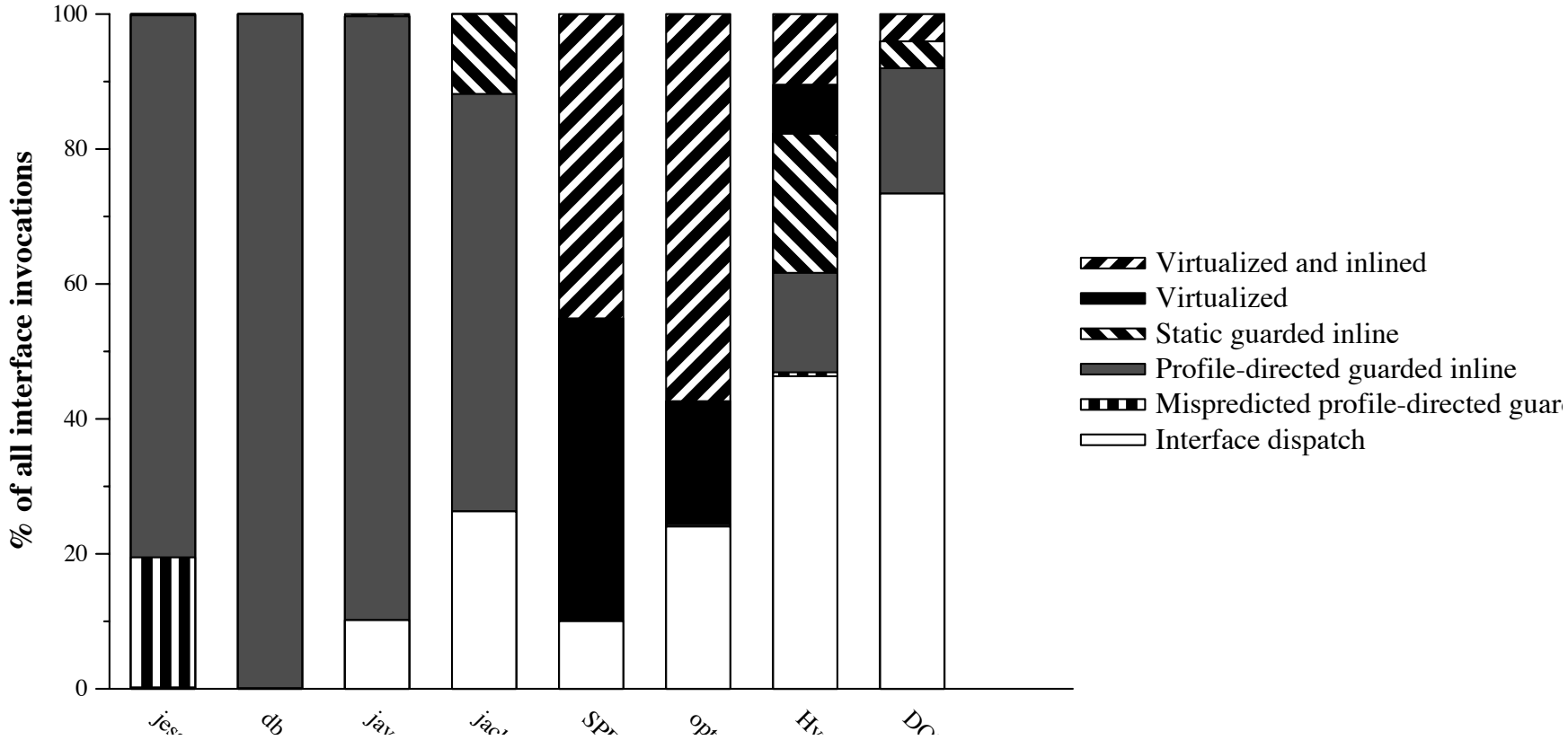
Benchmarks



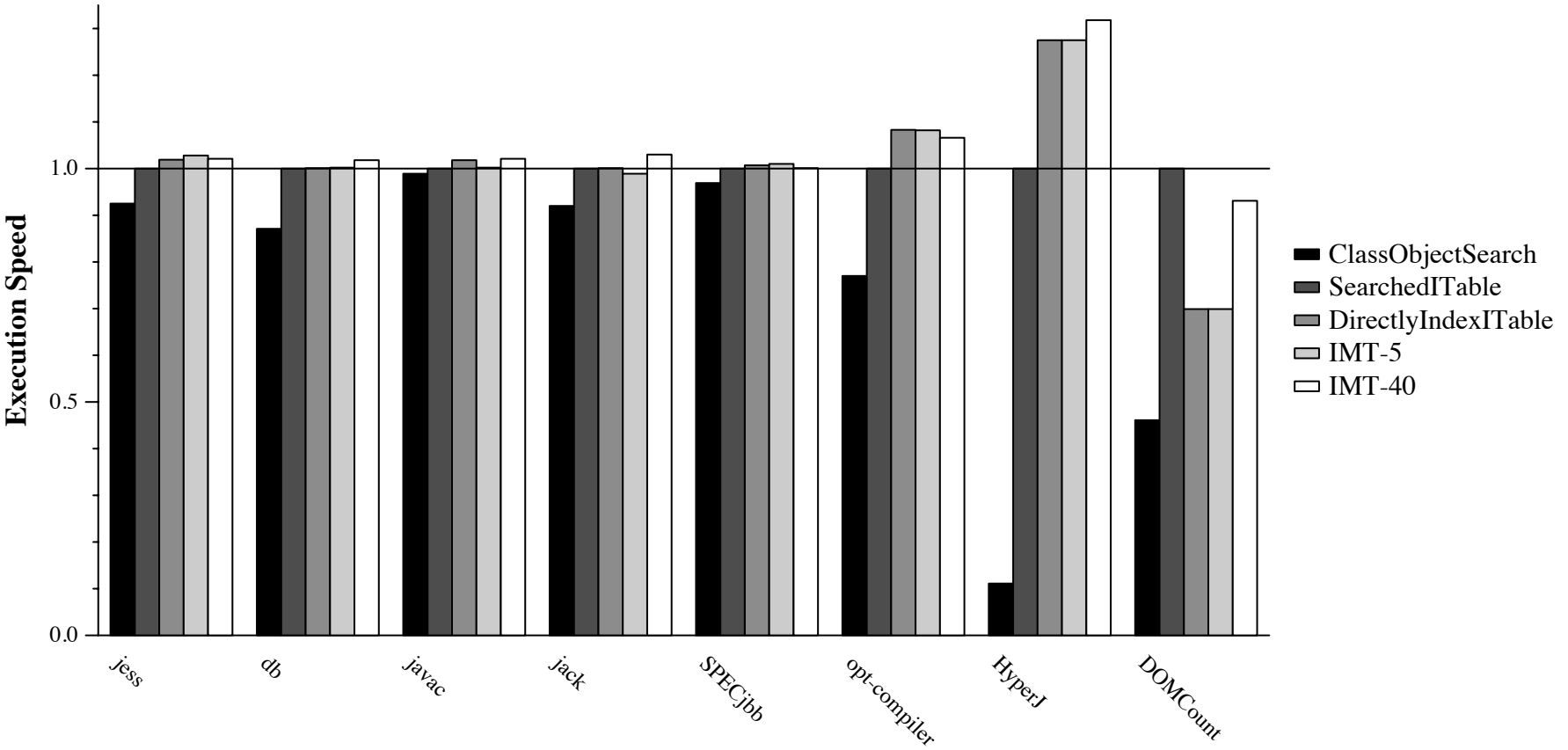
Dispatching Mechanism	Number of Methods in Interface	Trivial Callee
virtual	Not applicable	8.13
Embedded IMT-5	interface with 1 method (no IMT conflict)	8.23
Embedded IMT-5	interface with 100 methods (20 element stub)	20.25
Embedded IMT-40	interface with 1 method (no IMT conflict)	8.23
Embedded IMT-40	interface with 100 methods (2 or 3 element stub)	14.23
IMT-5	interface with 1 method (no IMT conflict)	10.18
IMT-5	interface with 100 methods (20 element stub)	22.20
IMT-40	interface with 1 method (no IMT conflict)	10.18
IMT-40	interface with 100 methods (2 or 3 element stub)	18.20
Directly Indexed ITables	interface with 1 method	12.18
Directly Indexed ITables	interface with 100 methods	12.18
Searched ITables	interface with 1 method	77.45
Searched ITables	interface with 100 methods	77.45
Class Object Search	interface with 1 method	352.55
Class Object Search	interface with 100 methods	1,743.13

Benchmarks

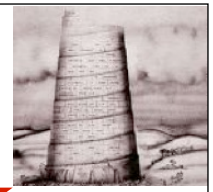


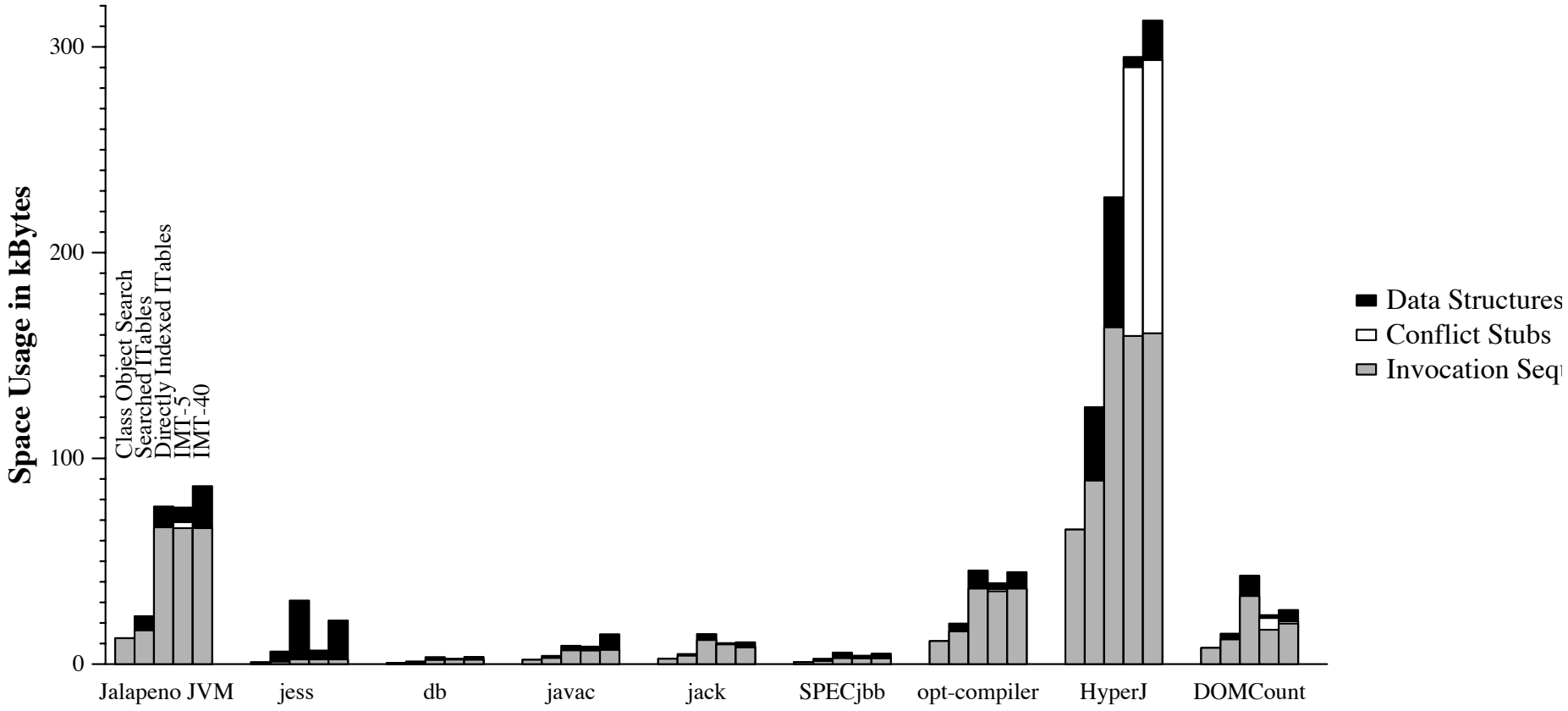

% of all interface invocations
100
80
60
40
20
0
db
Virtualized and inlined
Virtualized
Static guarded inline
Profile-directed guarded inline
Mispredicted profile-directed guar
Interface dispatch

Benchmarks


Execution Speed
1.0
0.5
0.0
jess
db
javac
jack
SPECjbb
opt-compiler
HyperJ
DOMCount
ClassObjectSearch
SearchedITable
DirectlyIndexITable
IMT-5
IMT-40

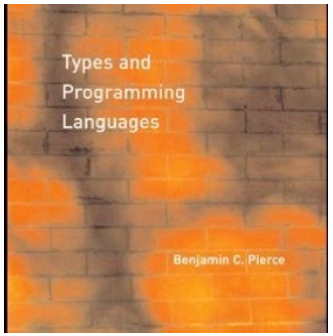
Benchmarks




Space Usage in kBytes
300
200
100
0
Class Object Search
Searched ITables
Directly Indexed ITables
IMT-5
IMT-40
Jalapeno JVM
jess
db
javac
jack
SPECjbb
opt-compiler
HyperJ
DOMCount
Data Structures
Conflict Stubs
Invocation Seq

Subtype tests

[Palacz,Vitek ECOOP'03]



Subtype tests

- Subtype tests are queries over a poset of types, (T, <:). Write A <: B if B belongs to transitive closure of parents of A.
- Dynamic subtype tests are needed to ensure run time type safety.
- Subtype tests have unpredictable time and space requirements.

Subtype tests in Java

- Dynamic subtype tests are frequent operations in many Java programs, consider:

```
X instanceof A

(A) x

A[0] = x

try{…}catch(E e)…
```

- Furthermore, the Generic Java compiler (gj) works by translating genericity away and inserting subtype tests to obtain verifying code.

Goals of this work

- **Predictability**: an algorithm for subtype tests in constant time and (almost) constant space
- **Incrementality**: Java allows dynamic class loading, any subtype test implementation must thus support incremental type hierarchy updates

Definitions

A <: B holds if A is a subtype of B

isSubtype(A,B) returns true if A <: B

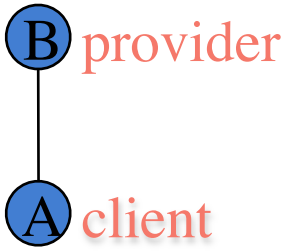
checkCast(A,B) throws an exception if not A<:B

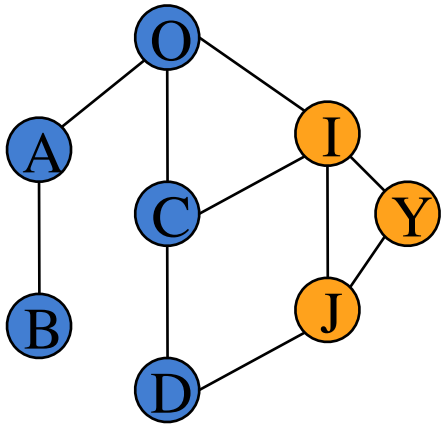
extends(A,B) iff A <: B and B is a class

implements(A, B) iff A <: B and B is an interface

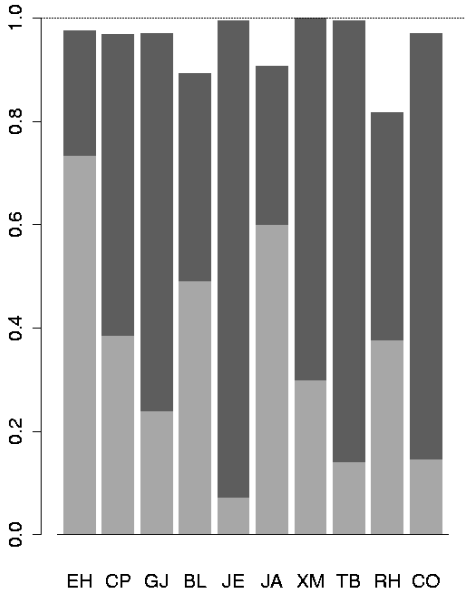
a subtype test is **successful** if isSubtype returns true or if checkCast does not throw an exception

A <: A is called a selftest (always true)

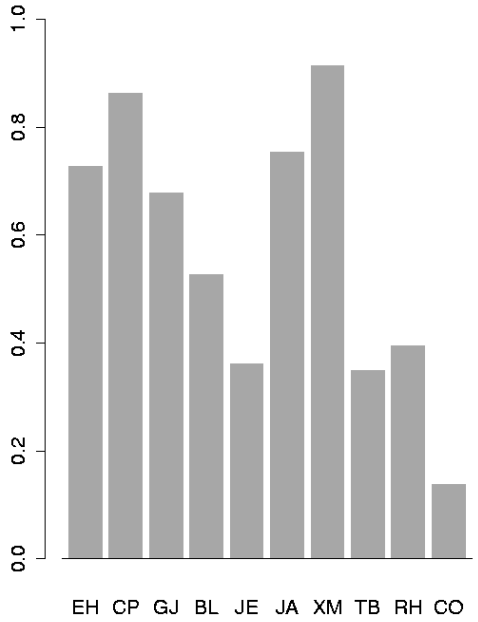




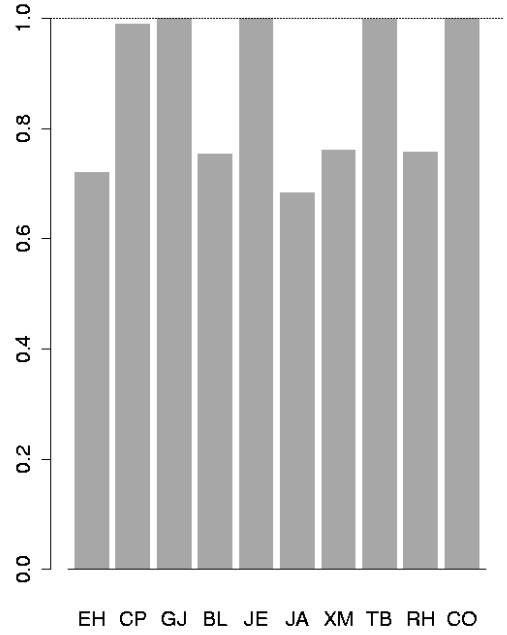
Dynamic characteristics of ST



Success rate



Checkcast rate



Extends rate

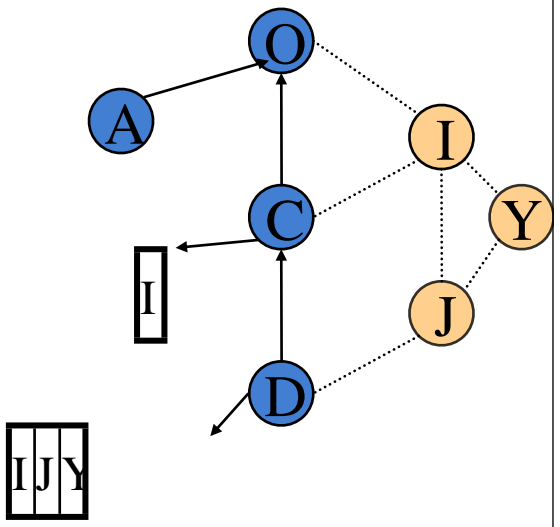
as % of dynamic tests

Observations

- Tests are mostly successful
  - A failed checkcast is often the last thing you do
  - Of course some programming styles invalidate this
- Selftests excepted, most types never in provider position
- Tests mostly `extends` tests
- `checkast` are highly dependent on programming style

Hierarchy traversal

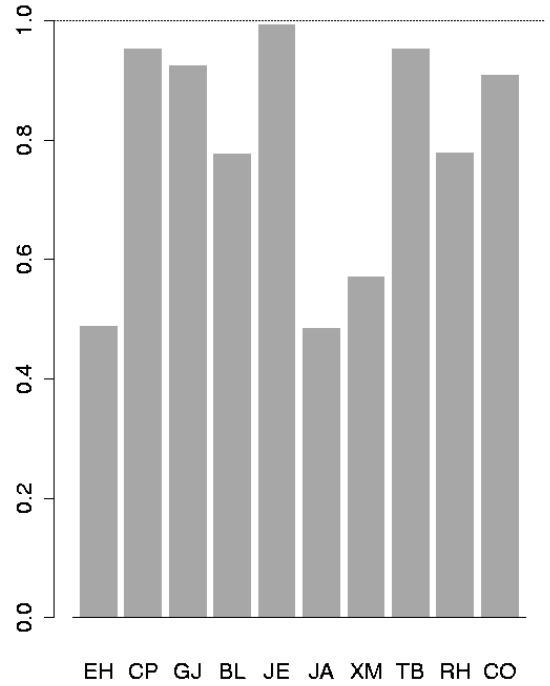
```
bool isSubtype(type cl, type pr) {
    if (cl.cache==pr) return true;
    if (isInterface(pr)) return extends(cl, pr);
    else return implements(cl, pr); }


bool implements(type cl, type pr) {
  for (int i=0; i < pr.interfaces.length; i++)
    if (cl==pr.interfaces[i]) {cl.cache=pr;return true;}
  return false; }

bool extends(type cl, type pr) {
    for (type sp=cl.parent; sp!=null; sp=sp.parent)
      if (sp==pr) {cl.cache=pr; return true;}
    return false; }
```



Truth in advertising

- The real subtype test procedure a bit more complex:
  - Cache test is JITed at the call site
  - Some interfaces may not be loaded
  - Arrays require comparison of element types
  - Some interfaces are treated specially (Cloneable and Serializable)
  - EVM uses two cache slots, one for array stores and one for other tests
  - `null instanceof A` throws an exception while `(A)null` is successful, go figure…

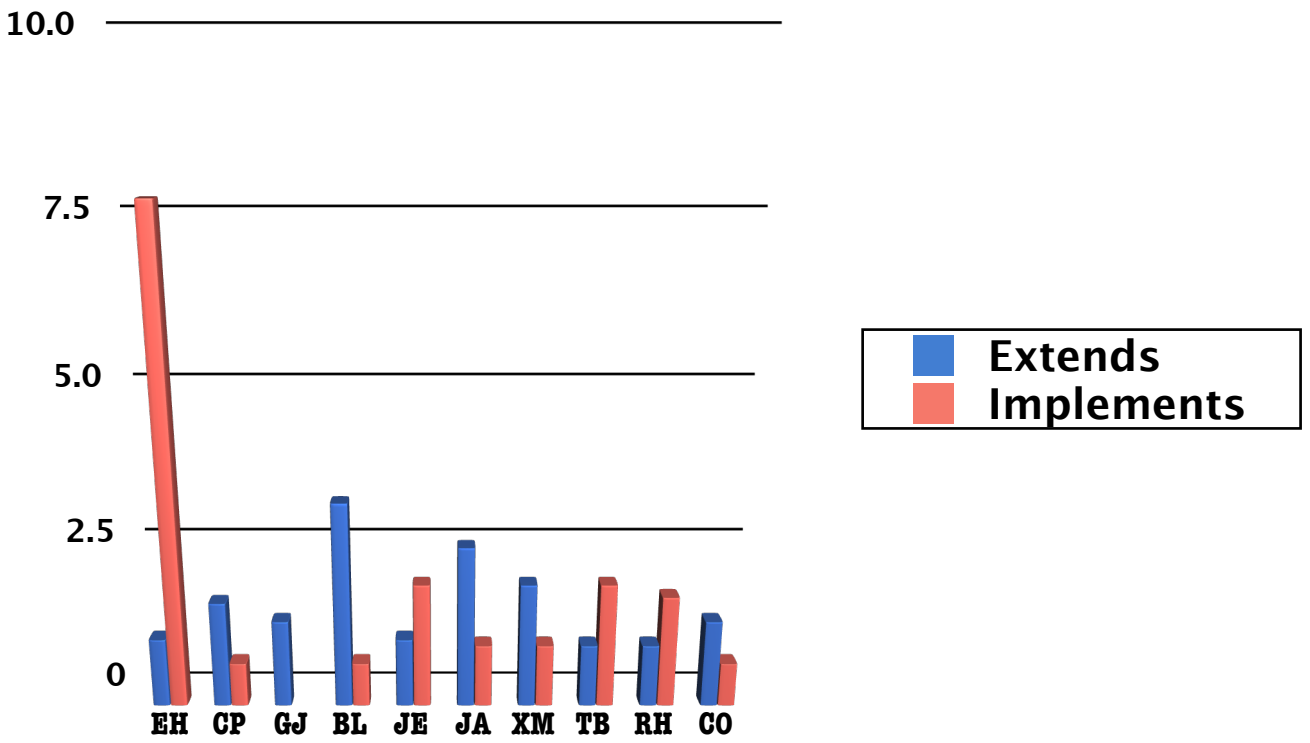
Evaluating the cache



Cache hit rates

as % of dynamic tests

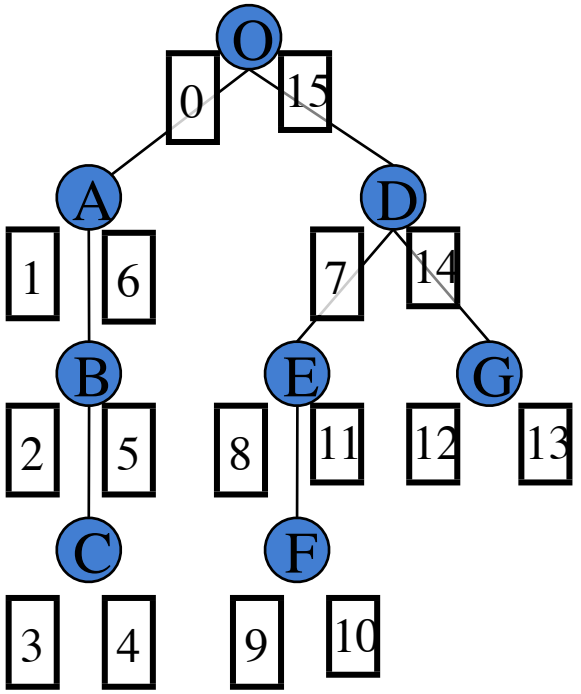
Hierarchy traversal cost



Avg number of iterations of extends and implements loop without a cache.

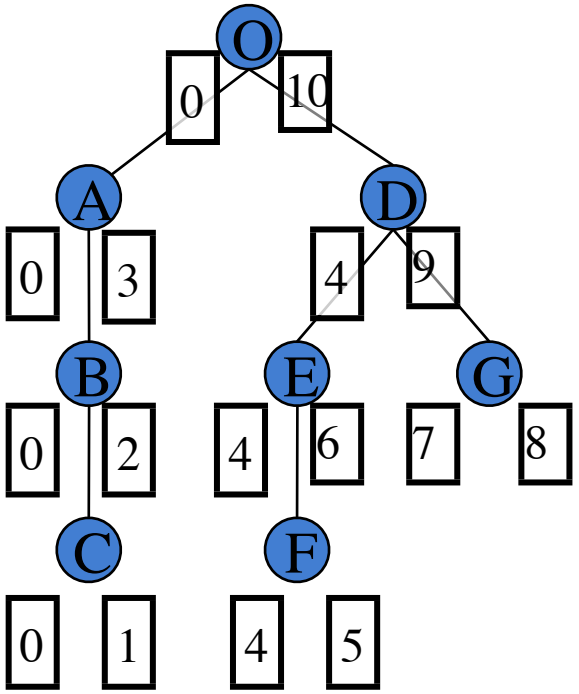
Constant time/space Extends

- Relative numbering
  - A.low < B.low && B.high < A.high



Extends

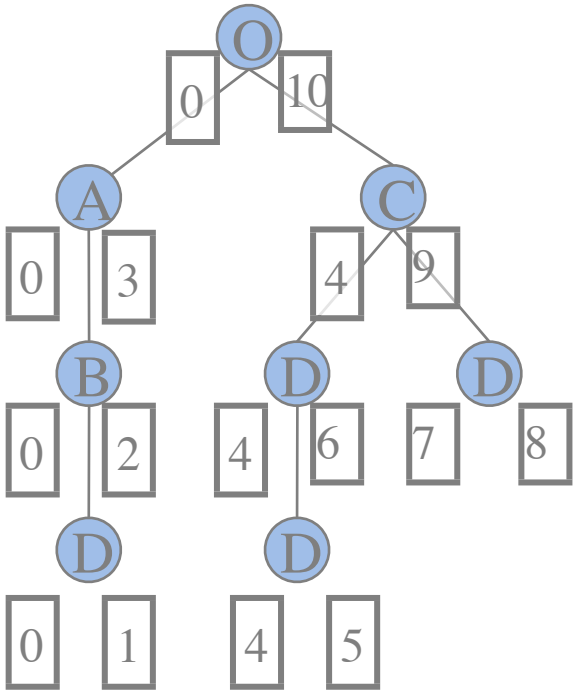
```
bool extends(type cl, type pr) {

    if (pr.low <= cl.low) {
         if (cl.low < pr.high)
                  return true;
    }
    return false;
}
```



Incremental computation

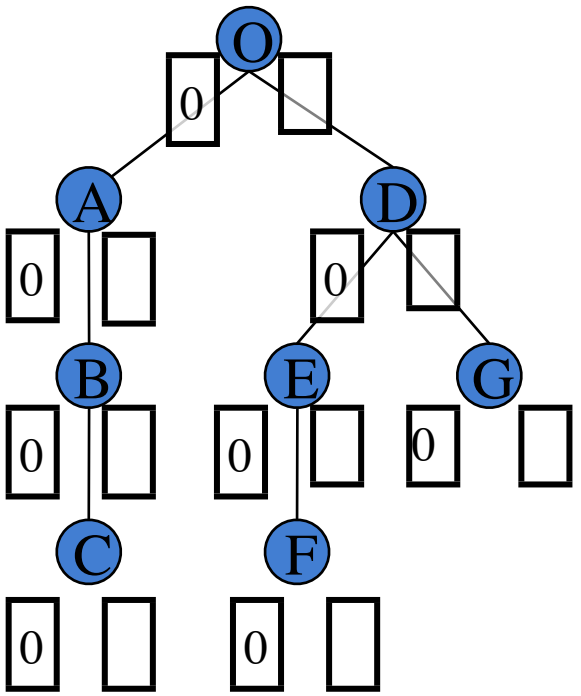
- Recall:
  - Most types are not in provider position
  - Success is the common case
- Define:

```
bool isInvalid(type cl) {
     return (cl.high == -1);
}
```



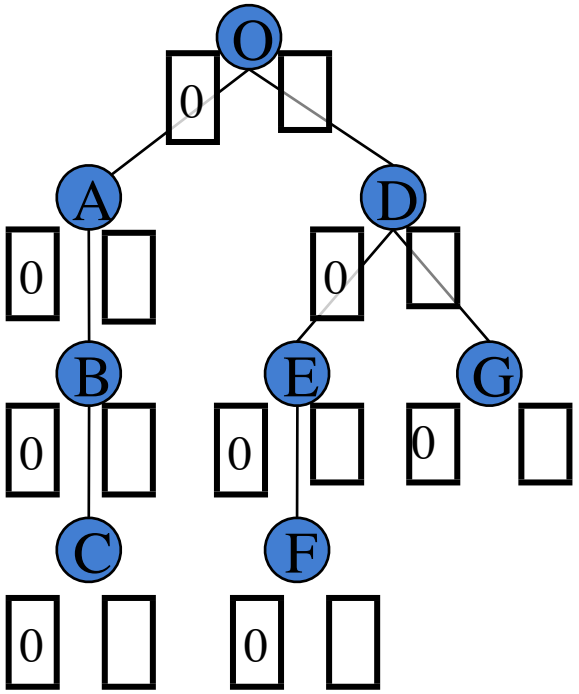
Incremental computation

```
bool extends(type cl, type pr) {
    if (pr == cl) return true;
    if (pr.low <= cl.low) {
         if (cl.low < pr.high)
                  return true;
         if (isInvalid(pr)) {
           recompute(pr);
           return extends(cl, pr);
        }
    }
    return false;
}
```



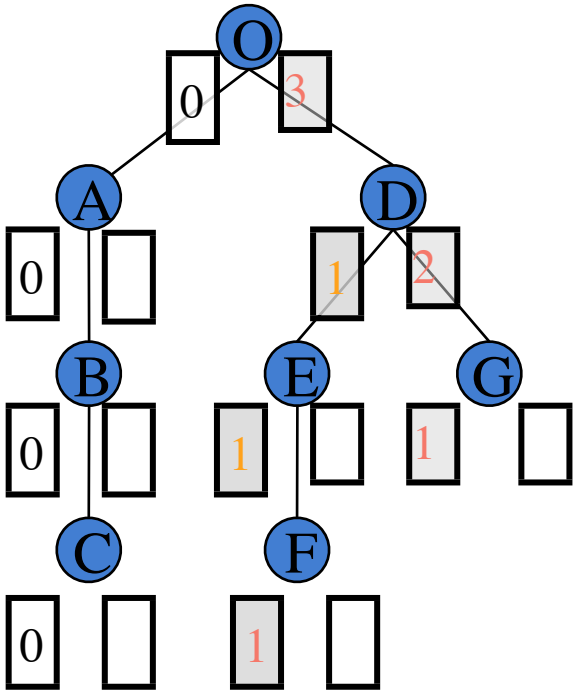
Incremental computation

```
extends(F,D)  ?
```



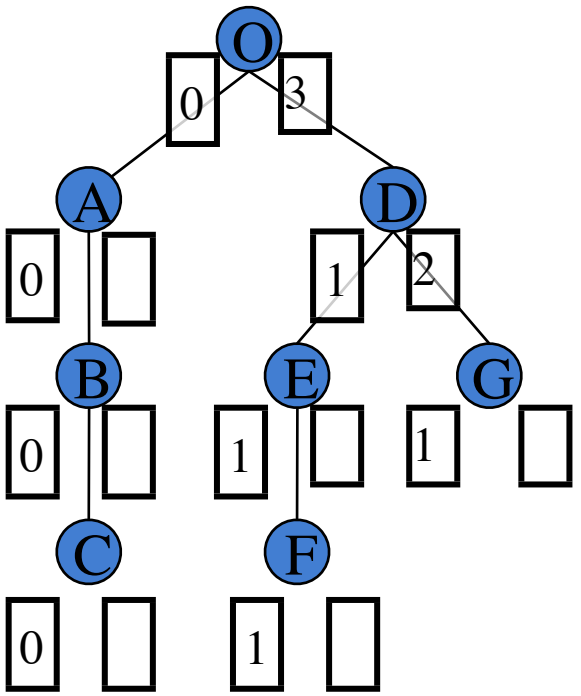
Incremental computation

`extends(F,D) ?`



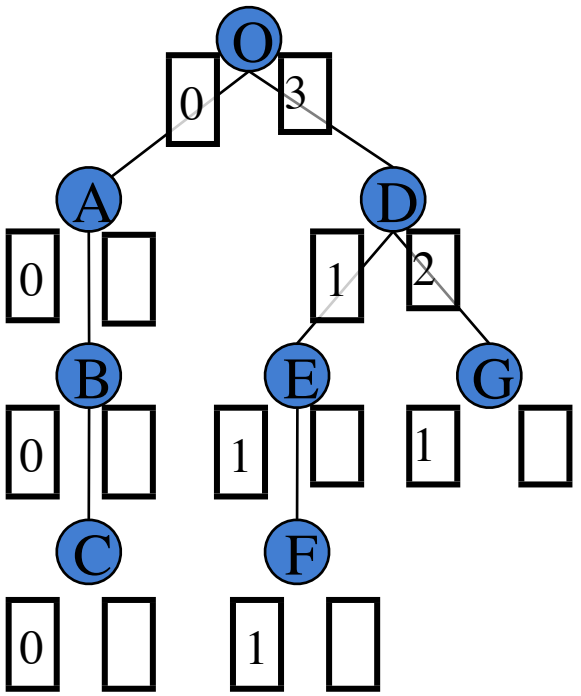
Incremental computation

```
extends(F,D)
extends(G,D)  ?
```



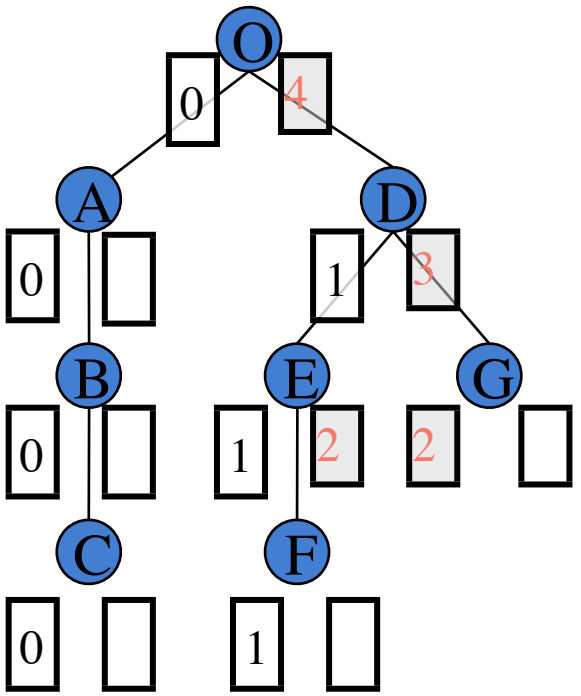
Incremental computation

```
extends(F,D)
extends(G,D)
extends(F,E)?
```



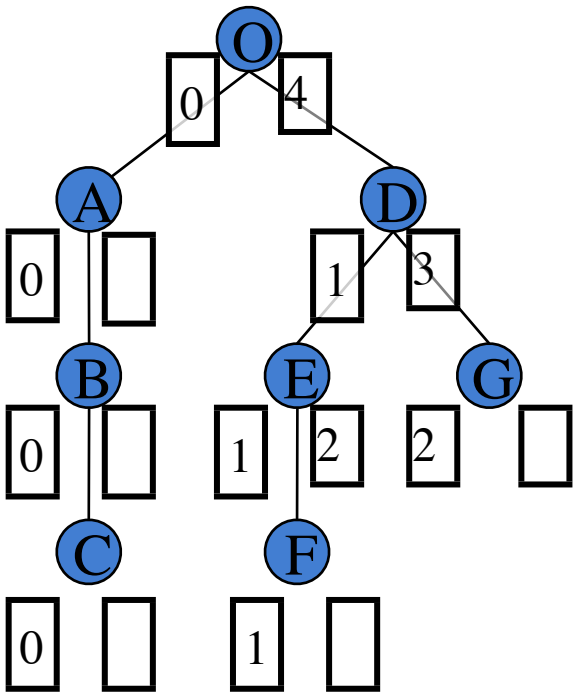
Incremental computation

```
extends(F,D)
extends(G,D)
extends(F,E)
extends(C,C)?
```



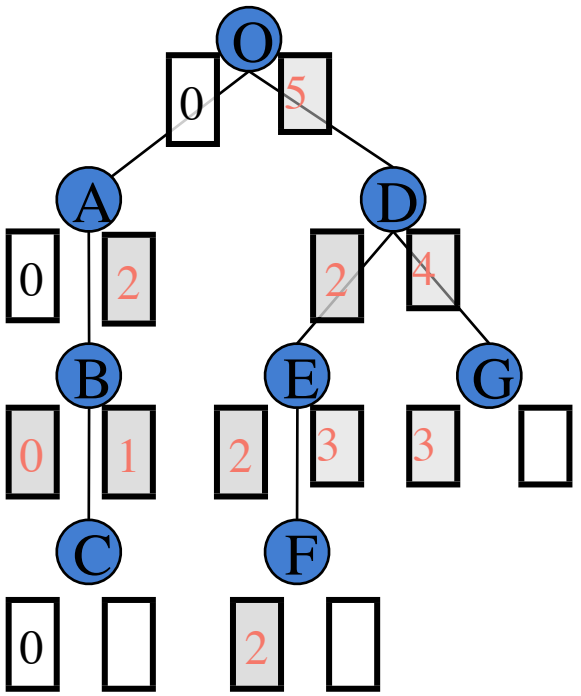
Incremental computation

```
extends(F,D)
extends(G,D)
extends(F,E)
extends(C,C)
extends(C,B)?
```



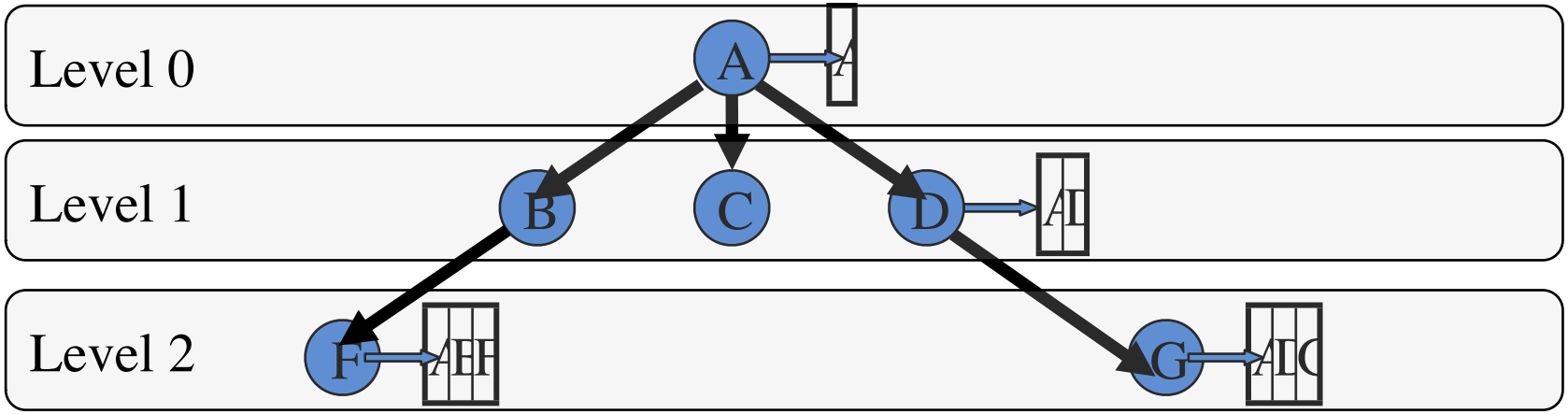
Incremental computation

```
extends(F,D)
extends(G,D)
extends(F,E)
extends(C,C)
extends(C,B)
```



Implements

Cohen's displays for single inheritance…



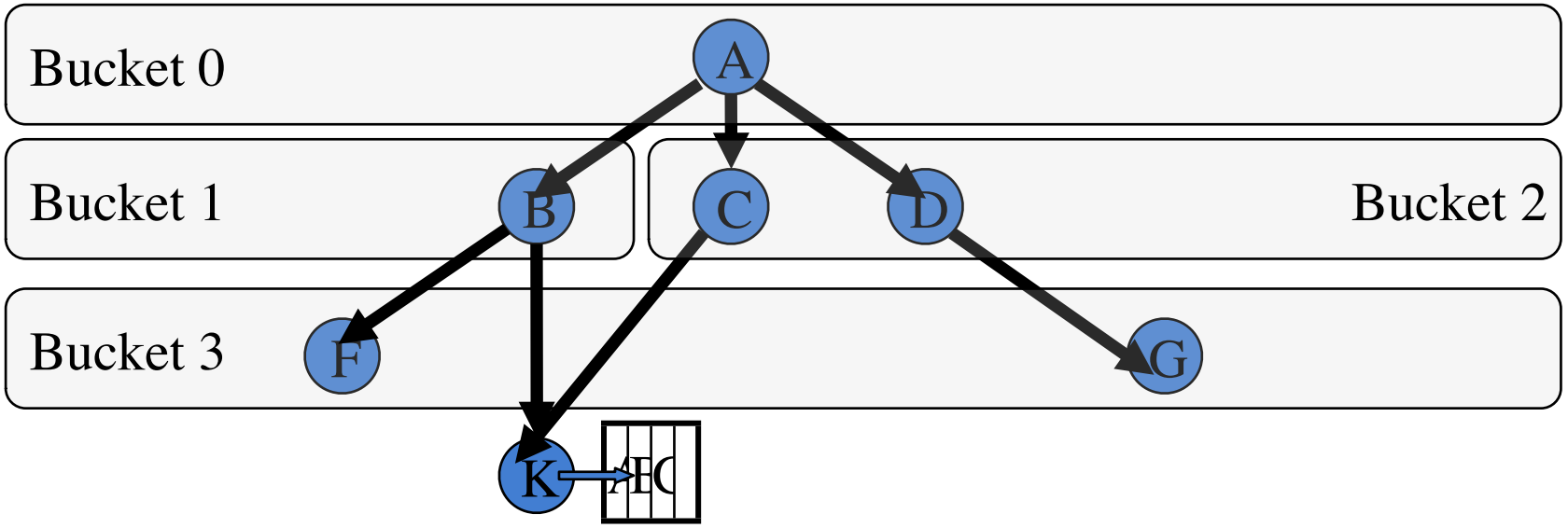
X <: Y

iff

X.display.length < Y.level && X.display[Y.level] == Y.id

Multiple subtyping

Interfaces in the same bucket have no common descendants




```
X <: Y
```

iff

```
X.display[Y.bucket_num] == Y.id
```

Display Updates

- Maintain invariant:

  `I.bucket_num == J.bucket_num`

  ⇔

  no `C` such that `C<:I && C<:J`

- When loading class **`A`**

  Foreach **`I:`** if **`A<:I`** do

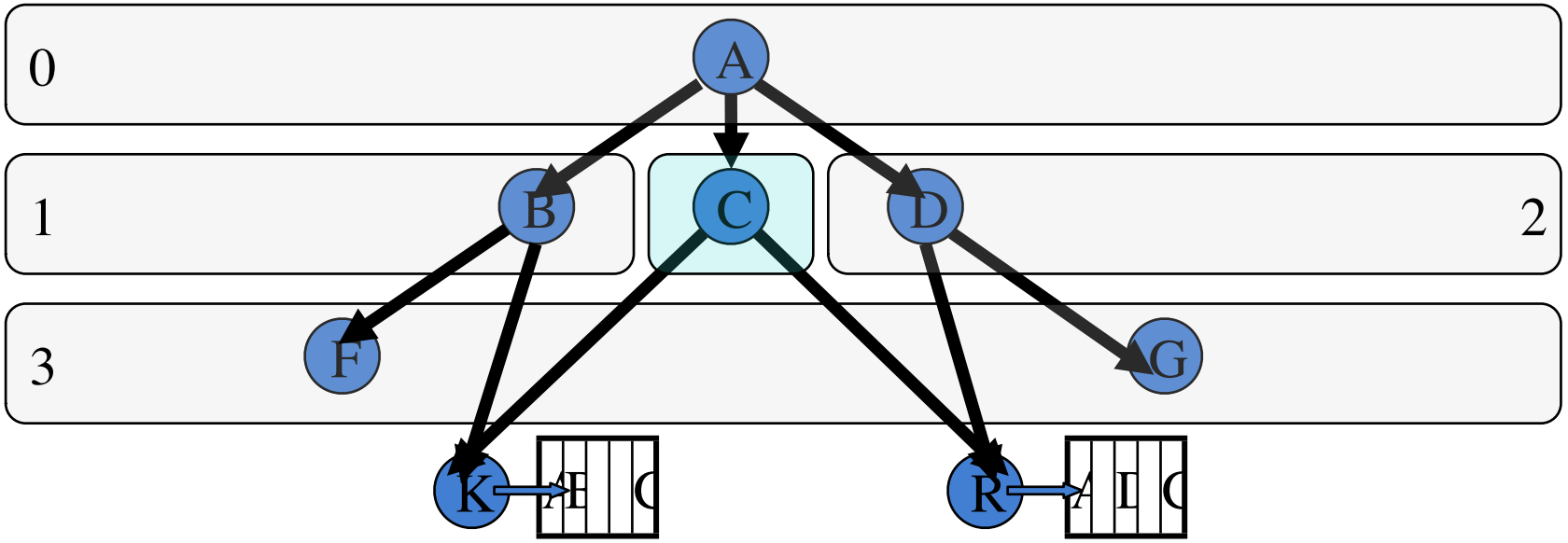
  `mark[I.bucket_num]++`

  Foreach `b:` if **`mark[b_num]==k && k<1`** do

  split **`b`** into **`k`** buckets of same size,
  assign **`A`**'s interfaces to different buckets

  recompute displays

Class Loading



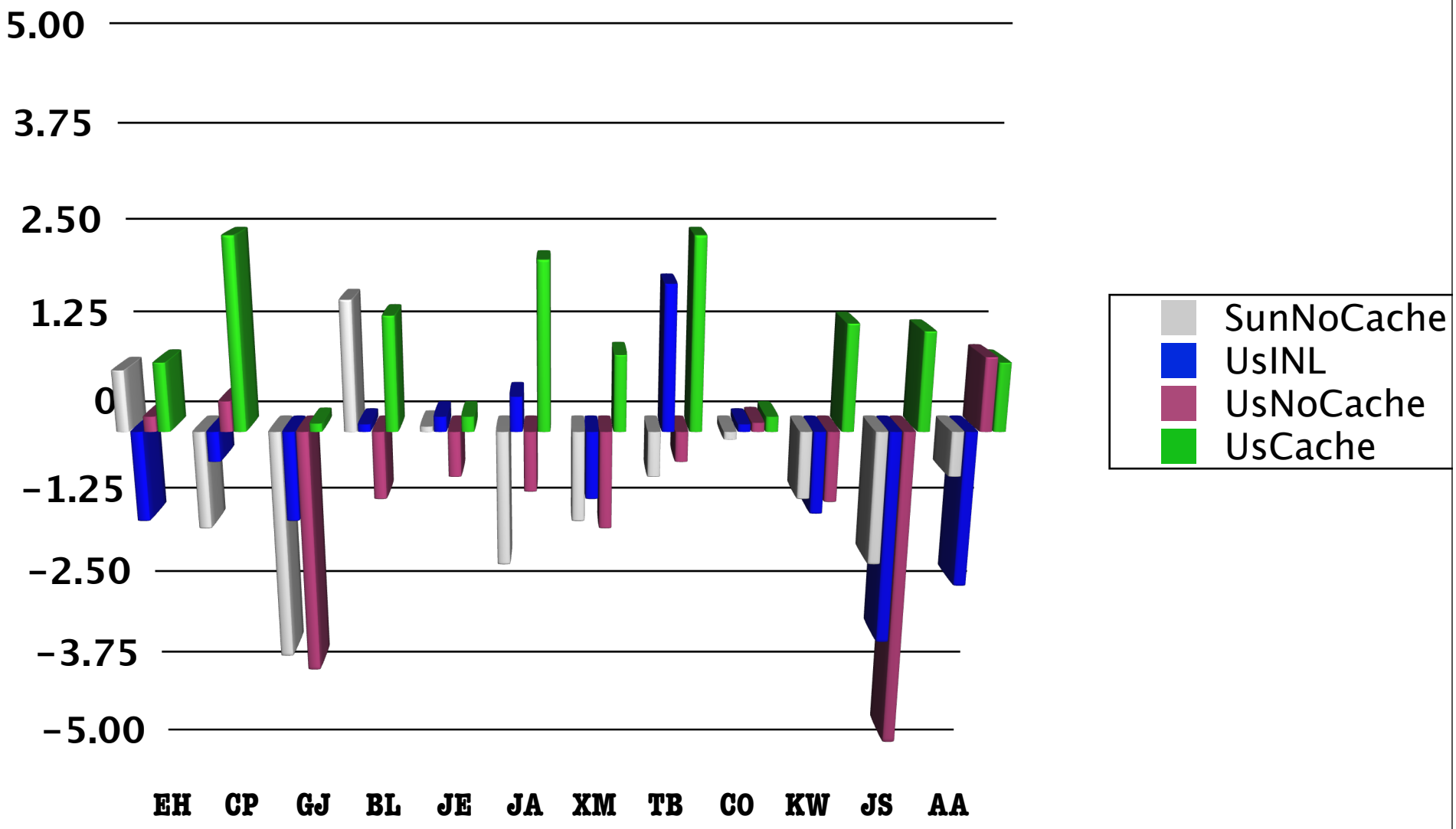
Implements, ct'd

- When loading interfaces: which bucket to add to?
  - choose smallest bucket among `m` most recently created buckets (for us `m=5`)
  - heuristics: avoids putting system interfaces together with application-defined interfaces while attempting to balance bucket sizes (likely to conflict)
- Simple but good enough (display size only twice maximum number of implemented interfaces)

Summary

- We have presented two simple techniques for dealing with extends and implements tests
- Some heuristics are particularly tuned to the characteristics of Java
- We have implemented these techniques in the SUN Research VM (aka EVM)

Results


5.00
3.75
2.50
1.25
0
-1.25
-2.50
-3.75
-5.00
EH
CP
GJ
BL
JE
JA
XM
TB
CO
KW
JS
AA
SunNoCache
UsINL
UsNoCache
UsCache