VIRTUAL EXECUTION ENVIRONMENTS

Jan Vitek

PURDUE UNIVERSITY

with material from Nigel Horspool and Jim Smith

(S³)

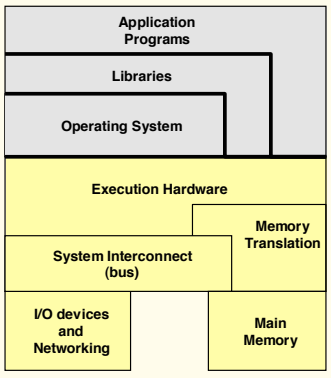
Virtualization

The "Machine"

- Different perspectives on what the *Machine* is:
- OS developer

Instruction Set Architecture

- ISA
- Major division between hardware and software



VEE '05 (c) 2005, J. E. Smith 8

- slides from Jim Smith's talk at VEE'05

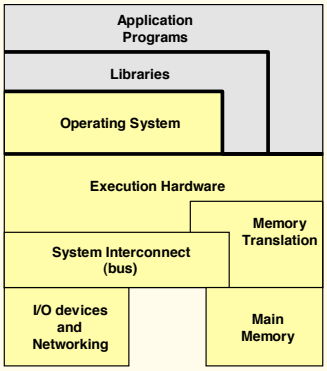
Virtualization

The "Machine"

- Different perspectives on what the *Machine* is:
- Compiler developer

Application Binary Interface

- ABI
- User ISA + OS calls



VEE '05 (c) 2005, J. E. Smith 9

- slides from Jim Smith's talk at VEE'05

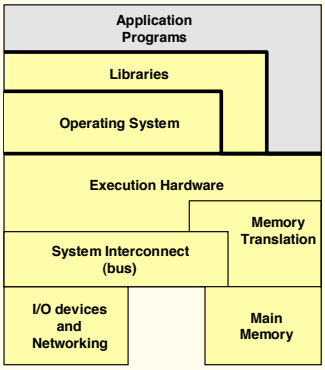
Virtualization

The "Machine"

- Different perspectives on what the *Machine* is:
- Application programmer

Application Program Interface

- API
- User ISA + library calls



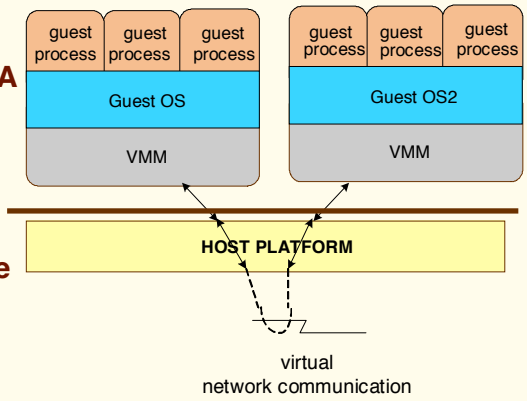
VEE '05 (c) 2005, J. E. Smith 10

- slides from Jim Smith's talk at VEE'05

Virtualization

System Virtual Machines

- Provide a system environment
- Constructed at ISA level
- Persistent
- Examples: IBM VM/360, VMware, Transmeta Crusoe



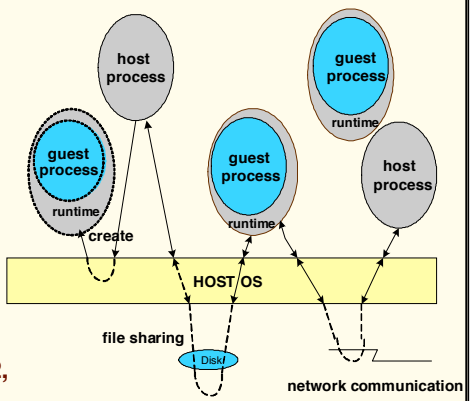
VEE '05 (c) 2005, J. E. Smith 11

- slides from Jim Smith's talk at VEE'05

Virtualization

Process Virtual Machines

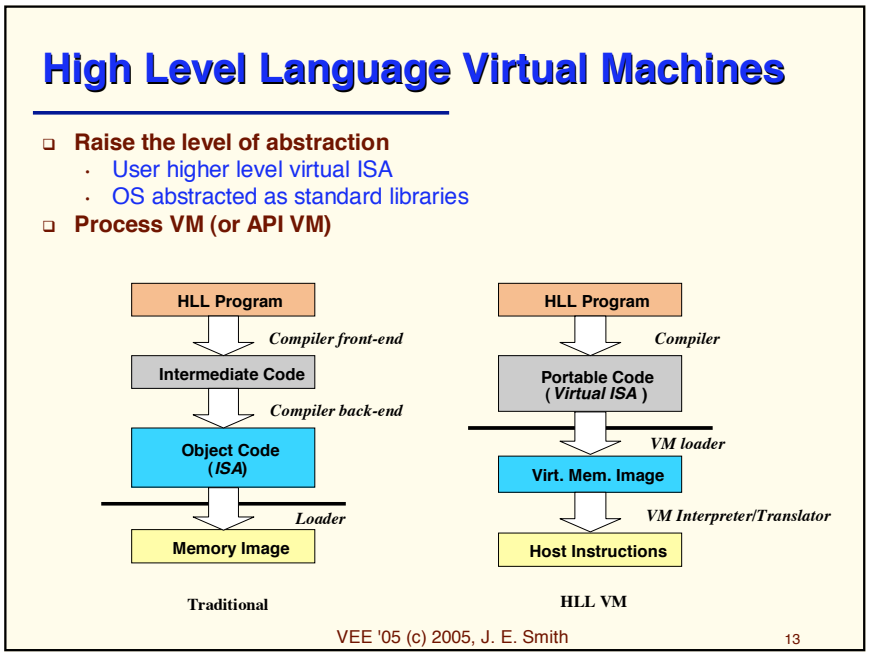
- Constructed at ABI level
- *Runtime* manages guest process
- Not persistent
- Guest processes may intermingle with host processes
- As a practical matter, guest and host OSes are often the same
- Dynamic optimizers are a special case
- Examples: IA-32 EL, FX!32, Dynamo



VEE '05 (c) 2005, J. E. Smith 12

- slides from Jim Smith's talk at VEE'05

Virtualization

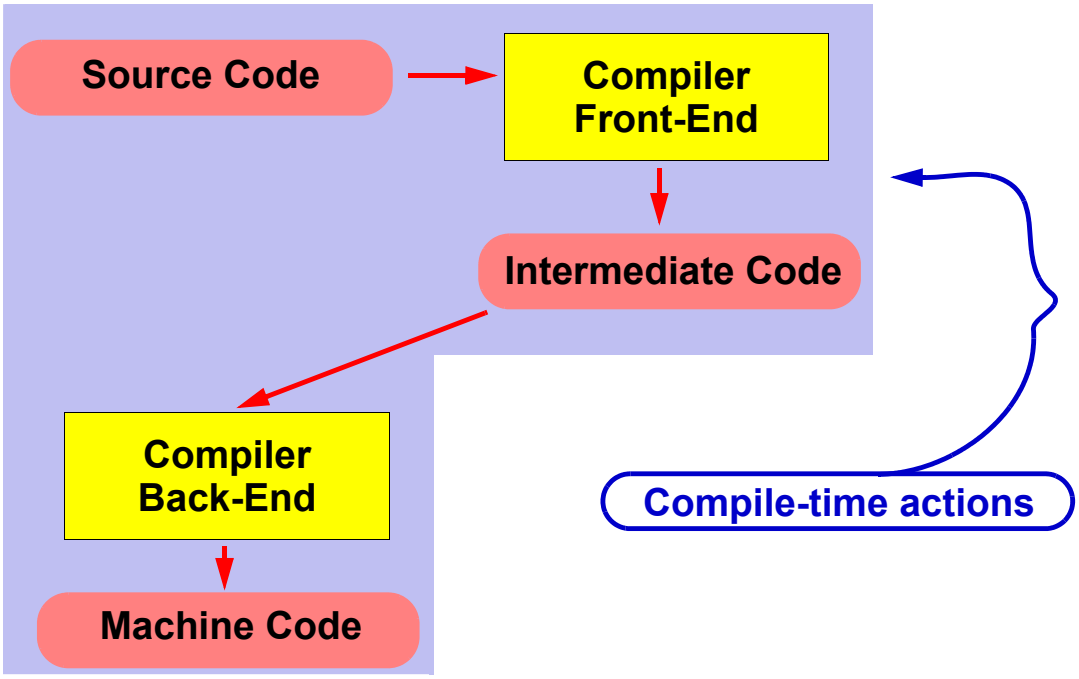


- slides from Jim Smith's talk at VEE'05

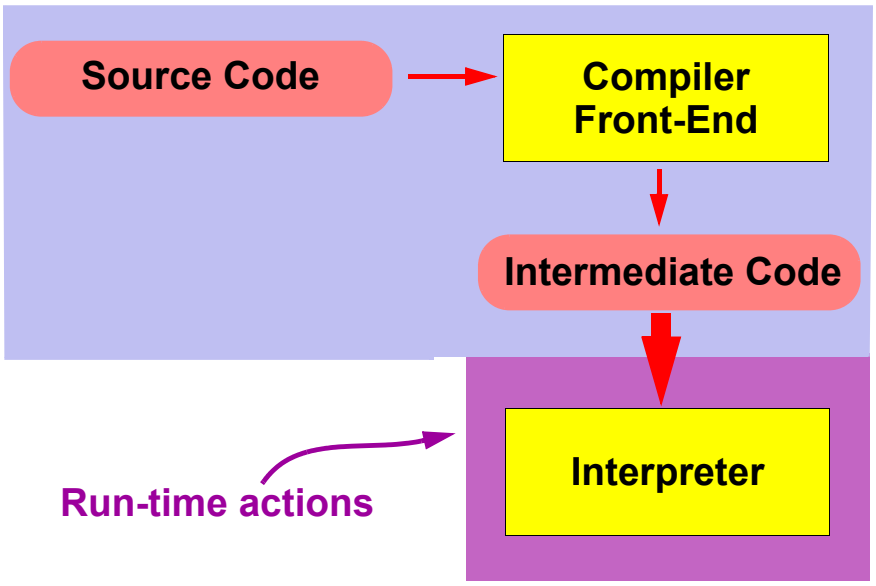
Implementation of Virtual Machines

Introduction

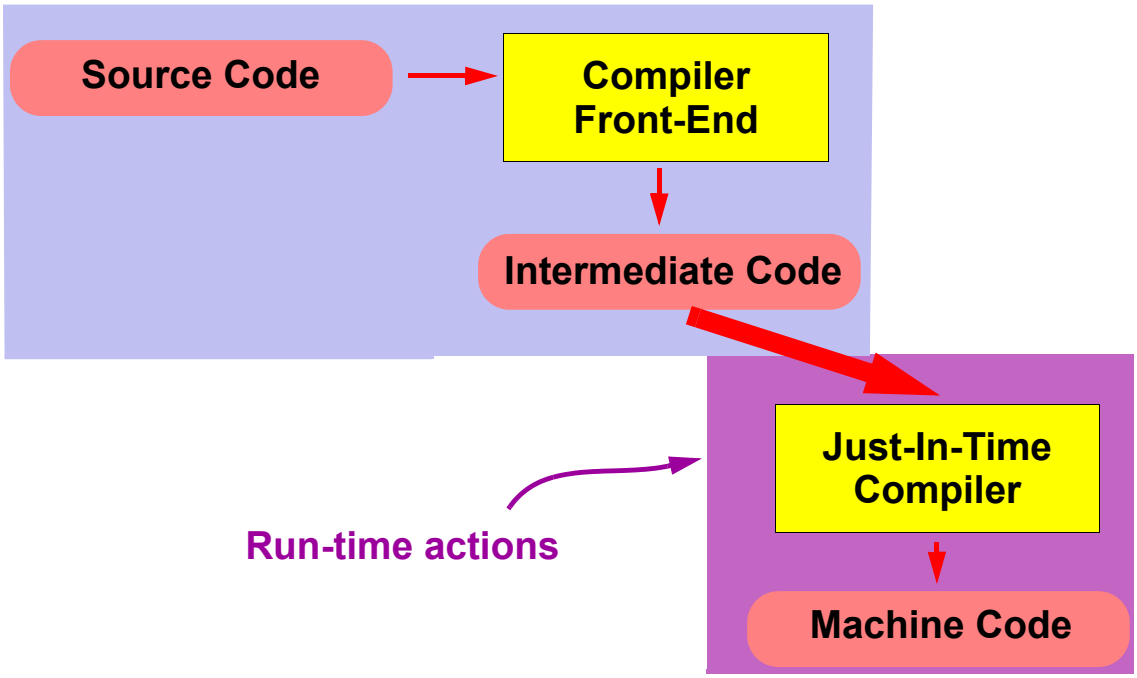
Usual Programming Language Implementation



Another Programming Language Implementation



And Another Implementation



An Overview

- Source code is translated into an intermediate representation, (IR)
- The IR can be processed in these different ways:
  1. compile-time (static) translation to machine code
  2. emulation of the IR using an interpreter
  3. run-time (dynamic) translation to machine code = JIT (Just-In-Time) compiling

What is IR?

IR is code for an idealized computer, a *virtual machine*.

Examples:

Language	IR	Implementation(s)
Java	JVM bytecode	Interpreter, JIT
C#	MSIL	JIT (but may be pre-compiled)
Prolog	WAM code	compiled, interpreted
Forth	bytecode	interpreted
Smalltalk	bytecode	interpreted
Pascal	p-code --	interpreted compiled
C, C++	--	compiled (usually)
Perl 6	PVM Parrot	interpreted interpreted, JIT
Python	--	interpreted
sh, bash, csh	original text	interpreted

Toy Bytecode File Format

We need a representation scheme for the bytecode. A simple one is:

- to use one byte for an opcode,
- four bytes for the operand of LDI,
- two bytes for the operands of LD, ST, JMP and JMPF.

As well as 0 for STOP, we will use this opcode numbering:

LDI	LD	ST	ADD	SUB	EQ	NE	GT	JMP	JMPF	READ	WRITE
1	2	3	4	5	6	7	8	9	10	11	12

The order of the bytes in the integer operands is important. We will use *big-endian* order.

The Classic Interpreter Approach

It emulates the fetch/decode/execute stages of a computer.

```
for( ; ; ) {
    opcode = code[pc++];
    switch(opcode) {
      case LDI:
          val = fetch4(pc);  pc += 4;
          push(val);
          break;
      case LD:
          num = fetch2(pc);  pc += 2;
          push( variable[num] );
          break;
      ...
      case SUB:
          right = pop();  left = pop();
          push( right-left );
      ...
```


The Classic Interpreter Approach, cont'd

```
      case JMP:
          pc = fetch2(pc);
          break;
      case JMPF:
          val = pop();
          if (val)
              pc += 2;
          else
              pc = fetch2(pc);
          break;
      ...
    }  /* end of switch */
}  /* end of for loop */
```

Critique

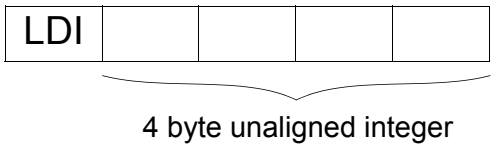
- The classic interpreter is easy to implement.
- It is flexible – it can be extended to support tracing, profiling, checking for uninitialized variables, debugging, ... anything.
- The size of the interpreter plus the bytecode is normally much less than the equivalent compiled program.
- But interpretive execution is *slow* when compared to a compiled program.

  The slowdown is 1 to 3 orders of magnitude (depending on the language).

What can we do to speed up our interpreter?

Improving the Classic Interpreter

1. Verification – verify that all opcodes and all operands are valid before beginning execution, thus avoiding run-time checks. We should also be able to verify that stacks cannot overflow or underflow.
2. Avoid unaligned data.



3. We can eliminate one memory access per IR instruction by expanding opcode numbers to addresses of the opcode implementations ...

Classic Interpreter with Operation Addresses

The bytecode file ... as in our example

READ; ST 0; READ; ST 1; LD 0; LD 1; NE; JMPF 54; LD 0; LD 1; GT; JMPF 41; LD 0; LD 1; SUB; ST 0; JMP 51; LD 1; LD 0; SUB; ST 1; JMP 8; LD 0; WRITE; STOP

would be expanded into the following values when loaded into the interpreter's bytecode array.

&READ	&ST	0	&READ	&ST	1	&LD	...

and so on.

Each value is a 4 byte address or a 4-byte operand.

Classic Interpreter, cont'd

Now the interpreter dispatch loop becomes:

```
        pc = 0;   /* index of first instruction */
DISPATCH:
        goto *code[pc++];

LDI:
        val = *code[pc++];
        push(val);
        goto DISPATCH;
LD:
        num = *code[pc++];
        push( variable[num] );
        goto DISPATCH;
        ...
```

The C code can be a bit better still ...

Classic Interpreter, cont'd

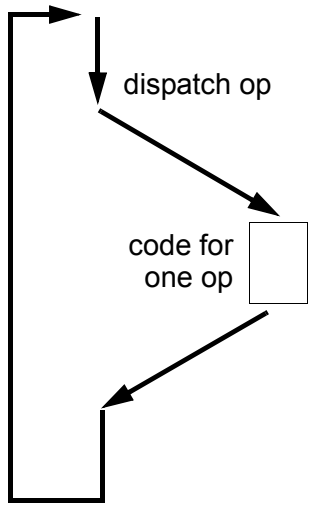
Recommended C style for accessing arrays is to use a pointer to the array elements, so we get:

```
        pc = &code[0];   /* pointer to first instruction */
DISPATCH:
        goto *pc++;
LDI:
        val = *pc++;
        push(val);
        goto DISPATCH;
LD:
        num = *pc++;
        push( variable[num] );
        goto DISPATCH;
        ...
```

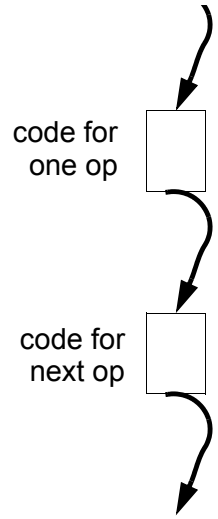
But let's step back and see a new technique –

(Direct) Threaded Code Interpreters

Reference: James R. Bell, Communications of ACM 1973



Classic Interpreter



Threaded Code Interpreter

Threaded Code Interpreters, cont'd

As before the bytecode is a sequence of addresses (intermixed with operands needed by the ops) ...

&LDI	99	&LDI	23	&ADD	&ST	5	...

The interpreter code looks like this ...

```
    /* start it going */
    pc = &code[0];
    goto *code[pc++];

LDI:
    operand = (int)*pc++;
    push(operand);
    goto *code[pc++];
```

```
ADD:
    right = pop();
    left = pop();
    push(left+right);
    goto *code[pc++];

    ...
```


Threaded Code Interpreters, cont'd

As before, better C style is to use a *pointer* to the next element in the code ...

```
    /* start it going */
    pc = &code[0];
    goto *(*pc++);

LDI:
    operand = (int)(*pc++);
    push(operand);
    goto *(*pc++);
```

```
ADD:
    right = pop();
    left = pop();
    push(left+right);
    goto *(*pc++);

    ...
```

This makes the implementation very similar to Bell's, who programmed for the DEC PDP11.

Further Improvements to Interpreters ...

A problem still being researched. (See the papers in the IVME annual workshop.)

Speed improvement ideas include:

1. Superoperators (see Proebsting, POPL 1995)
2. Stack caching (see Ertl, PLDI 1995)
3. Inlining (see Piumarta & Riccardi, PLDI 1998)
4. Branch prediction (see Ertl & Gregg, PLDI 2003)

Space improvement ideas (for embedded systems?) include:

1. Huffman compressed code (see Latendresse & Feeley, IVME 2003)
2. Superoperators – if used carefully (ibid)

The Java Virtual machine

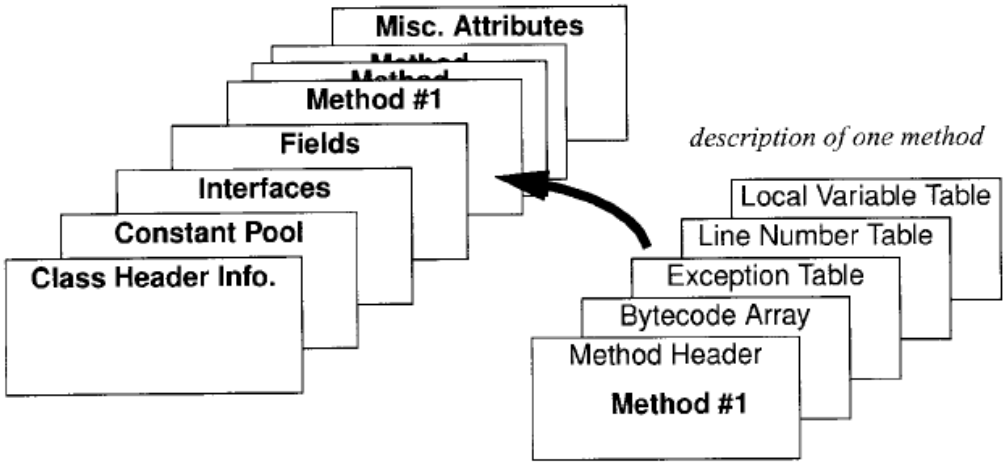
A Main Reference Source

The Java™ Virtual Machine Specification (2nd Ed)
by Tim Lindholm & Frank Yellin
Addison-Wesley, 1999

The book is on-line and available for download:

`http://java.sun.com/docs/books/vmspec/`

The Java Classfile



JVM Runtime Behaviour

- VM startup
- Class Loading/Linking/Initialization
- Instance Creation/Finalisation
- Unloading Classes
- VM exit

VM Startup and Exit

Startup

- Load, link, initialize class containing `main()`
- Invoke `main()` passing it the command-line arguments
- Exit when:
  - all non-daemon threads end, or
  - some thread explicitly calls the `exit()` method

Class Loading

- Find the binary code for a class and create a corresponding Class object
- Done by a class loader – bootstrap, or create your own
- Optimize: prefetching, group loading, caching
- Each class-loader maintains its own namespace
- Errors include: `ClassFormatError`, `UnsupportedClassVersionError`, `ClassCircularityError`, `NoClassDefFoundError`

Class Loaders

- System classes are automatically loaded by the bootstrap class loader
- To see which:

```
java -verbose:class Test.java
```

- Arrays are created by the VM, not by a class loader
- A class is unloaded when its class loader becomes unreachable (the bootstrap class loader is never unreachable)

Class Linking - 1. Verification

- Extensive checks that the .classfile is valid
- This is a vital part of the JVM security model
- Needed because of possibility of:
  - buggy compiler, or no compiler at all
  - malicious intent
  - (class) version skew
- Checks are independent of compiler and language

Class Linking - 2. Preparation

- Create static fields for a class
- Set these fields to the standard default values (N.B. not explicit initializers)
- Construct method tables for a class
- ... and anything else that might improve efficiency

Class Linking - 3. Resolution

- Most classes refer to methods/fields from other classes
- Resolution translates these names into explicit references
- Also checks for field/method existence and whether access is allowed

Class Initialization

Happens *once* just before first instance creation, or first use of static variable.

- Initialise the superclass first!
- Execute (class) static initializer code
- Execute explicit initializers for static variables
- May not need to happen for use of *final* static variable
- Completed before anything else sees this class

Instance Creation/Finalisation

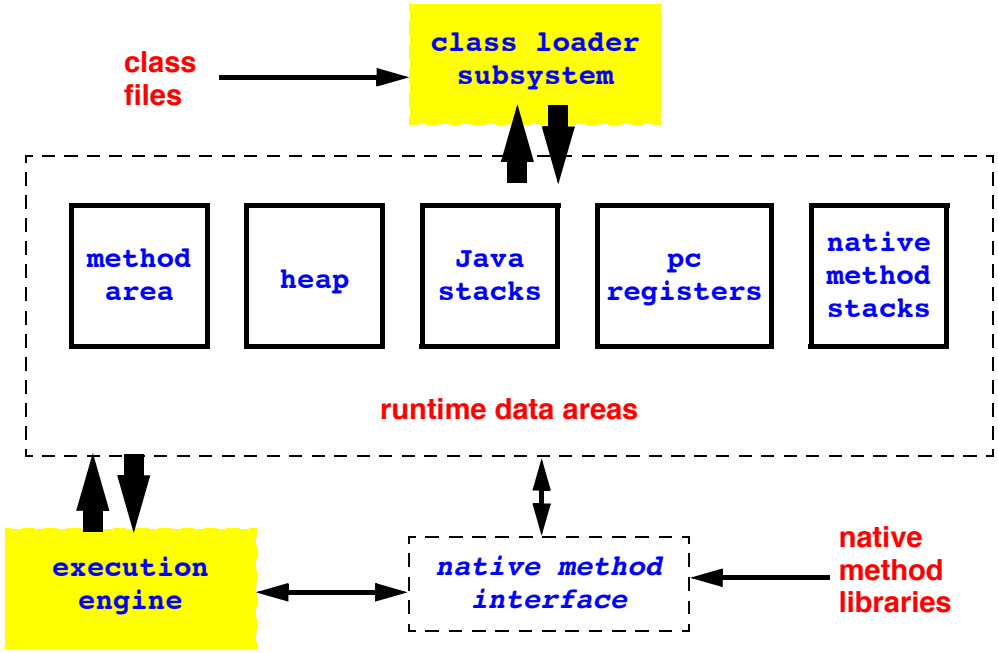
- Instances are created using `new`, or `newInstance()` from class `Class`
- Instances of `String` may be created (implicitly) for String literals
- Process:
  1. Allocate space for all the instance variables (including the inherited ones),
  2. Initialize them with the default values
  3. Call the appropriate constructor (do parent's first)
- `_ finalize()` is called just before garbage collector takes the object (so timing is unpredictable)

JVM Architecture

The internal runtime structure of the JVM consists of:

- One: (i.e. shared by all threads)
  - method area
  - heap
- For each thread, a:
  - program counter (pointing into the method area)
  - Java stack
  - native method stack (system dependent)

Run-Time Data Areas (Venners Figure 5-1)



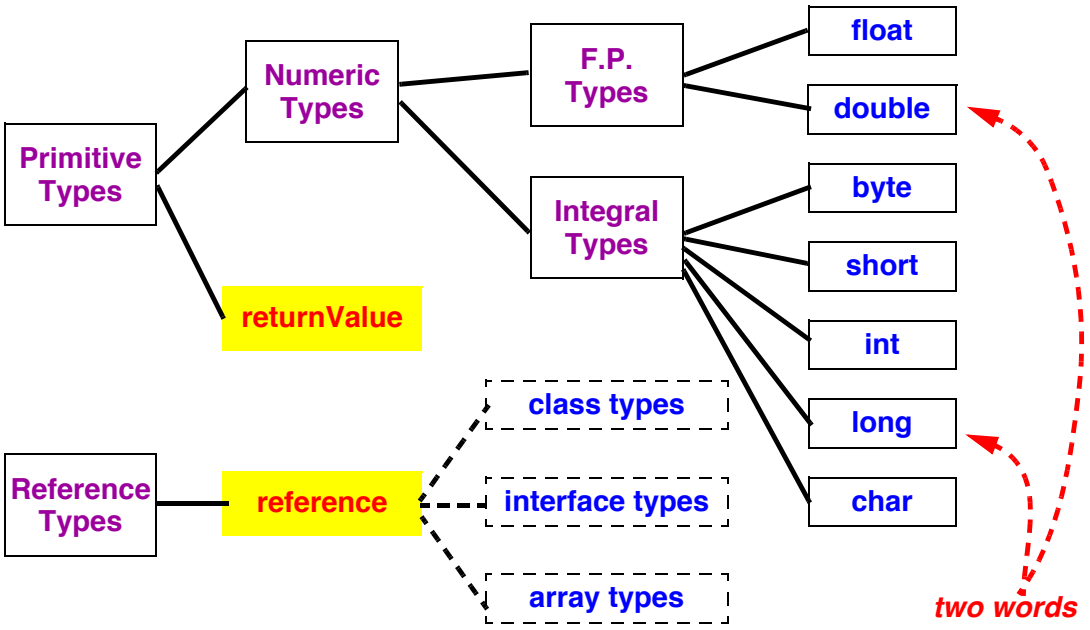
Java Bytecode

Java Intermediate Bytecode

- By James Gosling; presented at IR'95.

- Quick overview:
  - argue for the presence of type information in the bytecode
  - benefits for checkability (because speed/security)
  - reduced dependencies on environment

Datatypes of the JVM (Venners 5-4)



Control Transfer

- ifeq, iflt, ifle, ifne, ifgt, ifge
- ifnull, ifnonnull
- if_icmpeq, if_icmplt, if_icmple, if_icmpne, if_icmpgt, if_icmpge
- if_acmpeq, if_acmpne
- goto, goto_w, jsr, jsr_w, ret

Switch statement implementation

- tableswitch, lookupswitch

Comparison operations for long, float & double types

- lcmp, fcmpl, fcmpg, dcmpl, dcmpg

Load and Store Instructions

Transferring values between local variables and operand stack

- iload, lload, fload, dload, aload
  and special cases of the above: iload_0, iload_1 ...
- istore, lstore, fstore, dstore, astore

Pushing constants onto the operand stack

- bipush, sipush, ldc, ldc_w, ldc2_w, aconst_null, iconst_m1
  and special cases: iconst_0, iconst_1, ...

Arithmetic Operations

Operands are normally taken from operand stack and the result pushed back there

- iadd, ladd, fadd, dadd
- isub ...
- imul ...
- idiv ...
- irem ...
- ineg ...
- iinc

Bitwise Operations

- ior, lor
- iand, land
- ixor, lxor
- ishl, lshl
- ishr, iushr, lshr, lushr

Type Conversion Operations

Widening Operations

- i2l, i2f, i2d, l2f, l2d, f2d

Narrowing Operations

- i2b, i2c, i2s, l2i, f2i, f2l, d2i, d2l, d2f

Operand Stack Management

- pop, pop2
- dup, dup2, dup_x1, dup_x2, dup2_x2, swap

Object Creation and manipulation

- new
- newarray, anewarray, multinewarray
- getfield, putfield, getstatic, putstatic
- baload, caload, saload, iaload, laload, faload, daload, aaload
- bastore, castore, sastore, iastore, lastore, fastore, dastore, aastore
- arraylength
- instanceof, checkcast

Object Creation and manipulation

- new
- newarray, anewarray, multinewarray
- getfield, putfield, getstatic, putstatic
- baload, caload, saload, iaload, laload, faload, daload, aaload
- bastore, castore, sastore, iastore, lastore, fastore, dastore, aastore
- arraylength
- instanceof, checkcast

Method Invocation / Return

- invokevirtual
- invokespecial
- invokeinterface
- invokestatic
- ireturn, freturn, dreturn, areturn
- return

Java Intermediate Bytecode

- Observation:
  - Original goals where modularity, small footprint, verifiability, but not speed.
  - the bytecode had to be statically typed (speed/safety argument)
  - control flow merges must have the same incoming stack types
  - use symbolic references to environment (fragile base class)

Class Resolution

- CP entry tagged `CONSTANT_Class` can be either class/interface.
- Execution of an instruction that refers to a class:
  1. *search for class in the classloader hierarchy*
  2. *if not found, initiate class loading*
- ... much more to the story.

Method Invocation

```
INVOKEVIRTUAL,     - instance method
INVOKEINTERFACE,   - interface method
INVOKESPECIAL      - constructor/private/super method
INVOKESTATIC       - class method
```

```
foo/baz/Myclass/myMethod(Ljava/lang/String;)V
---------------         ---------------------
      |         --------          |
      |           |               |
   classname  methodname    descriptor
```

- When an invocation is executed the method must be resolved.

Method Resolution

1. Checks if C is class or interface.

   If C is interface, throw `IncompatibleClassChangeError`.

2. Look up the referenced method in C and superclasses:

   - Success if C has method with same name & descriptor
   - Otherwise, if C has a superclass, repeat 2 on super.

3. Otherwise, locate method in a superinterface of C

   - If found success.
   - Otherwise, fail.

Method Invocation

- Resolution is rather work intensive. Can this be done faster?

class initialization

- Before use of static field, static method, object creation, a class must be initialized.
- Initialization involves creating a new Class object, and running the static initializers.
- Every operation that could trigger initialization must check the status of the class.

subroutines

- Subroutines were added to the bytecode to reduce the space requirements of exception handler's finally clauses.

PURDUE UNIVERSITY

Example

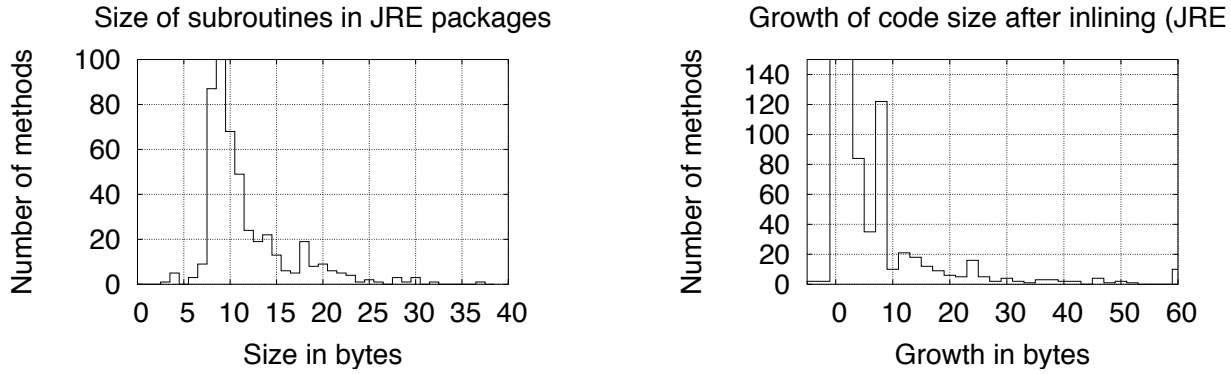
```
int bar(int i) {
    try {
        if (i == 3) return this.foo();
    } finally {
        this.ladida();
    }
    return i;
}
```

Region	Target
1–12	17
13–16	21

```
01  iload_1              // Push i
02  iconst_3             // Push 3
03  if_icmpne 10         // Goto 10 if i does not

04  aload_0              // Push this
05  invokevirtual foo    // Call this.foo
06  istore_2             // Save result of this.foo
07  jsr 13               // Do finally block before
08  iload_2              // Recall result from this
09  ireturn              // Return result of this.f

10  jsr 13               // Do finally block before

11  iload_1              // Push i
12  ireturn              // Return i
    // finally block
13  astore_3             // Save return address in
14  aload_0              // Push this
15  invokevirtual ladida // Call this.ladida()
16  ret 3                // Return to address save
    // Exception handler for try body
17  astore_2             // Save exception
18  jsr 13               // Do finally block
19  aload_2              // Recall exception
20  athrow               // Rethrow exception
    // Exception handler for finally body
21  athrow               // Rethrow exception
```

subroutines

- Over the JDK 1.1, subroutines save a total of 2427 bytes [Freund98].
- Java5 does not use them. They can be inlined by tools.



From Artho, Biere, Bytecode 2005.

Figure 7. Sizes of subroutines and size increase after inlining.

class compression

- Observation:
  - class file size dominated by symbolic information in the CP
  - JAR files (containing multiple classes) contain redundancies

	swingall	javac
Total size	3,265	516
excluding jar overhead	3,010	485
Field definitions	36	7
Method definitions	97	10
Code	768	114
Other	72	12
Constant pool	2,037	342
Utf8 entries	1,704	295
if shared	372	56
if shared and factored	235	26

[Pugh99]

compression

- Observation:
  - class file size dominated by symbolic information in the CP
  - JAR files (containing multiple classes) contain redundancies

File Format	Size	% orig. size
JAR file, uncompressed	260,178	100.0%
JAR file, compressed	132,600	51.0%
Clazz	97,341	37.4%
Gzip	97,223	37.4%
Jazz	59,321	22.8%

[Bradley,Horspool,Vitek,98]

Java Virtual Machine, part three

Verification

- Ensures that the type (i.e. the loaded class) obeys Java semantics, and
- will not violate the integrity of the JVM.

There are many aspects to verification

Verification, cont'd

Some Checks during Loading

- If it's a classfile, check the magic number (`0xCAFEBABE`),
- make sure that the file parses into its components correctly

Additional Checks after/during Loading

- make sure the class has a superclass (only Object does not)
- make sure the superclass is not final
- make sure final methods are not overridden
- if a nonabstract class, make sure all methods are implemented
- make sure there are no incompatible methods
- make sure constant pool entries are consistent

Additional Checks after/during Loading, cont'd

- check the format of special strings in the constant pool (such as method signatures etc)

A Final Check (required before method is executed)

- verify the integrity of the method's bytecode

This last check is very complicated (so complicated that Sun got it wrong a few times)

Verifying Bytecode

The requirements

- All the opcodes are valid, all operands (e.g. number of a field or a local variable) are in range.
- Every control transfer operation (goto, ifne, ...) must have a destination which is in range and is the start of an instruction
- Type correctness: every operation receives operands with the correct datatypes
- No stack overflow or underflow
- A local variable can never be used before it has been initialized
- Object initialization – the constructor must be invoked before the class instance is used

The requirements, cont'd

- Execution cannot fall off the end of the code
- The code does not end in the middle of an instruction
- For each exception handler, the start and end points must be at the beginnings of instructions, and the start must be before the end
- Exception handler code must start at the beginning of an instruction

Sun's Verification Algorithm

A *before* state is associated with each instruction.

The state is:

- contents of operand stack (stack height, and datatype of each element), plus
- contents of local variables (for each variable, we record *uninitialized* or *unusable* or the datatype)

A datatype is integral, long, float, double or any reference type

Each instruction has an associated *changed* bit:

- all these bits are false,
- except the first instruction whose changed bit is true.

Sun's Verification Algorithm, cont'd

```
do forever {
    find an instruction I whose changed bit is true;
    if no such instruction exists, return SUCCESS;
    set changed bit of I to false;
    state S = before state of I;

    for each operand on stack used by I
        verify that the stack element in S has correct datatype
        and pop the datatype from the stack in S;
    for each local variable used by I
        verify that the variable is initialized and
        has the correct datatype in S;
    if I pushes a result on the stack,
        verify that the stack in S does not overflow, and
        push the datatype onto the stack in S;
    if I modifies a local variable,
        record the datatype of the variable in S
                        ... continued
```


Sun's Verification Algorithm, cont'd

```
    determine SUCC, the set of instructions which can follow I;
        (Note: this includes exception handlers for I)

    for each instruction J in SUCC do
        merge next state of I with the before state of J
        and set J's changed bit if the before state changed;
        (Special case: if J is a destination because of an exception
        then a special stack state containing a single instance of
        the exception object is created for merging with the before
        state of J.)
} // end of do forever
```

Verification fails if a datatype does not match with what is required by the instruction, the stack underflows or overflows, or if two states cannot be merged because the two stacks have different heights.

Sun's Verification Algorithm, cont'd

Merging two states

- Two stack states with the same height are merged by pairwise merging the types of corresponding elements.
- The states of the two sets of local variables are merged by merging the types of corresponding variables.

The result of merging two types:

- Two types which are identical merge to give the same type
- For two types which are not identical:
  if they are both references, then the result is the first common superclass (lowest common ancestor in class hierarchy);
  otherwise the result is recorded as *unusable.*

Example (Leroy, Figure 1):

```
static int factorial( int n ) {
   int res;
   for (res = 1; n > 0; n--) res = res * n;
   return res;
}
```

Corresponding JVM bytecode:

```
method static int factorial(int), 2 variables, 2 stack slots
    0: iconst_1         // push the integer constant 1
    1: istore_1         // store it in variable 1 (res)
    2: iload_0          // push variable 0 (the n parameter)
    3: ifle 14          // if negative or null, go to PC 14
    6: iload_1          // push variable 1 (res)
    7: iload_0          // push variable 0 (n)
    8: imul             // multiply the two integers at top of stack
    9: istore_1         // pop result and store it in variable 1
   10: iinc 0, -1       // decrement variable 0 (n) by 1
   11: goto 2           // go to PC 2
   14: iload_1          // load variable 1 (res)
   15: ireturn          // return its value to caller
```


Sun's Analysis Algorithm

Chng'd	State before		Instruction	State after	
	Stack	Locals		Stack	Locals
X	()	(I,T)	0: iconst_1		
-	?	(?,?)	1: istore_1		
-	?	(?,?)	2: iload_0		
-	?	(?,?)	3: ifle 14		
-	?	(?,?)	6: iload_1		
-	?	(?,?)	7: iload_0		
-	?	(?,?)	8: imul		
-	?	(?,?)	9: istore_1		
-	?	(?,?)	10: iinc 0, -1		
-	?	(?,?)	11: goto 2		
-	?	(?,?)	14: iload_1		
-	?	(?,?)	15: ireturn		

where I = integral; T = *uninitialized/unusable*; ? = ⊥ = *unknown*

Sun's Analysis Algorithm - after 1 step

Chng'd	State before		Instruction	State after	
	Stack	Locals		Stack	Locals
-	()	(I,T)	0: iconst_1	(I)	(I,T)
X	(I)	(I,T)	1: istore_1		
-	?	(?,?)	2: iload_0		
-	?	(?,?)	3: ifle 14		
-	?	(?,?)	6: iload_1		
-	?	(?,?)	7: iload_0		
-	?	(?,?)	8: imul		
-	?	(?,?)	9: istore_1		
-	?	(?,?)	10: iinc 0, -1		
-	?	(?,?)	11: goto 2		
-	?	(?,?)	14: iload_1		
-	?	(?,?)	15: ireturn		

Sun's Analysis Algorithm - after 4 steps

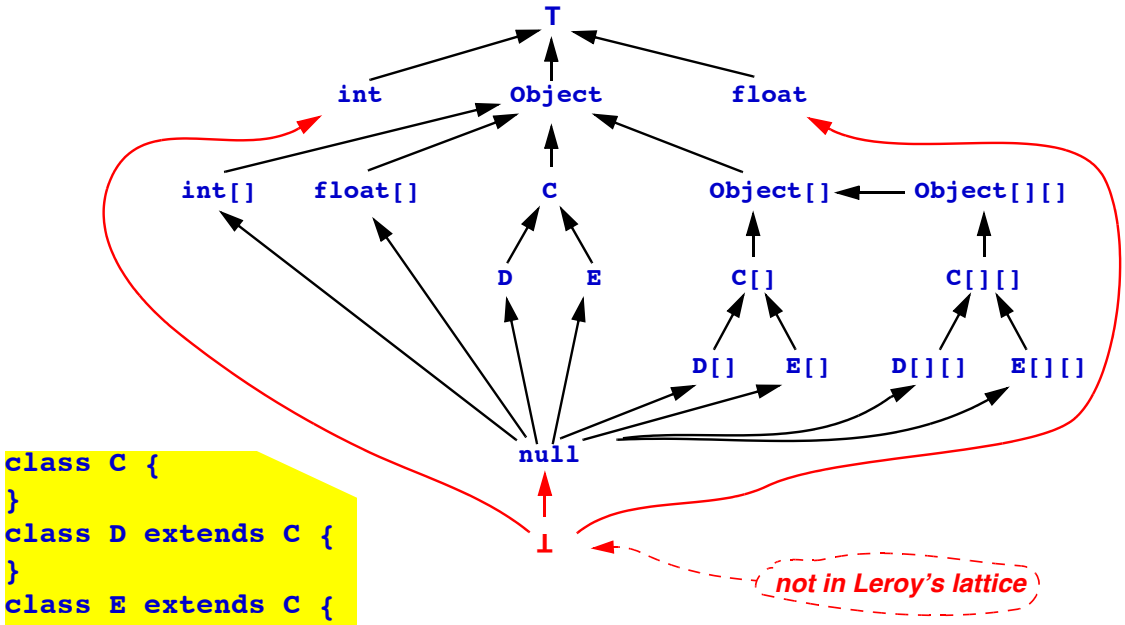
Chng'd	State before		Instruction	State after	
	Stack	Locals		Stack	Locals
-	()	(I,T)	0: iconst_1		
-	(I)	(I,T)	1: istore_1		
-	()	(I,I)	2: iload_0		
-	(I)	(I,I)	3: ifle 14	()	(I,I)
X	()	(I,I)	6: iload_1		
-	?	(?,?)	7: iload_0		
-	?	(?,?)	8: imul		
-	?	(?,?)	9: istore_1		
-	?	(?,?)	10: iinc 0, -1		
-	?	(?,?)	11: goto 2		
X	()	(I,I)	14: iload_1		
-	?	(?,?)	15: ireturn		

Analysis Algorithm - after 12 steps

Chng'd	State before		Instruction	State after	
	Stack	Locals		Stack	Locals
-	()	(I,T)	0: iconst_1		
-	(I)	(I,T)	1: istore_1		
-	()	(I,I)	2: iload_0		
-	(I)	(I,I)	3: ifle 14		
-	()	(I,I)	6: iload_1		
-	(I)	(I,I)	7: iload_0		
-	(I,I)	(I,I)	8: imul		
-	(I)	(I,I)	9: istore_1		
-	()	(I,I)	10: iinc 0, -1		
-	()	(I,I)	11: goto 2		
-	()	(I,I)	14: iload_1		
-	(I)	(I,I)	15: ireturn	()	(I,I)

and we have completed the verification without error.

Some of the Lattice of Types (Leroy, Figure 3)




```
class C {
}
class D extends C {
}
class E extends C {
}
```


Merging Types

- The lattice represents an ordering relation on types
- The lattice is derived from the semantics of Java (and is based on the class hierarchy)
- Given any two types t_1 and t_2, there is a least upper bound type, *lub*(t_1,t_2)
- Given any type t, the length of the path from t to top, T, is finite (the well-foundedness property).

The step in Sun's verification algorithm where types are merged is implemented as *lub*.

The finiteness property guarantees that Sun's algorithm will converge in a finite number of steps.

Garbage Collection – overview of the three classical approaches

(based on chapter 2 of Jones and Lins)

Reference Counting

- a simple technique used in many systems
- eg, Unix uses it to keep track of when a file can be deleted (references to files come from directories)
- each object contains a counter which tracks the number of references to the object;
  if the count becomes zero, the storage of the object is immediately reclaimed (put into a free list?)
- distributes the cost of gc over the entire run of a program

Pseudocode for Reference Counting

```
// called by program to get a
// new object instance
function New():
  if freeList == null then
    report an error;
  newcell = allocate();
  newcell.rc = 1;
  return newcell;

// called by program to overwrite
// a pointer variable R with
// another pointer value S
procedure Update(var R, S):
  if S != null then
    S.rc += 1;
  delete(*R);
  *R = S;
```

```
// called by New
function allocate():
  newcell = freeList;
  freeList = freeList.next;
  return newcell;

// called by Update
procedure delete(T):
  T.rc -= 1;
  if T.rc == 0 then
    foreach pointer U held
        inside object T do
      delete(*U);
    free(T);

// called by delete
procedure free(N):
  N.next = freeList;
  freeList = N;
```

`rc` is the reference count field in the object

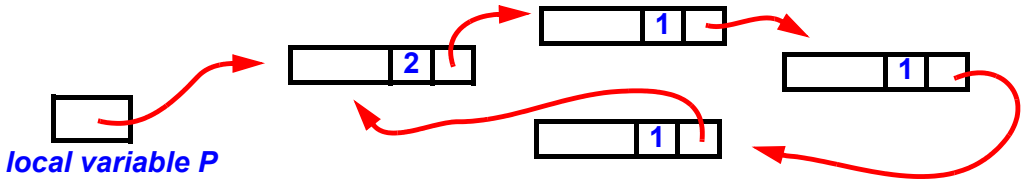
Benefits of Reference Counting

- GC overhead is distributed throughout the computation ==> smooth response times in interactive situations. (Contrast with a stop and collect approach.)
- Good memory locality 1 – the program accesses memory locations which were probably going to be touched anyway. (Contrast with a marking phase which walks all over memory.)
- Good memory locality 2 – most objects are short-lived; reference counting will reclaim them and reuse them quickly. (Contrast with a scheme where the dead objects remain unused for a long period until the next gc and get paged out of memory.)

Issues with Reference Counting, cont'd

- Extra storage requirements
  - Every object must contain an extra field for the reference counter. (And how big should it be?)

- **Does not work with cyclic data structures!!!**



Mark-Sweep (aka Mark-Scan) Algorithm

- First use seems to be Lisp
- Storage for new objects is obtained from a free pool
- No extra actions are performed when the program copies or overwrites pointers
- When the free pool is exhausted, the `New()` operation invokes the mark-sweep gc to return inaccessible objects to the free pool and then resumes

Pseudocode for Mark-Sweep

```
function New():
  if freeList == null then
    markSweep();
  newcell = allocate();
  return newcell;

// called by New
function allocate():
  newcell = freeList;
  freeList = freeList.next;
  return newcell;

procedure free(P):
  P.next = freeList;
  freeList = P;

procedure markSweep():
  foreach R in RootSet do
    mark(R);
  sweep();
  if freeList == null then
    abort "memory exhausted"
```

```
// called by markSweep
procedure mark(N):
  if N.markBit == 0 then
    N.markBit = 1;
    foreach pointer M held
        inside the object N do
      mark(*M);

// called by markSweep
procedure sweep():
  K = address of heap bottom;
  while K < heap top do
    if K.markBit == 0 then
      free(K);
    else
      K.markBit = 0;
    K += size of object
        referenced by K;
```

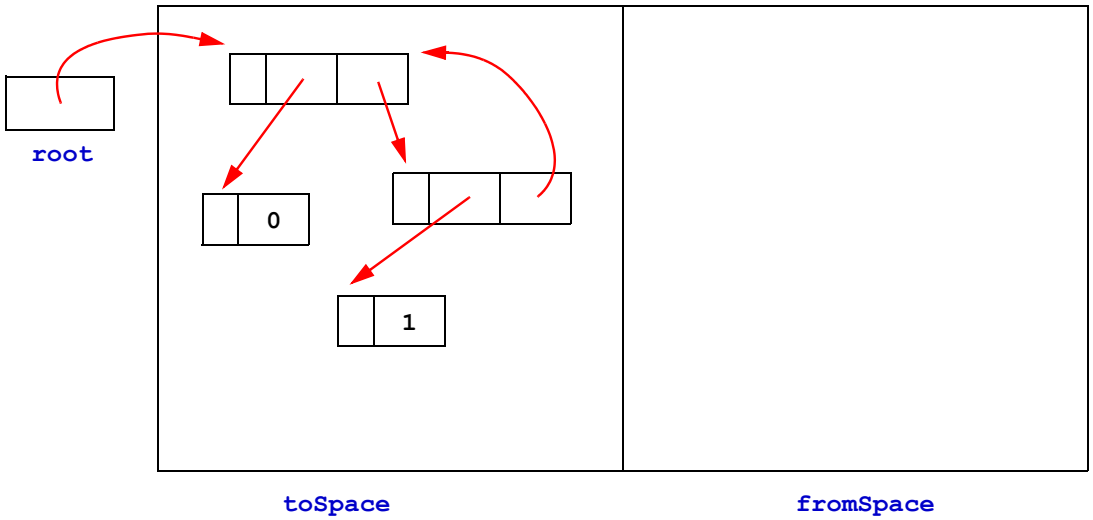

Pros and Cons of Mark-Sweep GC

- Cycles are handled automatically
- No special actions required when manipulating pointers
- It's a stop-start approach – in the 1980's, Lisp users got interrupted for about 4.5 seconds every 79 seconds.
- Less *total* work performed than reference counting.
- Tends to fragment memory, scattering elements of linked lists all across the heap
- Performance degrades as the heap fills up with active cells (causing more frequent gc)

Copying Garbage Collectors

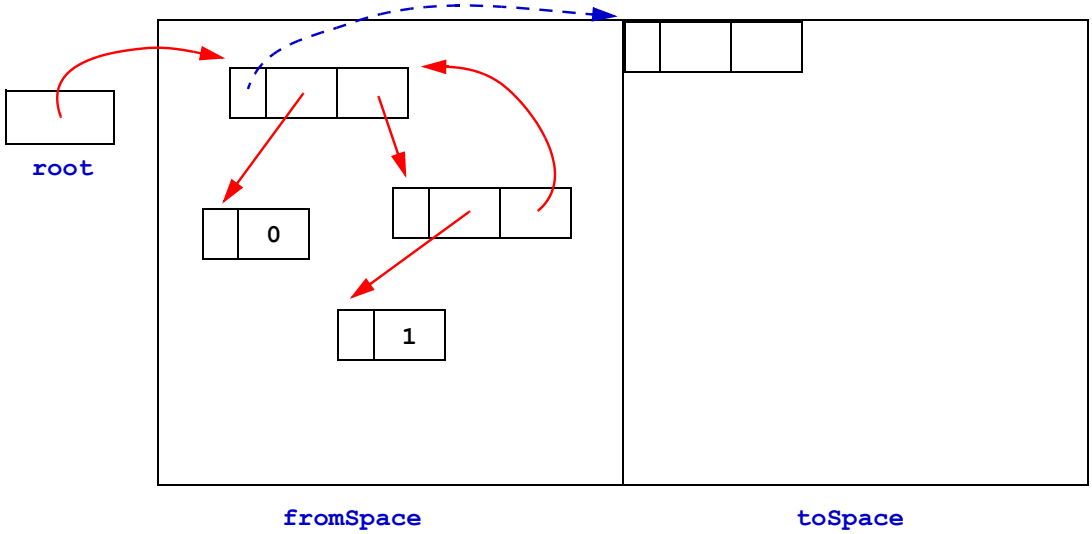
- The heap is divided into two equal sized regions – the *fromSpace* and the *toSpace.*
- The roles of the two spaces are reversed at each gc.
- At a gc, the active cells are copied from the old space (the *fromSpace*) into the new space (the *toSpace*), and the program's variables are updated to use the new copies.
- Garbage cells in the fromSpace are simply abandoned.
- Storage in the toSpace is automatically compacted during the copying process (no gaps are left).

Example of Copying Collector in Action



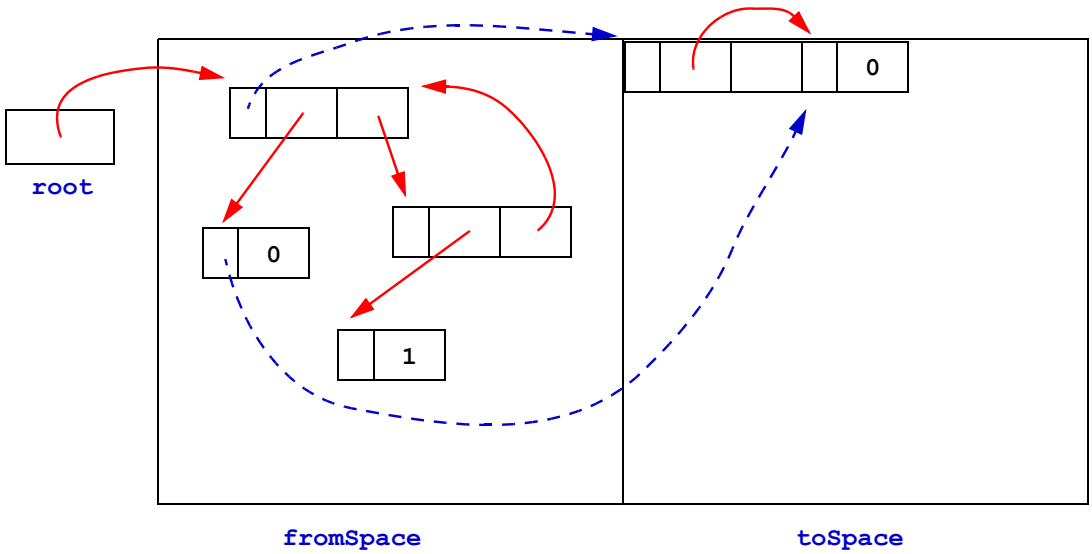
1. a gc is initiated; the fromSpace & toSpace are swapped ...

Example of Copying Collector in Action



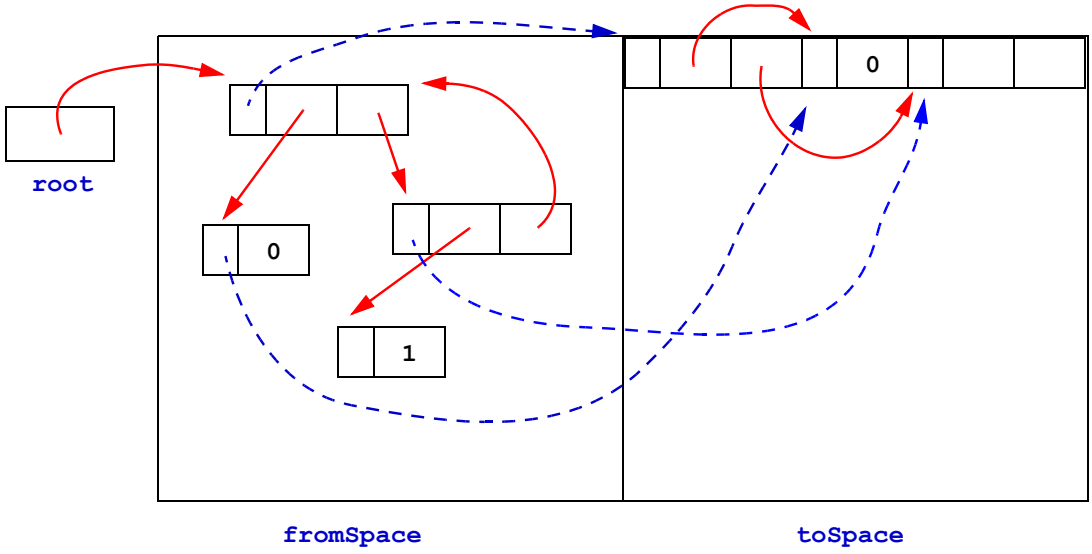
... the root node is copied, and a forwarding pointer added

Example of Copying Collector in Action



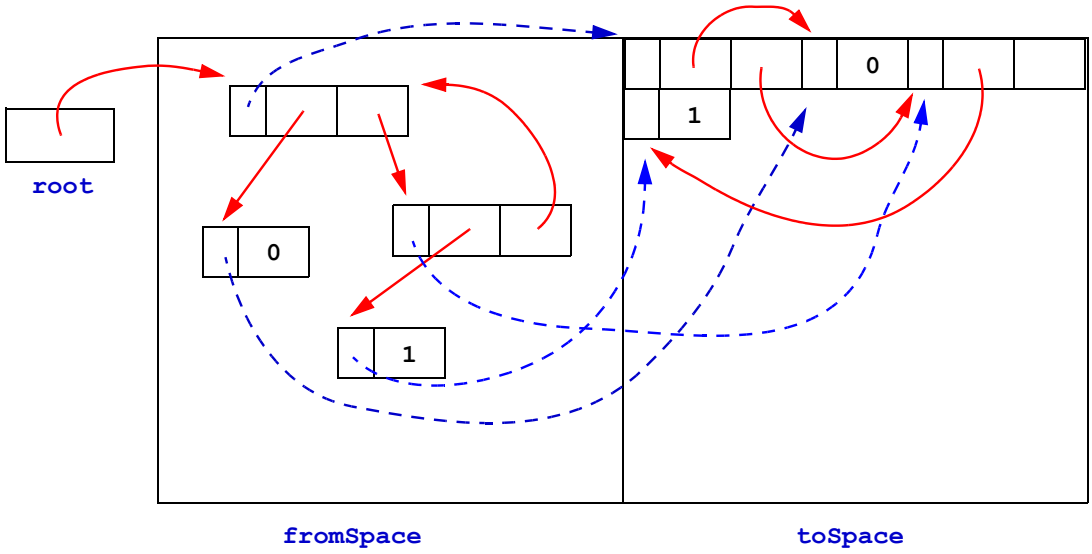
... the left child of first node is copied

Example of Copying Collector in Action



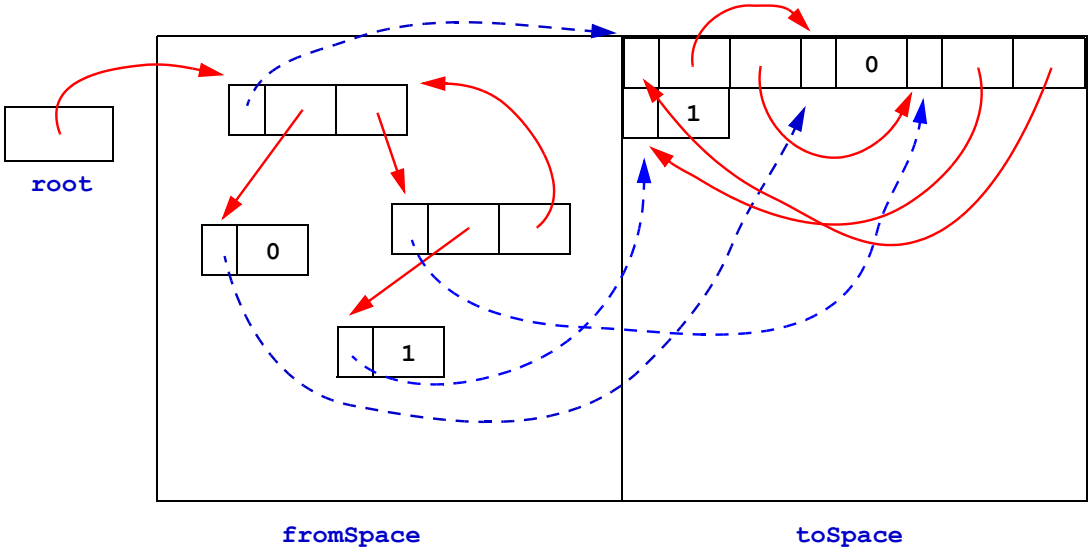
... and the right child of the first node is copied

Example of Copying Collector in Action



... and when the right child of the right child is copied ...

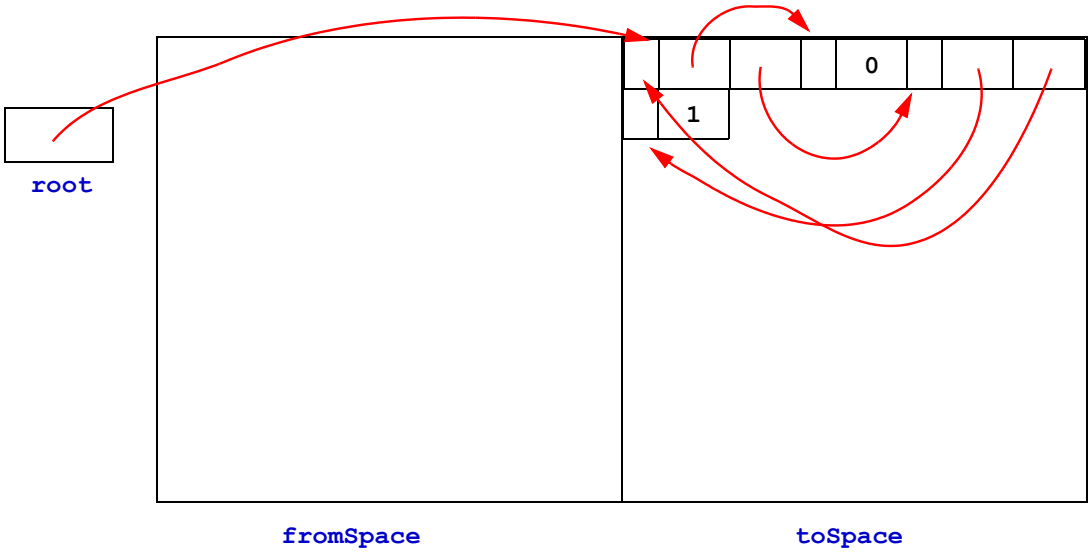
Example of Copying Collector in Action



... and we are almost finished

20 May 2005

Example of Copying Collector in Action



done ... and we carry on allocating new nodes in the toSpace

Pseudocode for a Copying Collector

```
procedure init():
  toSpace = start of heap;
  spaceSize = heap size / 2;
  topOfSpace =toSpace+spaceSize;
  fromSpace = topOfSpace+1;
  free = toSpace;

// n = size of object to allocate
function New(n):
  if free + n > topOfSpace then
    flip();
  if free + n > topOfSpace then
    abort "memory exhausted";
  newcell = free;
  free += n;
  return newcell;

procedure flip():
  fromSpace, toSpace =
        toSpace, fromSpace;
  free = toSpace;
  for R in RootSet do
    R = copy(R);
```

```
// parameter P points to a word,
// not to an object
function copy(P):
  if P is not a pointer
       or P == null then
    return P;
  if P[0] is not a pointer
       into toSpace then
    n = size of object
         referenced by P;
    PP = free;
    free += n;
    temp = P[0];
    P[0] = PP;
    PP[0] = copy(temp);
    for i = 0 to n-1 do
      PP[i] = copy(P[i]);
  return P[0];

// Note:
// The first word of an object,
// P[0], serves a dual role to
// hold a forwarding pointer.
```


Pros and Cons of Copying Collectors

- Very cheap allocation cost (just incrementing a pointer)
- Fragmentation of memory is eliminated at each gc

- At any time, at least 50% of the heap is unused
  (may not be a problem with virtual memory systems where we can have big address spaces)

Virtual and Interface Methods

PURDUE UNIVERSITY

(S³)

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)

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:

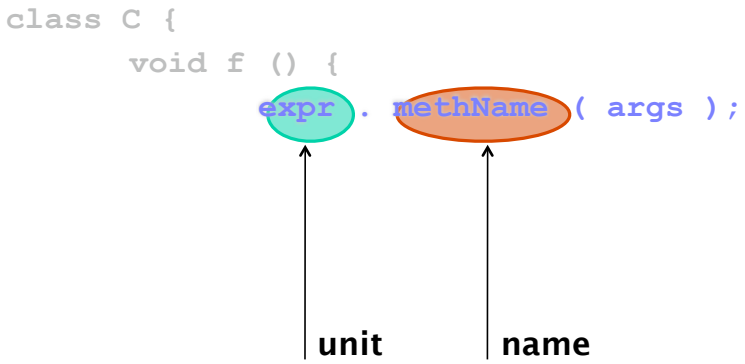
```
foo(arg)
obj.foo(arg)
HashTbl.foo(arg)
obj.foo(arg).bar()
```


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)



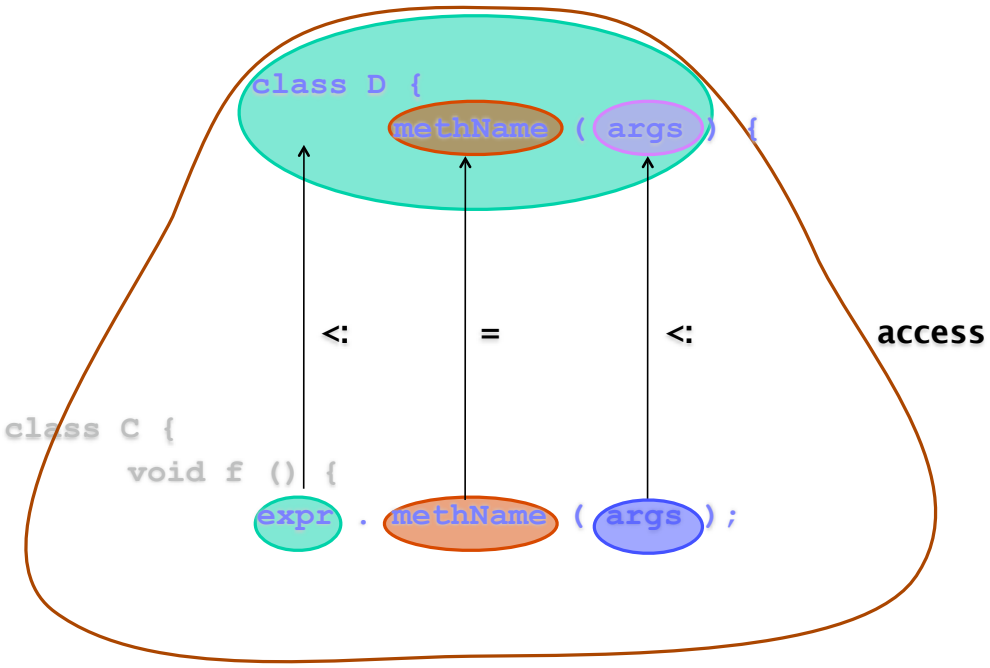
Determine Method Signature (step 2)

- 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.

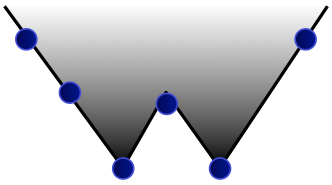
Given methods `T m( T1, ... Tn)` and `U m'( U1, ... Un)`, `m` is more specific than `m'` if

- `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

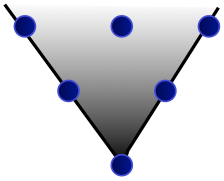
A method is said to be maximally specific for an invocation if it is applicable and accessible and there is no method more specific

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)
```

Figure 1: Sequences for virtual and IMT-based interface dispatch

Jikes IMTs [OOPSLA01]

```
    //    id1 < id2 < id3 < id4
    //    t0  contains the address of the receiving object ("this" parameter)
    //    s1  contains the interface method signature id ("hidden" parameter)
    //    LR  contains the return address in the caller
    //
    L     s0, tibOffset(t0) // s0 := TIB of the receiver
    CMPI  s1, id2           // compare hidden parameter to id of second method
    BLT   L1                // if less than branch to (id1..id1) search tree
    BGT   L2                // if greater than branch to (id3..id4) search tree
    L     s0, offset2(s0)   // load VMT entry for second method
    MTCTR s0                // move this address to the count register
    BCTR                    // branch to it (preserving contents of the LR)
L1: L     s0, offset1(s0)   // load VMT entry for the first method
    MTCTR s0                // move this address to the count register
    BCTR                    // branch to it (preserving contents of the LR)
L2: CMPI  s1, id3           // compare hidden parameter to id of third method
    BGT   L3                // if greater than branch to (id4..id4) search tree
    L     s0, offset3(s0)   // load VMT entry for the third method
    MTCTR s0                // move this address to the count register
    BCTR                    // branch to it (preserving contents of the LR)
L3: L     s0, offset4(s0)   // load VMT entry for the fourth method
    MTCTR s0                // move this address to the count register
    BCTR                    // branch to it (preserving contents of the LR)
```

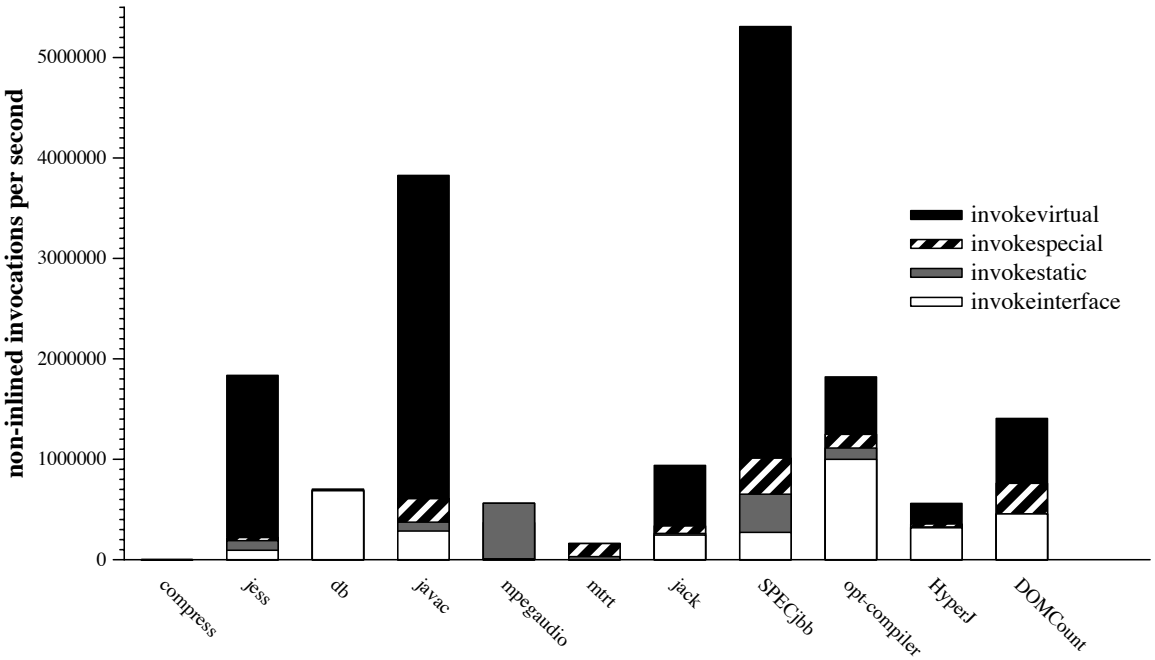
Figure 2: A conflict resolution stub with four entries.

Benchmarks

Dispatching Mechanism	Number of Methods in Interface	Trivial Callee	Normal Callee
virtual	Not applicable	8.13	19.18
Embedded IMT-5	interface with 1 method (no IMT conflict)	8.23	19.25
Embedded IMT-5	interface with 100 methods (20 element stub)	20.25	32.28
Embedded IMT-40	interface with 1 method (no IMT conflict)	8.23	19.25
Embedded IMT-40	interface with 100 methods (2 or 3 element stub)	14.23	27.40
IMT-5	interface with 1 method (no IMT conflict)	10.18	21.20
IMT-5	interface with 100 methods (20 element stub)	22.20	32.23
IMT-40	interface with 1 method (no IMT conflict)	10.18	21.20
IMT-40	interface with 100 methods (2 or 3 element stub)	18.20	28.23
Directly Indexed ITables	interface with 1 method	12.18	23.20
Directly Indexed ITables	interface with 100 methods	12.18	23.20
Searched ITables	interface with 1 method	77.45	88.50
Searched ITables	interface with 100 methods	77.45	88.50
Class Object Search	interface with 1 method	352.55	362.73
Class Object Search	interface with 100 methods	1,743.13	1,759.48

Table 1: Cost, in clock cycles, of round-trip method dispatch in Jalapeño, under each alternative interface dispatching mechanism.

Benchmarks



Benchmarks

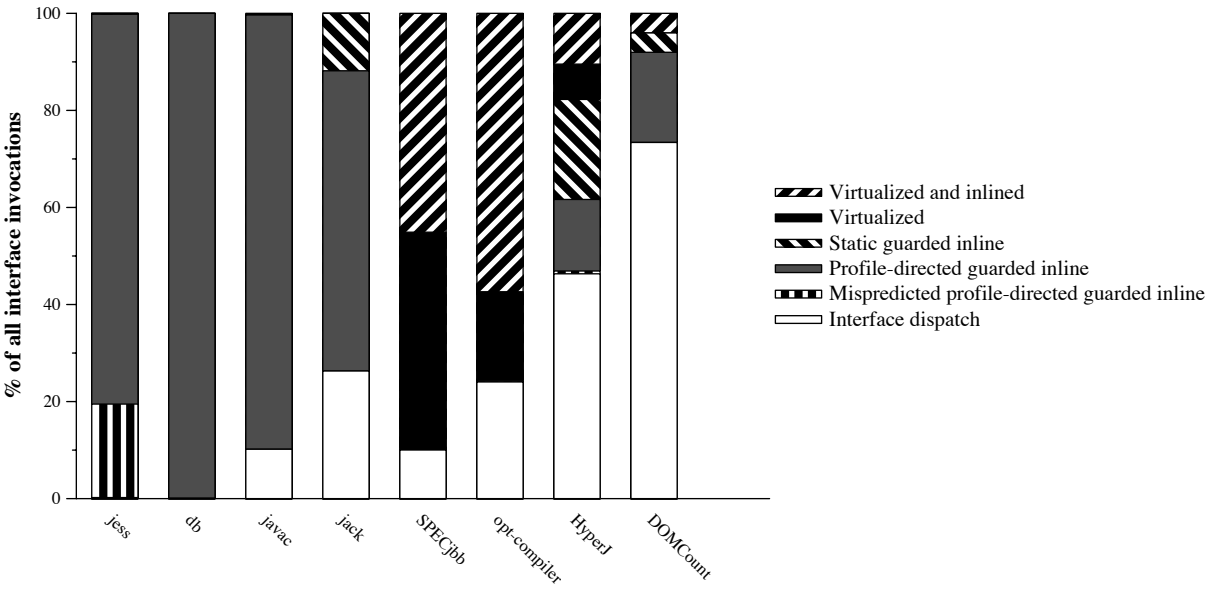


Figure 7: Dynamic percentage of interface invocations handled by each dispatch mechanism.

Benchmarks

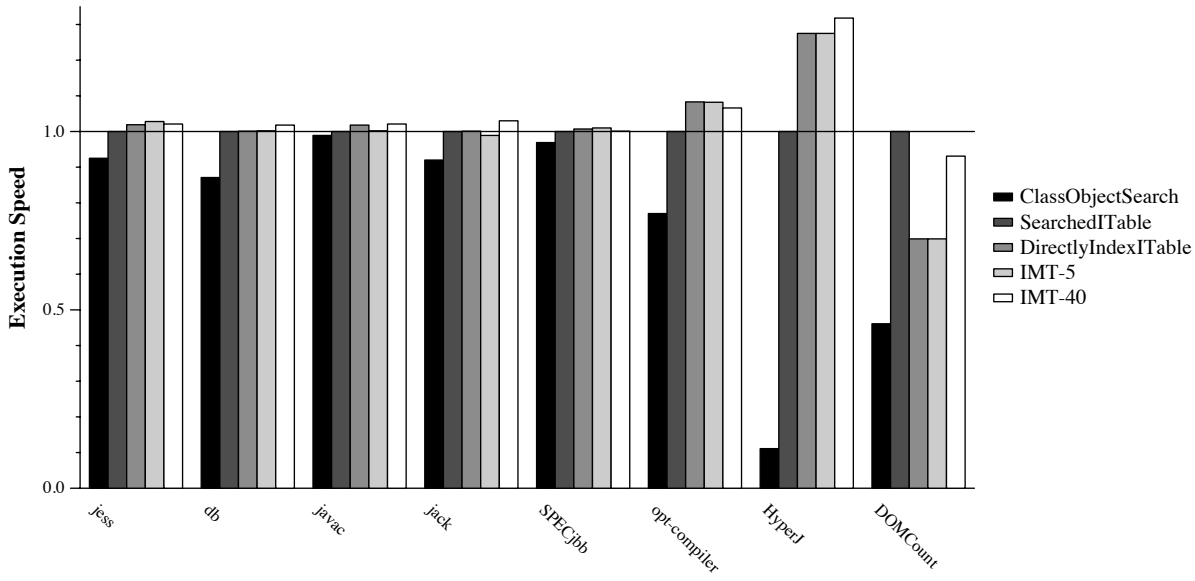
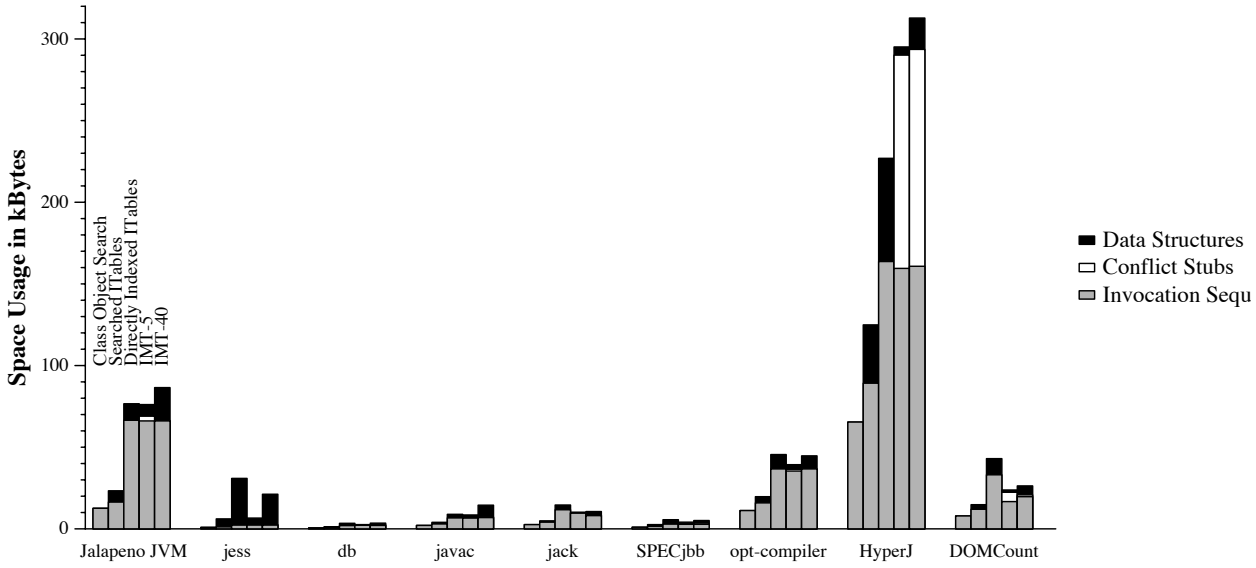


Figure 8: Performance impact of alternative interface method dispatching schemes. Speeds are norma to that of *SearchedITable*.

Benchmarks



Subtype tests

[Palacz, Vitek ECOOP'03]

PURDUE
UNIVERSITY

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)…
```

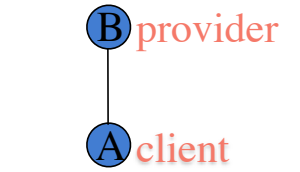
- Furthermore, the Generic Java compiler (gj) works by translating genericity away and inserting subtype tests to obtain verifying code.

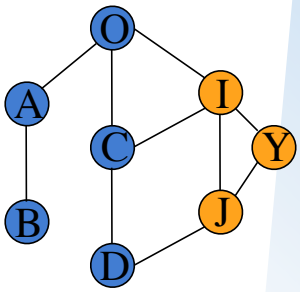
Goals of this work

- **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

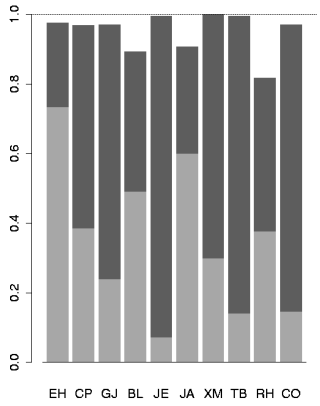




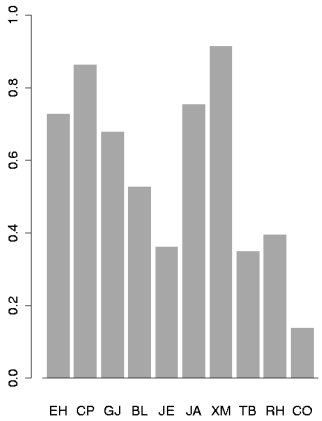
a subtype test is **successful** if isSubtype returns true or if checkCast does not throw an exception

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

Dynamic characteristics of ST

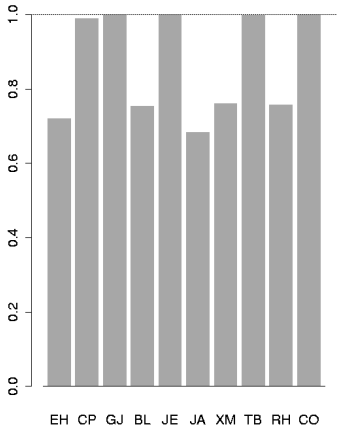


Success rate



Checkcast rate

as % of dynamic tests



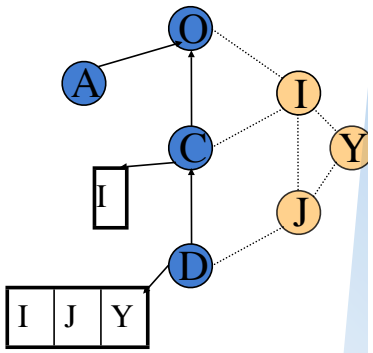
Extends rate

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

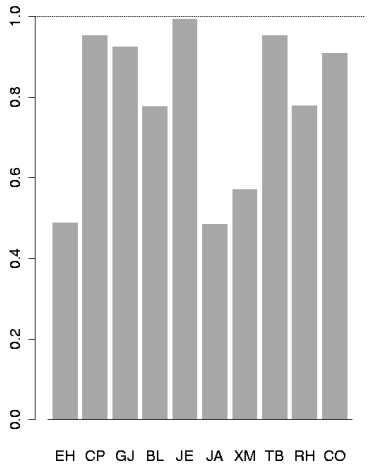
```
bool isSubtype(type cl, type pr) {
   if (cl.cache==pr) return true;
   if (isInterface(pr)) return extends(cl, pr);
   else return implements(cl, pr); }

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…

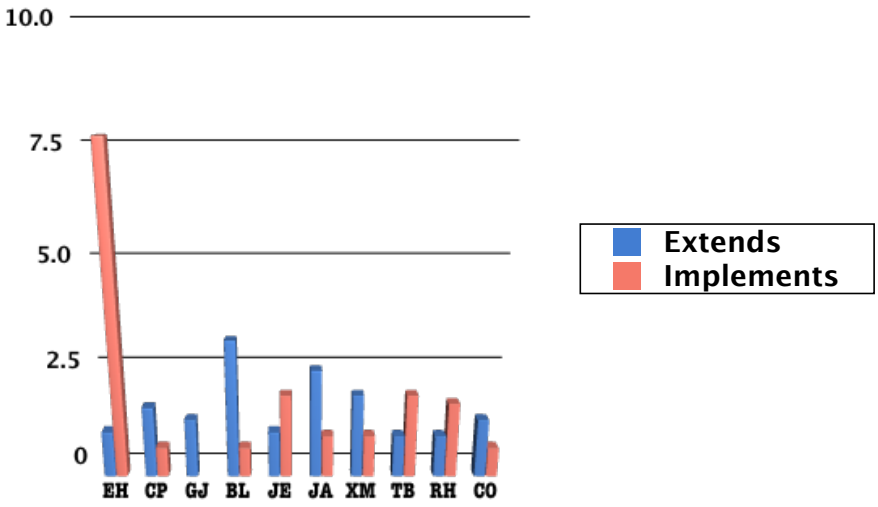
Evaluating the cache



Cache hit rates

as % of dynamic tests

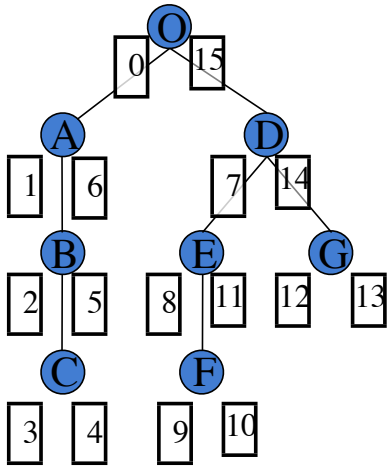
Hierarchy traversal cost



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

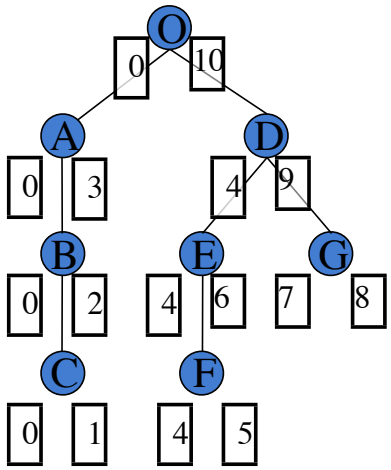
Constant time/space Extends

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



Extends

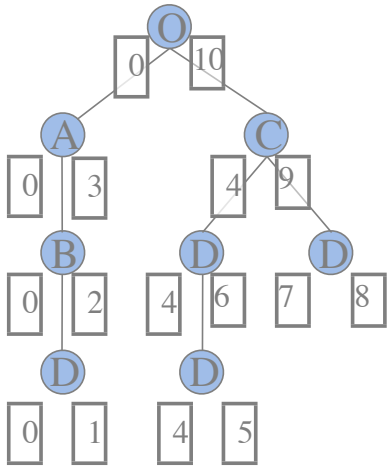
```
bool extends(type cl, type pr) {

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



Incremental computation

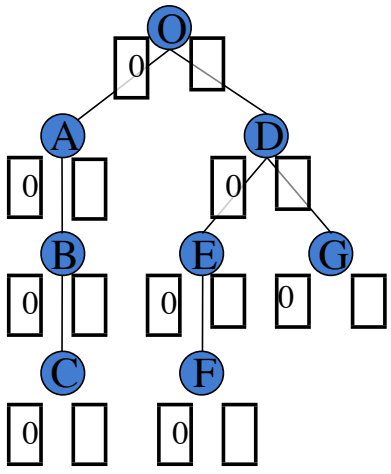
- Recall:
  - Most types are not in provider position
  - Success is the common case
- Define:

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



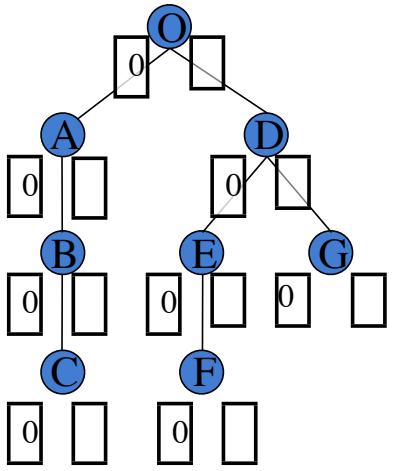
Incremental computation

```
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;
}
```



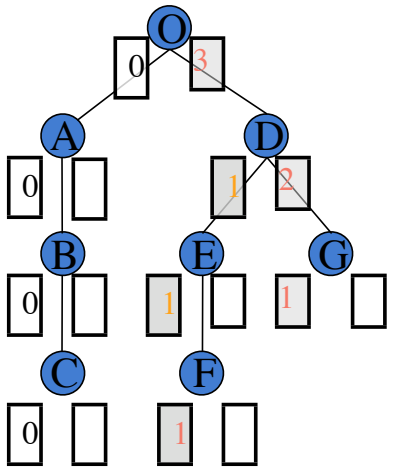
Incremental computation

`extends(F,D) ?`



Incremental computation

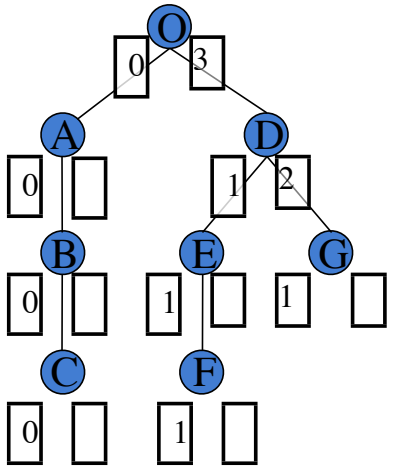
`extends(F,D) ?`



Incremental computation

extends(F,D)

extends(G,D) ?

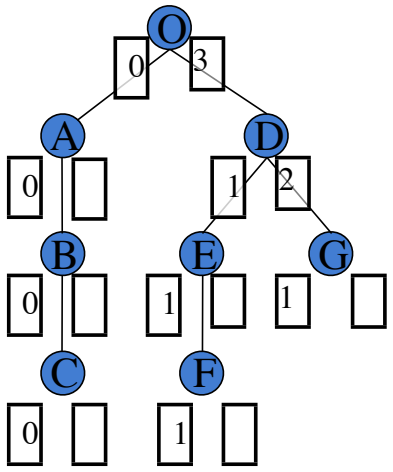


Incremental computation

extends(F,D)

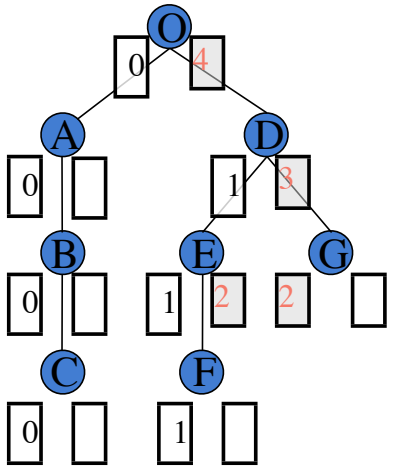
extends(G,D)

extends(F,E)?



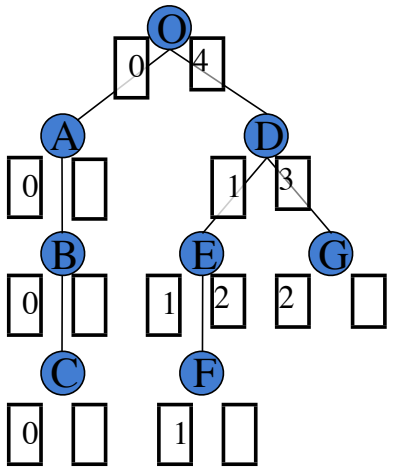
Incremental computation

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



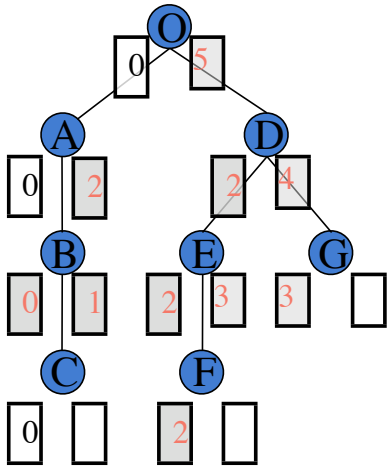
Incremental computation

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



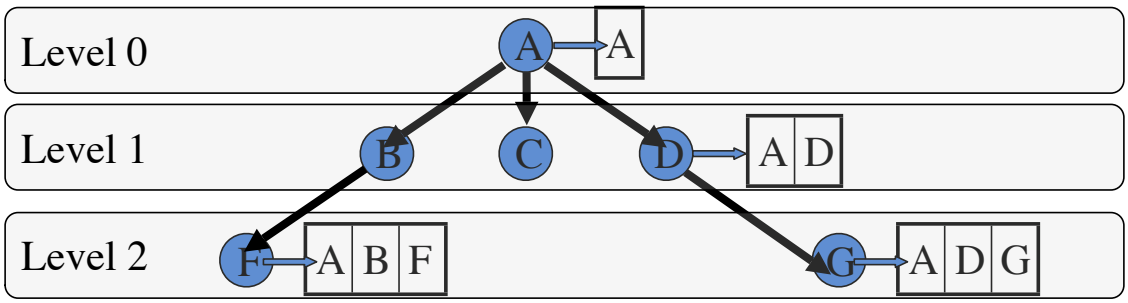
Incremental computation

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



Implements

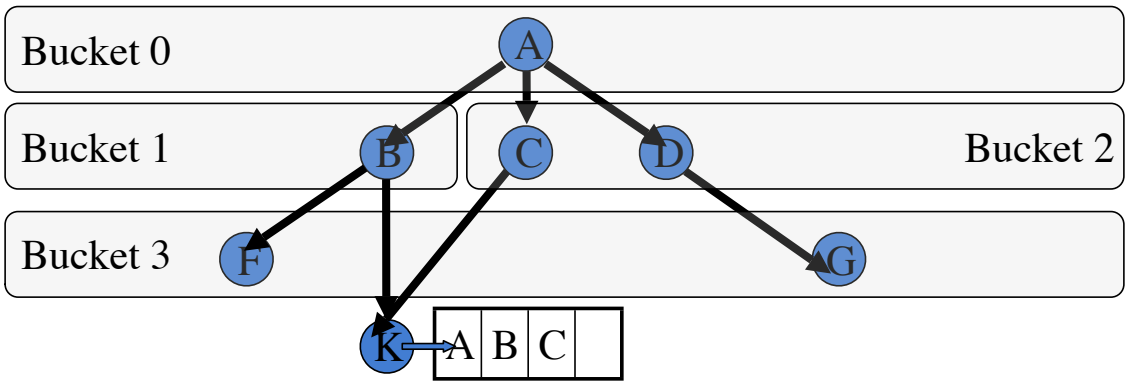
Cohen's displays for single inheritance…




```
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`**

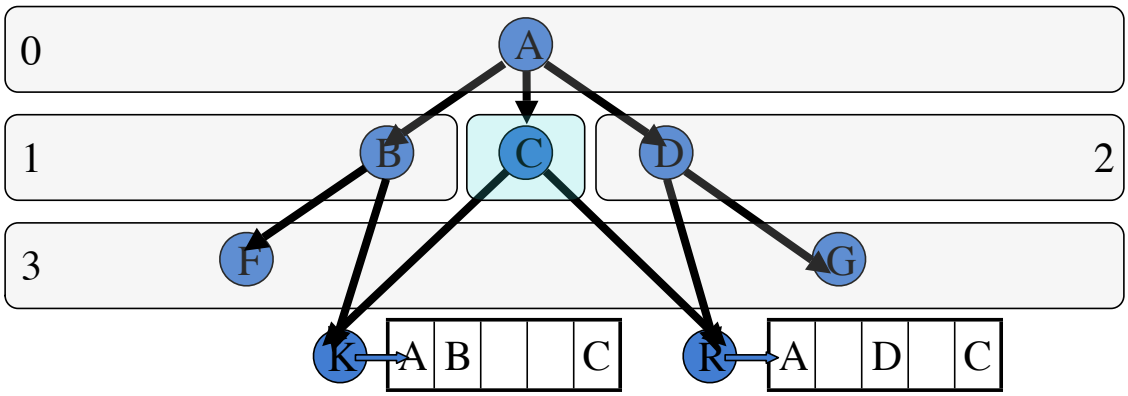
  Foreach **`I:`** if **`A<:I`** do
  `mark[I.bucket_num]++`

  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



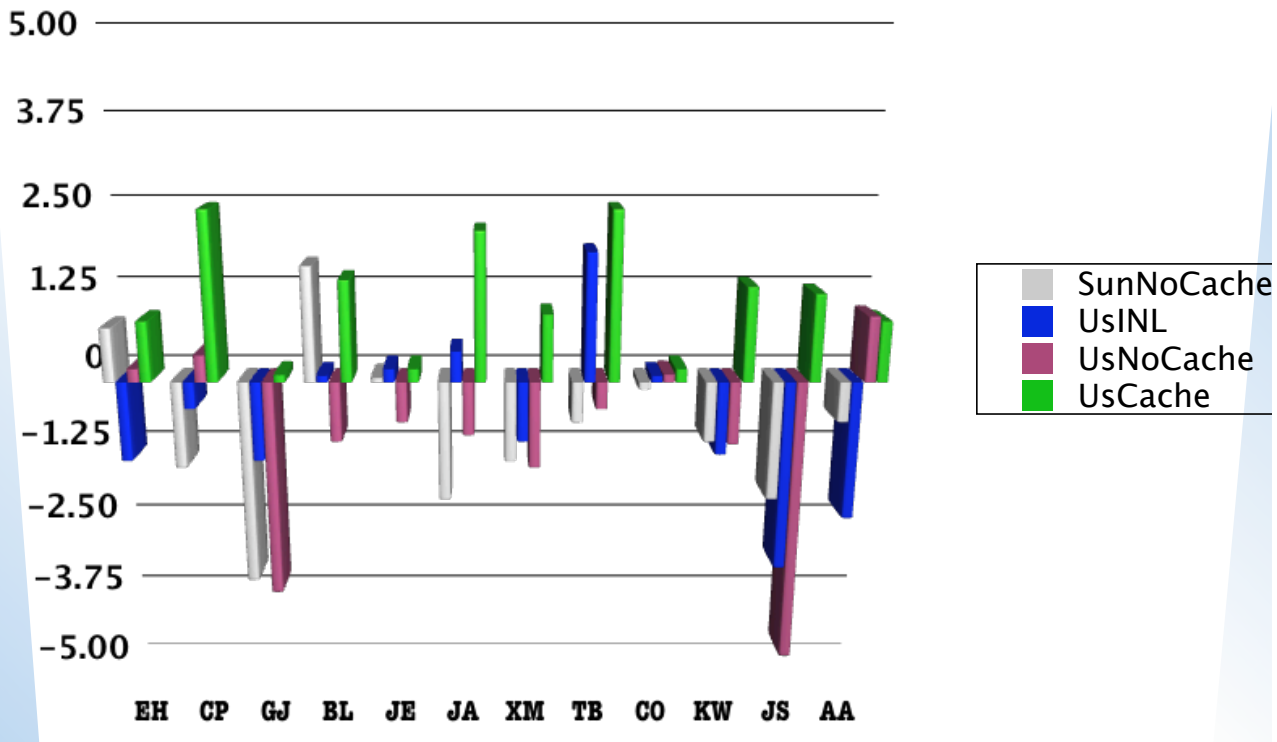
Implements, ct'd

- 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



A Java Virtual Machine for Hard Real-time Systems

Jan Vitek

PURDUE UNIVERSITY

Real-time Embedded Systems

- strict functional and non-functional requirements
  - Vast majority of microprocessors embedded
  - Must be reliable, especially if mission- or safety-critical
  - Must operate with limited resources, e.g. power, cpu, memory, price
- Traditional embedded system development
  - low-level languages (e.g. ASM, C) ⇒ low rate of reuse
- This model is not sustainable
  due to ***scale*** of modern embedded systems

300,000 lines of code for ScanEagle UAV

5,000,000 Loc for DD(X) destroyer

... Moore's law reflected in software

An Open Virtual Machine Framework

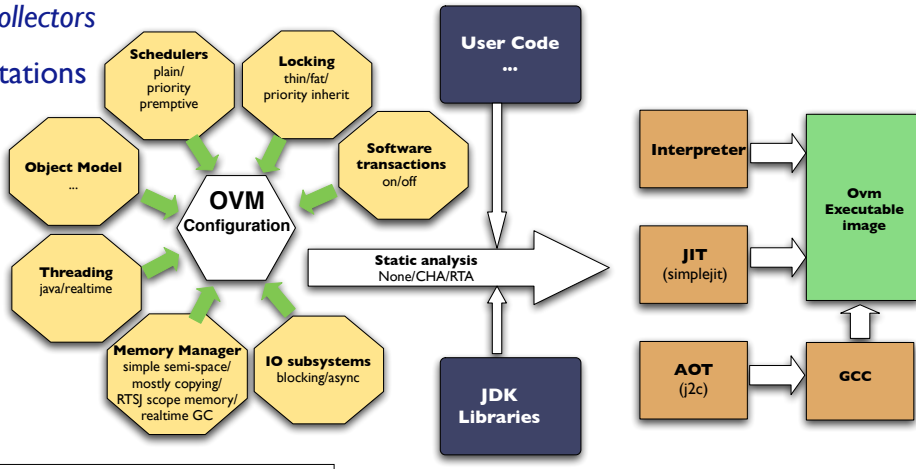
Purdue's Ovm

- Ovm is a configurable virtual machine framework
  - developed at Purdue, released under an Open Source licence
  - entirely written in Java (~150Kloc)
  - ported to x86, PPC, ARM
  - minimal OS dependencies
- Implementation of Real-time Java on top of Ovm
- An open source RTSJVM outperforms some commercial VMs

Baker, Cunei, Flack, Pizlo, Prochazka, Vitek, Armbuster+. A Real-time Java Virtual Machine for Avionics. **RTAS06**

A Configurable VM

- A framework used to generate virtual machines from specifications
  - multiple schedulers
    standard Java, priority preemptive
  - multiple execution engines & compilers
    ahead-of-time, just-in-time, interpreter
  - memory managers
    8 different garbage collectors
  - threading implementations
  - object models
  - ...

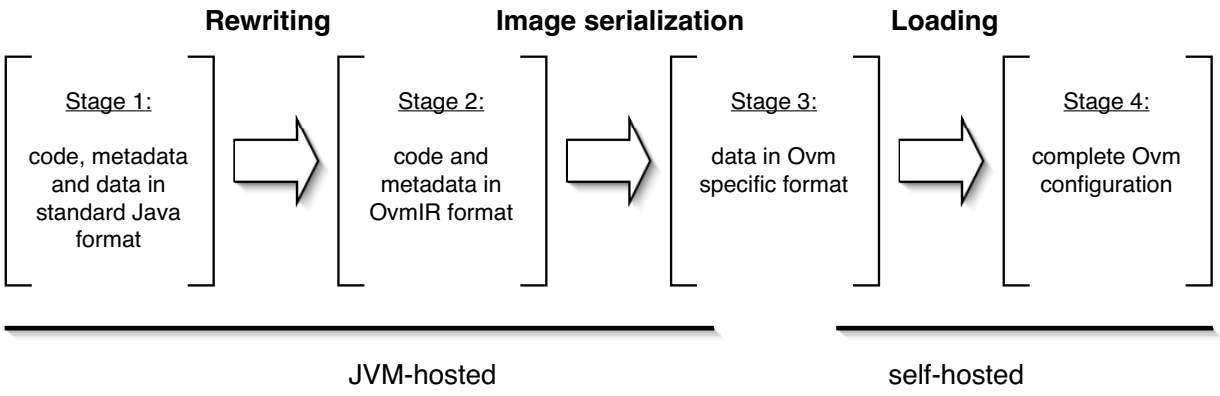


Palacz, Baker, Flack, Grothoff, Yamauchi, Vitek. Engineering an intermediate representation for Ovm. **SCP05**

Bootstrap
Hotspot hosted

VM Build Process

- Bootstrapped under Hotspot
- Configuration and partial evaluation
- Generate an executable image (data+code)



Java in Java

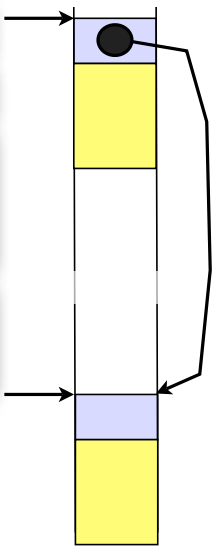
- Ovm written in Java
  - Fewer software defects / faster development times
  - Whole-system compilation inlines VM code into user code
- Approach:
  - Retain Java syntax but extend semantics to express unsafe operations
  - Idioms automatically recognized and translated to native code
  - Surprisingly few places where needed, most of VM type safe

Flack, Hosking, Vitek. Idiom recognition in Ovm. *Technical Report 2003*

Java in Java (1)

- Consider the implementation of moving garbage collector in Java. (*Java code*)

```
Oop updateReference(MovingGC oop) {
  int sz = oop.getBlueprint().getSize(oop);
  VM_Address loc = getHeapMem(sz);
  Mem.the().cpy(loc,oop,sz);
  oop.markAsForwarded(loc);
  return loc.asOop();
```



Java in Java (2)

```
updateReference(Updater*this,MovingGC*oop){
        sz = ...((*(*oop))->IfcTbl->v[23])(*oop,oop);
        loc = getHeapMem(this,sz);
        Mem_cpy(roots->v[37]),loc,oop,sz);
        return *(oop + 8) = loc;
```

- The bytecode of the VM rewritten at build time
  - With the j2c engine, the following C++ is generate
  - most OO operations disappear (eg.7 Java methods ⇒ one C++ indirect call)

Idioms (3)

- Going further in the representation of objects used in a moving GC:

```
interface MovingGC extends Oop.WithUpdate {
  void markAsForwarded(VM_Address a)throws PragmaModelOp;
  boolean isForwarded()throws PragmaModelOp;
  VM_Address getForwardAddress()throws PragmaModelOp;
}
```

- Ovm uses *idioms* to express low-level operations in Java
  - **MovingGC** is a representation of pointers to objects with moving gc attributes
  - GC code is insulated from the implementation of these operations
  - Java exceptions are used as annotations (this is pre Java 5.0 metadata)

Idioms (4)

```
interface WithUpdate extends Oop {
  void updateBlueprint(Blueprint v) throws PragmaModelOp;
  void updateReference(int k, Oop v) throws PragmaModelOp;
```

```
interface Oop {
  Blueprint getBlueprint() throws PragmaModelOp;
  VM_Address headerSkip() throws PragmaModelOp;
  Object asAnyObject() throws BCfiatcast
```

- Many kinds of object references in Ovm
  - Oop is simplest, minimal support for getting blueprint, hash, and skip header
  - **`PragmaModelOp`** denotes an operations dependent on the Object Model selected at build time

Idioms (5)

```
class VM_Address {

  VM_Address add(VM_Word i) throws BCiadd
     { ..hosted behavior.. }

  void setInt(int content) throws BCsetword {...}
```

```
r("setword", new byte[] {(byte)PUTFIELD_QUICK,0,0});
```

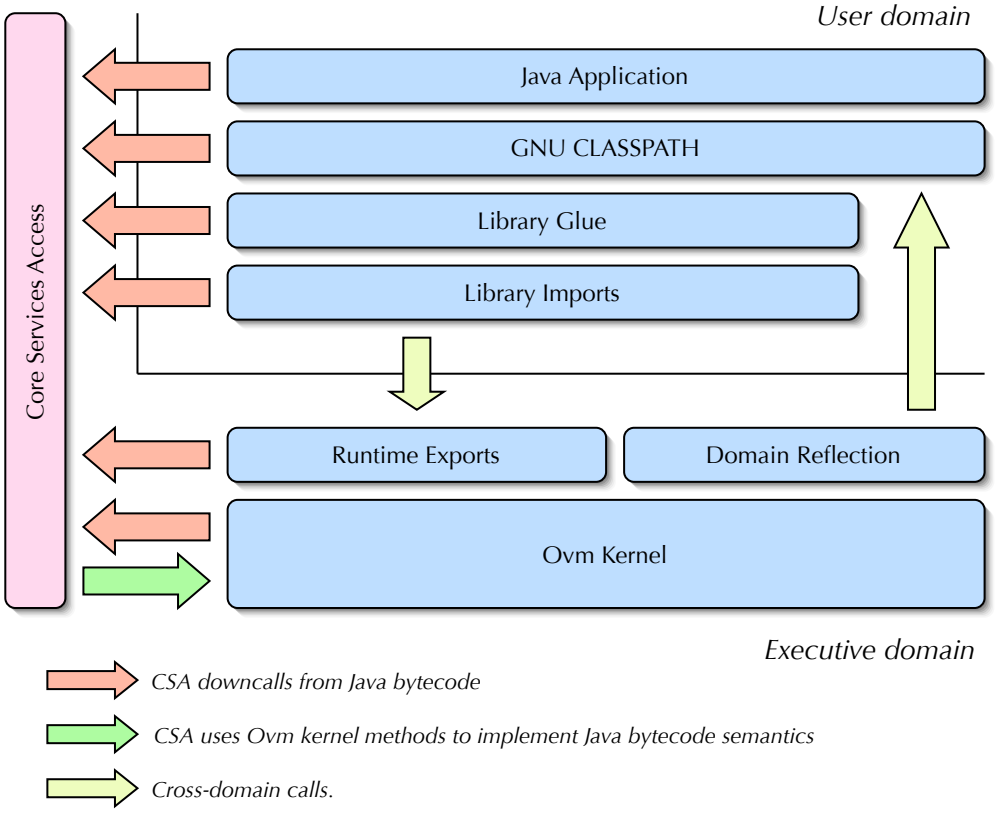
- Bare references are represented by the type **`VM_Address`**
  - core operation have "hosted" behavior written in Java and a translation specified with a pragma (e.g. BCiadd)
  - the translation is specified as a bytecode rewriting

Idioms (7)

```
class S3Model_B_M_F_H extends ObjectModel {

 Oop stamp( VM_Address a, Blueprint bp) {
   a.add(BP_OFFSET).setAddress(VM_Address.fromObject(bp));
   a.add(HASH_OFFST).setInt(SEQ_HASH?nextHash++:a.asInt());
   return super.stamp(a, bp);
 }
 String toCStruct() { return ("struct HEADER {\n"+
      "\tstruct s3_core_domain_S3Blueprint*_blue...
```

- The Object Model determines the bit-level layout of objects
  - size of header determined by the object model,
  - initialization of header performed by the **`stamp`**() method
  - mapping to C++ structs given by **`toCStruct`** (for AOT)

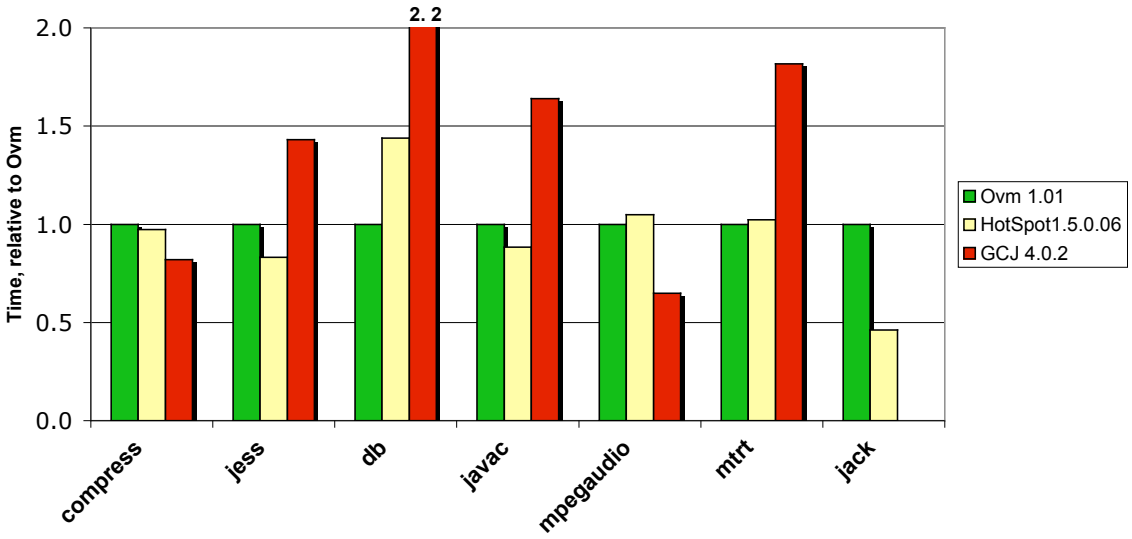
Virtual Machine Architecture



Domain Isolation

- Kernel/User domain isolation necessary
  - Single address space
  - enforced by the type system
  - requires java.lang.Object not built-in
- Cross-domain interaction must be treated specially
  - cross-domain references are opaque
  - cross-domain access is reflective
  - cross-domain exception require special handling

Performance



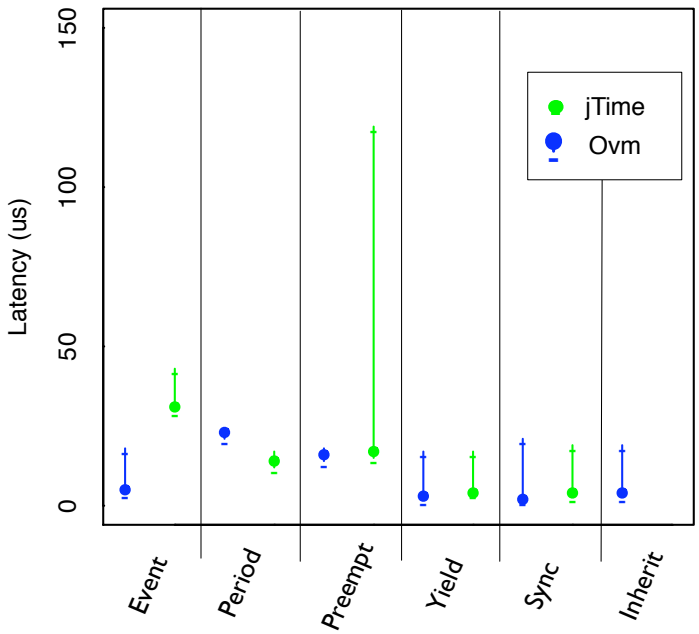
- SpecJVM98, steady state, best of three runs, normalized wrt Ovm
- competitive with SUN Hotspot and better than GCJ

AMD Athlon XP1900+, 1.6GHz, 1GB RTLinux, 2.4.7-timesys-3.1.214
Ovm Java VM, AOT

Implementation

Predictability of RTSJ operations

- Worst case execution times of Ovm vs. commercial RTVM



AMD Athlon XP1900+, 1.6GHz, 1GB RTLinux, 2.4.7-timesys-3.1.214

Cross Domain Calls

```
public ValueUnion call(Oop recv) {
    VM_Area area = MemoryManager.the().getCurrentArea();
    Object r1 = MemoryPolicy.the().enterScratchPadArea();
    try {
        InvocationMessage msg = makeMessage();
        VM_Area r2 = MemoryManager.the().setCurrentArea(area);
        try {
            ReturnMessage ret = msg.invoke(recv);
            ret.rethrowWildcard();
            return ret.getReturnValue();
        } finally { MemoryManager.the().setCurrentArea(r2); }
    } finally { MemoryPolicy.the().leave(r1); }
}
```

- Reflective data structures are allocated in temporary memory areas to circumvent need for GC
  - A scratchpad area is a thread local memory area

Scoped Memory Barriers

```
void readBarrier(VM_Address src)
      throws PragmaInline, PragmaNoBarriers, PragmaNoPollcheck {
    if (!doLoadCheck)  return;
    if (src.diff(heapBase).uLessThan(heapSize))  fail();
 }
```

- Scoped Memory areas require R/W barriers on all reference accesses
  - Read barriers are needed to prevent a NHRT Thread from accessing a heap location (which may be inconsistent if a GC is progress)
  - Write barriers prevent dangling pointers by checking that lifetime of the target object's scope is longer that of the object receiving the reference
  - both types are constant time

Palacz, Vitek. Subtype tests in real time. **ECOOP03**

Threads, Scheduling, Sync

- Compiler enforced cooperative scheduling (aka "green threads")
  - Provides consistent semantics across RT OSes
  - Cheap atomicity on uniprocessors (ie. most embedded systems)
- Preemption by low-overhead compiler-inserted polling

```
void someMethod() {
   ...
   while(...)  {
      ...
   }
}
```

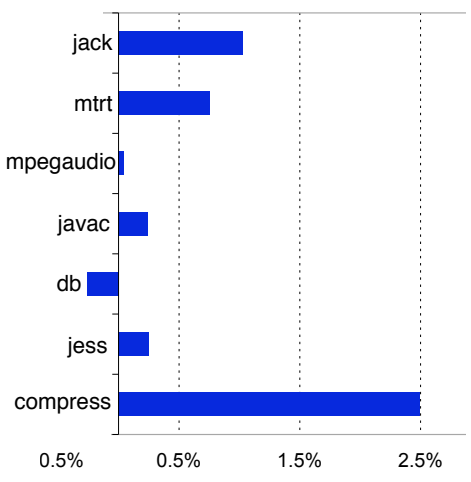
⇒

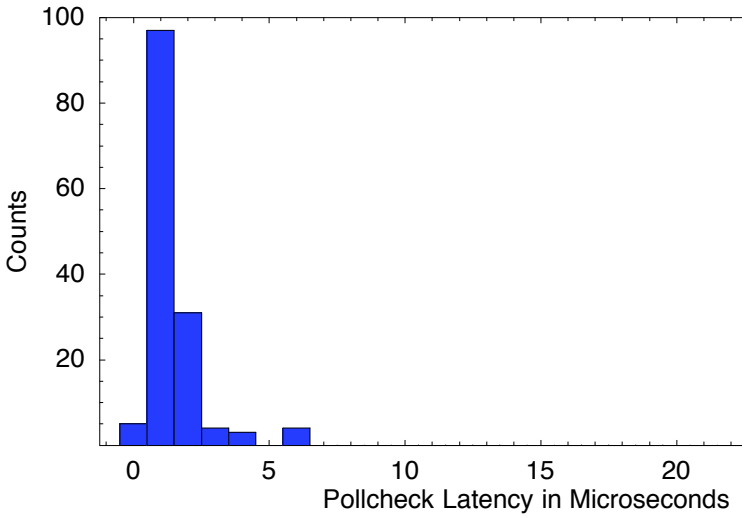
```
void someMethod() {
   POLLCHECK();
   ...
   while(...)  {
      ...
      POLLCHECK();
   }
}
```

```
POLLCHECK:
  if (pollUnion.pollWord == 0) {
     pollUnion.s.notSignaled = 1;
     pollUnion.s.notEnabled = 1;
     handleEvents();
  }
```

Threads, Scheduling, Sync

- Throughput and latency of polling is low.
  - There are some benefits in terms of atomicity which have not been quantified

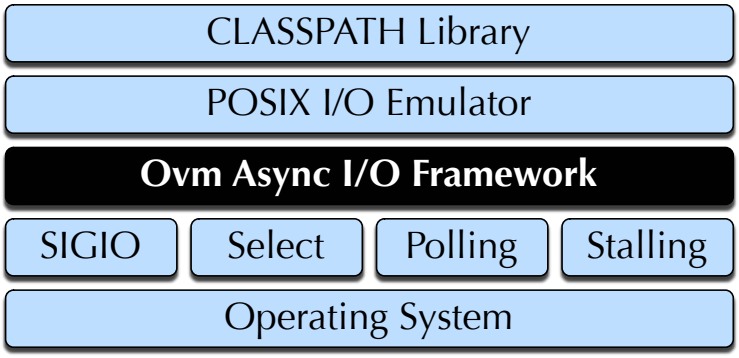




Armbuster, et.al. A Real-time Java Virtual Machine with Applications in Avionics. *To appear* 2006

Threads, Scheduling, Sync

- Cooperative multi-threading must deal with blocking IO calls
  - Ovm implements an Async I/O layer that emulates standard Java libraries
  - four different I/O configurations are provided



Static Analysis

	LOC	Classes	Data	Code
Boeing PRiSMj	108'004	393	22'944 KB	11'396 KB
UCI RT-Zen	202'820	2447	26'847 KB	12'331 KB
GNU classpath	479'720	1974	–	–
Ovm framework	219'889	2618	–	–
RTSJ libraries	28'140	268	–	–

Footprint. Lines of code computed overall all Java sources files (w. comments). Data/Code measure the executable Ovm image for two complete application (PRiSMj and Zen) on a PPC.

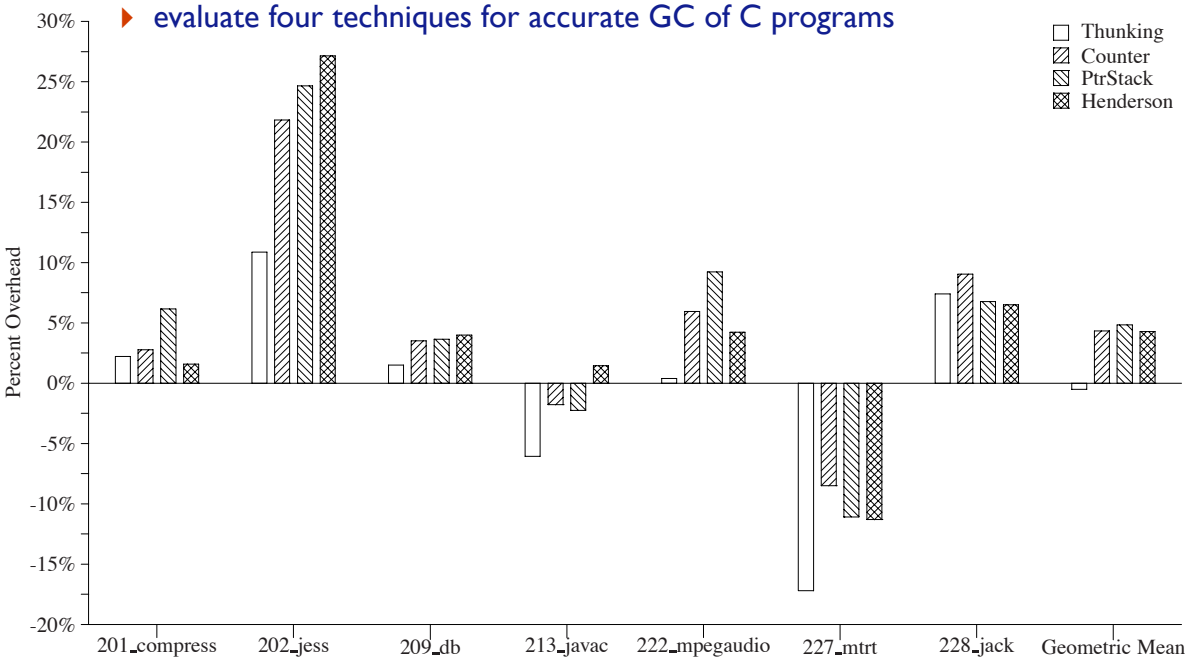
	classes loaded / used	methods defined / used	call sites (% devirt)	casts (% removed)
RTZEN	3266 / 941	20608 / 9408	67514 (89.7%)	5519 (37.7%)
PRiSMj	3446 / 953	13473 / 6616	46564 (89.8%)	73408 (96.9%)

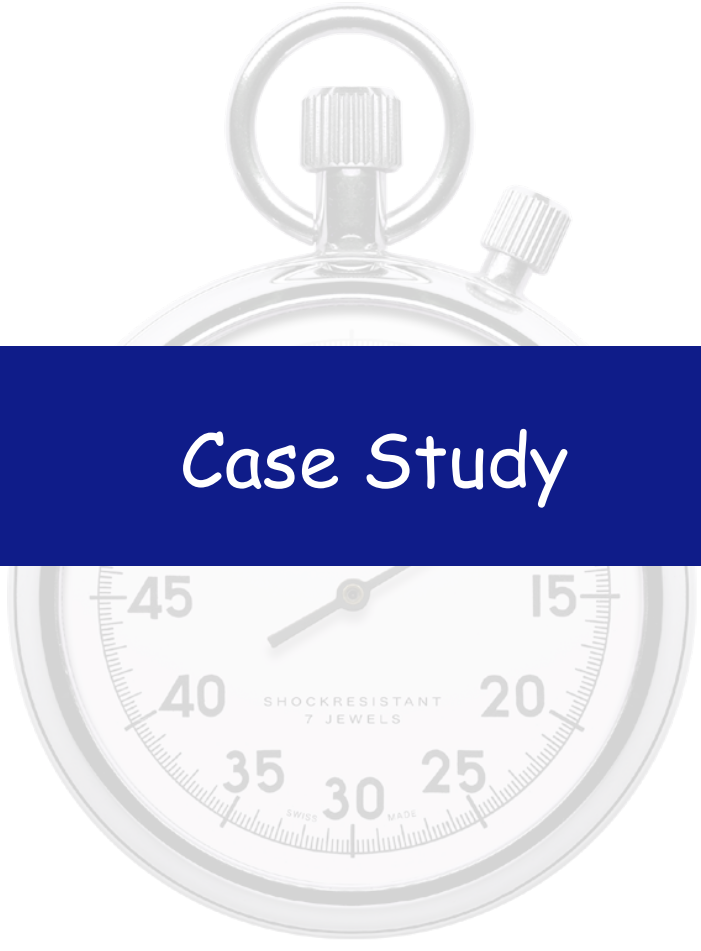
Impact of compiler optimizations.

- Ovm implements RTA, inlining and constant propagation at the bytecode level

Accurate Garbage Collection

- Ovm's ahead-of-time compiler generates C++
  - gcc does not provide information about pointers on the C stack
  - use a mostly copying collector, but for RT must avoid pathological cases
  - evaluate four techniques for accurate GC of C programs

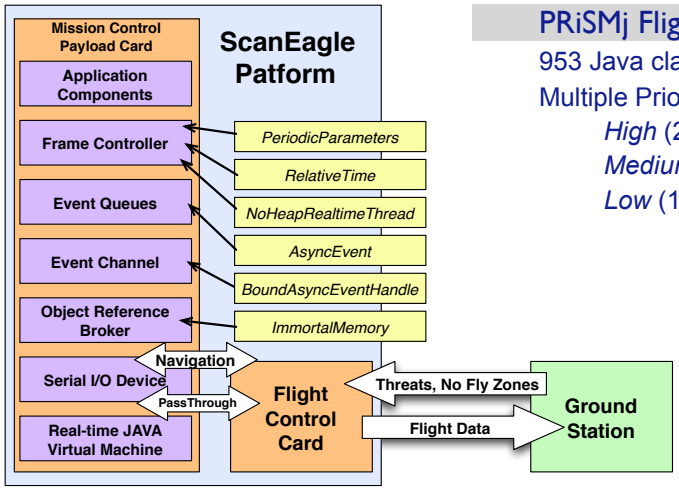




Case Study

Case Study

ScanEagle UAV provides surveillance and reconnaissance data



PRiSMj Flight Software:

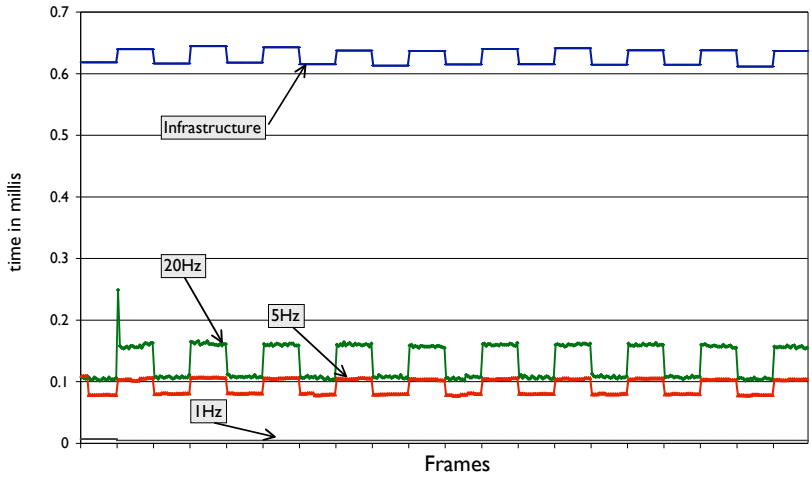
953 Java classes, 6616 methods.

Multiple Priority Processing:

- *High* (20Hz) - Communicate with Flight Controls
- *Medium* (5 Hz) - Computation of navigation data
- *Low* (1 Hz) - Performance Computation

Baker,Cunei,Flack,Pizlo,Prochazka,Vitek,Armbuster,Pla,Holmes. Real-time Java Virtual Machine for Avionics. **RTAS06**

Case Study



- Boeing PRiSMJ 100x synthetic workload
  - *Results faster than C++ version as reported in* [Sharp+ RTSS03]

```
300Mhz PPC, 256MB memory, Embedded Planet Linux
Ovm RTSJ VM, AOT, priority preemptive, PIP locks
```

Lessons Learned

- Case Study
  - First successful RTSJ live flight test.
  - Java development straightforward
  - C++ required testing on x86 & PPC ⇒ compiler incompatibilities ⇒ bugs
- Implementation
  - Open-sourced a compliant RTSJ VM
  - Good performance characteristics
- Impact
  - ScanEagle won a Duke's choice Award at JavaOne.

Acknowledgments

The Ovm team

Jason Baker, *Antonio* Cunei, *Chapman* Flack, *Filip* Pizlo, *Marek* Prochazka, *Christian* Grothoff, *Krista* Grothoff, *Andrei* Madan, *Gergana* Markova, *Jeremy* Manson, *Krzysztof* Palacz, *Jaques* Thomas, *Jan* Vitek, *Hiroshi* Yamauchi ✹ *Purdue University*

David Holmes ✹ *DLTeCH*

DARPA Program Composition for Embedded Systems

NASA/NSF HDCP - Assured Software Composition for Real-Time Systems

NSF Aspectual Configuration of Real-time Embedded Middleware

The Boeing Company