A High Performance Smalltalk-80 Interpreter

Allen Wirfs-Brock
Computer Research Lab.
Tektronix Inc.
Beaverton, Oregon

Agenda

1. Language Background

2. Early implementation experiences

3. Designing a high-performance implementation

Smalltalk History

Developed at Xerox PARC around 1972

Software component of Alan Kay's "Dynabook"

Influenced by Lisp and Simula

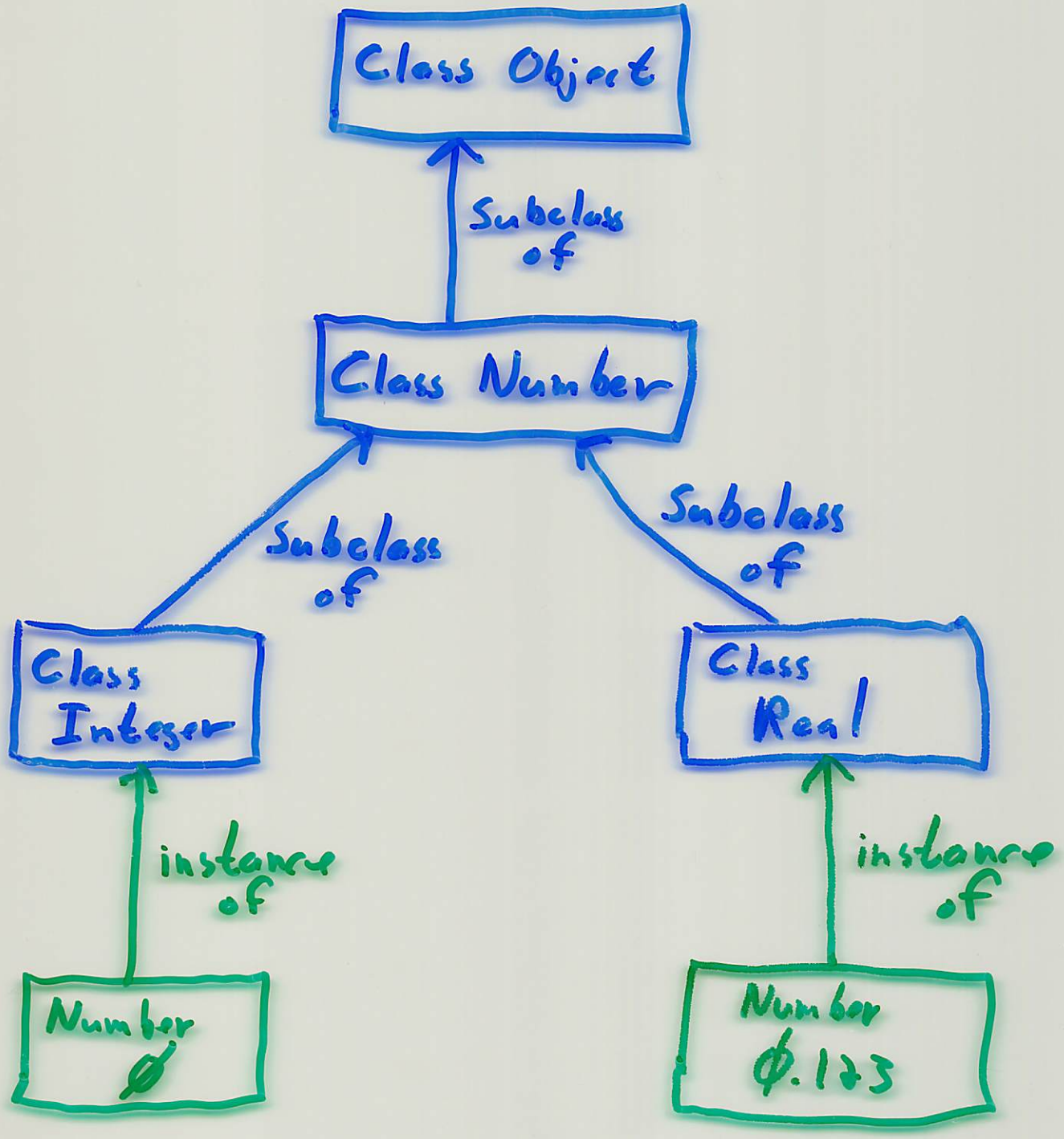
Evolved through several versions.

Objects

- Objects are entities which are able to respond to requests to perform computations. These requests are called *messages*.
- Objects are distinguished by the set of requests to which they respond.
- Objects encapsulate state. This state is only visible from "inside" the object.
- The state contained within an object consists of references to other objects.
- All objects are uniformly named and referenced.
- Objects are dynamically created and cease to exist if not referenced.

Classes

- A class is an object which defines the common characteristics of a group of objects
- Every object is an *instance* of some class
- A class defines the internal state representation and message responses for its instances
- A *subclass* is a hierarchical refinement of its *superclasses*
- A subclass may define additional internal state and add or modify message responses


Class Object
Subclass of
Class Number
Subclass of
Subclass of
Class Integer
Class Real
instance of
instance of
Number ∅
Number ∅.123

Messages

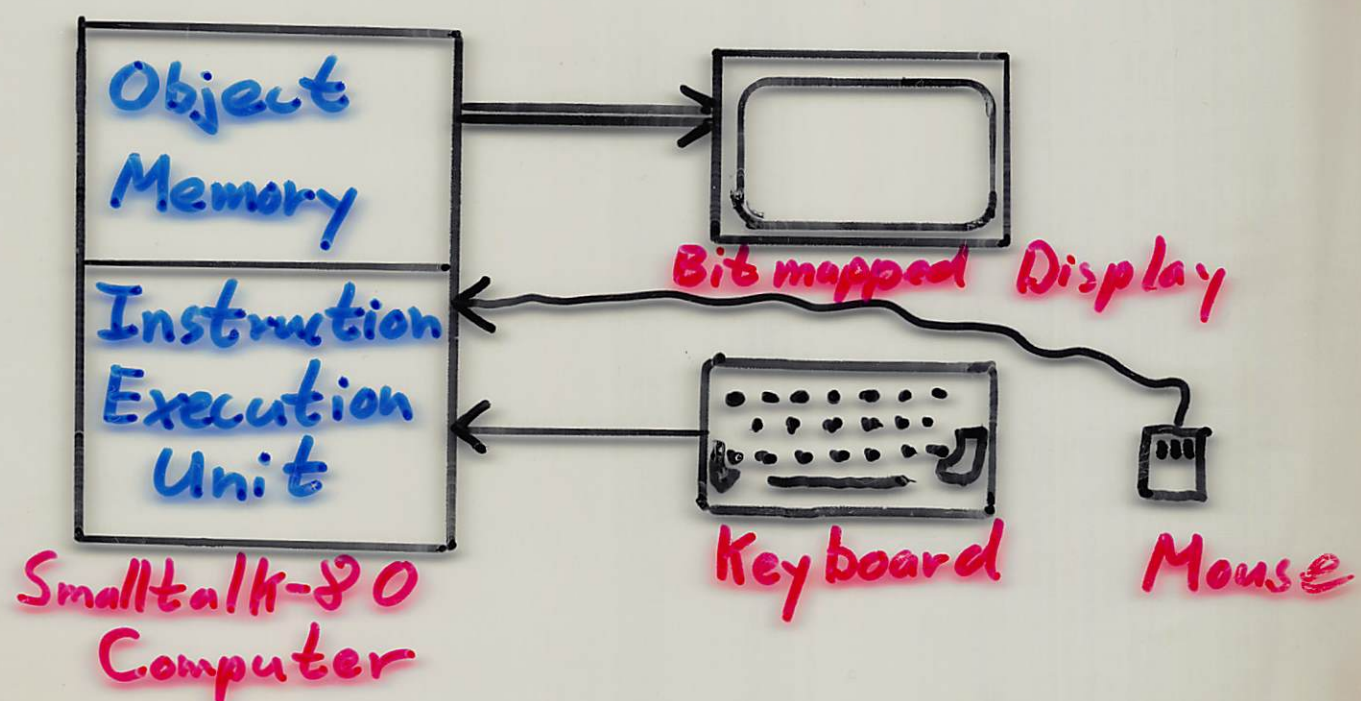
- A message is a request for an object to perform a computation
- An object responds to a message by executing a *method* provided by its class
- A method my modify the internal state of the receiving object and may send messages to other objects
- A message always returns a result to the sender of the message. Result values are object references.
- The method executed in response to a message it dynamically selected

Message Sending

- Find the class of the receiver of the message.
- Look for the message selector in the class' message dictionary.
- If the message selector is not found search in each of the superclasses' dictionaries until it is found.
- When the selector is found create a new context and start executing the compiled method specified in the dictionary.

The Smalltalk-80 compiler generates code for an abstract computer called:

"The Smalltalk-80 Virtual Machine"



Smalltalk-80 Virtual Machine

- The Smalltalk-80 Virtual Machine Specification is the computational model which defines the behavior of Smalltalk programs. The specification defines an abstract computer known as the Smalltalk-80 Virtual Machine.
- The Smalltalk-80 compiler generates code for this abstract machine.
- Smalltalk-80 is implemented on a host computer by building a simulator for the Smalltalk-80 Virtual Machine

Three Parts of the Virtual Machine

- Object Memory -- Provides storage for objects
- Interpreter -- Implements the semantics of message passing
- Primitive Methods -- Provide primitive responses to certain messages

The Object Memory

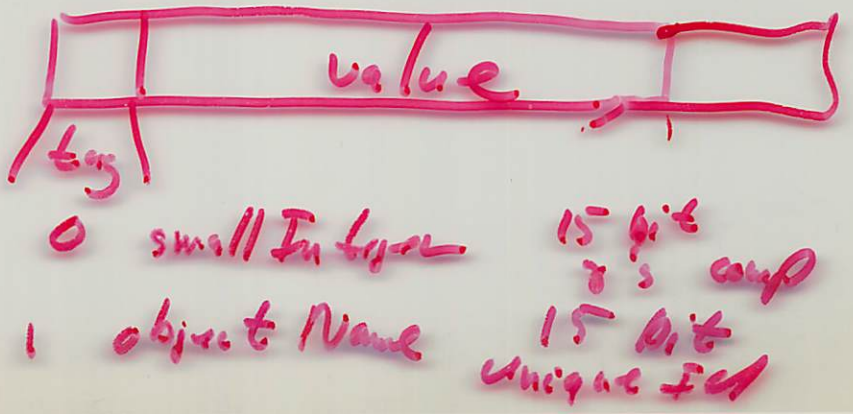
- The object memory provides the basic mechanisms for creating accessing and ultimately destroying objects.
- Objects are identified by unique 16-bit integer names
- The object memory is responsible for reclaiming any storage used by an object after the object is no longer accessible

Object Memory

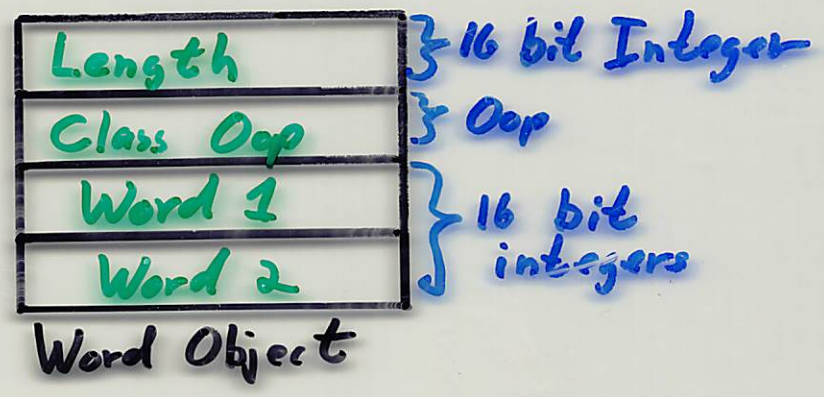
An Object-Oriented Memory Subsystem

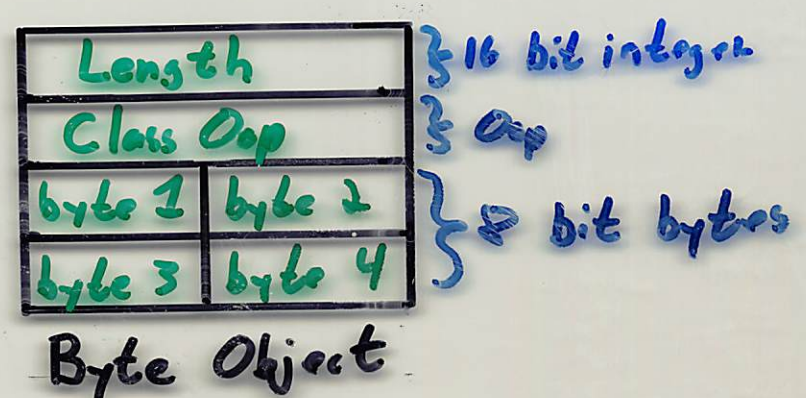
Create an Object
Store into an Object
Retrieve from an Object
Automatic "garbage collection"

Objects are referenced using "object names" not absolute addresses.



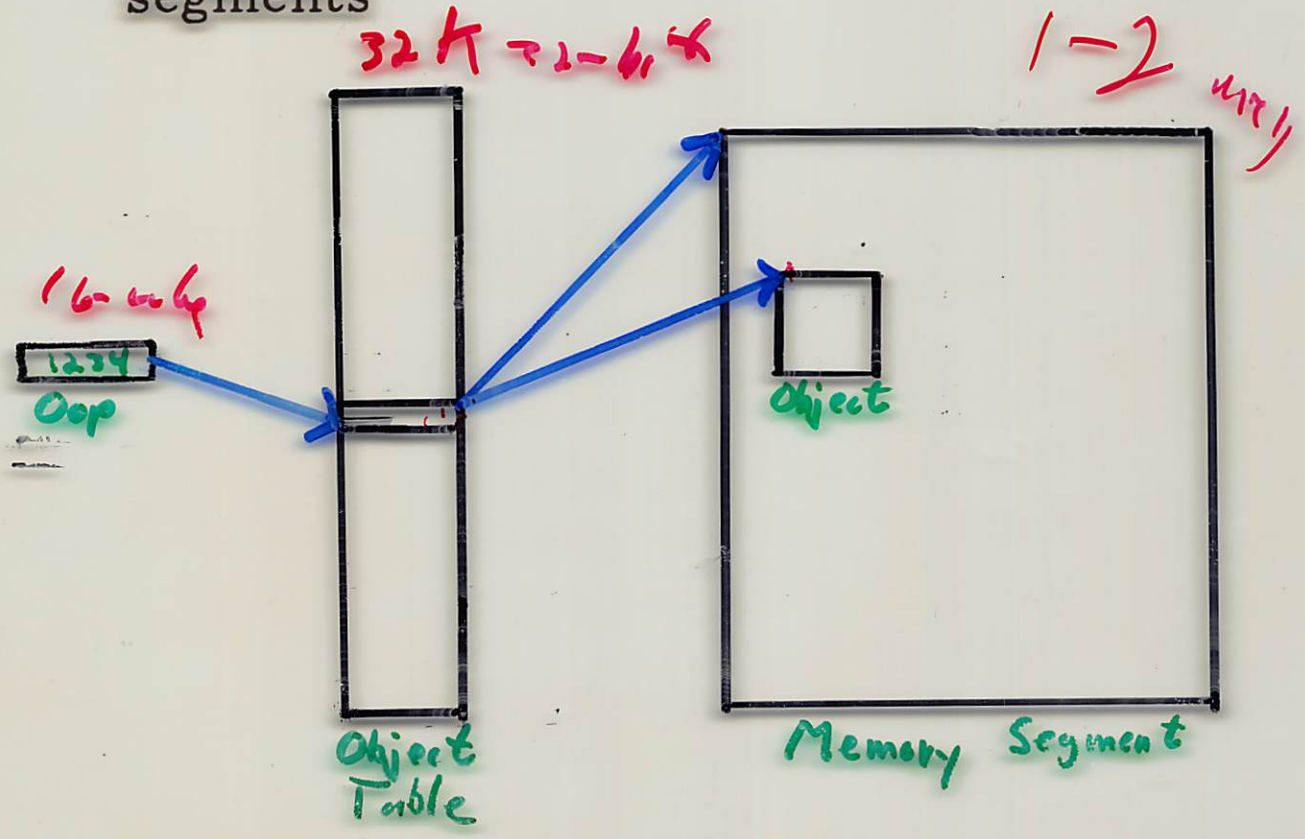
Object Formats





Object Management

- The object memory provides an interface for accessing and changing the fields of objects and for creating new objects
- An object table is used by the object memory to map oops (object names) to the actual memory locations of objects
- The object memory maintains a count of the number of references to each object. An object is deallocated when its reference counts reaches zero.
- The object memory views physical memory as a number of independent segments



The Interpreter

- The Smalltalk-80 interpreter implements the semantics of communicating objects.
- The Smalltalk-80 compiler converts smalltalk expresions into strings of *bytecodes* which are the instructions of for the virtual machine. The interpreter carries out the actions specified by the bytecodes.
- The interpreter uses *context* objects to represent its current state.
- Bytecodes are stored within instances of class CompiledMethod

Table 28-1 The Smalltalk Bytecodes

Range	Bits	Function
0-15	0000iiii	Push Receiver Variable #iiii
16-31	0001iiii	Push Temporary Location #iiii
32-63	001iiiii	Push Literal Constant #iiiii
64-95	010iiiii	Push Literal Variable #iiiii
96-103	01100iii	Pop and Store Receiver Variable #iii
104-111	01101iii	Pop and Store Temporary Location #iii
112-119	01110iii	Push (receiver, true, false, nil, -1, 0, 1, 2)[iii]
120-123	01111ii	Return (receiver, true, false, nil)[ii] From Message
124-125	0111110i	Return Stack Top From (Message, Block)[i]
126-127	0111111i	unused
128	10000000 jjkkkkkk	Push (Receiver Variable, Temporary Location, Literal Constant, Literal Variable)[jj] #kkkkkk
129	10000001 jjkkkkkk	Store (Receiver Variable, Temporary Location, Illegal, Literal Variable)[jj] #kkkkkk
130	10000010 jjkkkkkk	Pop and Store (Receiver Variable, Temporary Location, Illegal, Literal Variable)[jj] #kkkkkk
131	10000011 jjjkkkkk	Send Literal Selector #kkkkk With jjj Arguments
132	10000100 jjjjjjjj kkkkkkkk	Send Literal Selector #kkkkkkkk With jjjjjjjj Arguments
133	10000101 jjjkkkkk	Send Literal Selector #kkkkk To Superclass With jjj Arguments
134	10000110 jjjjjjjj kkkkkkkk	Send Literal Selector #kkkkkkkk To Superclass With jjjjjjjj Arguments
135	10000111	Pop Stack Top
136	10001000	Duplicate Stack Top
137	10001001	Push Active Context
138-143		unused
144-151	10010iii	Jump iii+1 (i.e., 1 through 8)
152-159	10011iii	Pop and Jump On False iii+1 (i.e., 1 through 8)
160-167	10100iii jjjjjjjj	Jump (iii-4)*256+jjjjjjjj
168-171	101010ii jjjjjjjj	Pop and Jump On True ii*256+jjjjjjjj
172-175	101011ii jjjjjjjj	Pop and Jump On False ii*256+jjjjjjjj
176-191	1011iiii	Send (+, -, <, >, <=, >=, =, ~=, *, /, \\, @, bitShift:, //, bitAnd:, bitOr:)[iiii]
192-207	1100iiii	Send (at:*, at:put:*, size*, next*, nextPut:*, atEnd*, ==, class, blockCopy:, value, value:, do:*, new*, new:*, x*, y*)[iiii] (* indicates a selector that can be changed by compiler modification)
208-223	1101iiii	Send Literal Selector #iiii With No Arguments
224-239	1110iiii	Send Literal Selector #iiii With 1 Argument
240-255	1111iiii	Send Literal Selector #iiii With 2 Arguments

Compiled Methods

- A compiledMethod is an object which hold the bytes which represent a Smalltalk method.
- A compiledMethods consists of a literal frame which contains references to other objects and an array of bytecodes.
- Bytecode operations including push and popping the values of temporary variables and the fields of the receiver, conditional and unconditional branches, and message sending.

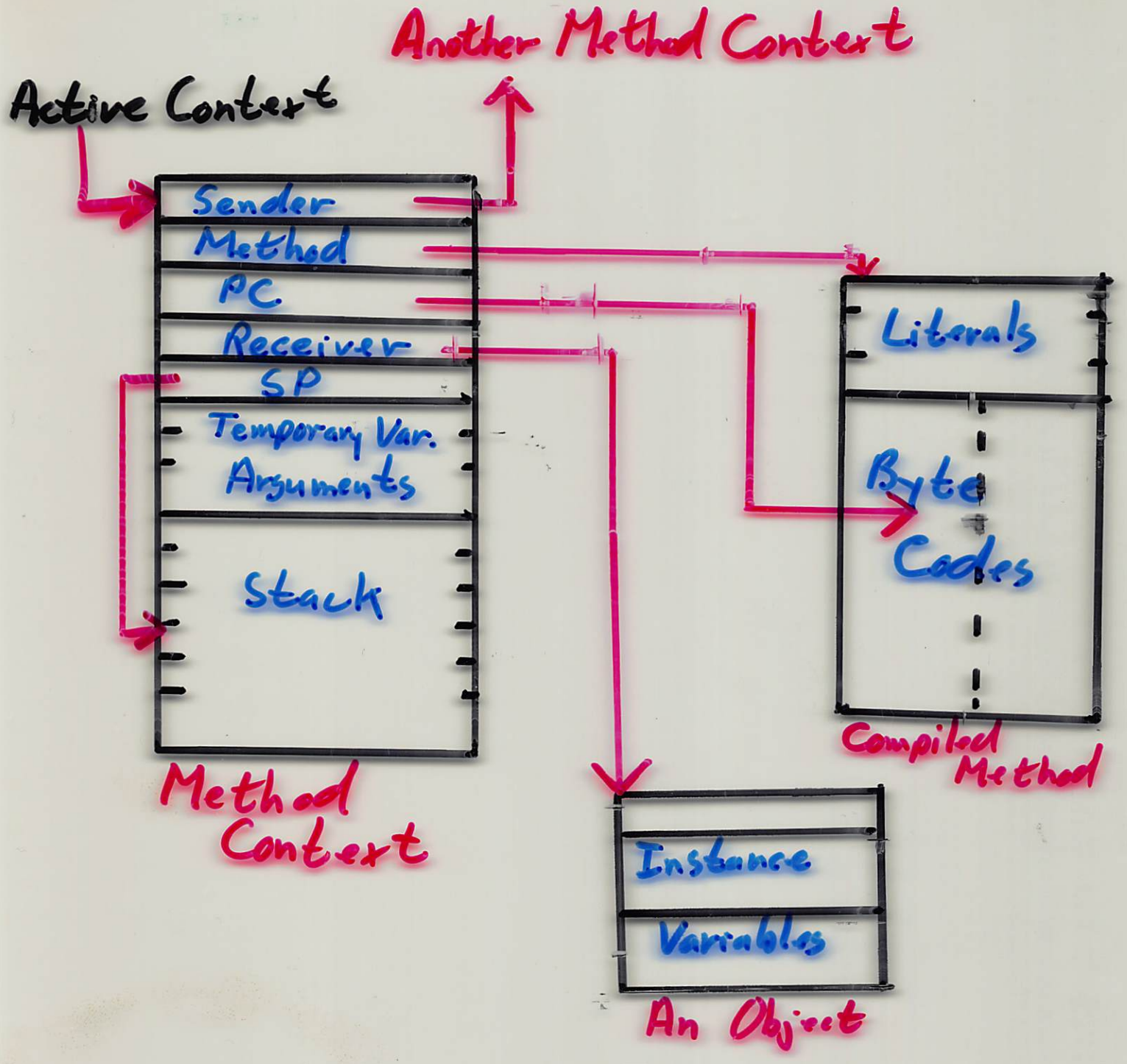
```
Point
    + delta
      |deltaPoint|
      deltaPoint ← delta asPoint.
      ↑ x+deltaPoint x @ (y + deltaPoint
                          y)
```

Compiled Method for Point +

```
Literal 1 : #asPoint
pushTemp: 0      ; delta
send Literal: 1  ; asPoint
popIntoTemp: 1   ; deltaPoint
pushReceiverField: 0   ; x
pushTemp: 1            ; deltaPoint
send: x
send: +
pushReceiverField: 1   ; y
pushTemp: 1            ; deltaPoint
send: y
send: +
send: @
returnTopOfStack
```

Contexts

- A context object represents the current or suspended execution state of the Smalltalk interpreter.
- A new context is created whenever a message is sent.
- A context contain space for temporary variables, an evaluation stack, and references to the receiver of the current message, the compiledMethod for the message, and the context that sent the message.
- Contexts are normal objects which will respond to messages, etc.



Smalltalk execution state

Primitive Methods

- Primitive methods are interpreter routines which implement the responses to selected method.
- Primitives are used when a method can not be easily expressed in Smalltalk or to improve performance.
- Examples of primitive methods include integer and floating point arithmetic, object creation, and Bitblt.

Virtual Machine Implementations

Xerox implementation were microcoded emulations of the Virtual Machine Specifications

In 1981 several groups attempted to build emulations upon conventional architectures.

Vax-11/780
68000

C
Pascal
Assembler

Achieved performance ranged between abysmal and poor!

UC Berkeley Implementation

Vax 11/780 - Unix

Implemented in C

Derived from H.P. implementation

"Assembler-like" coding style

Reference counting short-cuts

Performance .6 Dolphin

DEC

PDP-11/23

Assembly Language
Closely followed specification
Memory Addressability and size problems
"Very Slow"

Vax 11-780 VMS

Assembly Language
32-bit Oops
Modified Baker Garbage Collector
Some context reclaimation optimization
Microscopic performance better than Dolphin
Macro performance .5 Dolphin

Vax 11/730 VMS

Bliss-32
32-bit oops .1
Performance ~~1/2~~ Dolphin

Hewlett-Packard Implementation

Vax 11/780 - Unix

Implemented in C

Fairly literal translation of formal specification

Large object table entries

Heavily instrumented

Performance $\frac{1}{5}$ of Dolphin

Tektronix Implementation

Motorola 68000

Pascal + assembly language

Literal translation of specification
(espically memory manager)

Performance $\frac{1}{5}$ Dolphin

Where does a Smalltalk Interpreter spend its time?

20% Dispatch

20%-40% Message Sends
lookup + activation + return

40%-60% Storage Management
Reference Counting + allocation + freeing

85% of all dynamically created objects are contexts.

60% of all contexts are activated only once. (i.e. leaves of call graph)

11% of executed byte codes cause a context change

Average of 9 byte codes between context changes

Only 5% of contexts are explicitly referenced as objects.

Assembler coded push instance variable

28 instructions (+ dispatch)

most common path:

255 cycles ≡ 28.12 usec

If reference counting was eliminated

7 instructions

73 cycles ≡ 9.1 usec

Reference counting adds 200%

Assuming that LDINST is typical of simplist byte codes an upper bound for 8 mhz 68000 without reference counting is about 57,000 bc/sec

Reference Counting Alternatives

Classical Garabage Collection

Too slow: At 10K bc/sec
create ≈ 600 contexts/sec
≈ 30K bytes of garbage

Baker G.C.

Possible, but needs VM support

Bobbrow/Deutsch Deferred Ref. Counting

Since "almost all" stores are into contexts, eliminating counted references from contexts, eliminates almost all ref. counting overhead.

Deferred Reference Counting

① References in contexts are not counted

② All other references are counted normally

③ When an objects count goes to zero it may not be garabage so check all contexts to see if any reference it.

#3 is HARD

How do you find all contexts?

Searching all objects ≈ Mark sweep

Special lists of contexts have high over head

④ If no room was found in buffer throw out some entries. Periodic mark/sweep will get them

---

Object table change

2 bits taken from ref count

NotCounted: set if reference in this object are not counted (i.e. a context)

Necessary when contexts manipulated as objects

InΦCountBuffer: set if this object in the buffer

Question: How big should buffer be?

Too small — thrashing

Too big — too much time to flush

Guess: ≈ 100 entries

The ZERO-count buffer solution

Zero-count buffer:
a linear array. Entries added at upper limit.

All objects with uncounted references (contexts) + all objects whose ref cnt = ϕ are added to buffer

---

When buffer is full:

① Sweep buffer each non-context object whose count > ϕ is removed from buffer

② For each context in buffer, tag all ϕ-counted objects referenced by ~~obj~~ the context

③ All non-tagged objects in the buffer are deallocated.

Reducing Context Overhead

Possible object retention precludes stack allocation of contexts. (espically in presence of blocks)

But 60% of contexts are leaf activations

so

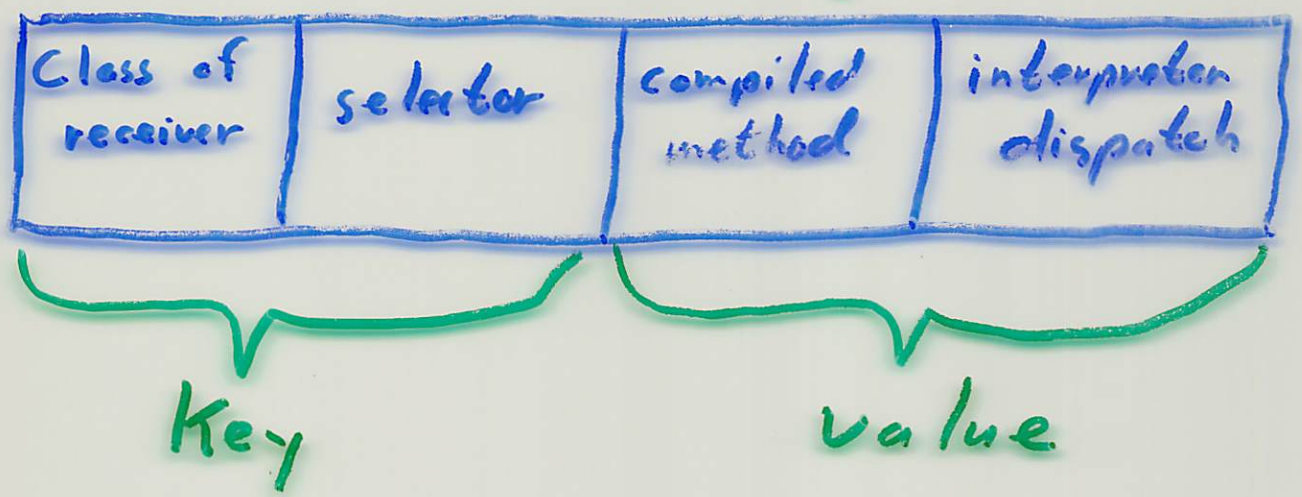
When a new context is needed don't allocate object or assign oop. Instead create a tentitive context.

If tentitive context does a send turn it into real object otherwise discard upon return

Message Lookup Cache

A hash table of message bindings

Table Entry



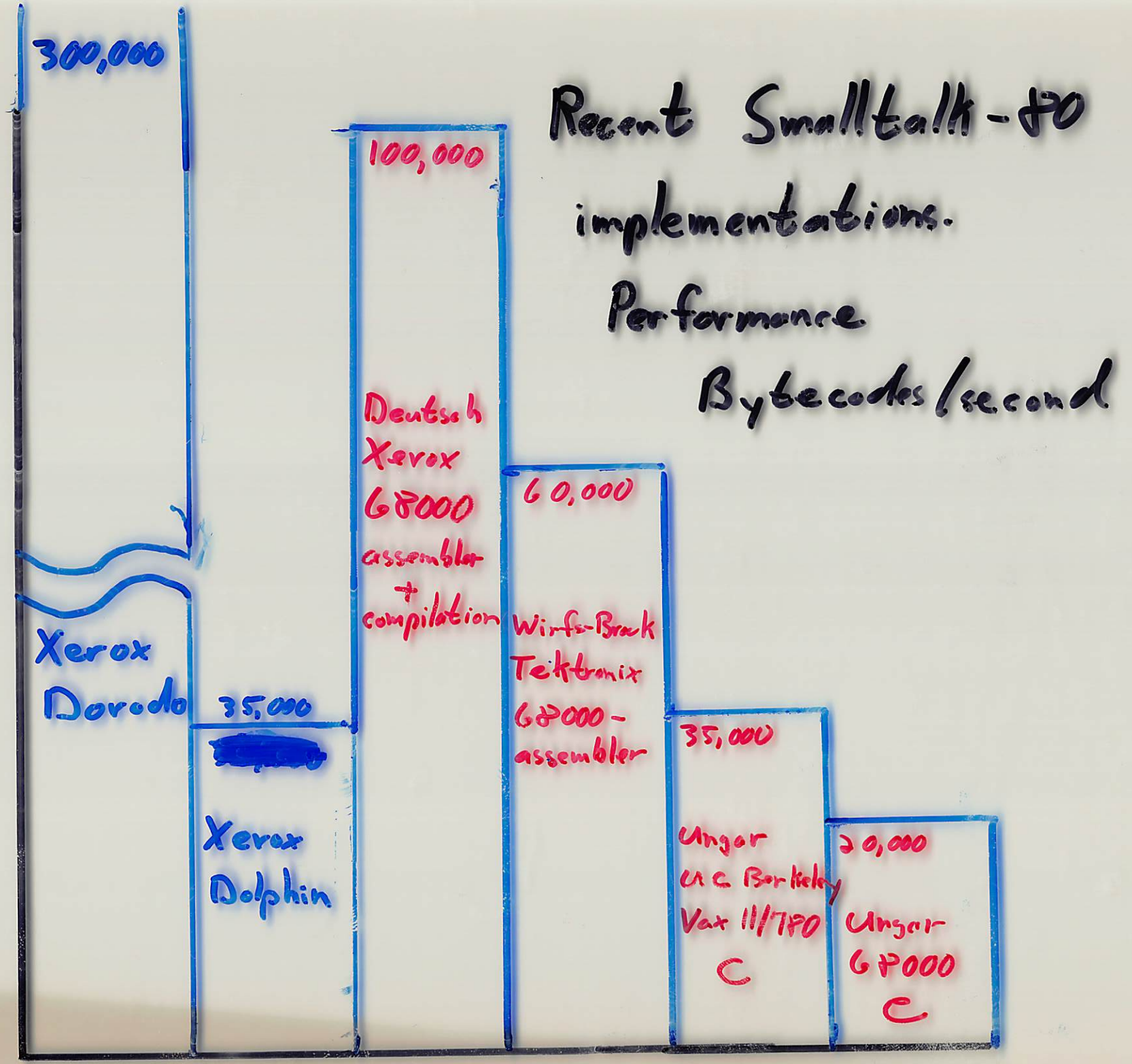
85% - 95% hit rate

Other Tricks

Optimize Data representations to match machine

Cache context states in machine optimized form.

Unwind/de-parameterize everything


300,000
Xerox
Dorado
35,000
[redacted]
Xerox
Dolphin
100,000
Deutsch
Xerox
6800
assembler
+
compilation
60,000
Wirfs-Brock
Tektronix
6800-
assembler
35,000
Ungar
U.C. Berkeley
Vax 11/780
C
20,000
Ungar
68000
C
Recent Smalltalk-80
implementations.
Performance
Bytecodes/second

Other Work

Peter Deutsch - Xerox

Dynamic translation to native code

Inline message caching

Dave Ungar - U.C. Berkeley

Generation G.C.

No object table

SOAR

References

Smalltalk-80 Language + System

Smalltalk-80: The Language and its Implementation
Goldberg + Robson, Addison-Wesley, 1983

Smalltalk-80: The Interactive Programming Environment
Goldberg, Addison-Wesley, 1983

Implementing Smalltalk

Smalltalk-80: Bits of History, Words of Advise
Krasner, Addison-Wesley, 1983

"Efficient Implementation of the Smalltalk-80 System", Deutsch + Schiffman, Proceedings of POPL, ACM, 1984

"Creating Efficient Systems for Object-Oriented Languages"
Suzuki + Terada, Proceedings of POPL, ACM, 1984