Space- and Time-Efficient Implementation of the Java Object Model

David F. Bacon    Stephen J. Fink    David Grove

IBM T.J. Watson Research Center

European Conference on Object-Oriented Programming (ECOOP)
Málaga, Spain
June 12, 2002
Overview

• What is the object model?
  – Internal, universal representation of objects

• Impact on Space
  – Per-object overhead

• Impact on Time
  – Access costs
  – Cache locality (related to space cost)
Goal: Minimize Time and Space

• Perform well for common case
  – Reduce header size from 2 or 3 words to 1
  – Speedup or no slowdown
• Support multiple object models
  – Rapid prototyping
  – Tune object model to GC and other aspects of system
• Can be applied to similar languages
• Engineering, not rocket science
Outline

✓ Introduction
✓ Motivation
➢ Abstract Java Object Model
  • Existing Compression Techniques
  • Development of Object Models
  • New Compression Techniques
  • Measurements
  • Conclusions
class Object {
    Class getClass();
    int hashCode();
    void wait();
    void wait(long);
    void wait(long,int);
    void notify();
    void notifyAll();
    Object clone();
    boolean equals();
    void finalize();
}
Outline

✓ Introduction
✓ Motivation
✓ Abstract Java Object Model
  ➢ Existing Compression Techniques
    • Development of Object Models
    • New Compression Techniques
    • Measurements
    • Conclusions
Compression Techniques: Hashing
(Agesen ’97, Bacon et al.’98)

• Observations
  – Objects usually die before they move
  – Objects usually are not hashed
  – The address of an object is a good hash code (or seed)

• Use 3-state encoding
  – unhashed, hashed, hashed&moved
  – In states unhashed and hashed hash code is address
  – On GC, hashed object has address copied to new object
  – In state hashed&moved, hash code is retrieved from end
Compression Techniques: Locking
(Bacon et al.’98)

• Observations
  – Most objects are not locked
  – Nesting of locks is shallow
  – Most locked objects are not contended

• Encode as 24-bit thin lock
  • In thin case: $fat\ bit=0$, owning thread, nest level
  • In fat case, $fat\ bit=1$, index of inflated lock structure
  • In usual thin case, only 1 compare&swap needed

• Numerous variants and improvements
Outline

✓ Introduction
✓ Motivation
✓ Abstract Java Object Model
✓ Existing Compression Techniques
  ➢ Development of Object Models
  • New Compression Techniques
  • Measurements
  • Conclusions
Original Sun Object Model ('95)
IBM JVM without Handles (’97)
IBM JVM with Thin Locks (’98)
Outline

✓ Introduction
✓ Motivation
✓ Abstract Java Object Model
✓ Existing Compression Techniques
✓ Development of Object Models
➢ New Compression Techniques
  • Measurements
  • Conclusions
Compression Techniques: Class
(Not Previously Implemented in Java)

• Bit-stealing
  • Use low-order bits; mask out when accessing class
  • Cost is 1 dependent ALU instruction

• Indexing
  • Use class number (via table) instead of direct pointer
  • Cost is at least 1 ALU, 1 load, both dependent

• Implicit (a.k.a. BIBOP)
  • All objects on same “page” have same class

• Folklore: these techniques are too costly (class is “hot”)

Compression of Locks

- **Observations**
  - Most objects are not locked
  - Most locked objects have *synchronized* methods

- **Treat lock as an *implicit field***
  - Defined by first *synchronized* method in hierarchy
  - *synchronized* methods will know the offset
  - *synchronized* blocks may need to look up offset
The Lock Nursery

- For objects without *synchronized* methods
  - Hash code is used to look up lock in *lock nursery*
  - With copying GC, nursery can be evacuated
- Nursery can be tuned
  - Single hash table
  - Partitioned hash table
  - Per-hash thin lock (see paper for details)
- In practice, nursery is rarely used
Implementation

- Pluggable object model - `VM_ObjectModel`
  - `VM_JavaHeader`
  - `VM_AllocatorHeader`
  - `VM_MiscHeader`

- Purely procedural interface (heavy inlining)
  - All object fields abstracted

- Support for exposing/requesting bits

- Size computation and allocation support
Single-Word Masked (no lock)
Single-Word Masked (with lock)
Single-Word Indexed (no lock)
Forwarding Pointers
Outline

✓ Introduction
✓ Motivation
✓ Abstract Java Object Model
✓ Existing Compression Techniques
✓ Development of Object Models
✓ New Compression Techniques

➢ Measurements

• Conclusions
Measurement Environment

• Benchmarks
  – SPECjvm98, SPECjbb2000
  – jBYTEEmark, CaffeineMark

• Jikes RVM (baseline 2-word object model)
  – IBM RS/6000 Enterprise Server F80
  – AIX 4.3.3
  – 6 500 MHz PowerPC RS64 III CPUs
  – 4GB RAM, 4MB L2 cache/CPU
Dynamic Lock Type

- Nursery
- Fat
- Thin Slow
- Thin Fast

jBYTEmark, CaffeineMark, 201.compress, 202.jess, 209.db, 213.javac, 222.mpegaudio, 227.mtrt, 228.jack, SPECjbb2000, GEO. MEAN
Lock-Free Objects and Potential Space Savings

![Bar chart showing lock-free objects and potential space savings.](image-url)
Performance (Mark-Sweep Collector)
Performance (Semispace Copying Collector)

- JBYTEmark
- CaffeineMark
- 201.compress
- 202.jess
- 209.db
- 213.javac
- 222.mpegaudio
- 227.mtrt
- 228.jack
- SPECjbb2000
- GEO. MEAN

Legend:
- Green: Bit-stealing
- Purple: Index
Related Work

- Shuf et al: Prolific types
- Dieckmann+Hölzle: Profiling
- Locks
  - Run-time (Krall+Probst, Bacon et al, Agesen et al)
  - Compile-time (Bogda+Hölzle, Aldrich et al, Ruf)
- Stroustrup, Meyer: Multiple Inheritance
- Tip+Sweeney: Class Hierarchy Specialization
Conclusions

• Object model can be flexible
  – Allows tuning to GC, other characteristics

• Single-word header is better
  – Always in space: usually about 14% smaller
  – Usually in time: 0.5-4% faster

• Conventional wisdom wrong on class pointers
  – With a good optimizing compiler
Where to Get it

oss.software.ibm.com/developerworks/oss/jikesrvm

- Or Google to “Jikes RVM”
- Included in Jikes RVM release 2.1.0
- Feedback encouraged/appreciated
- Mechanism for user contributions exists