Accordion Arrays
Selective Compression of Unicode Arrays in Java

Craig Zilles
University of Illinois
Overview

Unicode is overkill for many Java character arrays
- Make char arrays polymorphic (un/compressed)
- Dynamically dispatch for caload/castore
- “Inflate” to handle full Unicode characters

Evaluated in Harmony DRLVM w/Jbb05 & DaCapo
- Speedups of 8% to -1% (2% on average)
Outline

- Motivation: char array usage characteristics
- Accordion compression
  - Allocation / Object model / Dynamic Dispatch
  - Inflation / Garbage Collection
- Experimental Results
- Open Issues / Future Work
Motivation (1)

Much of heap is Unicode character arrays:

- 38% of allocated memory (by space)
- 35% of heap live at GC
Motivation (2)

ASCII is sufficient for most text in much of world:
- Americas, W. Europe, much of Oceania/Africa

Most characters fit in a single byte:

- <.1% of arrays allocated have a >8b char
- 99.7% of char array memory “compressible”
Accordion Compression

Simple compression, can still directly index
- Only works for arrays with all 8-bit chars

Benefits:
- Lower allocation rate: less frequent GCs
- Reduce working set: less GC work, caching
Two Constraints

1. Most objects are short lived, to get benefit:
   **Must Allocate Compressed**

2. Predominantly mutable:
   **No guarantee that 16b char won’t be written**
A Speculative Technique

1. Predict that an array will be compressible.
   - (Unless we can know it won’t be …)

2. Detect when writing a 16b char into 8b array.

3. “Inflate” the character array in this case.
Object implementation

- Three accordion array types

- Encode type as part of length field
- Steal top two bits of length
Object allocation

- Four allocation paths:
  - String from UTF-8  
    - Can know type at alloc. time
  - Clone()
  - newarray  is most of the allocations
  - JNI  not seen in these programs

---

Craig Zilles

University of Illinois
Polymorphic caload/castore

- Somewhat surprisingly, performance doesn’t stink
Optimize compressed case

Make test for compressed cheap:
  • uncompressed, inflated types have top bit set

Two added branches for isolated castore

castore:  lw  rlength, 8(rbase)  # 1 load length field (w/type)
blt  rlength, 0, uncomp1       # 2 branch if not compressed
compres: bge  rindex, rlength, out-of-bounds # 3 perform bounds check
        bgt  rchar, 255, inflate         # 4 char too large? need to inflate
        add  rtmp, rbase, rindex         # 5 add base and index
        sb   rchar, 12(rtmp)             # 6 perform 8b store

  • one for caload

In loops, opportunity for unswitching (not impl.)
Dynamically executed code

- Added operations are off the critical path
- Dispatch branch is highly biased -> predictable
- OOO processors speculate, superscalar eats ILP
Inflations

Allocate new object, copy, set up forwarding

- Must be done in thread-safe manner
  - Big hammer: stop-the-world (at GC-safe pts.)
  - Expensive, even if rare...
Inflations, cont.

Inflations often happen early in object lifetime

```java
char[] characters = new char[wordLength];
word.getChars(0, wordLength, characters, 0);
output = new char[count];
i = o - offset;
System.arraycopy(value, offset, output, 0, i);
```

Stop-the-world unneeded if not escaped

- Simplistic analysis works (local scope post inlining)

On a per-castore basis statically select:

1. Inflate with stop-the-world, or
2. Inflate without stop-the-world
Inflation w/o relocation

Often inflated object is the most recently allocated
- For bump ptr. allocator, can be inflated in place
- At inflation time, check if alloc. can be extended
GC modifications

A mark-sweep collector:
- Needs to know size of object for copying
- Must treat forwarding pointer as “slot”
  - Liveness and relocation

- Optimization: remove level-of-indirection
  - Point pointers to “inflated” to “uncompressed”
- Little opportunity for “deflation”
Other modifications

- arraylen
  - Strip type from length field
- System.arraycopy()
  - Modified to do dyn. dispatch on both array types
- JNI
Evaluation Setup

- Apache Harmony DRL JVM (x86)
- Benchmarks:
  - SPECJBB2005, 8 warehouses, 512MB heap
  - 9 DaCapo (2006-10), 64MB heap (hsqldb: 256MB)
- 2.66Ghz Core 2 Duo, 2GB memory
- Red Hat Enterprise Linux (2.6.9-42.08.ELsmp)
Performance

- 8% to -1% speedup (2% on average)
- Limit on speedup, little time spent in GC
Reduction in GC freq, time

- GC time reduction more than just fewer GC’s
- Note lusearch and xalan in particular
Reduction in live set at GC

- Smaller live sets reduce GC copying overhead
Conclusion/Future Work

Accordion Arrays: Unicode array compression
- Can significantly reduce heap allocation rate
- Non-trivial performance benefit for heap-intensive workloads

Open questions/future work:
- Support 16-bit character intensive code
- Better mechanisms for handling inflation
- Reducing instruction overhead for narrow machines
Future Work