Interfacing Java to the Virtual Interface Architecture

Chi-Chao Chang
Dept. of Computer Science
Cornell University

(joint work with Thorsten von Eicken)

Preliminaries

High-performance cluster computing with Java

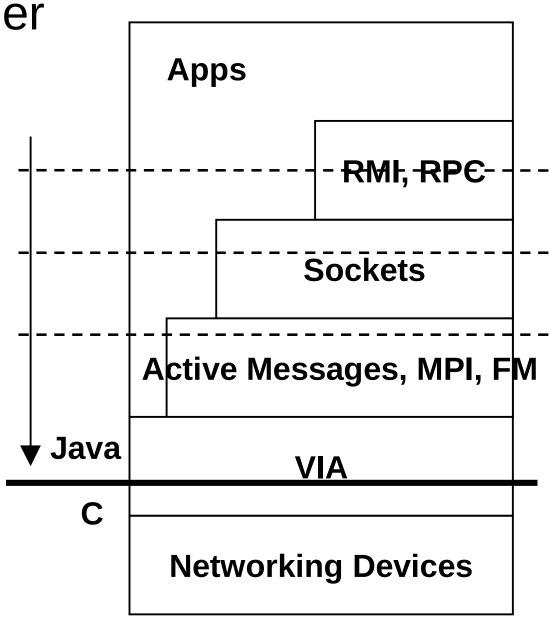
- on homogeneous clusters of workstations

User-level network interfaces

- direct, protected access to network devices
- Virtual Interface Architecture: industry standard
  - Giganet's GNN-1000 adapter

Improving Java technology

- Marmot: Java system with static bc→x86 compiler



Javia: A Java interface to VIA

✔ bottom-up approach

✔ minimizes unverified code

✔ focus on data-transfer inefficiencies

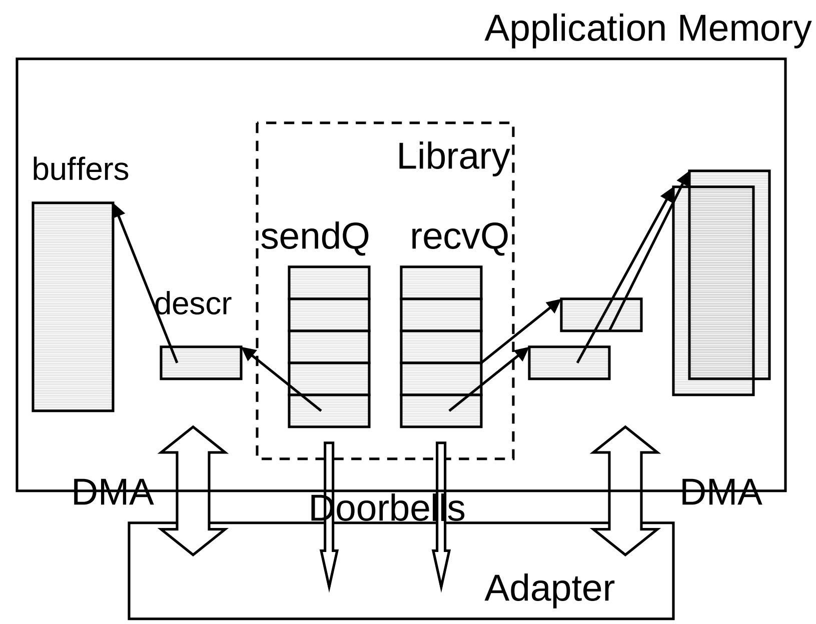
VIA and Java

VIA Endpoint Structures

- buffers, descriptors, send/recv Qs
- pinned to physical memory

Key Points

- direct DMA access: zero-copy
- buffer mgmt (alloc, free, pin, unpin) performed by application
  - buffer re-use amortizes pin/unpin cost (~ 5K cycles on PII-450 W2K)



Memory management in Java is automatic...

- no control over object location and lifetime
  - copying collector can move objects around
- clear separation between Java heap (GC) and native heap (no GC)
  - crossing heap boundaries require copying data...

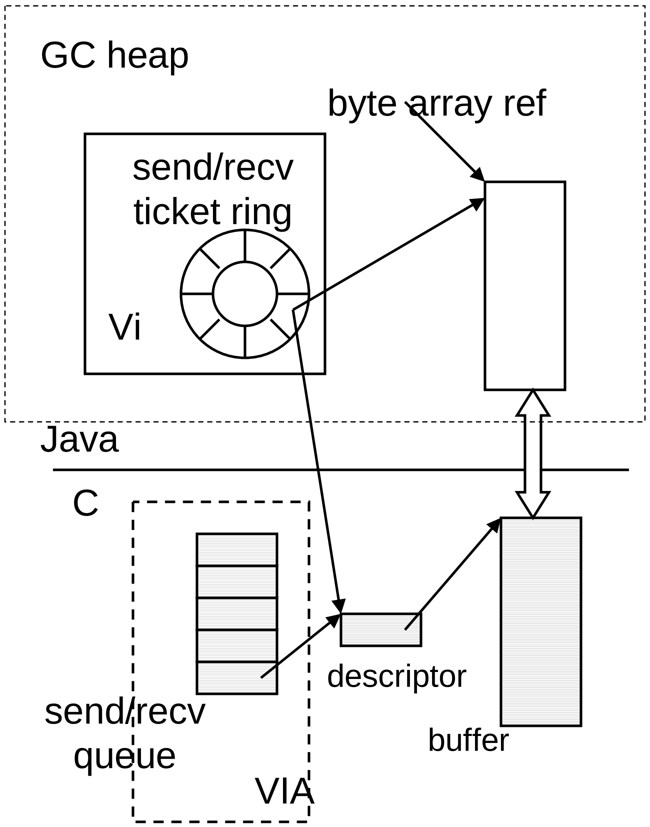
Javia-I

Basic Architecture

- respects heap separation
  - buffer mgmt in native code
- Marmot as an “off-the-shelf” system
  - copying GC disabled in native code
- primitive array transfers only

Send/Recv API

- non-blocking
- blocking
  - bypass ring accesses
  - copying eliminated during send by pinning array on-the-fly
  - recv allocates new array on-the-fly
- cannot eliminate copying during recv



Javia-I: Performance

Basic Costs (PII-450, Windows2000b3):

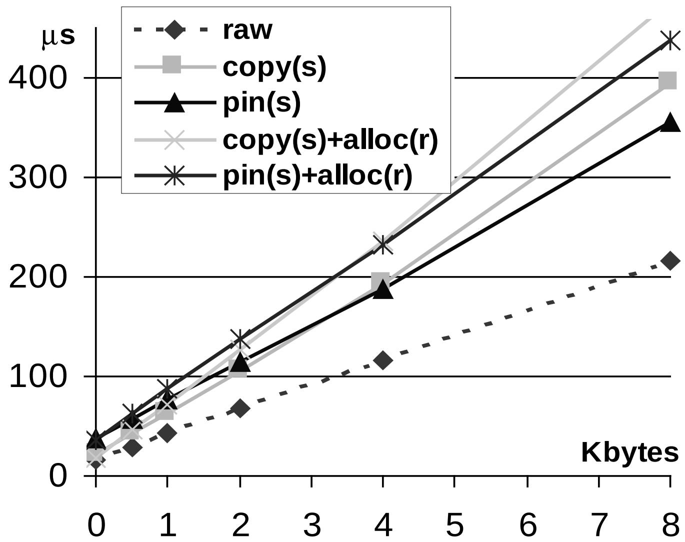
VIA pin + unpin = (10 + 10)us

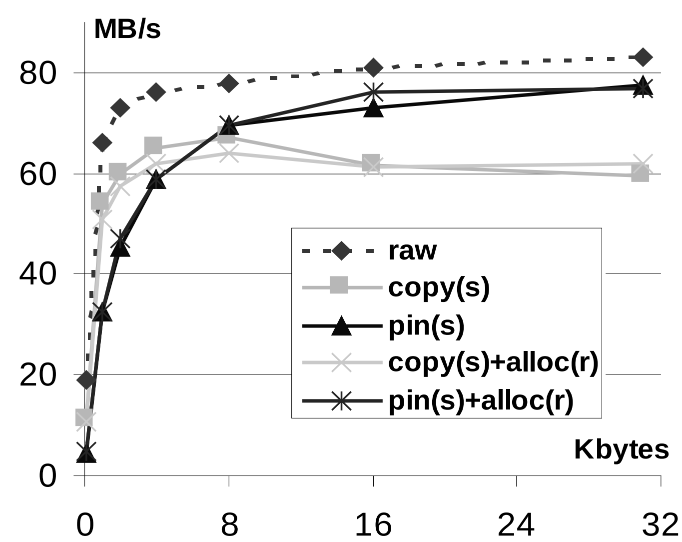
Marmot: native call = 0.28us, locks = 0.25us, array alloc = 0.75us

Latency: N = transfer size in bytes

16.5us + (25ns) * N	raw
38.0us + (38ns) * N	pin(s)
21.5us + (42ns) * N	copy(s)
18.0us + (55ns) * N	copy(s)+alloc(r)

BW: 75% to 85% of raw, 6KByte switch over between copy and pin





jbufs

Motivation

- hard separation between Java heap (GC) and native heap (no GC) leads to inefficiencies

Goal

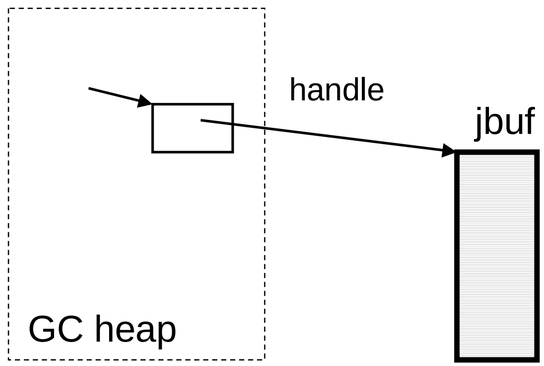
- provide buffer management capabilities to Java without violating its safety properties

jbuf: exposes communication buffers to Java programmers

1. *lifetime control*: explicit allocation and de-allocation
2. *efficient access*: direct access as primitive-typed arrays
3. *location control:* safe de-allocation and re-use by controlling whether or not a jbuf is part of the GC heap
   - heap separation becomes *soft* and *user-controlled*

jbufs: Lifetime Control

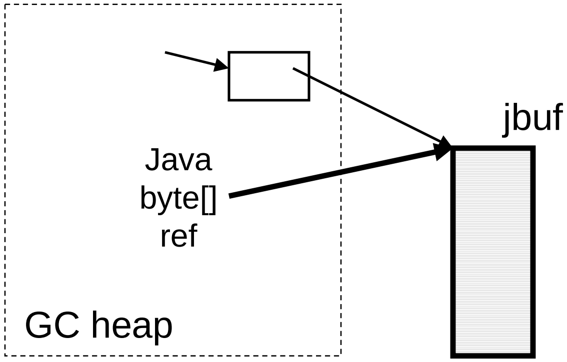
```
public class jbuf {
 public static jbuf alloc(int bytes);/* allocates jbuf outside of GC heap */
 public void free() throws CannotFreeException; /* frees jbuf if it can */
}
```



1. jbuf allocation does not result in a Java reference to it
   - cannot access the jbuf from the wrapper object
2. jbuf is not automatically freed if there are no Java references to it
   - `free` has to be explicitly called

jbufs: Efficient Access

```
public class jbuf {
 /* alloc and free omitted */
 public byte[] toByteArray() throws TypedException;/*hands out byte[] ref*/
 public int[] toIntArray() throws TypedException; /*hands out int[] ref*/
 . . .
}
```



3. (*Storage Safety*) jbuf remains allocated as long as there are array references to it
   - when can we ever free it?
4. (*Type Safety*) jbuf cannot have two differently typed references to it at any given time
   - when can we ever re-use it (e.g. change its reference type)?

jbufs: Location Control

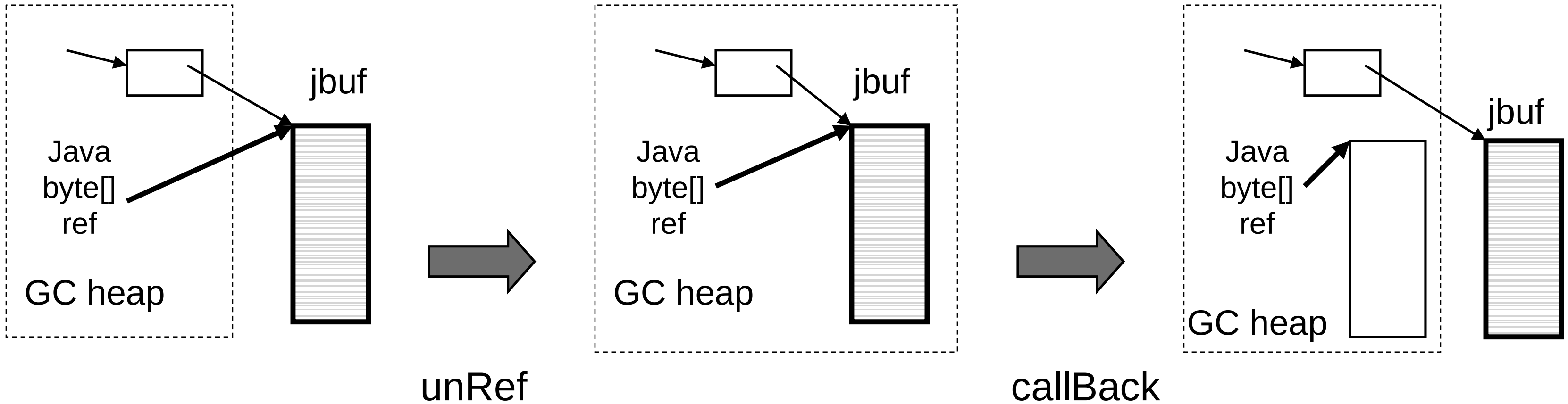
```
public class jbuf {
 /* alloc, free, toArrays omitted */
 public void unRef(CallBack cb); /* app intends to free/re-use jbuf */
}
```

Idea: Use GC to track references

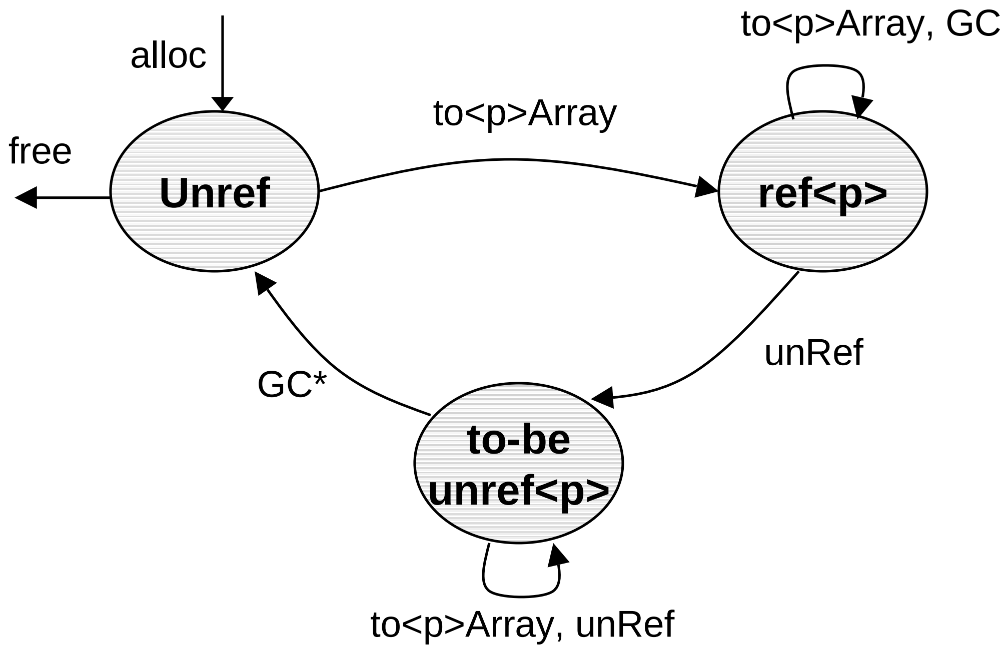
unRef: application claims it has no references into the jbuf

- jbuf is added to the GC heap
- GC verifies the claim and notifies application through callback
- application can now free or re-use the jbuf

Required GC support: change scope of GC heap dynamically



jbufs: Runtime Checks



Type safety: *ref* and *to-be-unref* states parameterized by primitive type

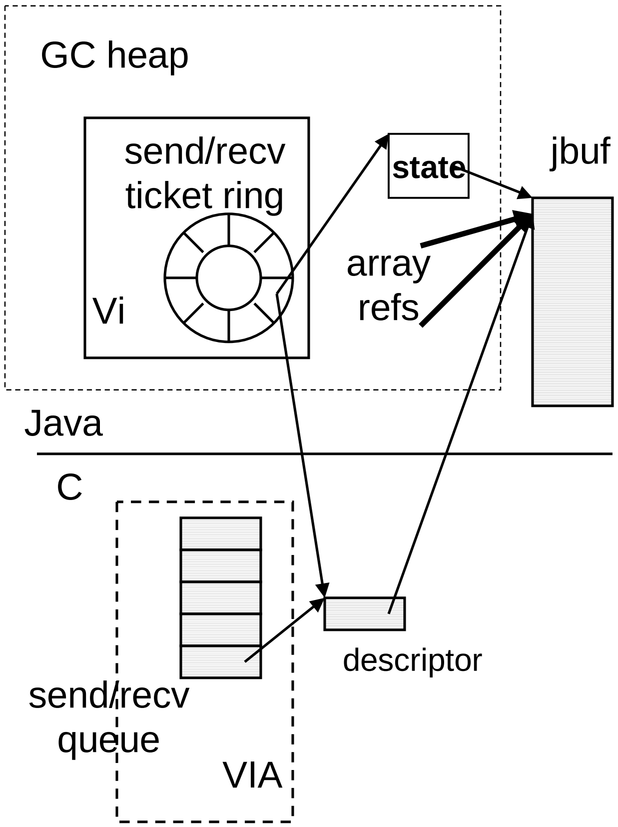
GC* transition depends on the type of garbage collector

- non-copying: transition only if all refs to array are dropped before GC
- copying: transition occurs after every GC

Javia-II

Exploiting jbufs

- explicit pinning/unpinning of jbufs
- only non-blocking send/recvs
- additional checks to ensure correct semantics



Javia-II: Performance

Basic Costs

allocation = 1.2us, to*Array = 0.8us, unRefs = 2.5 us

Latency (n = xfer size)

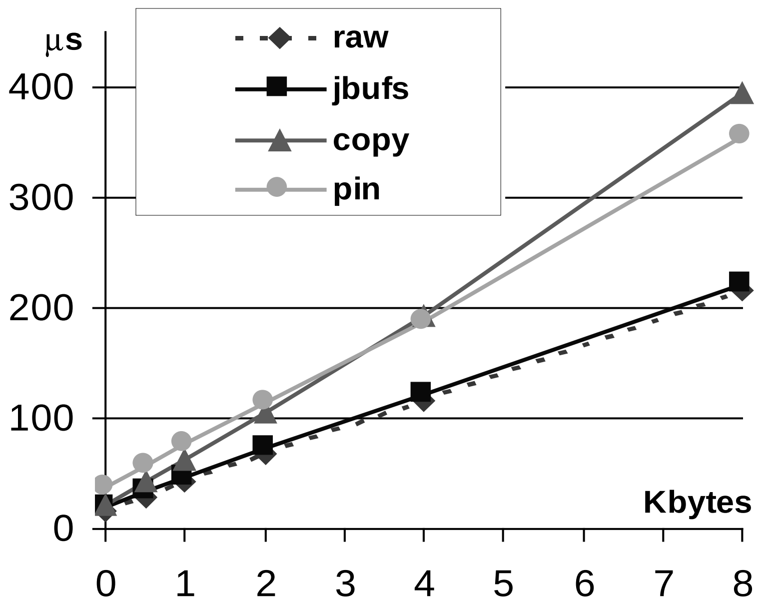
16.5us + (0.025us) * n   raw

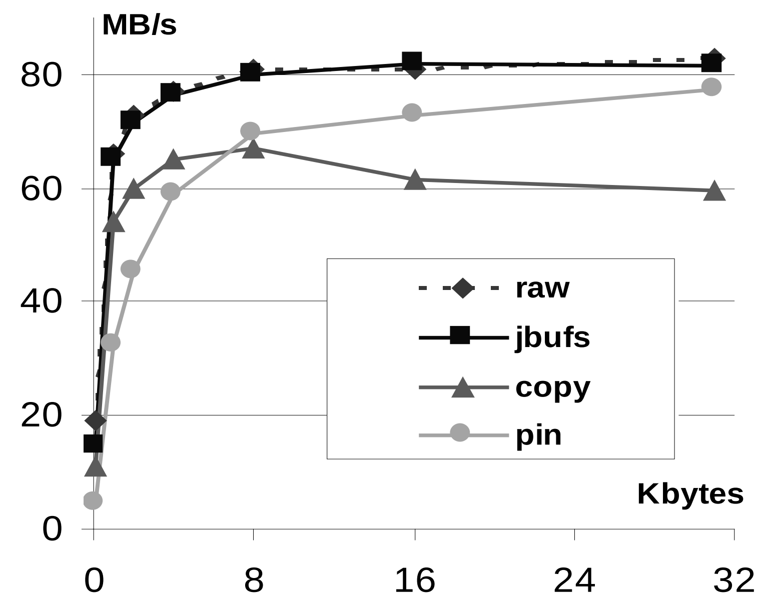
20.5us + (0.025us) * n   jbufs

38.0us + (0.038us) * n   pin(s)

21.5us + (0.042us) * n   copy(s)

BW: within margin of error (< 1%)





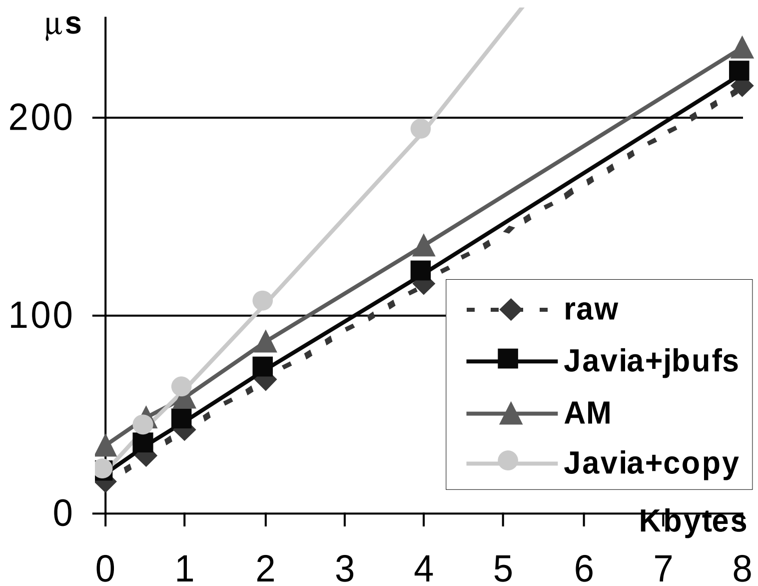
Exercising Jbufs

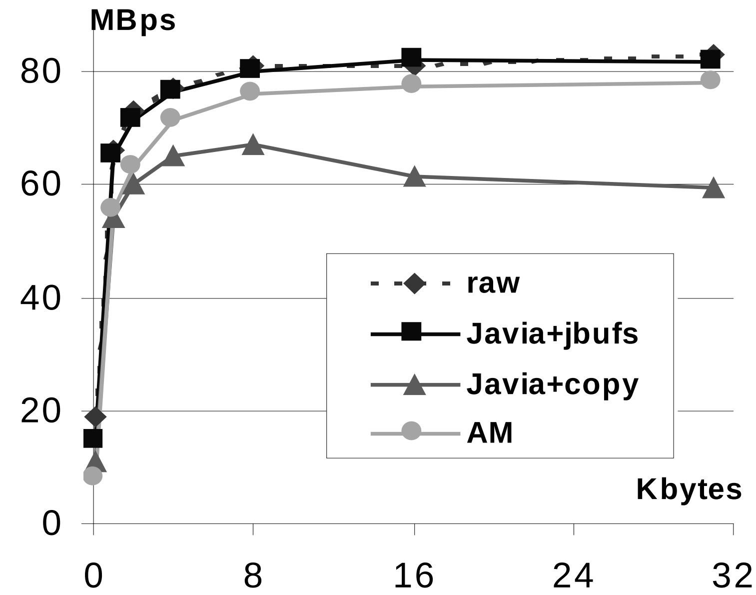
Active Messages II

- maintains a pool of free recv jbufs
  - jbuf passed to handler
  - `unRef` is invoked after handler invocation
  - if pool is empty, alloc more jbufs or reclaim existing ones
- copying deferred to GC-time *only if needed*

```
class First extends AMHandler {
  private int first;
  void handler(AMJbuf buf, …) {
    int[] tmp = buf.toIntArray();
    first = tmp[0];
  }
}
class Enqueue extends AMHandler {
  private Queue q;
  void handler(AMJbuf buf, …) {
    int[] tmp = buf.toIntArray();
    q.enq(tmp);
  }
}
```

AM-II: Preliminary Numbers





Latency about 15μs higher than Javia

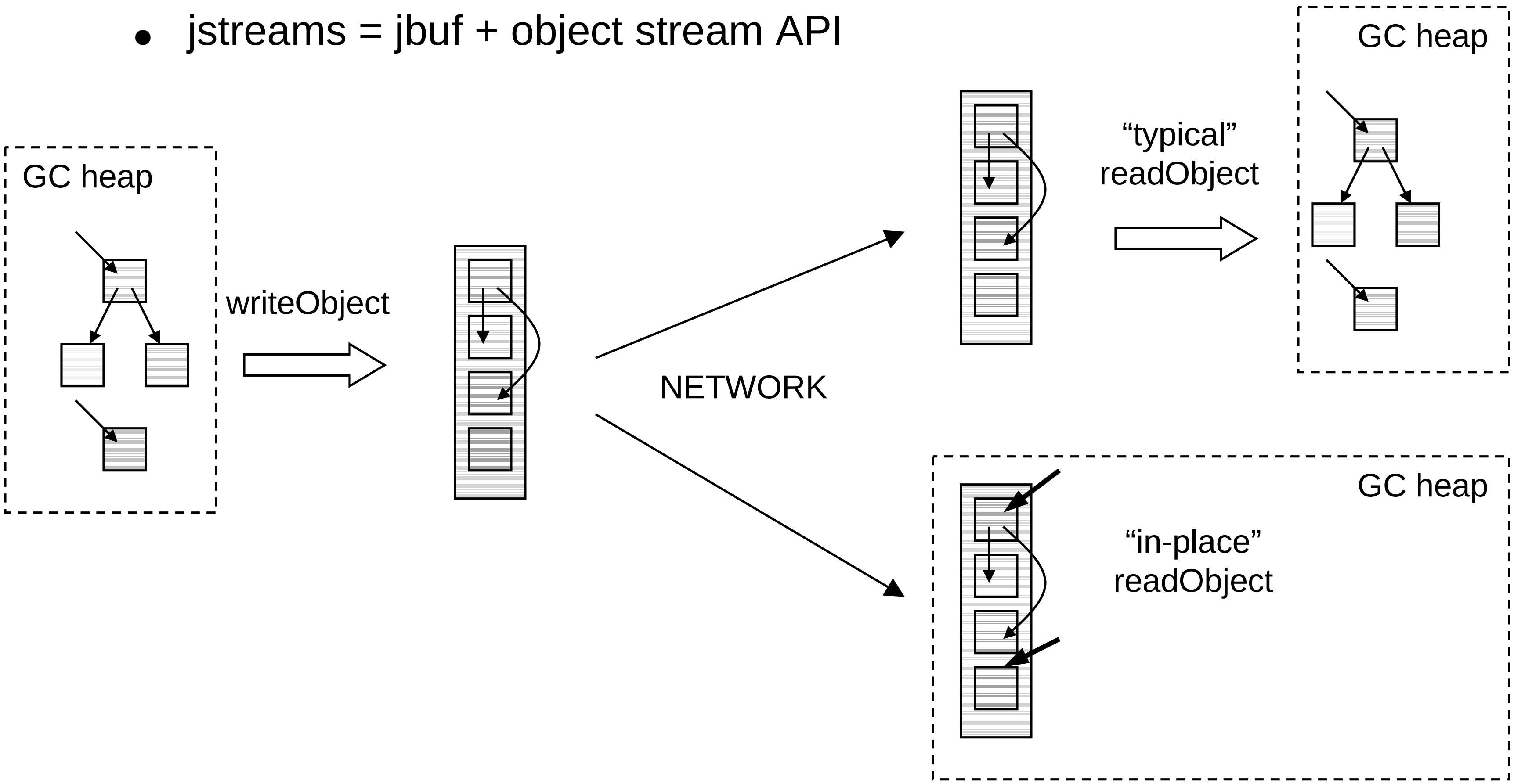
- synch access to buffer pool, endpoint header, flow control checks, handler id lookup
- room for improvement

BW within 3% of peak for 16KByte messages

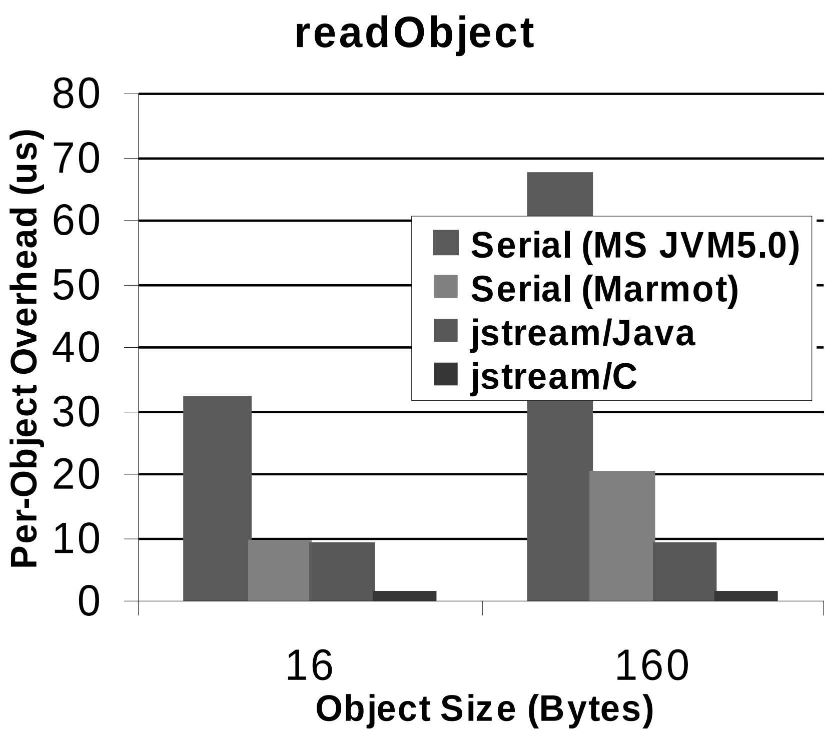
Exercising Jbufs again

“in-place” object unmarshaling

- assumption: homogeneous cluster and JVMs
- defer copying and allocation to GC-time if needed
- jstreams = jbuf + object stream API



jstreams: Performance



readObject cost constant w.r.t. object size

- about 1.5µs per object if written in C
- pointer swizzling, type-checking, array-bounds checking

Summary

Research goal:

Efficient, safe, and flexible interaction with network devices using a safe language

Javia: Java Interface to VIA

- native buffers as baseline implementation
  - can be implemented on off-the-shelf JVMs
- jbufs: safe, explicit control over buffer placement and lifetime
  - ability to allocate primitive arrays on memory segments
  - ability to change scope of GC heap dynamically
- building blocks for Java apps and communication software
  - parallel matrix multiplication
  - active messages
  - remote method invocation