Region-based Memory Management

for Real-time System

Jan Vitek

PURDUE UNIVERSITY

This talk

- High-level programming languages facilitate software development by abstracting error prone or tedious tasks
- Memory management is error prone and tedious, but real-time systems sometimes need fine grained control
- This talk shows how to regain control over memory ... when needed
- The proposed solution is a hybrid model, a language running mostly on GC but with spurts of manual allocation
- A generalization of this problem is to devise practical techniques for constraining heap reference structures
- Our solution leverages a long line of results from type theory, the key contribution is that the solution is surprisingly simple

Embedded Real-time Systems

- Real-time embedded systems are central to many applications from avionics, to to automotive industry; they are the largest installed base of microprocessors
- The size of embedded systems is growing steadily; up to multi-million line systems (e.g. DD(X) battleship control)
- Very low level languages (e.g. assembly) are not viable; low-level ones (eg. C) are barely tolerable; Ada has unfortunately not found the degree of adoption it deserved...
- ... new programming language abstractions for embedded real-time abstractions are sorely needed.
- Reusability is becoming mandatory, even Boeing changing from the old way (recode from scratch) in favor of COTS

- *Caveat: this talk is Java centric; this may not happen real soon, research advances take a long time (~10 years) to percolate down into products;*

Failure Modes

- Memory management is an important source of software failures. *e.g. software intrusions are often violations of memory safety due to a combination of unsafe languages and sloppy programming*
- For RT memory management is critical. Memory is a finite resource, running out is a system failure.
- The time tested way of statically preallocating memory is not tenable. Systems are complex and the effort to ensure deterministic memory requirements conflicts with the black box reuse of COTS
- Dynamic memory allocation (malloc) is unsafe because it assumes discipline use

  References should be used as follows: A (R|W)* F

  With reference aliasing, reasoning about compliance to the protocol is not local, it requires whole program analysis

  Memory leak: A (R|W)*

  Double free: A (R|W)* F+

  Memory overwrite: A(R|W)* F W+

Automated Memory Management

- Garbage collection algorithms along with memory-safe languages prevent all memory errors.
- The cost is an increase in memory requirements (or a decrease in performance, take you pick) with 2x over the optimal not uncommon.
- Memory leaks are hard to diagnose.

- But the main issue is pause times. Stop-the-world collectors require the application to pause for >100ms. In a RT system this can mean missing deadlines.
- Recently, real-time collectors have been able to bound pauses to <20ms, this comes at some cost in throughput
- The state of the art in RT GC is good enough for many RT applications, but not for some hard real-time subsystems

Region-based allocation

- Region-based allocation is an alternative to GC and manual allocation. It follows from the observation that data exhibits liveness locality (allocation and deallocation times are correlated).

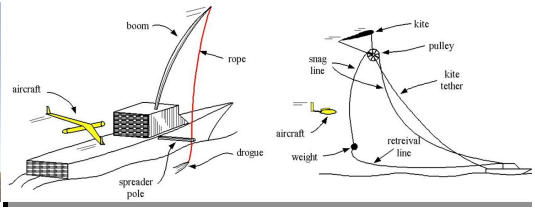
- Idea: create regions which are pools of memory that can be used to store data and bulk deallocated in constant time.

- This simplifies the protocol because instead of having to track individual pointers - we can deal with entire regions.

- A stack of regions allows to deal with different lifetimes.
  Parent regions outlive child regions; data is allocated in the region most closely matching its lifetime.
  Degenerate case: a single region for the whole program.
- In itself, region-based allocation is not safe. Programming language techniques must be devised to enforce safety.

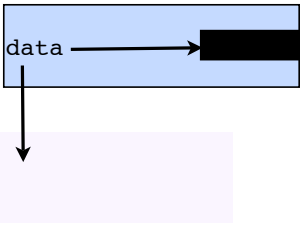
Real-time Java

- The Real-time Specification for Java (RTSJ) is an extension to Java,(JSR-1) which supports real-time processing.
- Many influences from Ada, but also many differences.
- Use GC in non-RT parts of the application and Regions in HRT codes.
- We have implemented a RTSJ virtual machine called Ovm. It has been flight tested in the Scaneagle Boeing UAV.



Nested Lifetimes

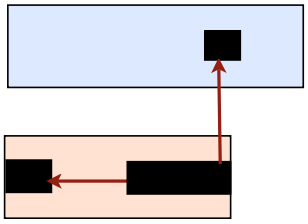
```
byte[] data=new data[12]

    new region
    if ( ) global.data=new byte[]{2,3}

if (   ) data[0]=3
```



- Invariant: data in a nested region is not used after deallocation.
- Dynamic checks:
  - ☑ eager by write barrier, or
  - ☑ just-in-time by time-stamped references (not practical).
- Tradeoff between actual errors and potential errors.
- Improves on unchecked errors by better fault locality. In unsafe languages, errors appear random -- only hint is the ordering bw. deallocation and memory overwrite.

Valid reference patterns

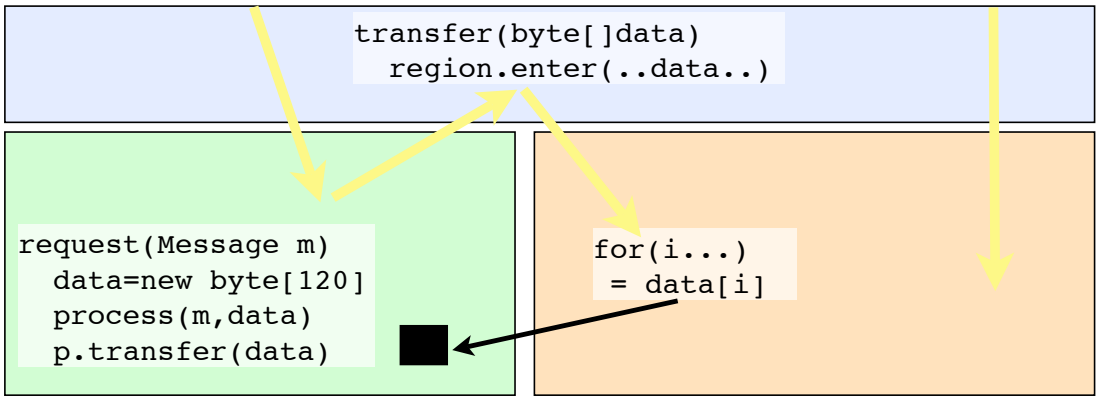
```
Object o1=new Object()
    new region
    Object o2=new Object()
    Object[] up=new Object[2]
    up[0]=o1
    up[1]=o2
```



- All references that respect LIFO ordering of regions are valid
- In concurrent setting Regions form a tree (or cactus stack)
- Write barriers inserted by the compiler on every reference store
- Can be implemented by a Cohen-style subtype test

```
O=o.region; X=x.region
if(O.level<=X.level && O.id==X.display[O.level])
    o.f = x
else ERR
```

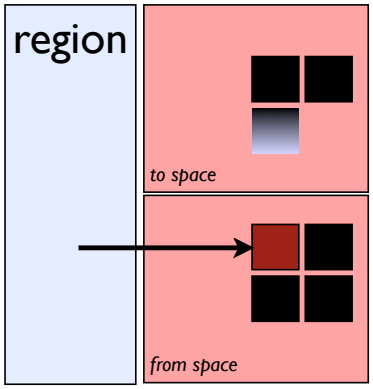

Pipelined computations



- LIFO region discipline is not suited if data requires successive rounds of filtering
- Pipelined computation can be set up by lending a reference to a sibling region for a limited time
- Intuition: a region is pinned by a thread while the thread is active within it. If a thread crosses regions, it is safe to access the sibling scope until the thread returns
- This is safe as long as sibling references are not stored

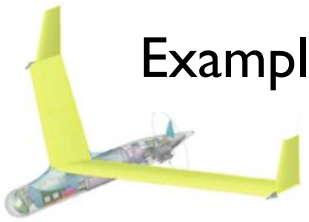
GC-safety

- In a hybrid setting, hard real-time code running in a region must not pause for the GC.
- Two scenarios for experiencing GC pauses
  - real-time code executes code in the heap which triggers an allocation,
  - the real-time thread is released in midst of GC: either wait for GC to complete or risk observing inconsistent data.



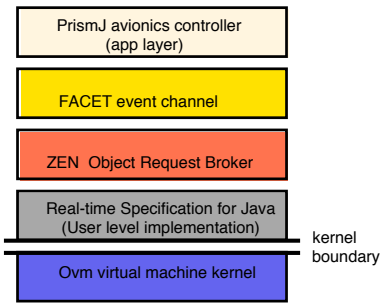
- Dynamic checks can ensure that a hard real-time thread does not load a reference pointing into the heap.
- No known way to prevent transitive priority inversion scenarios between RT, non RT and GC.

Example: PRISMj



Mission critical avionics DRE
Boeing, Purdue, UCI, WUSTL
Route computation, Threat deconfliction algorithms
ScanEagle UAV

System	K LOCs
PRISMJ	109K
FACET EVENT CHANNEL	15K
ZEN CORBA ORB	179K
RTSJ LIBRARIES	60K
CLASSPATH LIBRARIES	500K
OVM VIRTUAL MACHINE	220K



3 rate groups (20, 5, 1Hz)
performance 2x jTime,
≈ Sun product VM

Embedded Planet PowerPC 8260
Core at 300 MHz 256 Mb SDRAM 32 Mb FLASH PC/104 mechanical sized Embedded Linux

A sample fault

The Zen CORBA ORB's Request processing code

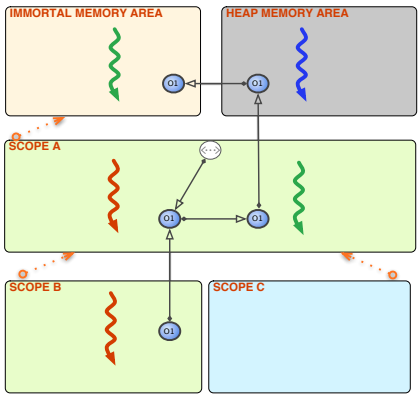
```
class MSGRunnable {
      Servant servant; CDROut
      CDROutputStream reply;
        try { servant = (Servant)RealtimeThread.getCurrentMemoryArea().newInstance(srv.getUnstablePart());
        } catch (Exception e) { throw new Error(e);  }
        ResponseHandler rh = new ResponseHandler(rm);
      if (rm.getOperation().equals("_is_a")) {
        boolean _result = servant._is_a(rm.istream.read_string()); reply = rh.createReply();
        reply.write_boolean(_result);
        } else if (rm.getOperation().equals("_non_existent")) {
            boolean _result = servant._non_existent(); reply = rh.createReply(); reply.write_boolean
            (_result);
        } else CDROutputStream.instance()                                          rh);
      if (rm.getResponseExpected() == 1) {
        reply.updateLength(); final WriteBuffer wb = reply.getBuffer();
        ExecuteInRunnable eir = new ExecuteInRunnable();
        eir.init(new Runnable() {public void run() {
         ((IIOP                                                    oryArea()).getPortal()).send(wb);}},
        rm.getTransport());
        try { orb.outerGate.orbScope.executeInArea(eir); } catch (Exception e) { e.printStackTrace(); }}
        reply.free();
      }
}
public class CDROutputStream extends OutputStream {
      WriteBuffer buffer;
      private static Queue cdrCache = new Queue();
      public static CDROutputStream instance() {
        try { if (cdrCache.isEmpty()) return new    CDROutputStream();
            else return (CDROutputStream) cdrCache.dequeue();
        } catch (Exception e) { throw new Error(e); }
      }
```

```
class CDROutputStream extends OutputStream
    static Queue cdrCache = new Queue()
    static CDROutputStream instance()
        if (cdrCache.isEmpty()) return new CDROutputStream()
        else                    return  cdrCache.dequeue()
```


Summary

- Memory-safe and GC-safe is crucial for assurance and predictability
- The Real-time Specification for Java (RTSJ) provides a memory-safe and (mostly) GC-safe hybrid memory management model
- The RTSJ distinguishes between real-time and non-real-time threads, RT threads can run in regions and non-RT threads run in the heap
- Safety is achieved through dynamic checks

  they entail performance overheads, errors are reported eagerly and software can experience runtime exceptions
- Evaluation: RTSJ code is brittle. Memory violation occur due to trivial programming errors, e.g. reuse of code that was not intended to run in scopes, etc. These errors may exhibit themselves once the software is deployed and thus be hard to find.
- **Goal: catch errors at compile time**



Region types

- Tofte and Talpin invented the notion of region types circa 1992.
  (Their 1994 POPL paper on the topic was voted most influential paper for that year.)
- *Region-types guarantee static-memory safety of a region-based programs.*
- This is achieved by extending a type system with information describing the allocation context of data.
- The paper targets a single-threaded, functional language (ML).
- The T&T type system uses both polymorphism and effects for expressivity.
- Their implementation, the MLKit, has been used successfully to write large systems without garbage collection.
- Region types also appear in Cyclone (a dialect of C)

Region types

- Regions in T&T are lexically scoped, and single threaded,
  An example in the Java-like syntax of Christiansen's MSc thesis:

```
letregion r in {
  Pt[r] point = new Pt[r](1,2);
  ...
}
class Pt[ρ] { int y,x;
  ρ Pt[ρ](int x, int y) {this.x=x;this.y=y;}
  void move[ρ2]( Pt[ρ2] p) {x+=p.x;y+=p.y;}
}
```

- Types are annotated with the variables that denote regions, methods have effect annotations. Polymorphism allows for one type to be used in different allocation contexts
- We need to extend T&T region types with multi-threading, GC-safety, and O-O

Ownership Type Systems

- Ownership types were developed independently from region-types with the goal of imposing structural constraints on graphs of references in object-oriented systems [Noble,Potter,Vitek ECOOP97], [Clarke et.al OOPSLA98]
- While not originally intended for memory management ownership types are quite close to what we need
- Boyapati et.al. have proposed an ownership type system for RT Java
- But it is complex and is not backwards compatible with Java
- Also, it fails to capture idioms used in most real RTSJ programs

```
class Producer<CommunicationRegion r> {
  void run(RHandle<r> h) {
    while(true) {
      (RHandle<FrameBuffer buffer> hbuffer = h.buffer)
        Frame<buffer> frame = new Frame<buffer>;
        grab_image(frame);
        hbuffer.f = frame;
      }
      ... // wake up the consumer
      ... // wait for the consumer
}}}

class Consumer<CommunicationRegion r> {
  void run(RHandle<r> h) {
    while(true) {
      ... // wait for the producer
      (RHandle<FrameBuffer buffer> hbuffer = h.buffer)
        Frame<buffer> frame = hbuffer.f;
        hbuffer.f = null;
        process_image(frame);
      }
      ... // wake up the producer
}}}
```

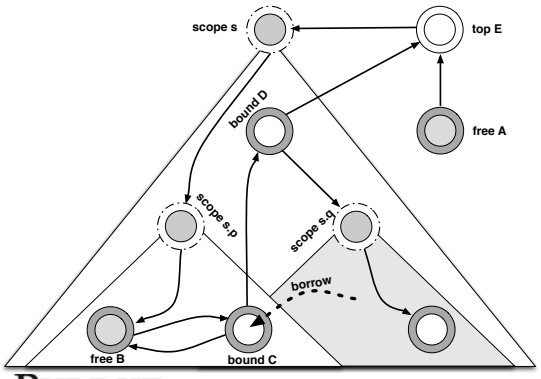

Subtyping

- One key issue with object-oriented languages is subtyping
- The problem is
  we use types to track allocation contexts, but subtyping lets you "forget" some type information

```
Object obj  = new Pt[r]();   // solution either disallow or make Object[r]

Object[r1] o= new Foo[r1,r2](); // ?? have two regions param for Object

Pt[r] p = (Pt[r]) obj;    // ? is there a way to do a dynamic cast?

p.doStuff()  // if doStuff is an inherited method, p is silently widened to parent type
```

Scopes

- Scopes are a new abstraction for programming highly-responsive applications in a statically memory-safe and gc-safe fashion
- The design goals were to be backwards compatible with Java, simple to use, expressive and efficient
- Scopes have been integrated in a RTSJ virtual machine
- The type system has been proven sound



(1) *Scopes* (`s`, `s.p`, `s.q`) may refer to objects within their allocation context and to bound classes of ancestor scopes.

(2) *Bound classes* (e.g. `C`, `D`) may refer to any co-located object, and to bound classes allocated in a parent scope.

(3) *Free class instances allocated in a scope* (e.g. `B`) may refer to any co-located object.

(4) *Free class instances allocated in the heap* (e.g. `A`) may refer to permanent classes.

(5) *Permanent classes* (e.g. E) may refer to other permanent classes and top-level scopes.

(6) *Borrowed reference* may refer to any bound class.

Programming with Scopes

- Scopes are like objects, they have fields and methods.
- Bound classes are defined within a scope and are always allocated in that scope at runtime
- Scopes are first-class values, when a thread invokes a scope method, the allocation context changes to that of the scope
- Invoking any method of a bound type also triggers an allocation context switch

```
r = new processor;
while (b = nextRequest()) {
   r.getMessage(b,m);
   m.dispatch();
   release r;  }
```

```
1   scope processor {
2      class Unpacker { ...
3          void parse(Bytes b) {...}
4          void write(Message m) {...}
5      }
6      void getMessage(Bytes b, Message m) {
7          Unpacker p = new Unpacker();
8          p.parse(b);
9          p.write(m);
10  }    }
```

A Scope Pool

- Scopes can be nested. The enclosing scope provides the allocation context for its subscopes
- Deallocating the contents of a scope entail deallocating all child scopes
- Scopes can be shared by multiple threads and multiple threads can enter a scope simply by calling its methods
- Scopes are only deallocated when all threads have finished

```
1  scope pool {
2    area[] a;  int cnt;
3    void create(int i) {
4      a=new area[i];
5      for(int j=0;j<i;j++) a[j]=new area;
6    }
7    void run(Message m, Bytes data){
8      int i = cnt++ % a.length;
9      a[i].run(m, data);
10     release a[i];
11   }
12   scope area {
13     void run(Message m, Bytes data){
14       ... // service request
15 } } }
```


Crossing Scopes

- Bound classes allocated in parent scopes are visible from code running in a subscope
- Invoking a method of one of those bound classes temporarily changes the allocation context to that of the parent, when the method returns the allocation context is restored

```
1  scope a {
2    class Box {
3      c s2;
4      void enter() { s2.mc(); }
5    }
6    scope b { void mb(Box box) { box.enter(); } }
7    scope c { void mc(){...} }
8    void ma() {
9      b s1 = new b;  c s2 = new c;
10     Box box = new Box(); box.s2 = s2;
11     s1.mb(box);
12 }  }
```

Implicit Polymorphism

- Classes that are not lexically enclosed within a scope are called free classes
- These classes are implicitly scope polymorphic
- Any free class can be allocated within a scope, but it is only visible from that scope (and not from its subscopes); it can not escape the scope
- This allows the reuse of most libraries classes (with small restrictions)

```
1  class List {   List next;   Object value; }
2
3  scope user {
4      List l;
5      void create(int i) {
6        List t;
7        for(int j=0;j<i;j++)
8        {  t = new List(); t.next =l; l = t; }
9  }  }
```


Quasi Linear Types

- To support pipelined computation, a quasi linear type annotation "borrow" is added
- borrow can annotate method arguments and local variables
- a borrowed reference can not be stored in a field
- anything retrieved from a borrowed reference is borrowed
- it is not allowed to store a reference into a field of a borrowed object

```
1    scope r {
2      class Bridge {  q qscp;
3          void run(p pscp, q  qscp) {
4               this.qscp=qscp;  pscp.enter(this)
5          }
6          void handoff(borrow byte[] data) {
7               qscp.enter( data);
8      }   }
9      scope p {
10         void enter(Bridge b) {
11            byte[] data=new byte[20];
12            ... // some computation
13            b.handoff( data);
14     }   }
15     scope q {
16         void enter(borrow byte[] data) {
17              ... //use cross-scope reference
18   }  }   }
```

Type safe casts

- Free types can be cast to any parent type (with the caveat that free types cannot extend a bound type)
- Bound types can be cast to
  - any bound type defined in the same scope
  - any free type
- Downcasts from any type are allowed.

```
1    scope a {
2       Object[] obj;
3       class B { }
4       ...
5            obj[1] = new B();
6            B  b = (B) obj[1];
```


Conclusions

- High-level languages for Real-time systems need support for memory-safe, and gc-safe, memory management techniques
- Region-based allocation provides linear time allocation and bulk deallocation
- Type systems can prevent all memory violations at compile time
- The scopes abstraction introduced in this talk provides static safety without requiring drastic changes to Java,
- Scopes are simple to use and allow the reuse of existing code