Practical Off-heap Memory for Scala

Denys Shabalin

This talk is *not* going to feature:

- new language features
- formal systems

Overview

1. Introduction
2. Encoding regions
3. Memory management runtime
4. 0.1 roadmap for ScalaDays SF
5. Conclusion

Introduction

"In computer science, region-based memory management is a type of memory management in which each object is assigned to a region... that can be *efficiently deallocated all at once*."

Wiki

Widely used in production:

- Apache HTTP Server
- PostgreSQL
- Google Chrome

Apache memory pools in C:

```
struct Point { int x; int y; };

void main() {
    apr_pool_t *mp;
    apr_pool_create(&mp, NULL);        // create a fresh pool
    for (i = 0; i < 100; ++i) {        // allocate in region
        struct Point *p = apr_palloc(pool, sizeof(struct Point))
        p->x = i;
        p->y = i * 2;
        hello(p);
    }
    apr_pool_destroy(mp);              // free everything at once
}

void hello(struct Point *p) {          // points are friendly
    printf("Hi, I'm a point (%d, %d)", point->x, point->y)
}
```

Dynamic regions in Cyclone:

```
struct Point { int x; int y; };

void main() {
    region<`r> h;
    for (i = 0; i < 100; ++i) {
        Point *@region(`r) p = rnew(h) Point(i, i * 2);
        hello(p);
    }
}

void hello(Point *@region(`r) p) {
    printf("Hi, I'm a point (%d, %d)", point->x, point->y)
}
```

Encoding regions

Main challenges

1. Null dereference safety
2. Dangling pointers safety
3. Boilerplate
4. Performance

Three experiments

1. Unchecked
2. Statically checked
3. Dynamically checked

Unchecked

A simple API to allocate in regions:

```
@offheap class Point(x: Int, y: Int)

object Main extends App {
  Region { implicit r =>
    for (i <- 0 to 99) {
      val p = Point(i, i*2)
      hello(p)
    }
  }
  def hello(p: Point) =
    println(s"Hi, I'm a Point (${p.x}, ${p.y})")
}
```

Desugaring:

```
class Point[R <: Region[_]] private (
  private val addr: runtime.Address
) extends AnyVal {
  def x: Int =
    if (runtime.isNull(addr)) throw ...
    else runtime.read[Int](addr)
  def y: Int = ...
}

object Point {
  def apply(x: Int, y: Int)(implicit r: Region): Point = {
    val addr = runtime.regionAlloc(r, runtime.sizeof[Point])
    runtime.write[Int](addr, x)
    runtime.write[Int](addr, y, offset = runtime.sizeof[Int])
    new Point(addr)
  }
}
```

Statically checked

Model region as an implicit capability to:

1. Allocate objects
2. Access fields

Encoding region checking through unique region types:

```
@region(R) class Point(x: Int, y: Int)

object Main extends App {
  implicit val r = Region.open       // r: Region[42]
  for (i <- 0 to 99) {
    val p = Point(i, i*2)            // p: Point[Region[42]]
    hello(p)                         // hello[Region[42]](p)(r)
  }
  @region(R) def hello(p: Point[R]) =
    println(s"Hi, I'm a Point (${p.x}, ${p.y})")
}
```

Regions are first-class types with compile-time unique ids:

```
trait Region[Id <: Int]
object Region {
  def open[Id <: Int](implicit fid: FreshId[Id]): Region[Id] =
    runtime.regionOpen().asInstanceOf[Region[Id]]
}
```

Can't use syntax used in the previous API:

```
Region { implicit r => ... }
```

Due to inability to infer the parameter type for f:

```
def apply[Id <: Int, T](f: Region[Id] => T)(implicit fid: FreshId[Id])
```

Hello method desugaring:

```
def hello[R <: Region[_]](p: Point[R])(implicit r: R) =
  println(s"Hi, I'm a Point (${p.x(r)}, ${p.y(r)})")
```

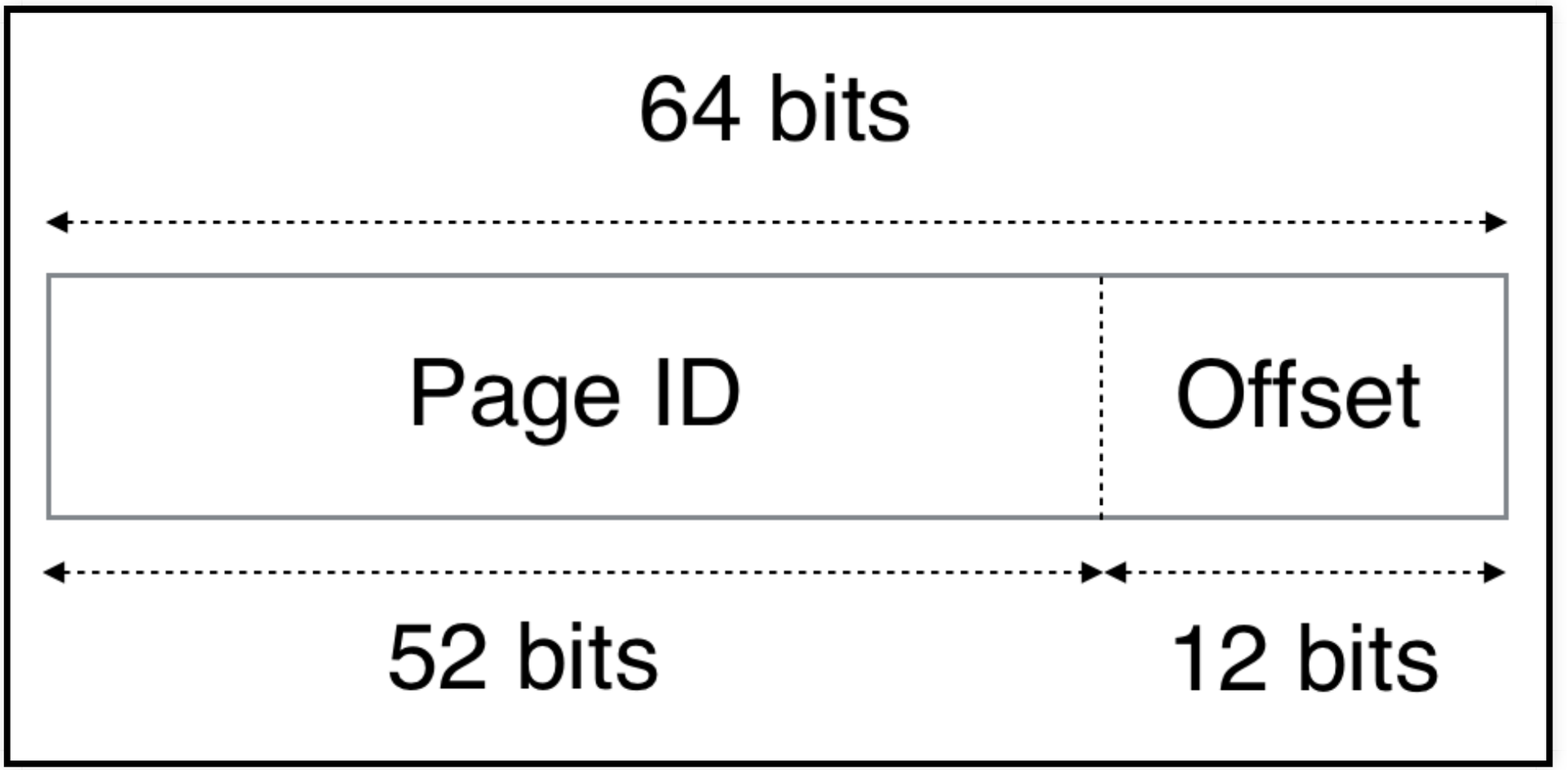
Desugaring:

```
class Point[R <: Region[_]] private (
  private val addr: runtime.Address
) extends AnyVal {
  def x(implicit r: R): Int =
    if (runtime.isNull(addr)) throw ...
    else runtime.read[Int](addr)
  def y(implicit r: R): Int = ...
}

object Point {
  def apply[R <: Region[_]](x: Int, y: Int)(implicit r: R): Point[R] = {
    val addr = runtime.regionAlloc(r, runtime.sizeof[Point])
    runtime.write[Int](addr, x)
    runtime.write[Int](addr, y, offset = runtime.sizeof[Int])
    new Point[R](addr)
  }
}
```

Dynamically checked

API is the same as in unchecked one. The major difference lies in representation of pointers:



Memory management runtime

sun.misc.unsafe

```
package sun.misc

class Unsafe {
    public native long allocateMemory(long bytes)
    public native long reallocateMemory(long address, long bytes)
    public native void freeMemory(long address)
    public native void putByte(long address, byte x)
    public native void putShort(long address, short x)
    ...
    public native byte getByte(long address)
    public native short getShort(long address)
    ...
}
```

Runtime API

```
trait runtime {
  type Address
  type Size
  def isNull(addr: Address): Boolean
  def read[T](addr: Address, offset: Size): T
  def write[T](addr: Address, value: T, offset: Size): Unit
  def sizeof[T]: Size
  def regionOpen(): Region
  def regionClose(r: Region): Unit
  def regionAlloc(r: Region, size: Size): Address
}
```

Memory recycling runtime

Performance

0.1 Roadmap

(for ScalaDays SF)

Offheap traits

Implemented as tagged unions.

```
@offheap trait Tree
@offheap class Add(left: Tree, right: Tree) extends Tree
@offheap class Const(value: Int) extends Tree
```

Offheap arrays

With familiar collection-like API.

```
import offheap._

Region { implicit r =>
  val arr = Array(1, 2, 3)
  arr.foreach(println)
}
```

Thread safety

Of the region allocator.

Summary

Challenge	**Unchecked**	**Static**	**Dynamic**
Null pointers	Dynamic	Dynamic	Dynamic
Dangling pointers	Unchecked	Static	Dynamic
Boilerplate	Least	Most	Least
Performance	1.0x	1.2x	4.0x

Conclusions

1. There is no silver bullet
2. Scala is (almost) expressive enough to encode statically safe region-based memory as a library
3. Managed offheap memory can have competitive performance on JVM

Questions?