Compiler Optimized Remote Method Invocation

Ronald Veldema

University of Erlangen-Nuremberg

Computer Science 2

Programming Languages

Remote Method Invocation (RMI)

- Object-Oriented Remote Procedure Call
  - Polymorphism
  - Inheritance
- Heterogeneity
- Portable

Problem Statement

- Java RMI for parallel programming too slow ?
  - Millisecond level
  - Many improvements already made
    - 30-200 microsecond level
      - PPoPP’99 Manta-RMI
      - JavaGrande 99 KaRMI
      - etc

JavaParty

- Layer on top of RMI
- JavaParty programs compile to RMI programs
- Advantages
  - No remote interfaces
  - No registry
  - Network exceptions masked by JavaParty
  - Object Distribution is handled by JavaParty RTS
  - Simpler programming model = code easier to analyze by a compiler

RMI mechanism (1)

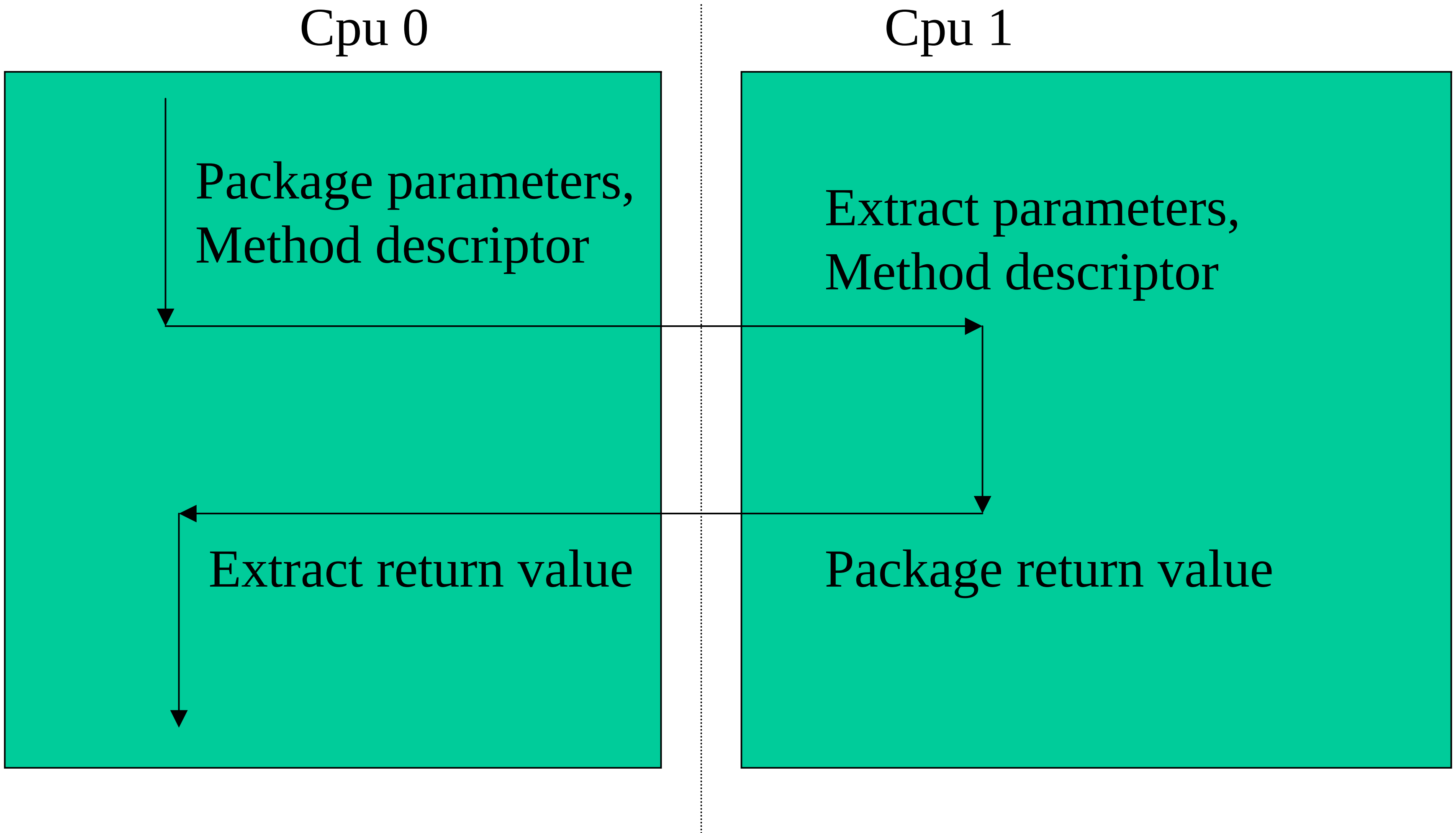
CPU 0

```
void zoo(Hello hi) {
    hi.go();
}
```

CPU 1

```
remote class Hello {
    void foo() {
        System.out.println("Hello !");
    }
}
```

RMI mechanism (2)



Package Parameters: Earlier Work (1)

- Traverse the reachable object graph and serialize it to a byte array

```
class Data {
    int temp;
    String str;
}

Data param1 = new Data();
object.foo(param1);
```

```
Serialize_Data(Stream s, Data this) {
  write_class_descriptor(s, Data.class);
  Serialize_int(s, this.temp);
  Serialize_Object(s, this.str);
}
```

Extract Parameters: Earlier Work (2)

- Read byte stream and reconstruct parameters by creating *new* objects

```
class Data {
        int temp;
        String str;
}

Data param1 = new Data();
object.foo(param1);
```

```
Object deserialize(Stream s) {
  ObjectDescriptor d =
                 read_object_descriptor();
   return d.deserializer(s);
}

// for each class generate:
Object Data.deserializer(Stream s) {
        Data this = new Data();
        this.temp = read_int(s);
        this.str = deserialize(s);
        return this;
}
```

Problems With Naïve Approaches (1)

- Cannot handle cycles
  - The runtime system cannot look into the future
  - A compiler can look into the future
- For each deserialized object, a new object is created
  - The runtime system does not know if after an RMI the old object is garbage

Problems With Naïve Approaches (2)

- Each reference causes an indirect call to locate the (de)serializer for that type
  - If the compiler knows for each reference what it points to, this can be avoided
- Search needs to be performed to locate the meta object for the deserialized object
  - If the compiler knows for each reference what it points to, this can be avoided

Problems With Naïve Approaches (3)

- Cycle table lookups
  - Even for objects where programmer knows no cycles exist !

```
Serialize_Data(Stream s, Data this) {
  if already_serialized(this) {
      write_stream_backreference(s);
  } else {
      write_class_descriptor(s, Data.class);
      Serialize_int(s, this.temp);
      Serialize_Object(s, this.str);
  }
}
```

Intermezzo: Heap Analysis

- Statically analyze which types of objects allocated from where in a program a reference can point to
- Algorithm
  - Convert program to SSA form
  - Assign to each ‘new’ a unique number
  - Propagate numbers throughout the program
  - Continue until a stable state is reached

Heap Approximation (1)

```
class B {}

class A {
  B b;

  A() {
     B b = new B();
     this.b = b;
   }

   void foo() {
        // what do we know of “this.b” ?
   }

   public static void main(String args[]) {
         A a = new A();
         a.foo();
   }
}
```

Heap Approximation (2)

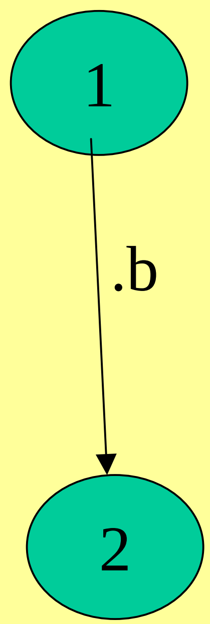
```
class B {}

class A {
  B b;

  A() {
    B b = new B();                    // b = {1}            1
    this.b = b;
  }

  void foo() {
        // what do we know of "this.b" ?
  }

  public static void main(String args[]) {
        A a = new A();                // a = {2}            2
        a.foo();
  }
}
```

Heap Approximation (3)

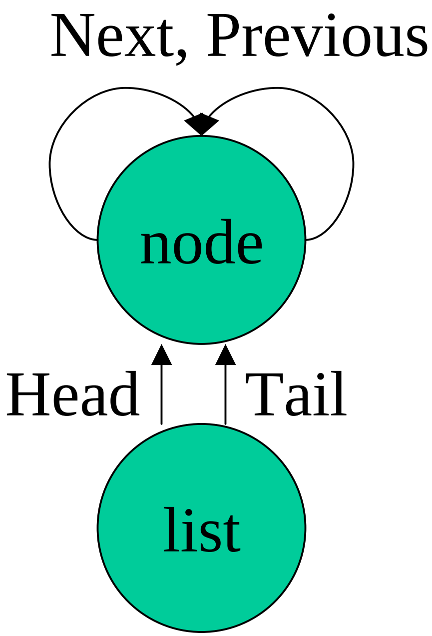
```
class B {}

class A {
  B b;

  A() {                                   // this = {2}
     B b = new B();                       // b = {1}           1
     this.b = b;
   }

   void foo() {                           // this = {2}
        // what do we know of "this.b" ?
   }

   public static void main(String args[]) {
         A a = new A();                   // a = {2}           2
         a.foo();
   }
}
```

Heap Approximation (4)

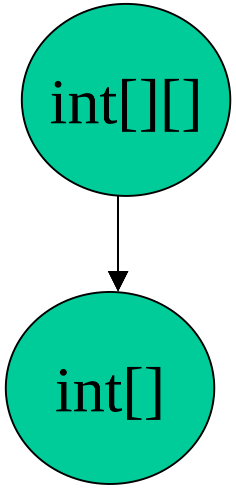
```
class B {}

class A {
  B b;

  A() {                                              // this = {2}
     B b = new B();                                  // b = {1}
     this.b = b;      //{2}.b = {1}
  }

  void foo() {                                       // this = {2}
       // what do we know of "this.b" ?
  }

  public static void main(String args[]) {
        A a = new A();                               // a = {2}
        a.foo();
  }
}
```



Heap Approximation (5)



Linked List



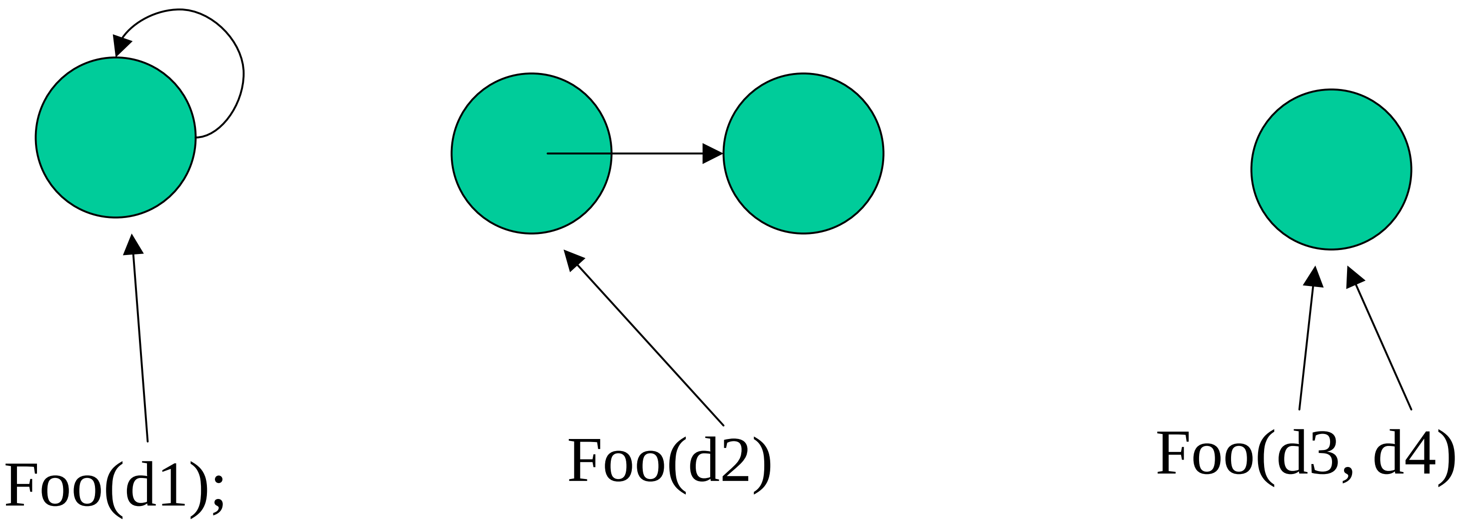
2 dimensional array

Heap Approximation Results (1)

- Whole program analysis:
  - For each variable we know its possible origins
    - We know that variable V was the result of *new* from locations X, Y, Z in source code
    - For each reference we know where it may point to
    - For each object field we know where it may point to

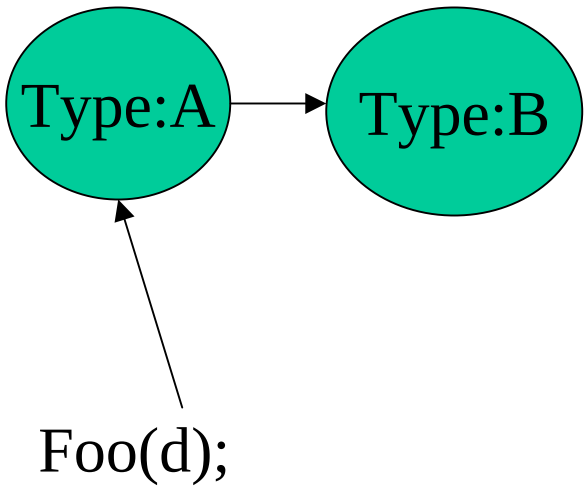
Heap Approximation Results (2)

- Given a heap graph, we can see if the graph rooted in the RMI call's parameter contains cycles



Heap Approximation Results (3)

- If we know the specific shape/types of nodes in the heap graph: generate specialized code *for that callsite*



ISO calling "foo" call:

```
call_foo_remotely(A d) {
        write_method_descriptor("foo");
        serialize_fields_of_A(d);
        serialize_fields_of_B(d.b);
}
```

RMI Escape Analysis (1)

- If after an RMI the entire parameter tree is garbage, reuse the tree
  - Avoid garbage collection cost
  - Avoid costs of allocating new parameter objects
- Note: normal escape analysis is defined on single objects

RMI Escape Analysis (2)

No !

```
// modified
static Data temp;

void foo(Data d) {
  temp = d;
}

void zoo(Data d) {
  temp = d.x.y.z.field;
}
```

Yes !

```
// read only:
static int temp;

void foo(Data d) {
  print (d.field);
}

int zoo(Data d) {
  return d.field * temp;
}
```

Results

- 19 % performance gain for simple programs
- Most gain by generating specialized code for each callsite
  - Removes type lookups from code
  - Cycle elimination for duplicate reference tests
- Escape analysis doesn’t help much
  - Garbage collector already efficient