Miniboxing

Load-time Specialization on the JVM

DSLAB, EPFL, 19th of March 2103

Vlad Ureche

vlad.ureche@epfl.ch

Generic Code

```
def identity[T](t: T): T = t
```

- Reuse, type safety
- Uniform interface
- Low level code: **non-uniform**
  - Registers, stack
  - Sizes and semantics
  - Current solution on the JVM → **erasure**

C++ and .NET do it right, why can't the JVM?

```
identity(3) // = unbox_int(identity(box_int(3)))
```

Optimal Low Level Code

- C++
  - **Compile-time on-demand** template expansion
  - Undermined separate compilation
- .NET CLR
  - **Load-time on-demand** bytecode specialization
  - Type system burnt in the virtual machine
- Specialization in Scala
  - **Compile-time static** bytecode specialization

What's the difference?

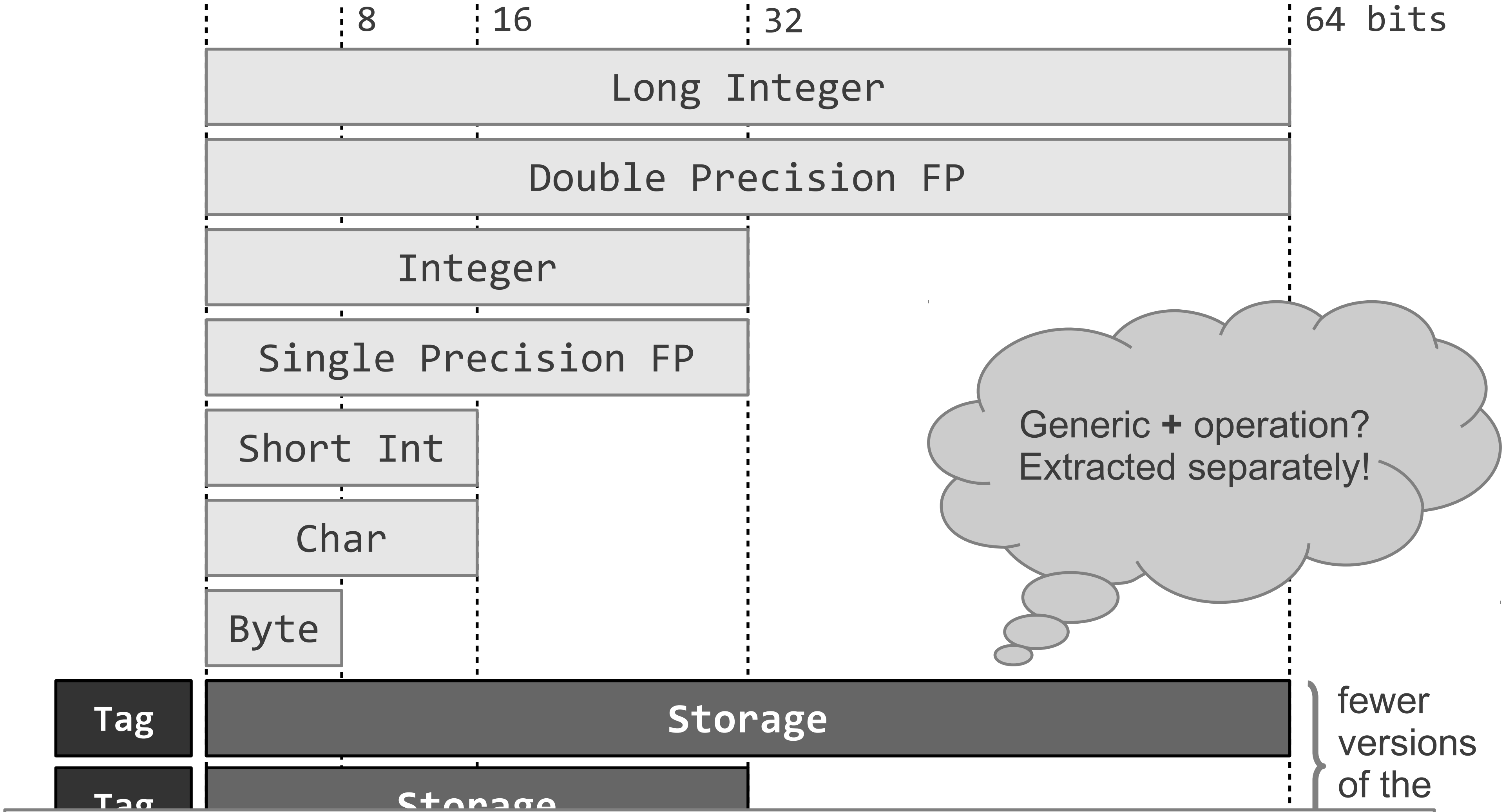
Specialization

```
def identity[T](t: T): T = t
```

```
def identity(t: Any): Any = t
def identity_D(t: Double): Double = t
def identity_J(t: Long): Long = t
def identity_I(t: Int): Int = t
def identity_F(t: Float): Float = t
def identity_S(t: Short): Short = t
def identity_C(t: Char): Char = t
def identity_B(t: Byte): Byte = t
def identity_Z(t: Bool): Bool = t
def identity_U(t: Unit): Unit = t
```

```
def apply[T, R](arg: T): R // 10^2 versions
```

Miniboxing



8 16 32 64 bits

Long Integer

Double Precision FP

Integer

Single Precision FP

Short Int

Char

Byte

Tag Storage

Tag Storage

Generic + operation?
Extracted separately!

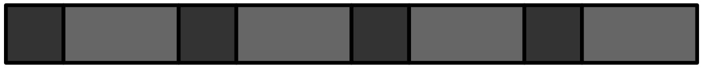
fewer
versions
of the

Looks like tagged union...

But It's Not

Tagged Union

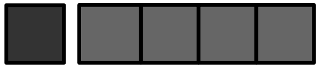
- a single version of the code, uniform
- dynamic
  - attaches type tags to each value



- dispatch overhead
- no optimizations across dispatches

Miniboxing

- multiple versions of the code, as specialization
- static
  - tags attached to classes and methods



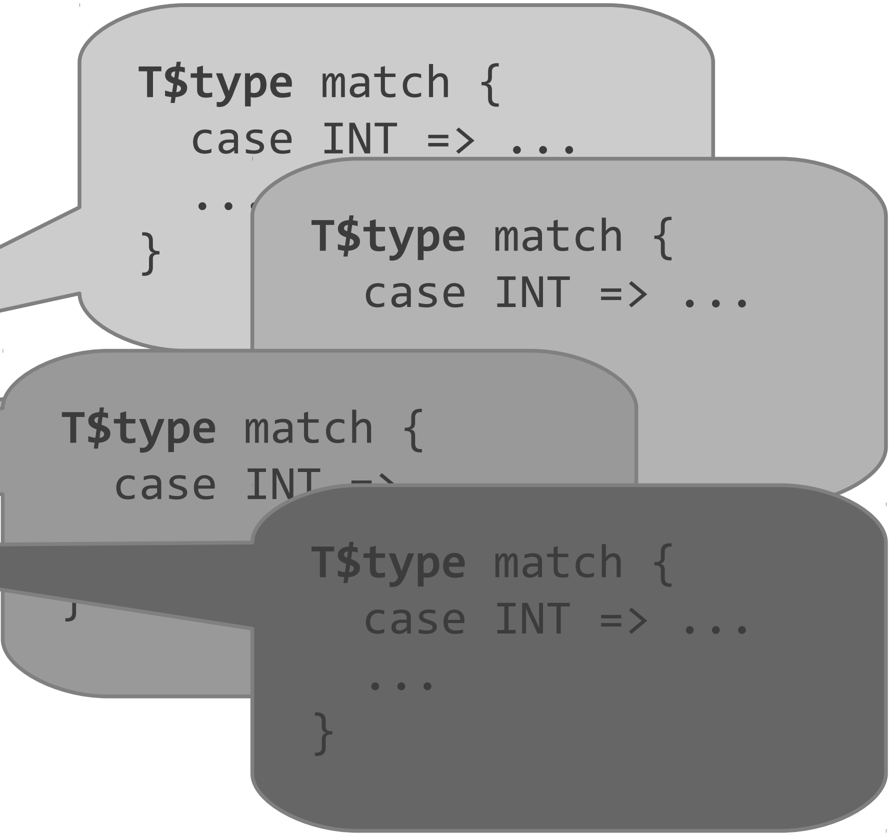
- eliminates dispatch overhead
- optimizations across

What's the magic?

Dispatch Overhead

- `ArrayBuffer.reverse()`

```
def reverse(): Unit {
  var index = 0
  while (index * 2 < length) {
    val opposite = length-index-1
    val tmp1: T = array(index)
    val tmp2: T = array(opposite)
    array(index) = tmp2
    array(opposite) = tmp1
    index += 1
  }
}
```



```
T$type match {
  case INT => ...
  ...
}
```

```
T$type match {
  case INT => ...
```

```
T$type match {
  case INT =>
}
```

```
T$type match {
  case INT => ...
  ...
}
```

Lift the `T$type match { ... }` above the loop?

Object-Oriented Dispatching

- Dispatch object
  - Encodes array interactions

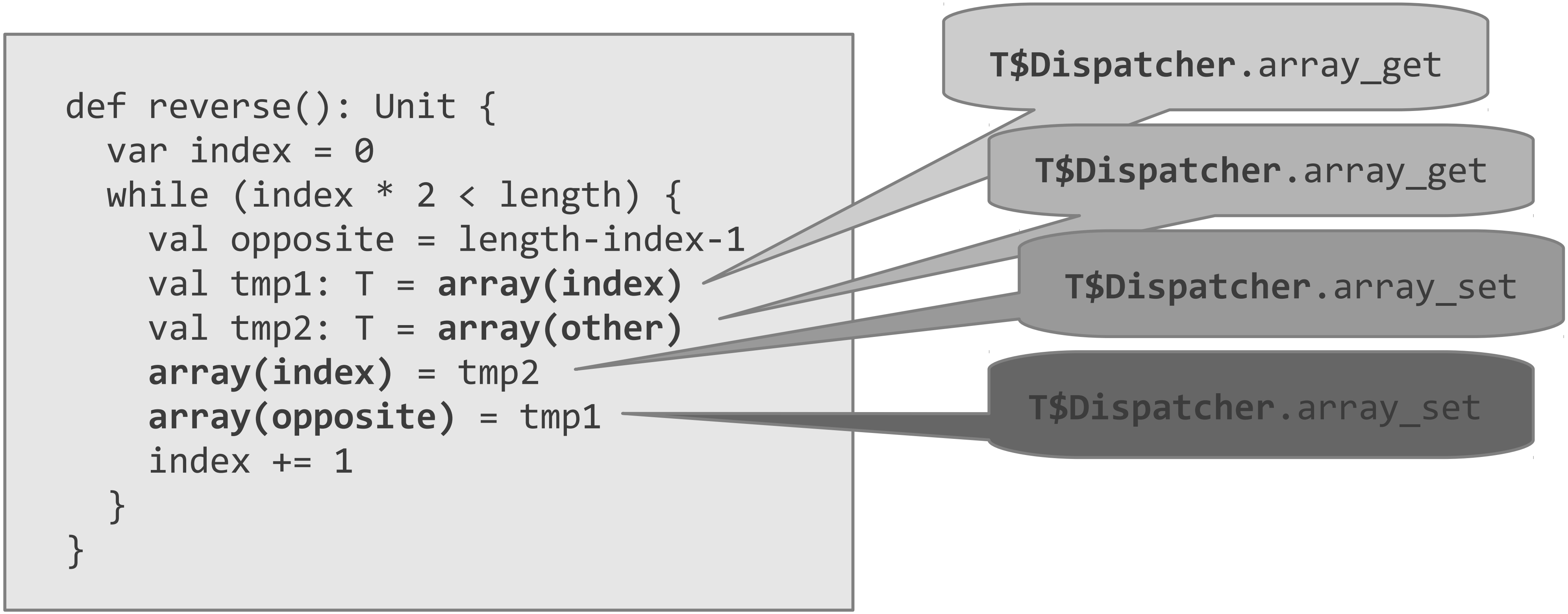
```
class Dispatcher[T] {
  def array_get(arr, idx): Storage
  def array_set(arr, idx, value: Storage): Unit
}
```

```
object IntDispatcher extends Dispatcher[Int] {
  ...
}
```

Passing a dispatcher = switch already done

Object-oriented Dispatching

- ArrayBuffer.reverse()



```
def reverse(): Unit {
  var index = 0
  while (index * 2 < length) {
    val opposite = length-index-1
    val tmp1: T = array(index)
    val tmp2: T = array(other)
    array(index) = tmp2
    array(opposite) = tmp1
    index += 1
  }
}
```

T$Dispatcher.array_get

T$Dispatcher.array_get

T$Dispatcher.array_set

T$Dispatcher.array_set

Call site becomes megamorphic

Load-time Specialization

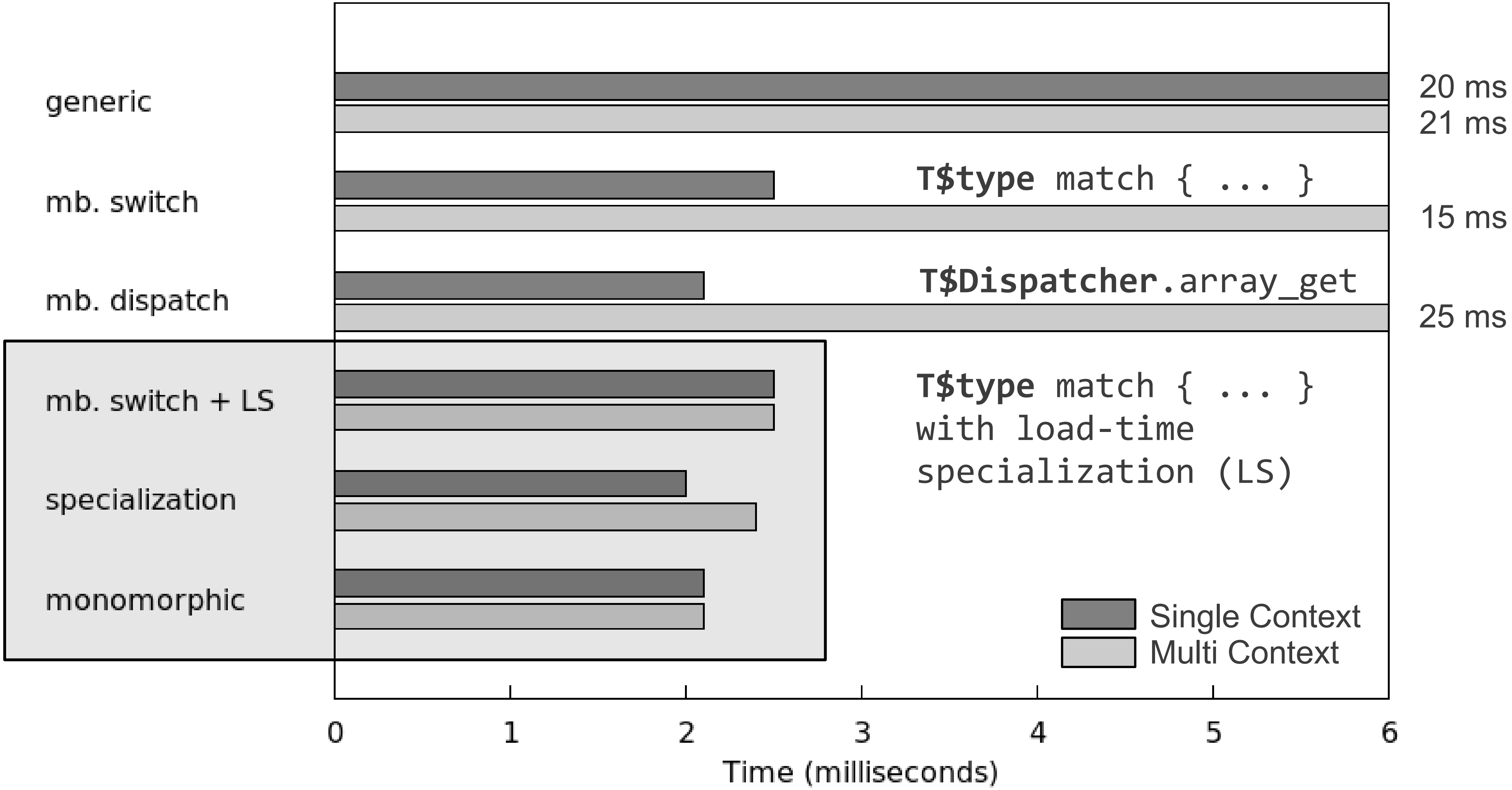
- Switch-based dispatching

```
T$type match {
  case INT => ...
  ...
}
```

- At load-time
  - set **T$type** statically
  - perform constant propagation
  - perform dead code elimination
- We get the specialized class
  - no dispatch overhead
  - optimizations can kick in across dispatches

Does it work in practice?

Evaluation – `reverse` on 3M elements



T$type match { ... }

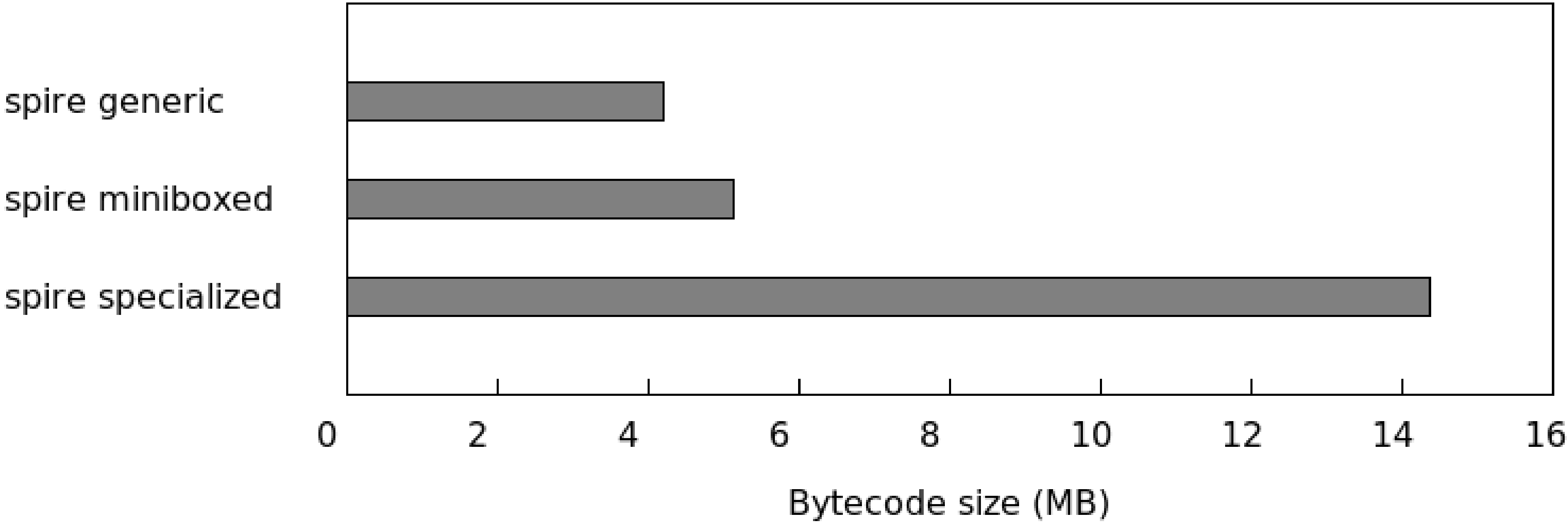
T$Dispatcher.array_get

T$type match { ... } with load-time specialization (LS)

Similar results on other benchmarks

Evaluation – Code Size

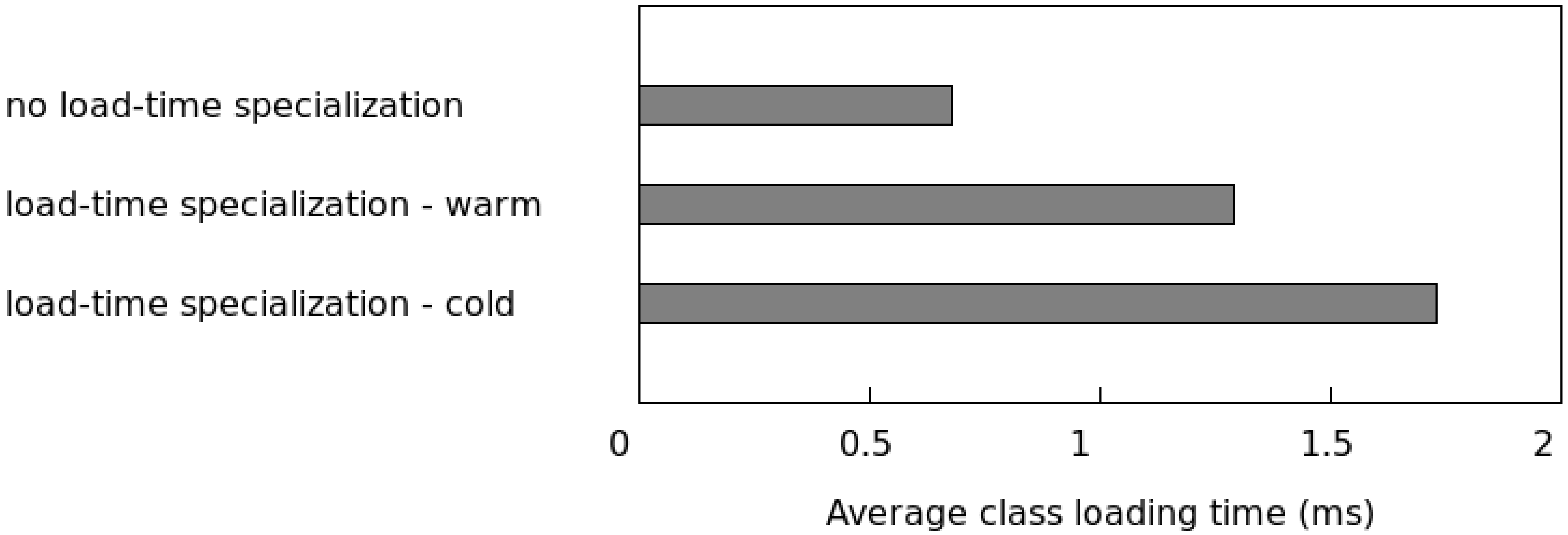
- Spire – numeric abstractions library (12KLOC)



2.8x bytecode reduction (4.7x for `Vector` in std. lib)

Evaluation – LS Overhead

- `scala.collection.immutable.Vector`



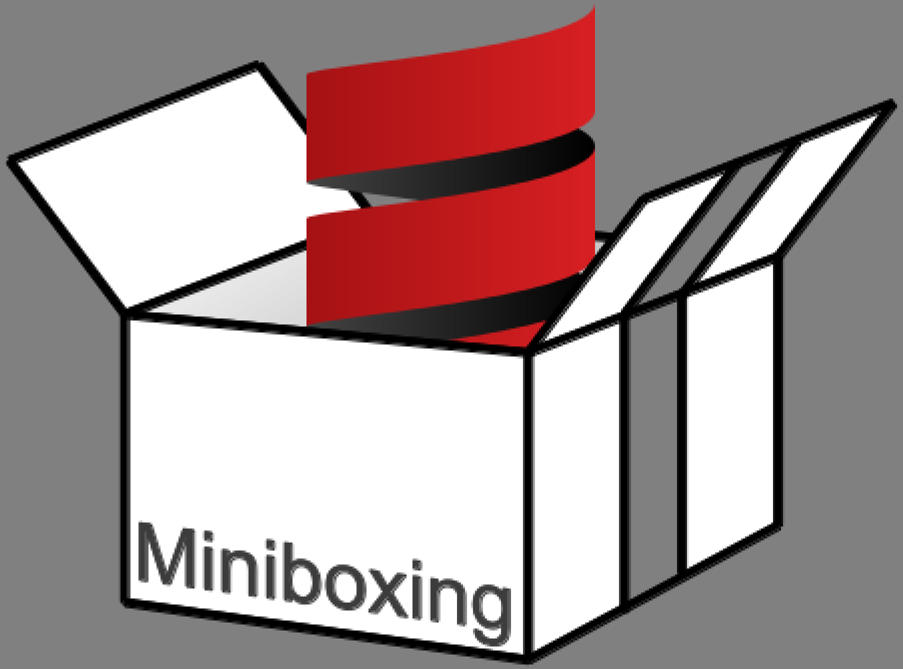
Load-time specialization overhead – at most 2.5x

Conclusions

- Miniboxing – same performance, less bytecode
  - Encoding
  - Transformation
  - Runtime Support
  - Load-time Specialization
- Virtual machines and performance
  - **High-level self-modifying code**

Thank you!

github.com/miniboxing
vlad.ureche@epfl.ch



WE ARE HERE

Backupslides

Generic Code

```
def identity[T](t: T): T = t
identity(3) // = unbox_int(identity(box_int(3)))
```

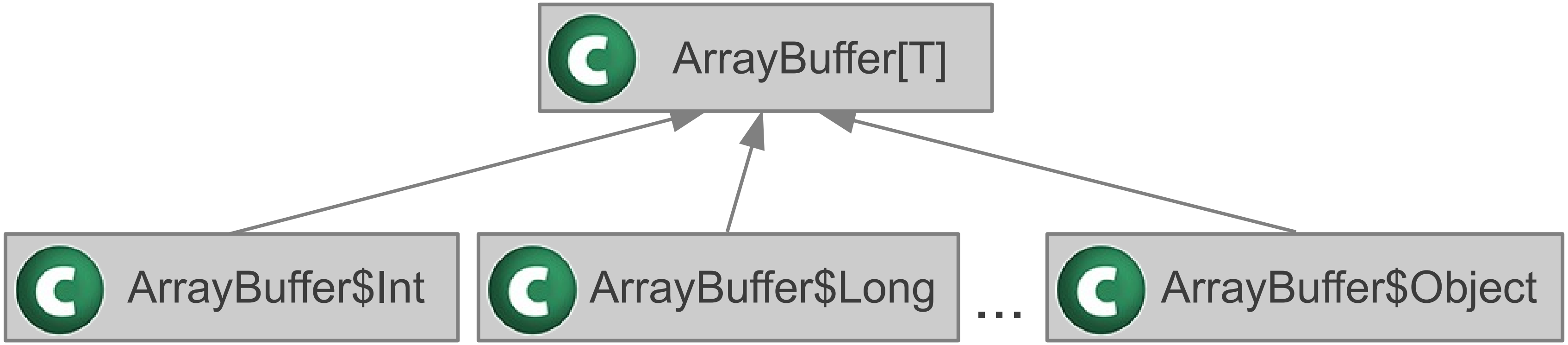
- common representation – boxing?
  - **allocation** and **garbage collection**
  - **wasteful** use of heap memory
  - affects **cache locality**

Specialization

- duplicate and adapt methods

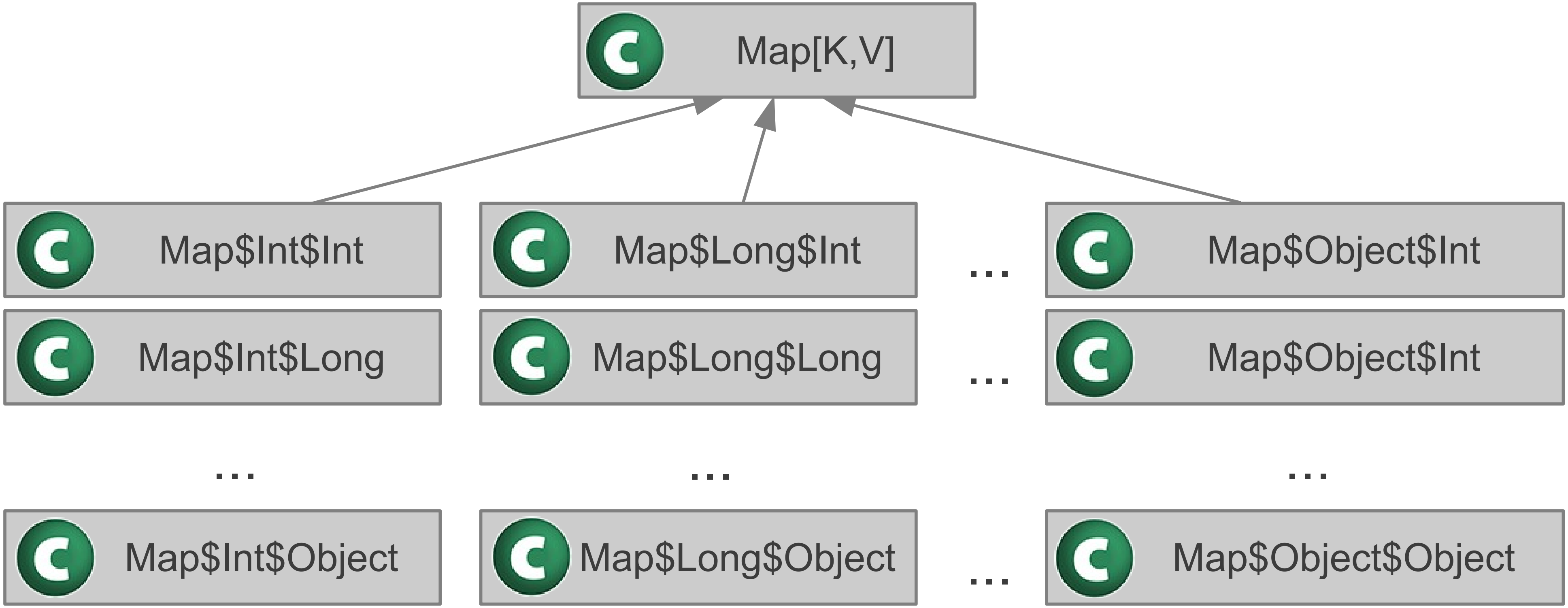
```
def identity[T](t: T): T = t
def identity$Int(t: Int): Int = t
...
```

- duplicate and adapt classes



Specialization - Scaling

- static expansion, in the library



Specialization - Compile-time

- Link-time
  - Could be seen as JAR assembly
  - Not very common, people prefer mix-n-matching
- Load-time
  - Storing ASTs or source code
  - Compilation is a heavyweight process
    - especially for scalac
  - **Need a lightweight solution**

Specialization - Static

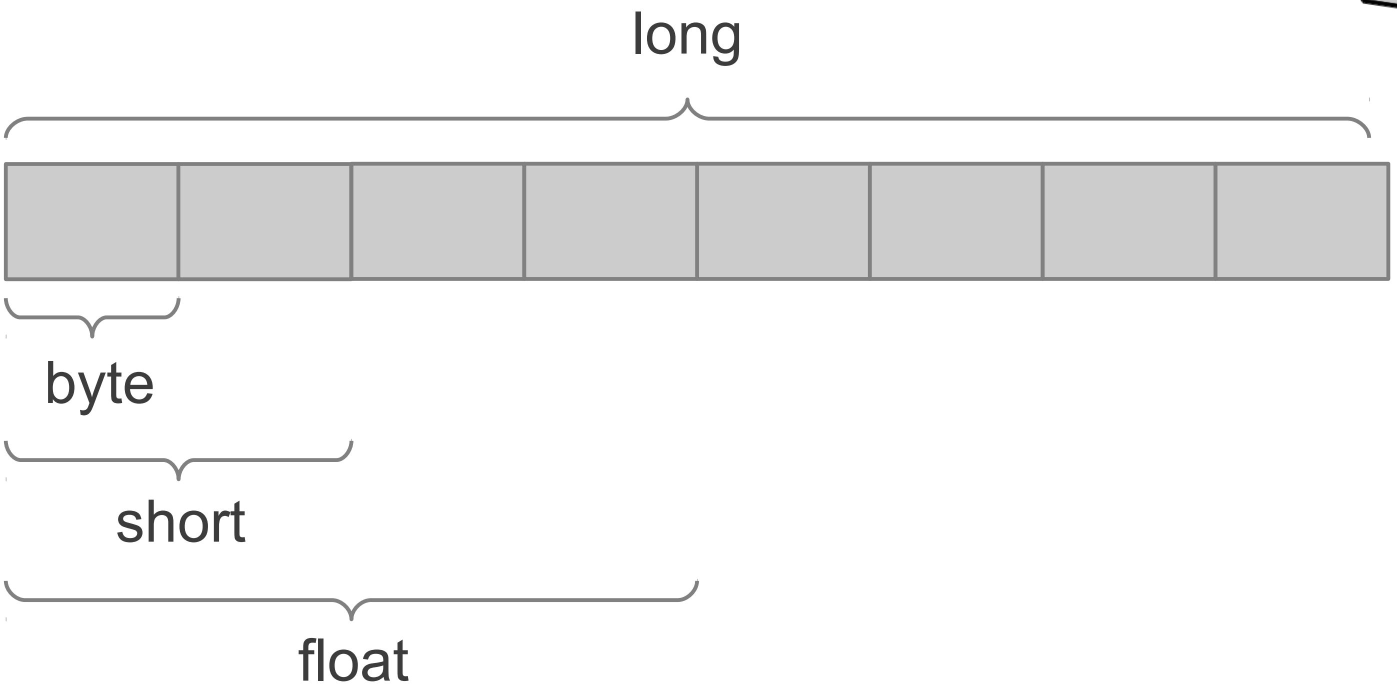
- On-demand
  - Suppose we attached the AST or the source code
  - And expanded it at compile-time
  - Multiple versions of common classes:
    - Function[Int, Int]
      - in the standard library
      - in Akka and other middleware
      - in your application
    - We can't perform de-duplication like C++ linkers do
      - since few people do JAR assemblies

Specialization – On source code

- On bytecode
  - Complex machinery, would prefer a higher level
  - Full Specialization would be easy
    - But it requires reified types
    - And those are slow*
  - Opportunistic specialization
    - Compatibility is very complex to maintain

* PhD Thesis of Michel Schinz, EPFL, 2005

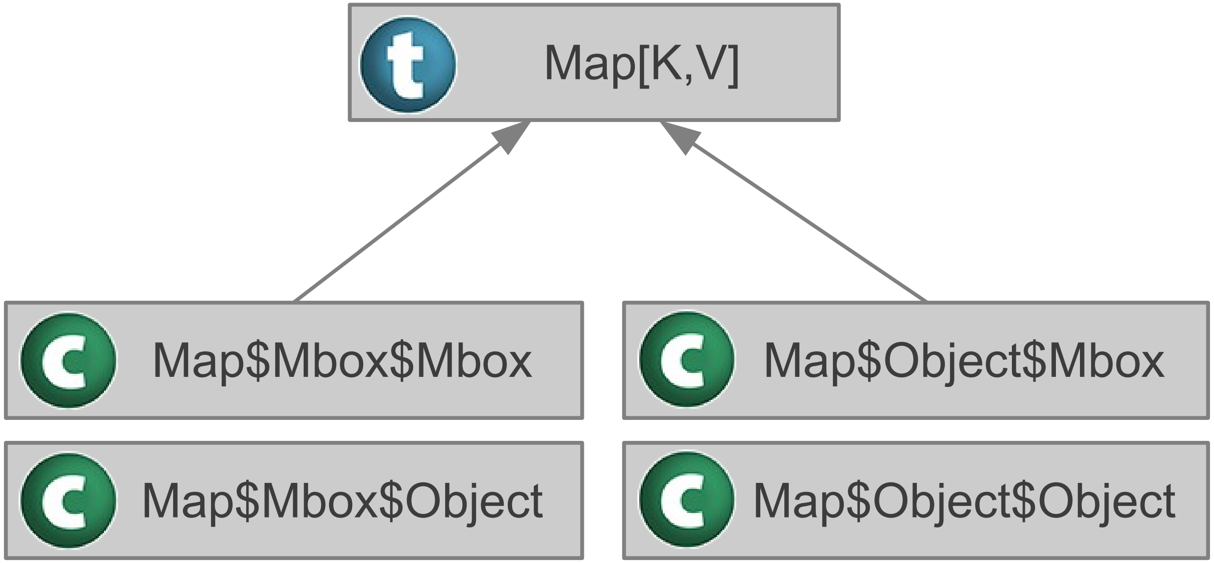
Miniboxing - Insight



- Only applied to stack values
- We'll look at arrays separately

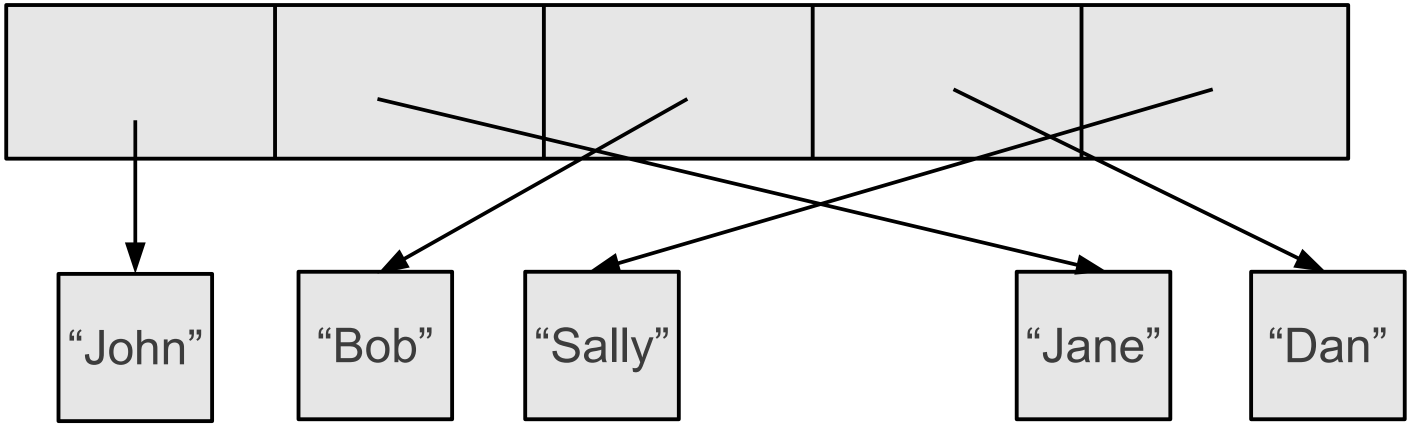
Miniboxing - Scaling

- static expansion, in the library



Arrays

- Monomorphic
  - Array[String], Array[Int], Array[Float]



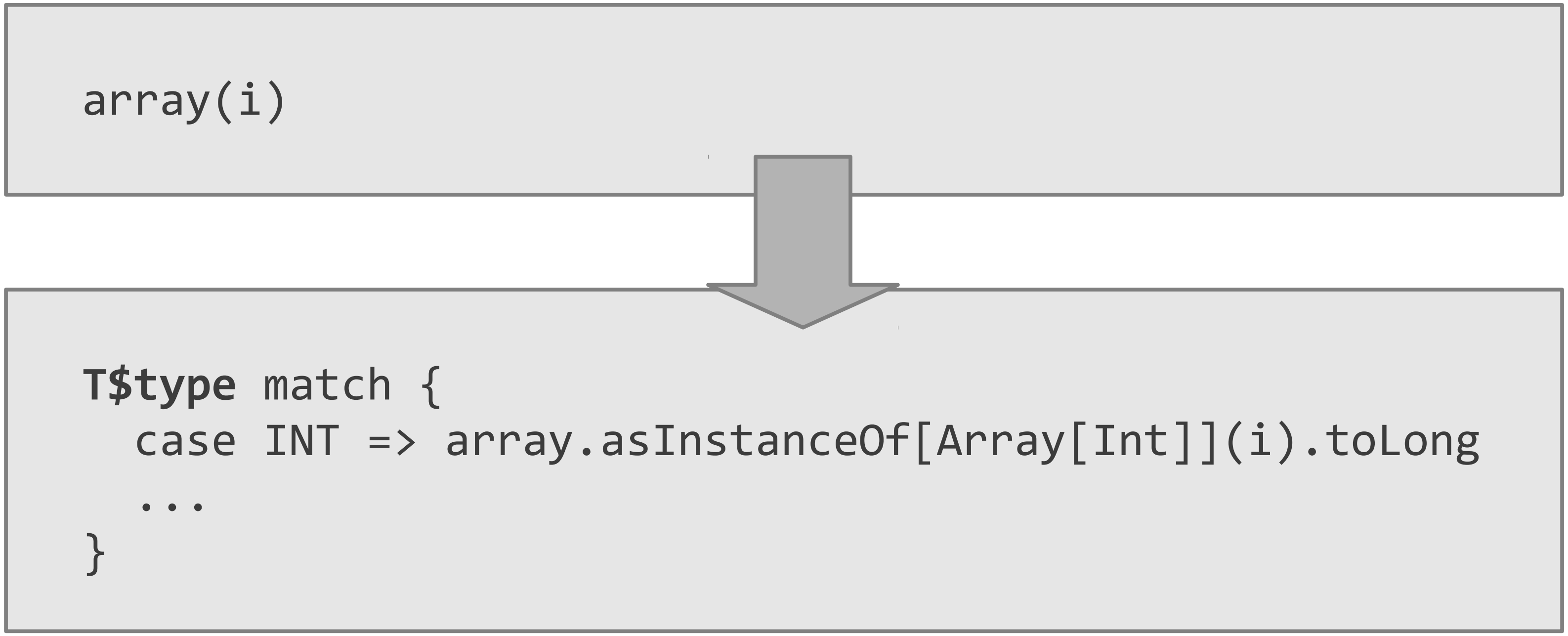
“John” “Bob” “Sally” “Jane” “Dan”

2500	3500	1000

'S'	'a'	'l'	'l'	'y'

Library Support

- Executing code on value types (.hashCode...)
- Interfacing with arrays



```
array(i)
```

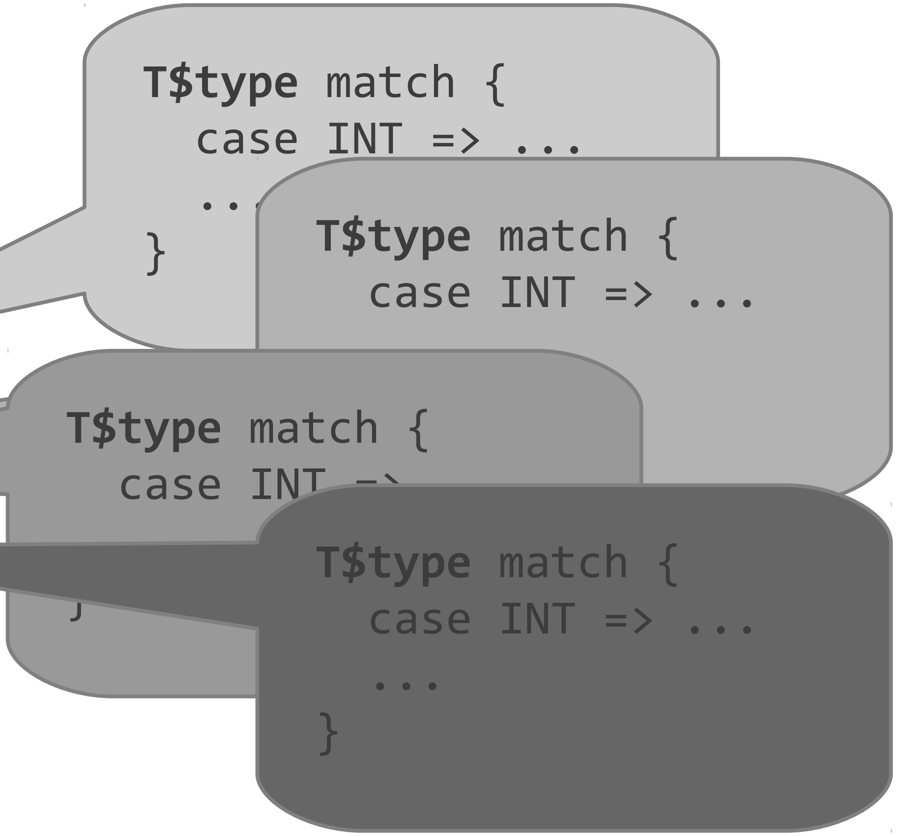
```
T$type match {
  case INT => array.asInstanceOf[Array[Int]](i).toLong
  ...
}
```

Casting and converting to long is fast, noop-fast

Library Support

- `ArrayBuffer.reverse()`

```
def reverse(): Unit {
  var index = 0
  while (index * 2 < length) {
    val opposite = length-index-1
    val tmp1: T = array(index)
    val tmp2: T = array(opposite)
    array(index) = tmp2
    array(opposite) = tmp1
    index += 1
  }
}
```



```
T$type match {
  case INT => ...
  ...
}
```

```
T$type match {
  case INT => ...
```

```
T$type match {
  case INT =>
}
```

```
T$type match {
  case INT => ...
  ...
}
```

Can the JVM optimize such code?

Java Virtual Machine Support

- We confuse the JVM heuristics by inlining manually
- The JVM will do its best
  - Using a single value of T$Type (one type)
    - hoist the switch outside the loop (loop unswitching)
    - get rid of multiple casts (common subexpression elim)
    - get rid of muliple bounds checks (check elimination)
    - **good speed**
  - Using more types…
    - can't hoist the switch outside → bad performance
    - **we need to optimize this code too**

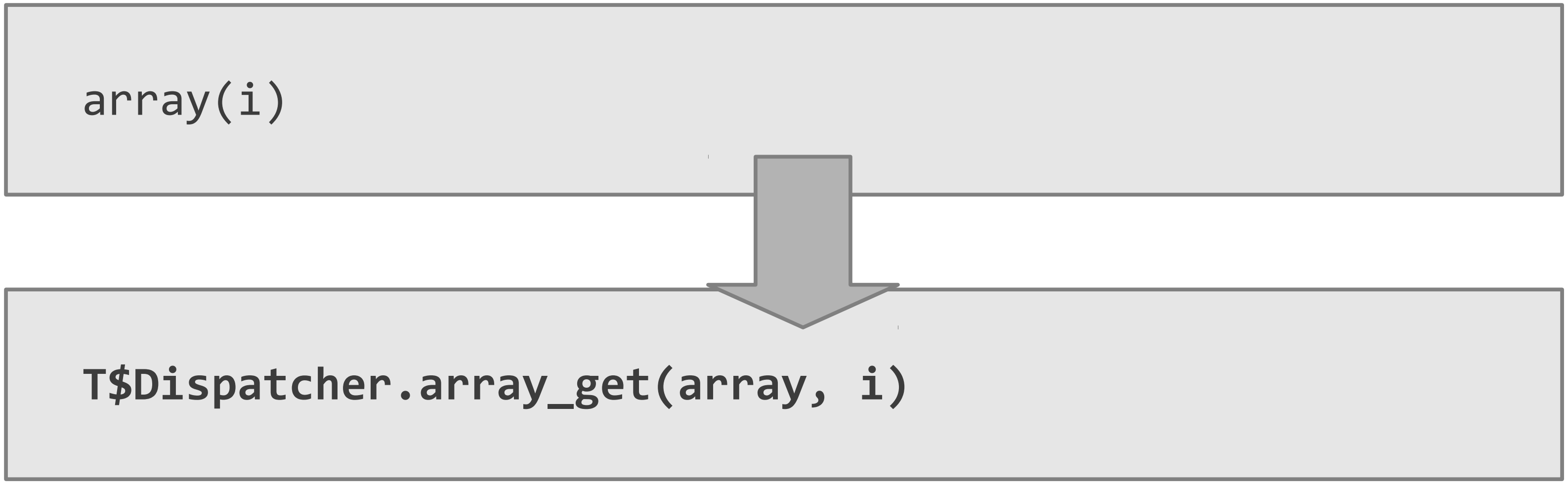
Can we lift the switch ourselves?

Dispatching

- Systems approach: `T$type`
  - byte value
  - encodes the type of a parameter (`INT`, `...`)
  - `switch` to do the right operation
- Object-oriented approach
  - common operations object (Dispatcher)
  - specialized instances for all types
  - a type class for Haskell folks

Object-oriented Dispatching

- `T$Dispatcher: Dispatcher`
  - `IntDispatcher, LongDispatcher, ...`
- Common operators



```
array(i)
```

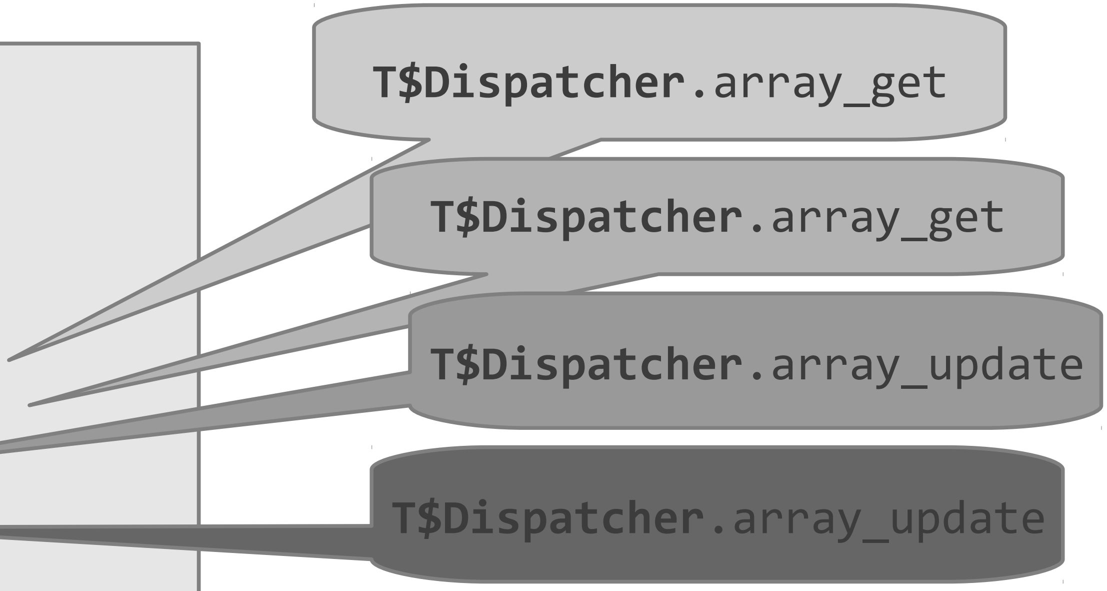
```
T$Dispatcher.array_get(array, i)
```

So far so good

Object-oriented Dispatching

- `ArrayBuffer.reverse()`

```
def reverse(): Unit {
  var index = 0
  while (index * 2 < length) {
    val other = length-index-1
    val tmp1: T = array(index)
    val tmp2: T = array(other)
    array(index) = tmp2
    array(other) = tmp1
    index += 1
  }
}
```



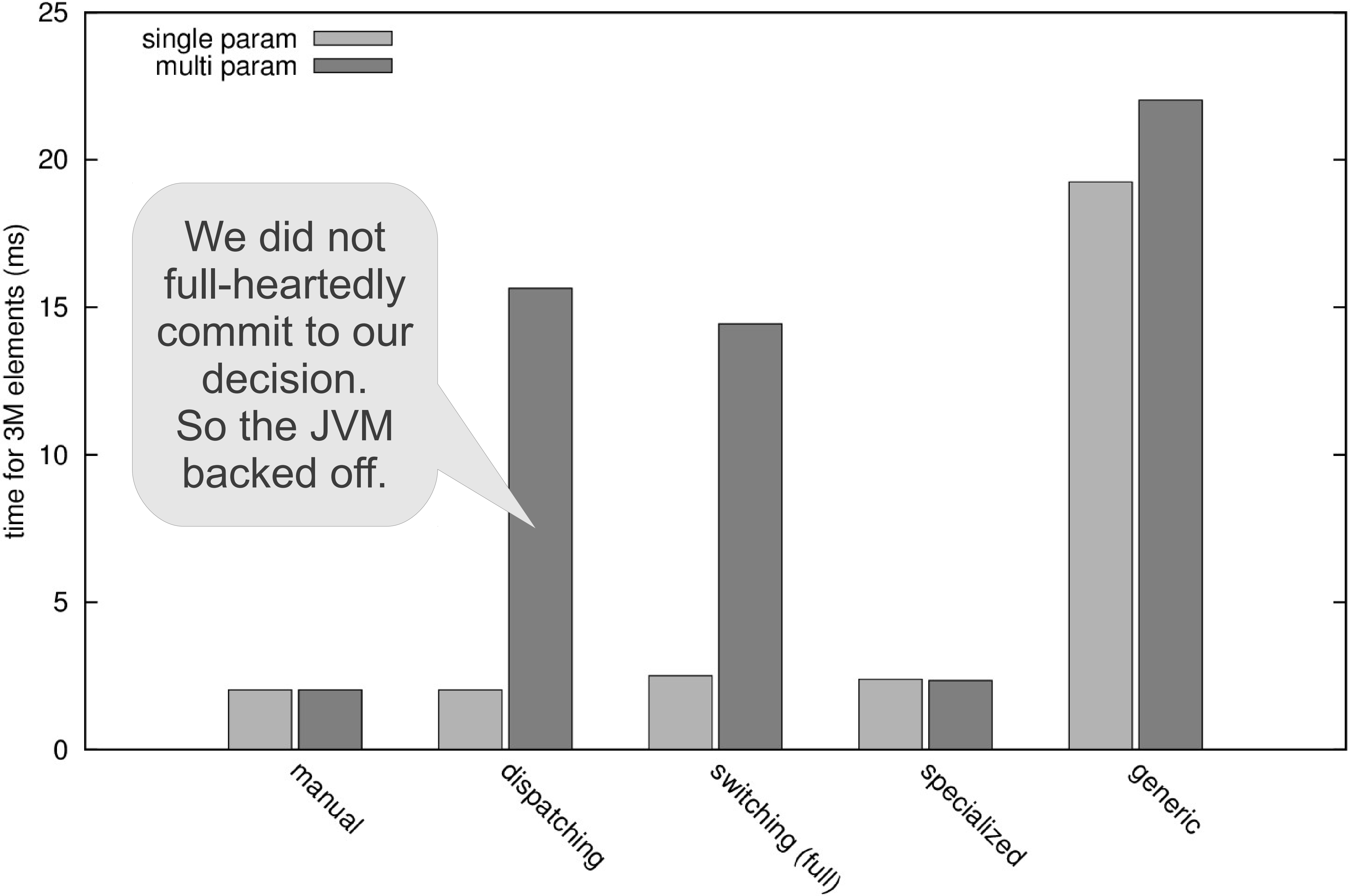
T$Dispatcher.array_get

T$Dispatcher.array_get

T$Dispatcher.array_update

T$Dispatcher.array_update

We committed to the data representaion before



We did not full-heartedly commit to our decision.
So the JVM backed off.

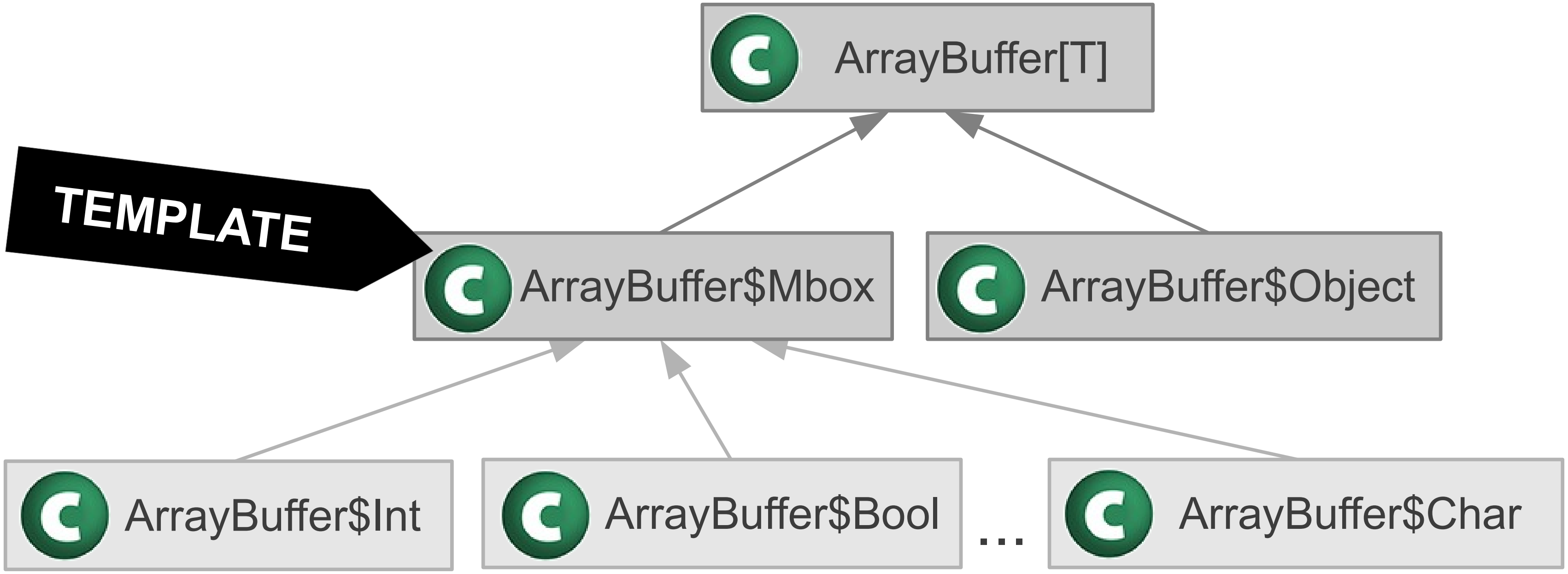
ArrayBuffer.reverse()

OO Dispatching Performance

- One or two dispatchers in a callsite
  - as fast as manually specialized code
- Three+ dispatchers
  - call to `array_get` becomes **megamorphic**
  - it is not inlined anymore
  - no optimization in the loop
  - awful results, in some cases slower than generic

Runtime Specialization

- In practice, dispatcher set per INSTANCE
- Semantically - dispatcher per SPECIALIZATION
- Remember the ArrayBuffer diagram?



Runtime Specialization

- `ArrayBuffer$Int`
  - has `IntDispatcher` set statically
- Operations can be inlined
  - since `IntDispatcher` is final
  - calls never become megamorphic
  - optimizations can be done inside the loop

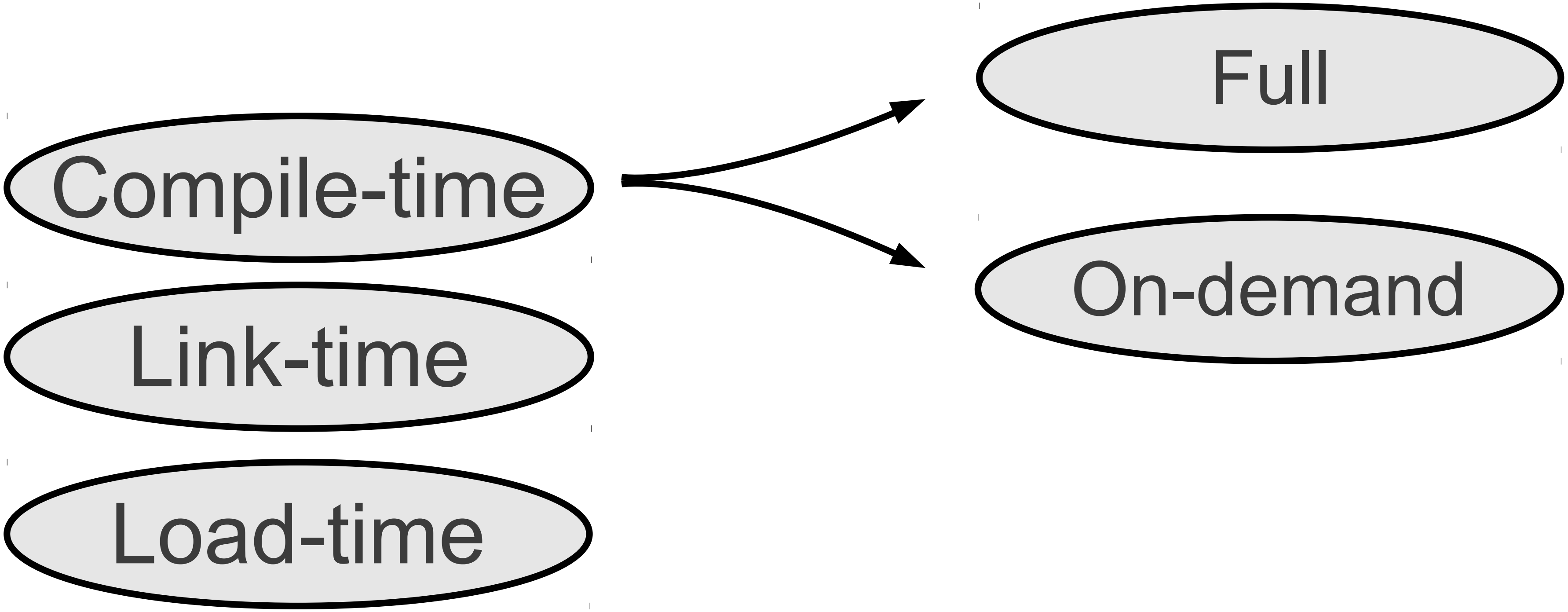
Class modification and loading cost?

Numbers

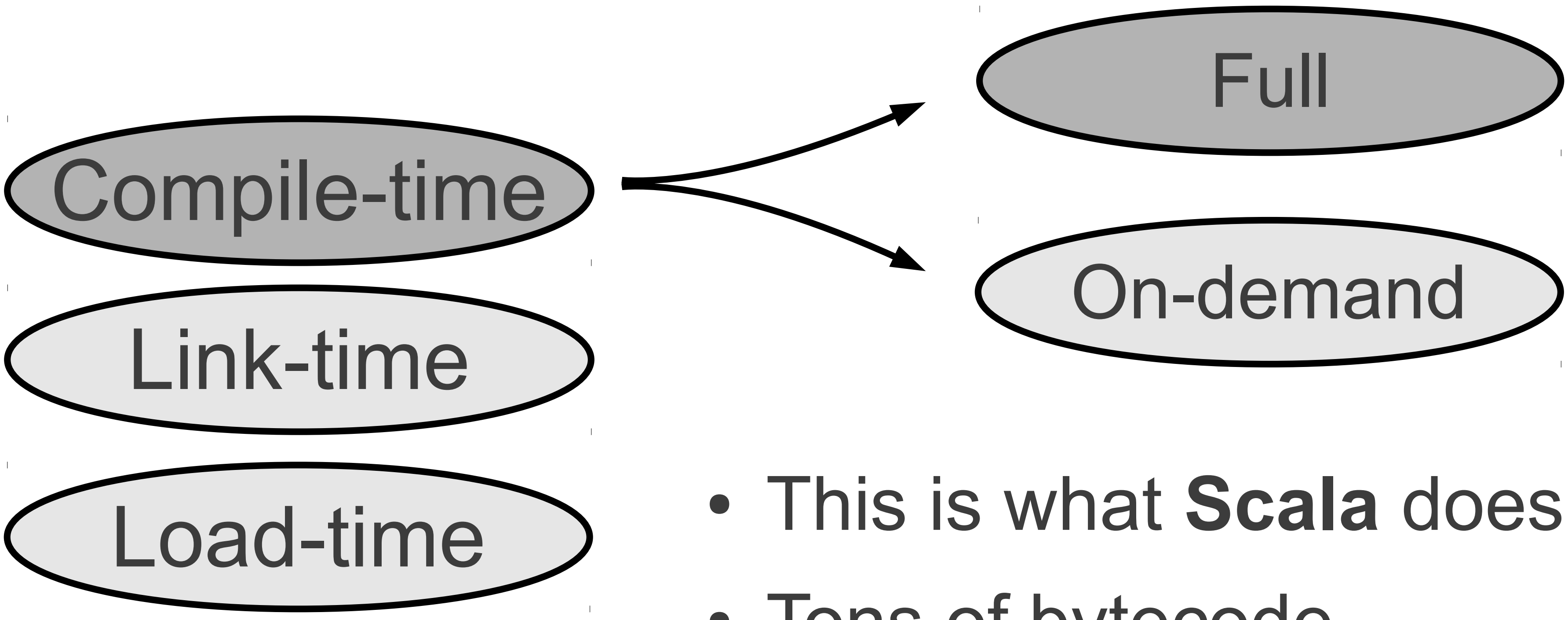
- `ArrayBuffer.reverse()`

```
                       ideal :     0.85883 ms +/-   0.05687
     miniboxed switching mono :     0.62966 ms +/-   0.21116
     miniboxed switching mega :     4.21423 ms +/-   1.15425
     miniboxed switching w/cl :     0.87526 ms +/-   0.05992
      miniboxed dispatch mono :     0.75230 ms +/-   0.21551
      miniboxed dispatch mega :     4.90191 ms +/-   1.52366
 miniboxed dispatch w/cl mono :     0.77984 ms +/-   0.14872
 miniboxed dispatch w/cl mega :     0.80932 ms +/-   0.14671
            specialized mono :     0.75813 ms +/-   0.16000
            specialized mega :     0.61374 ms +/-   0.20144
                     generic :     9.76250 ms +/-   1.55297
```

Specialization in different languages



Specialization in different languages



Compile-time

Link-time

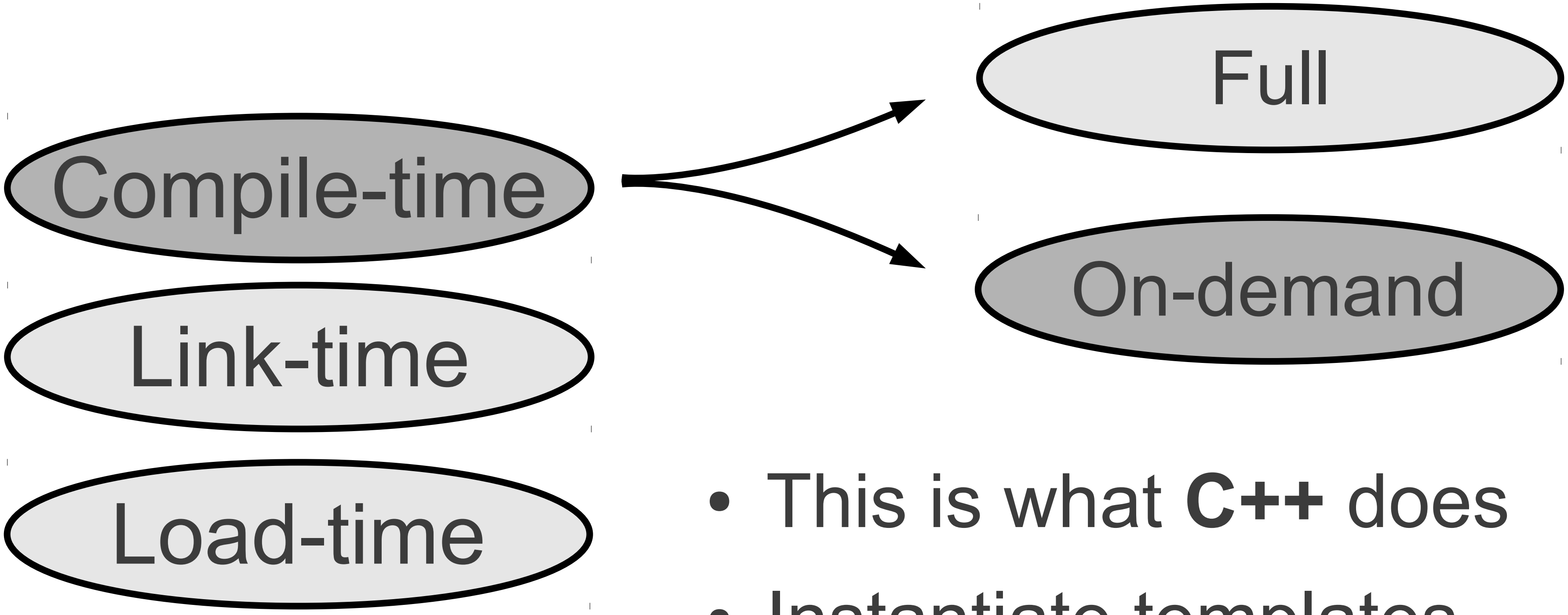
Load-time

Full

On-demand

- This is what **Scala** does
- Tons of bytecode

Specialization in different languages



Compile-time

Link-time

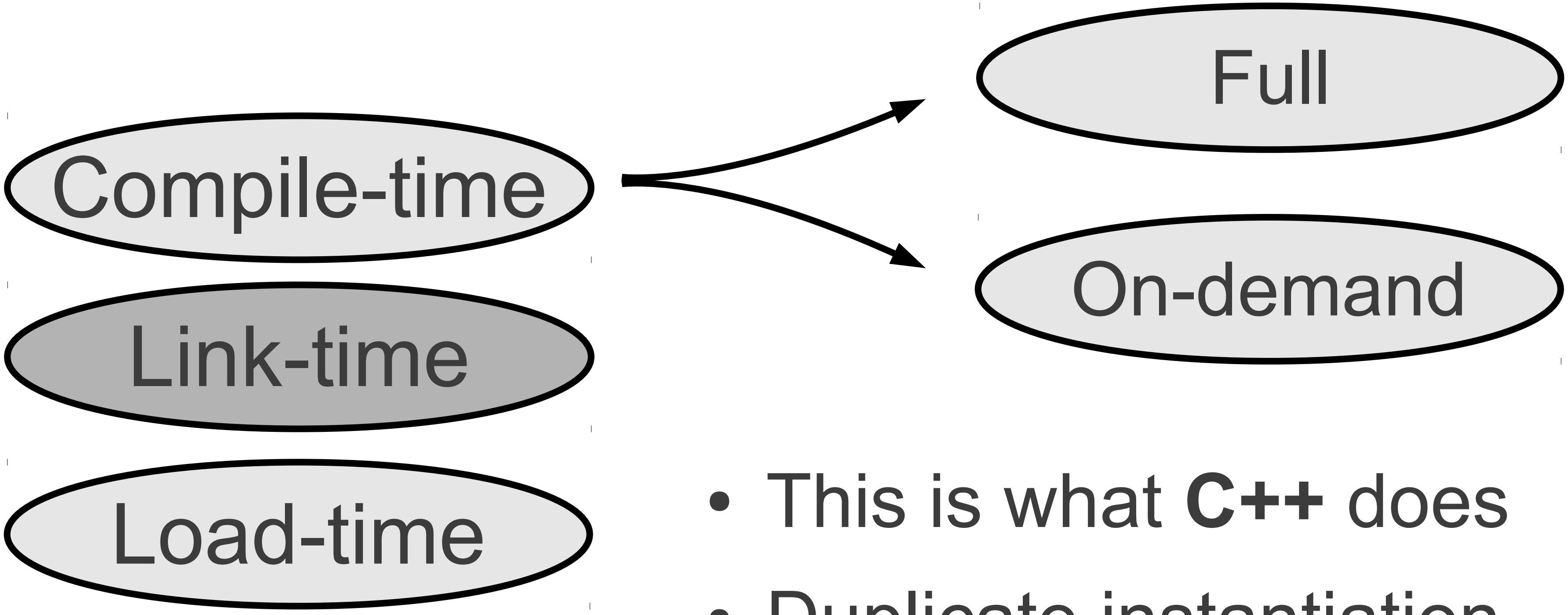
Load-time

Full

On-demand

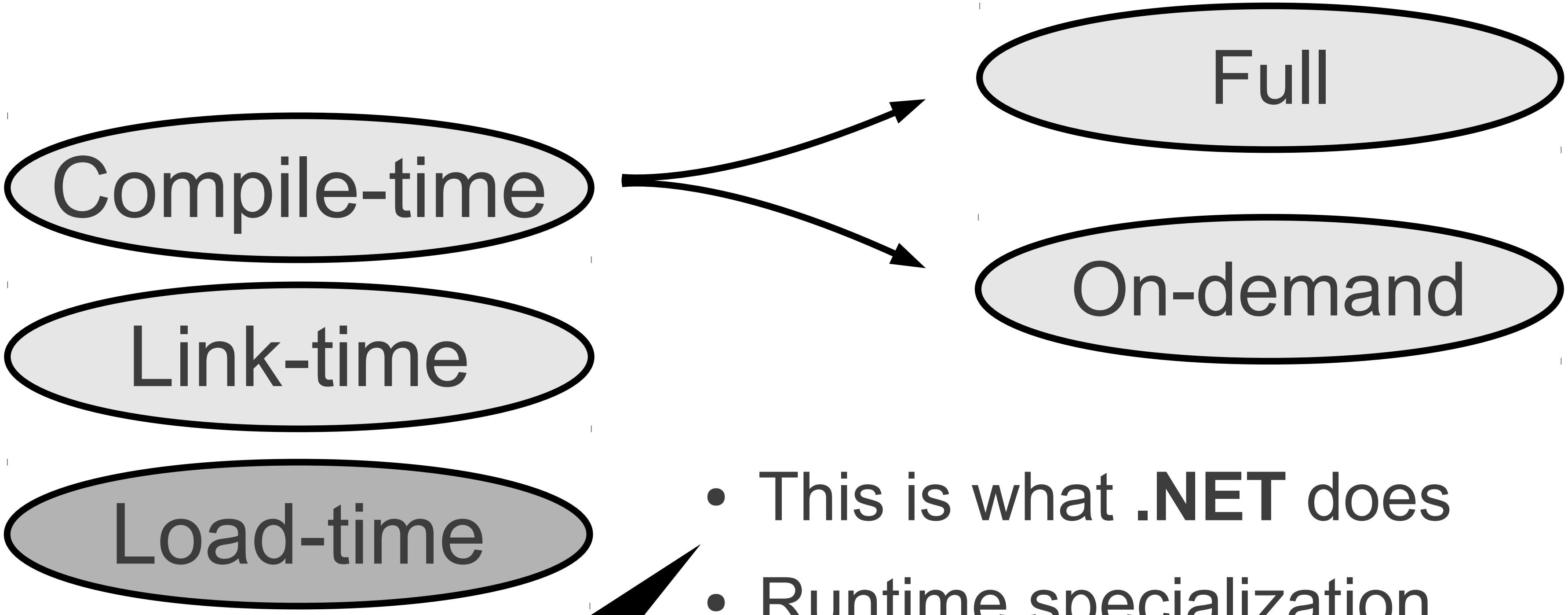
- This is what **C++** does
- Instantiate templates
  - Libraries are mostly headers
- Duplicate instantiation

Specialization in different languages



- This is what **C++** does
- Duplicate instantiation
- Cleaned up by the linker

Specialization in different languages

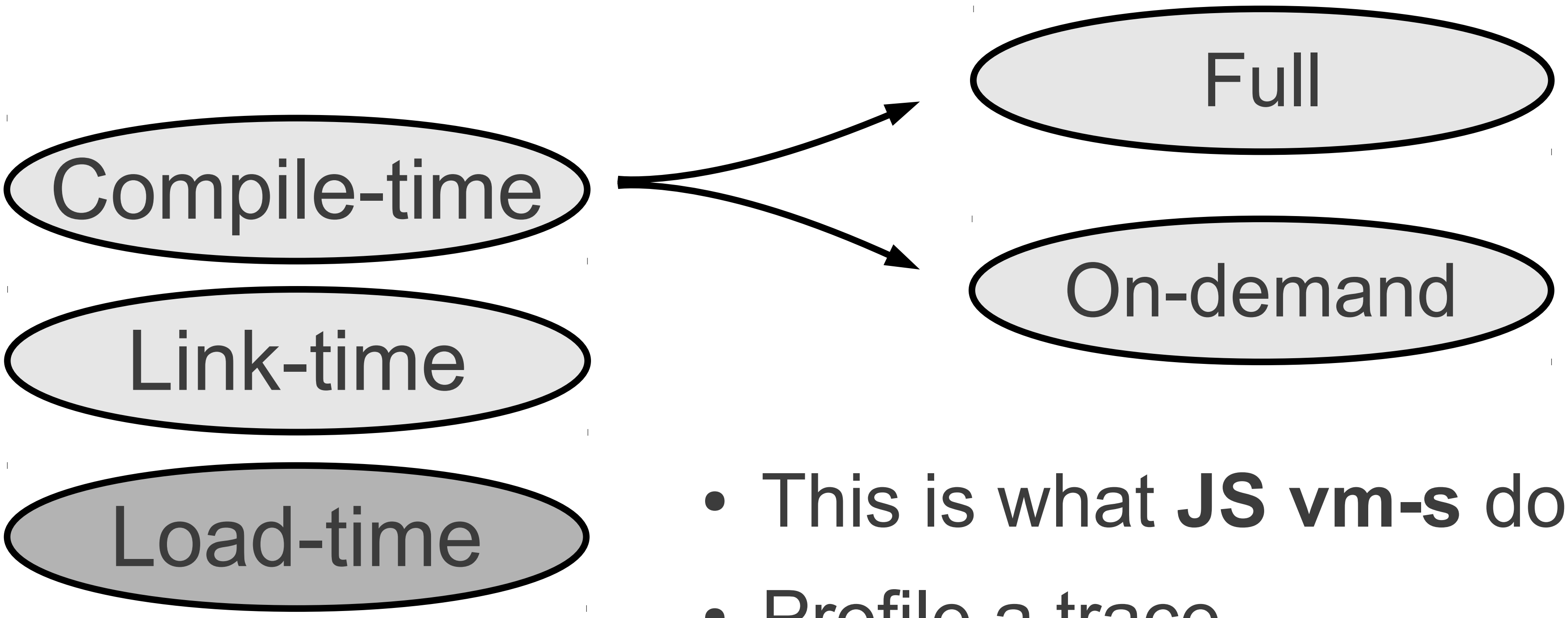


- This is what **.NET** does
- Runtime specialization
  - Based on templates
- Type system fixed
  - in the virtual machine

One more ACE up its sleeve:

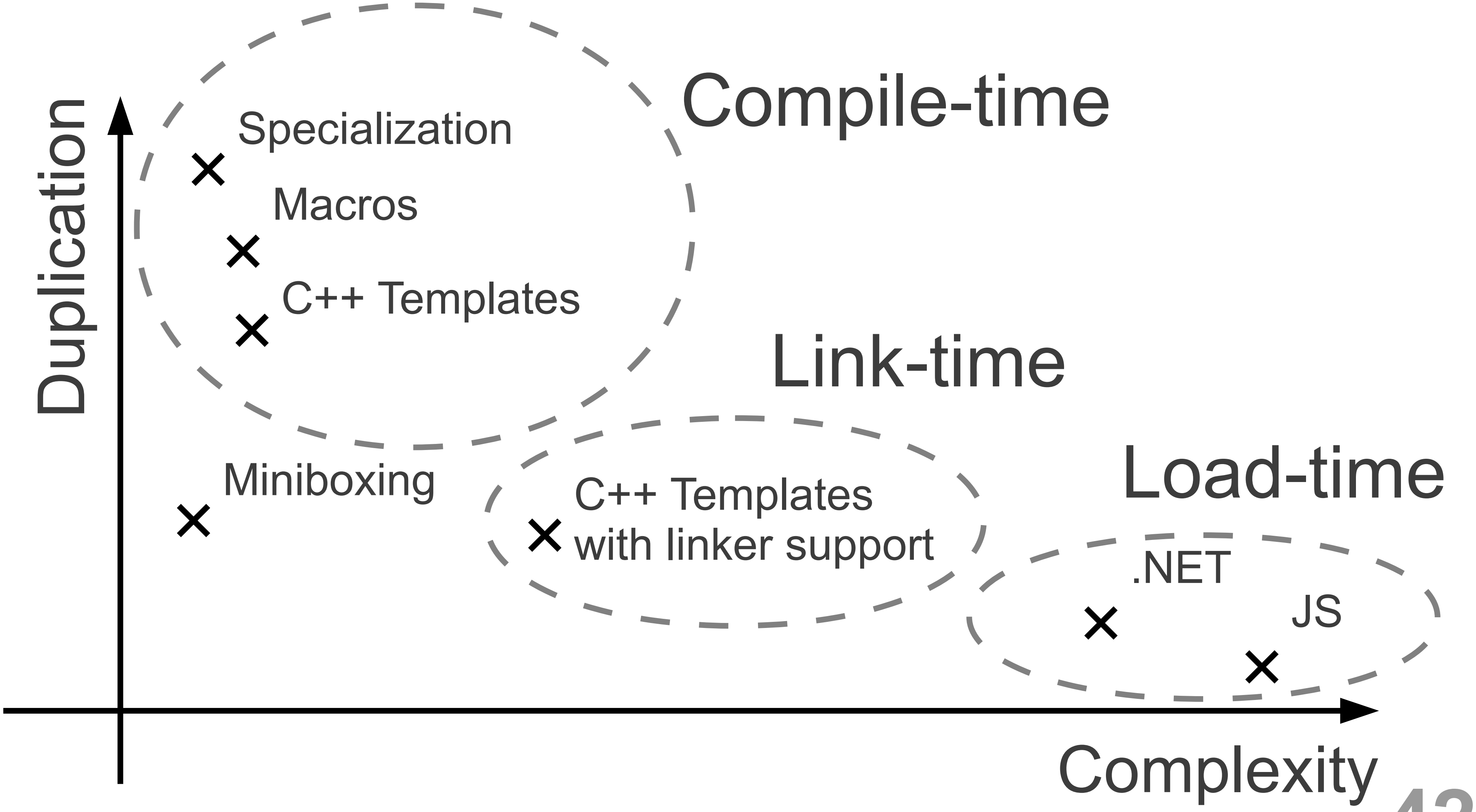
Reified types

Specialization in different languages



- This is what **JS vm-s** do
- Profile a trace
- Specialize it
- Fallback to interpretation

In perspective


Compile-time
Specialization
Macros
C++ Templates
Duplication
Link-time
Miniboxing
C++ Templates
with linker support
Load-time
.NET
JS
Complexity