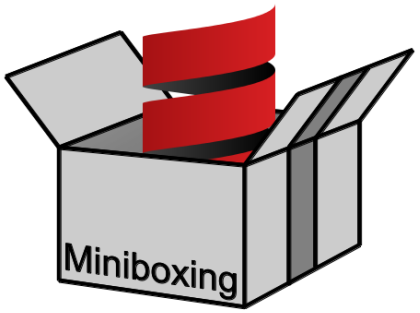


Miniboxing

Runtime Specialization on the JVM

Preliminary benchmark results

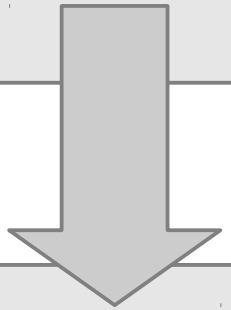
LAMP, EPFL, 7th of March 2013

Vlad Ureche

vlad.ureche@epfl.ch

Miniboxing transformation

```
class C[@minispec T](implicit mf: Manifest[T]) {
  var a = new Array[T](5)
}
```



```
abstract class C[T](implicit mf: Manifest[T]) {
  def a: Array[T]
  def a_=(_a: Array[T])
  def a_J: Array[T]
  def a_=_J(_a: Array[T])
}
```

Miniboxing transformation

```
class C_J[T] extends C[t]{
  private[this] _a: Array[T] // erased to Object

  def <init>(T$Type: Byte) = {        // call to the
    this._a = mbarray_new(T$Type, 5) // library but
  }                                   // inlined by
                                      // the backend
  // actual implemen      ns
  def a: Array[T] = a
  def a_=(_a: Array[T                 = a = J(_a)
  def a_J: Array
  def a_=_J(_a:
}
```

```
T$type match {
  case INT => new Array[Int](5)
  ...
}.asInstanceOf[Array[T]]
```

Main benchmark

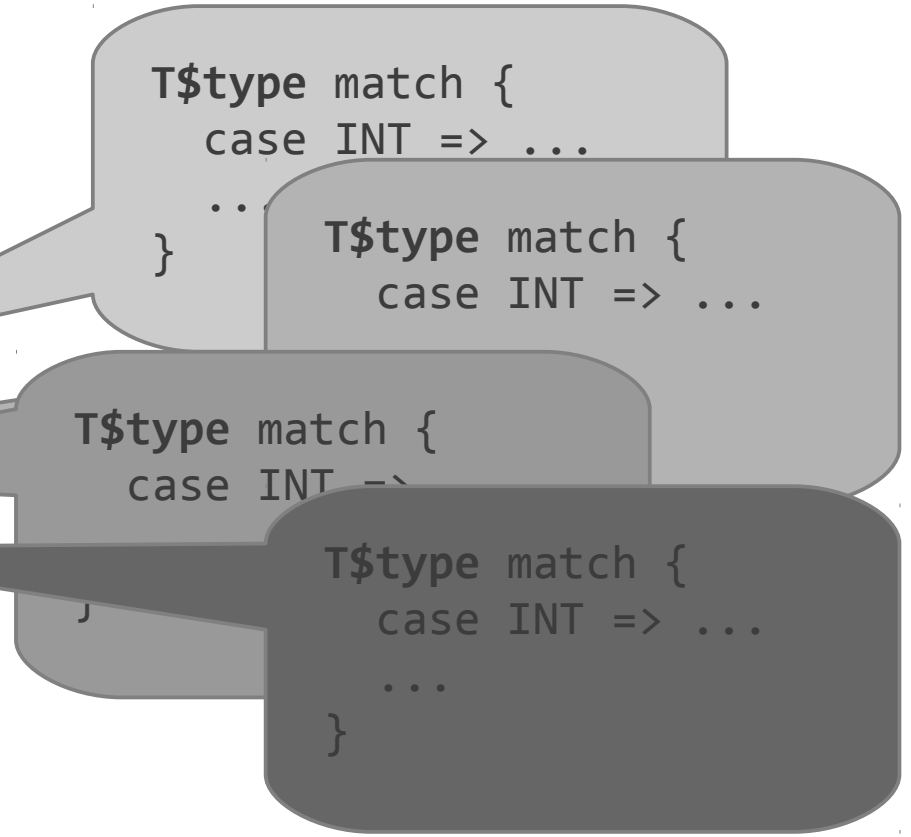
- `ArrayBuffer.reverse()`
  - VM can still perform array ops
  - **toughest benchmark**
  - **12x speedup over generic**

```
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
  }
}
```

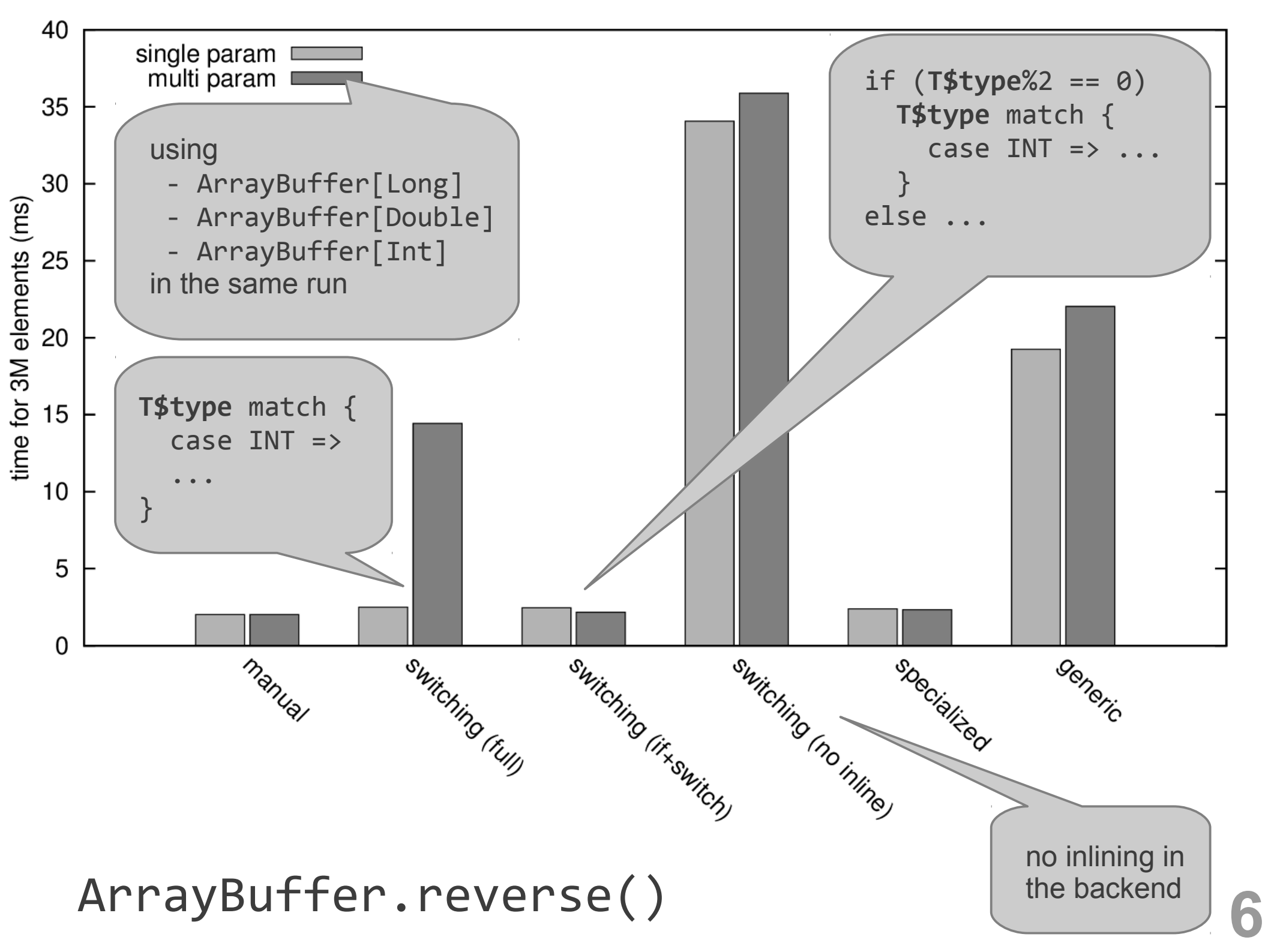
Library Support

- `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
  }
}
```



Can the JVM optimize such code?



ArrayBuffer.reverse()

Library Support

- switch+if works without slowdowns for up to 6 type parameters:

```
if (T$type%2 == 0)
  T$type match {
    case INT => ...    // this path is being benchmarked, fast path
    case CHAR => ...
    case BOOLEAN => ...
    case BYTE => ...
  }
else
  T$type match {
    case DOUBLE => ... // this path has been exercised before, slow path
    case FLOAT => ...  // this path has been exercised before, slow path
    case LONG => ...   // this path has been exercised before, slow path
    case UNIT => ...   // this path has been exercised before, slow path
    case SHORT => ...  // this path has been exercised before, slow path
  }
```

But it's still not a complete solution

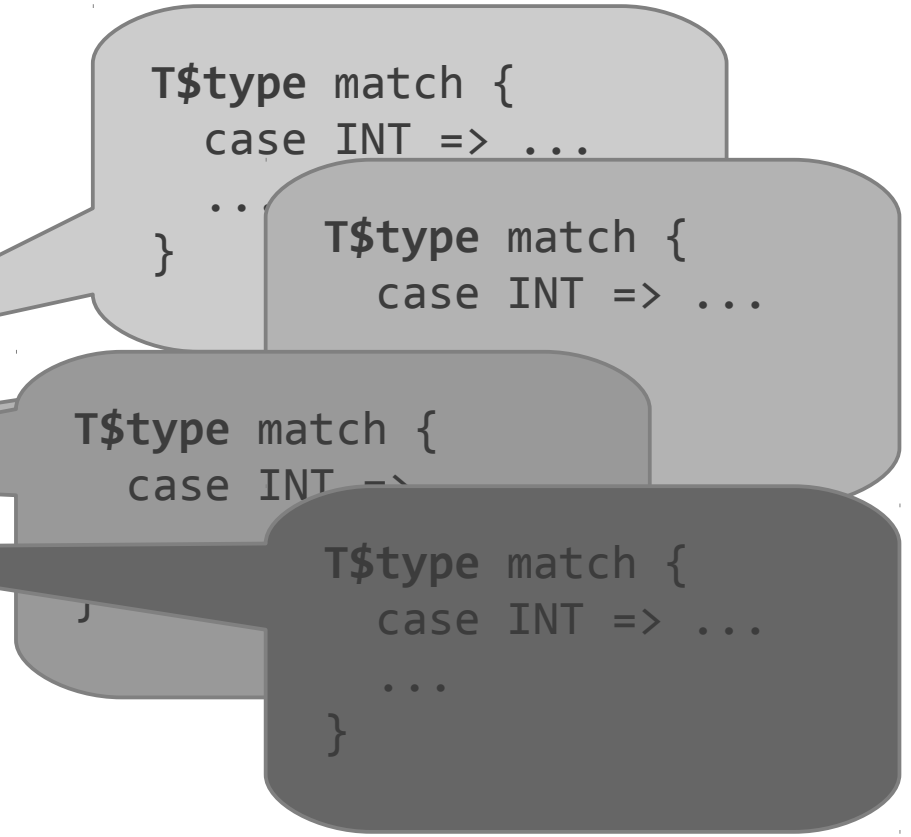
Library Support

- Insight:
  - Let's lift the switch on the entire body of `reverse`
  - The JVM can surely optimize that
  - What if the loop is not in the current method?
    - We'd still be doing the switching at each loop
    - We need a more generic way, instance-wide

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(opposite)
    array(index) = tmp2
    array(opposite) = tmp1
    index += 1
  }
}
```



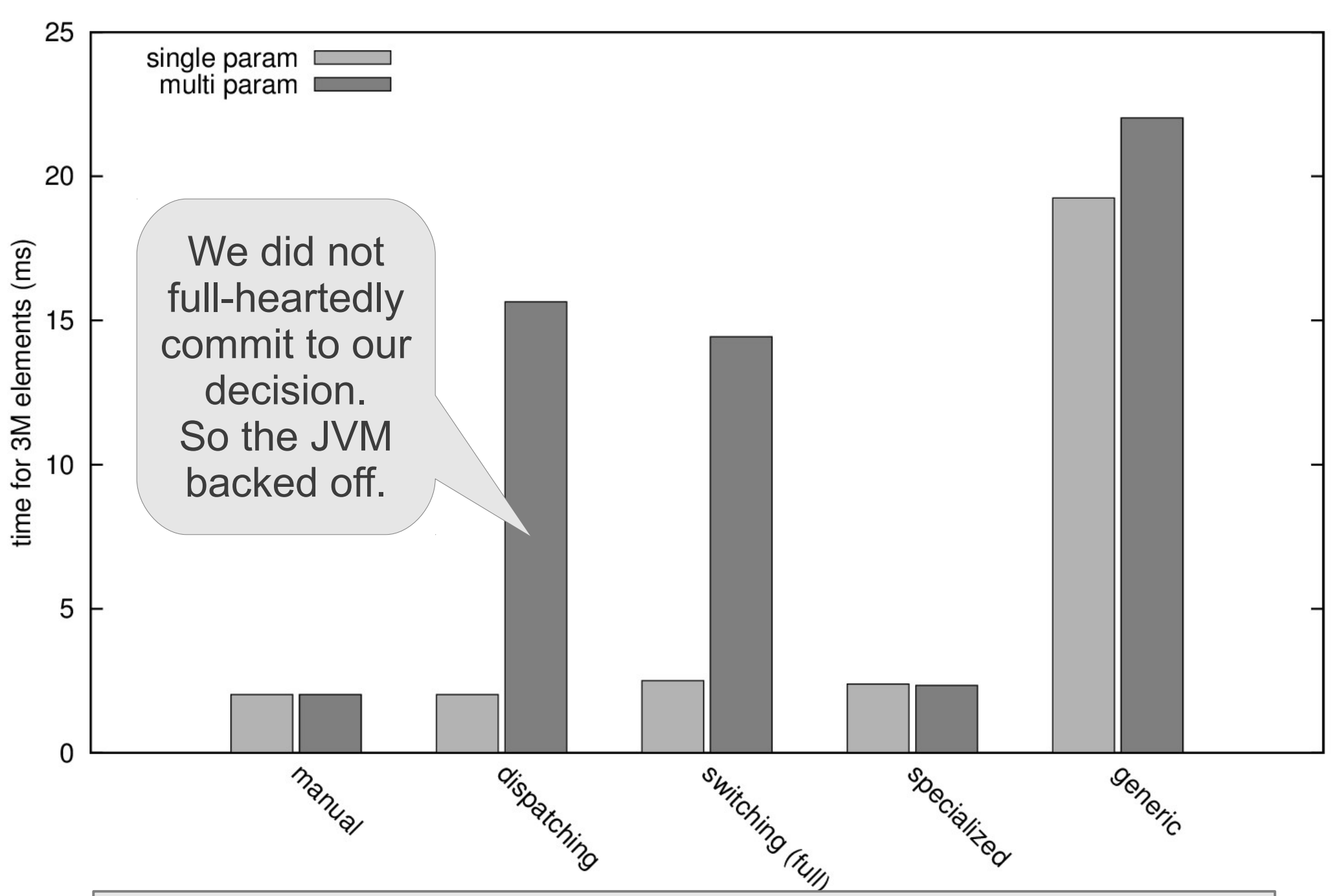
Need to lift the switches outside the loop

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
  }
}
```

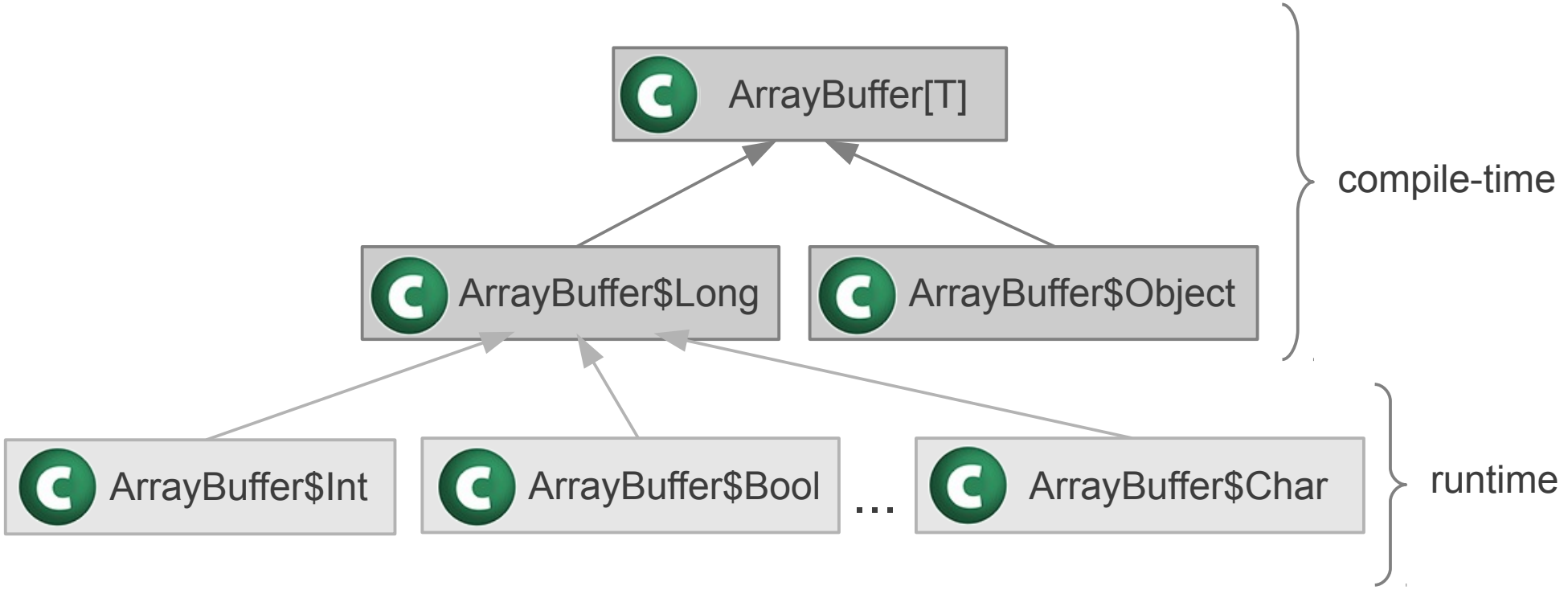
We committed to the data representaion early



We need to make a promise and stick to it

Runtime Specialization

- dot net-like runtime specialization
  - two stages: compile-time and runtime



What do we gain?

Runtime Specialization

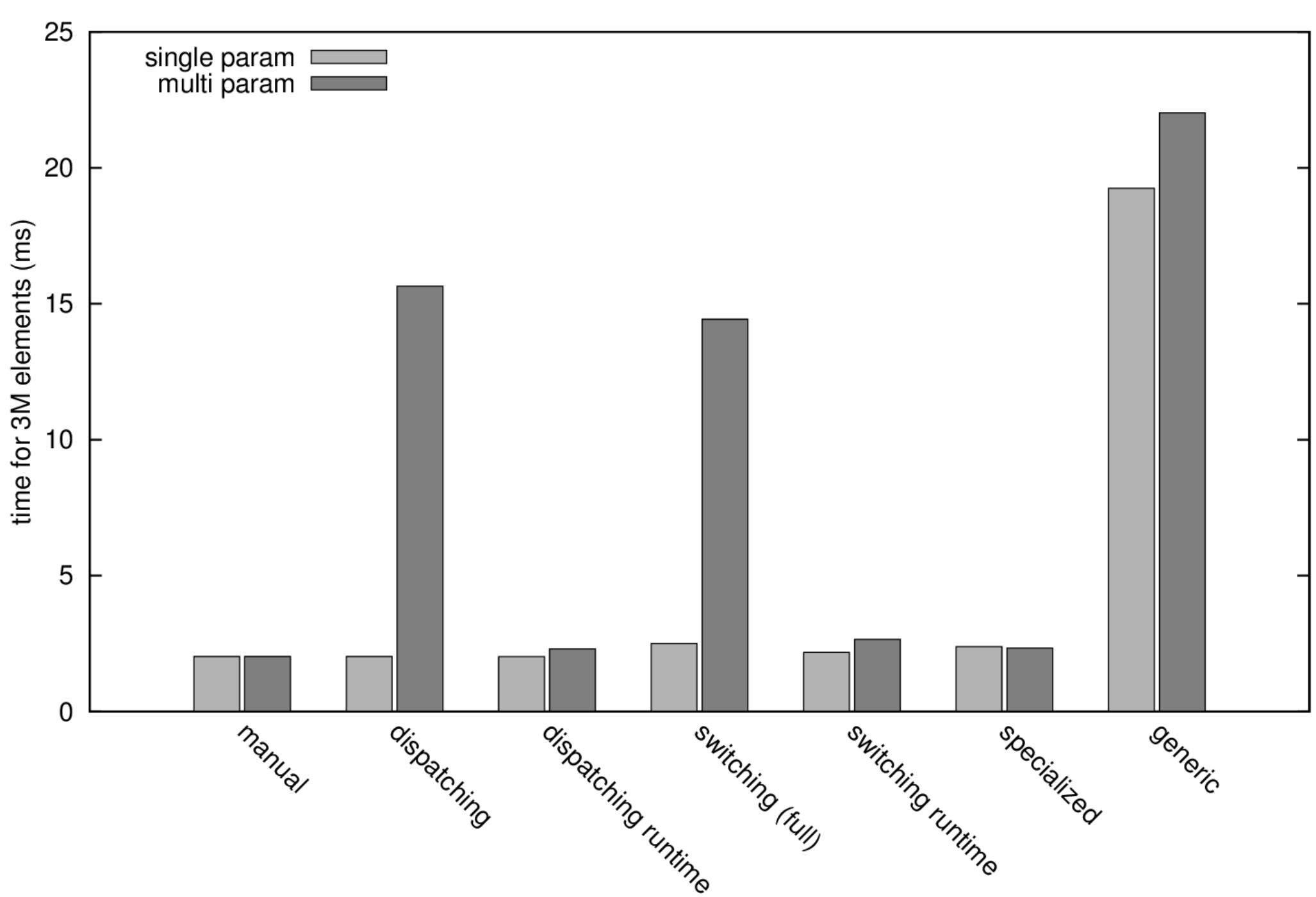
- `ArrayBuffer$Int`
  - has `IntDispatcher` set statically
- Operations can be inlined
  - since `IntDispatcher` is final
  - calls never become megamorphic
  - optimizations can be done inside the loop
- Separate call sites for `array_get`

Class modification and loading cost?

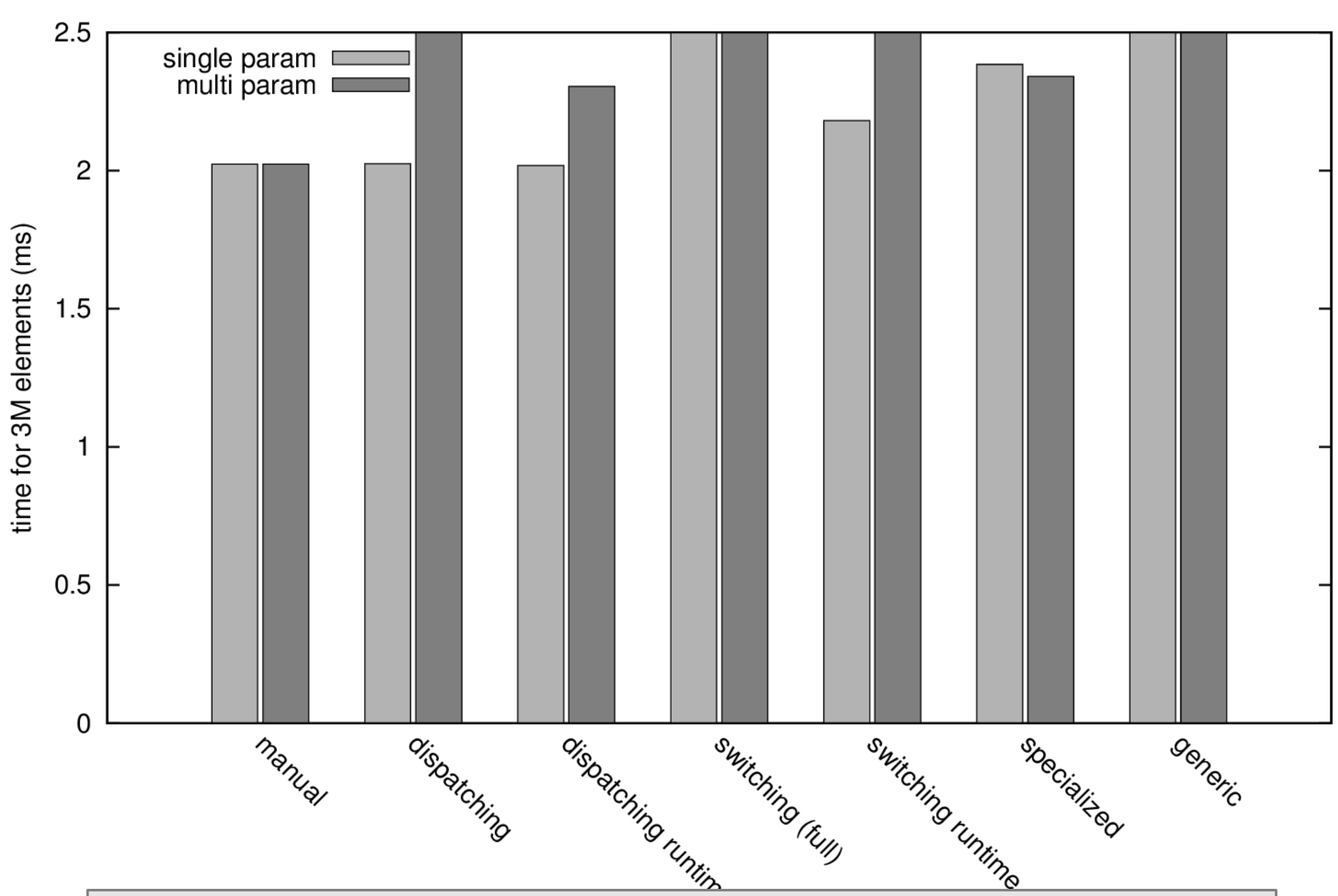
Runtime Specialization

- One-time non-negligible cost
  - load `...$Long` bytecode
  - modify it
  - build a `Class[_]` object
  - instantiate it
- Apply the cost to the factory
  - instead of the individual class
  - turns out the cost amortizes well (10-50 uses)
  - Only local classloader required

Class modification and loading cost?



`ArrayBuffer.reverse()`



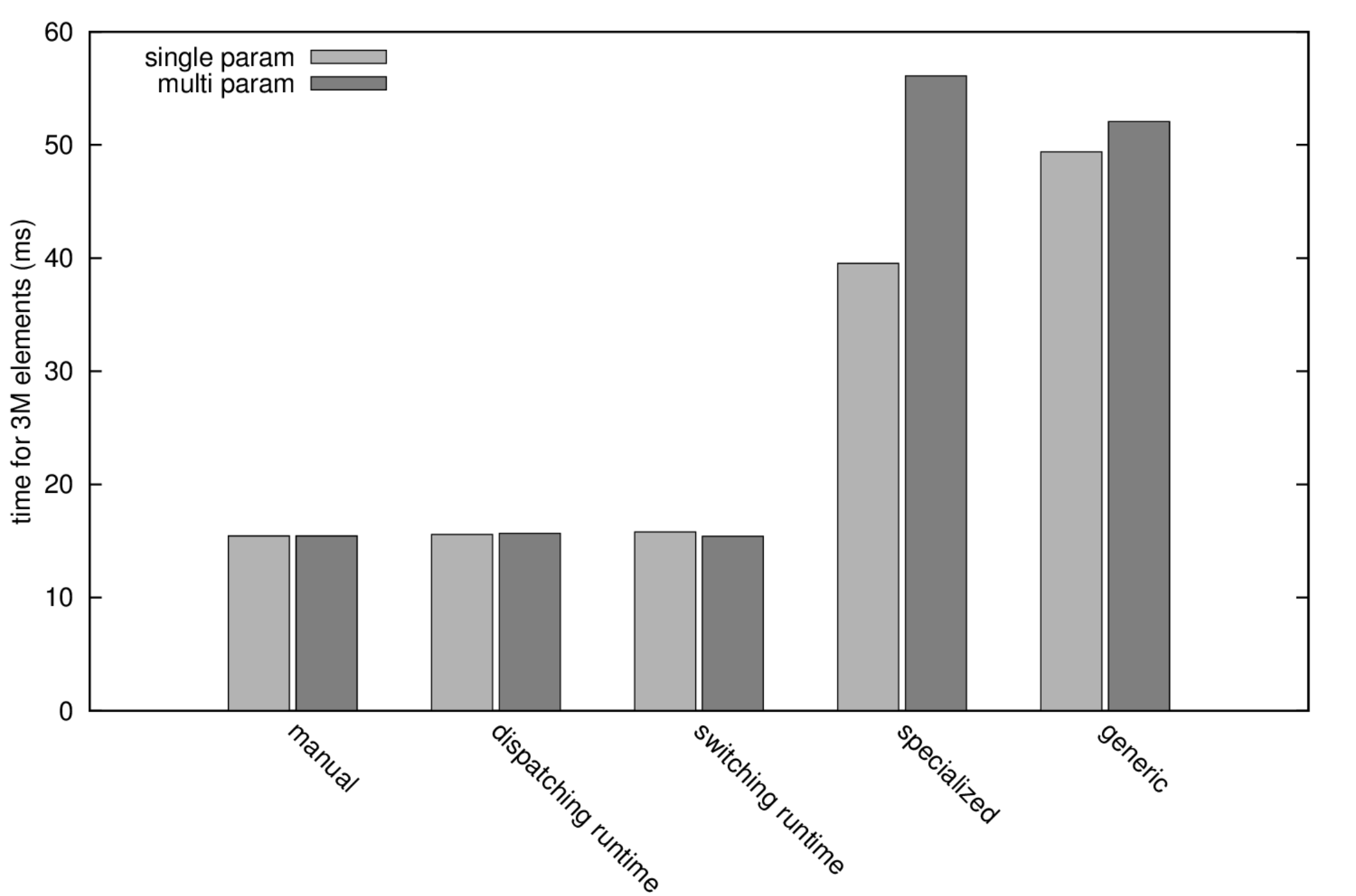
Looks good on this benchmark. What about others?

ArrayBuffer.reverse()

All benchmarks

- `ArrayBuffer.insert()`
  - nested calls performance
    - confusing the JVM inline heuristics => this blows up

```
def add(elem: T) = {
  extend()
  array(elemCount) = elem
  elemCount += 1
}

def extend(): Unit = {
  if (elemCount == size) {
     ...
  }
}
```

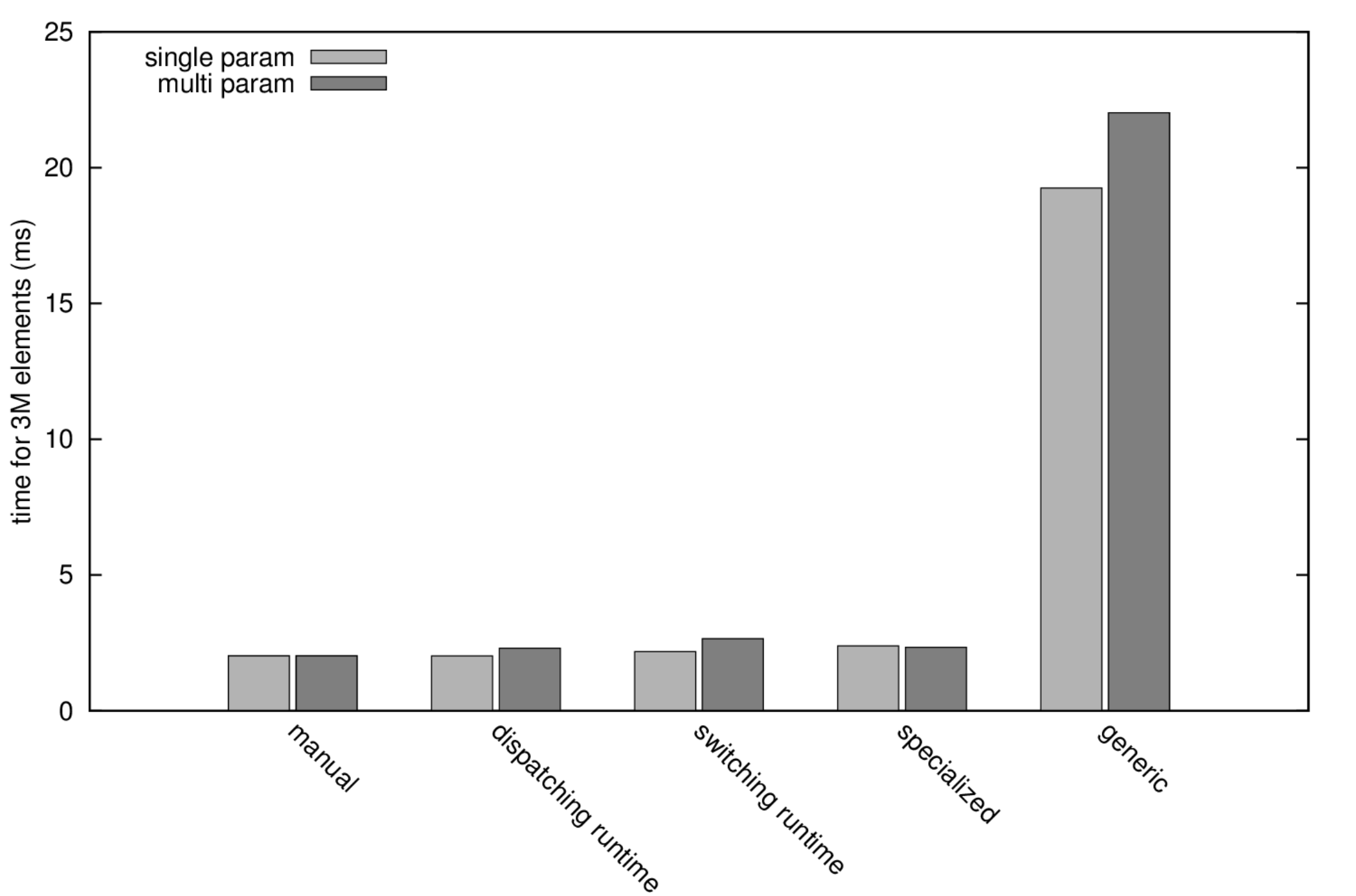


`ArrayBuffer.insert()`

All benchmarks

- `ArrayBuffer.reverse()`
  - VM can still perform array ops
  - **toughest benchmark**
  - **10x+ speedup over generic**

```
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
  }
}
```

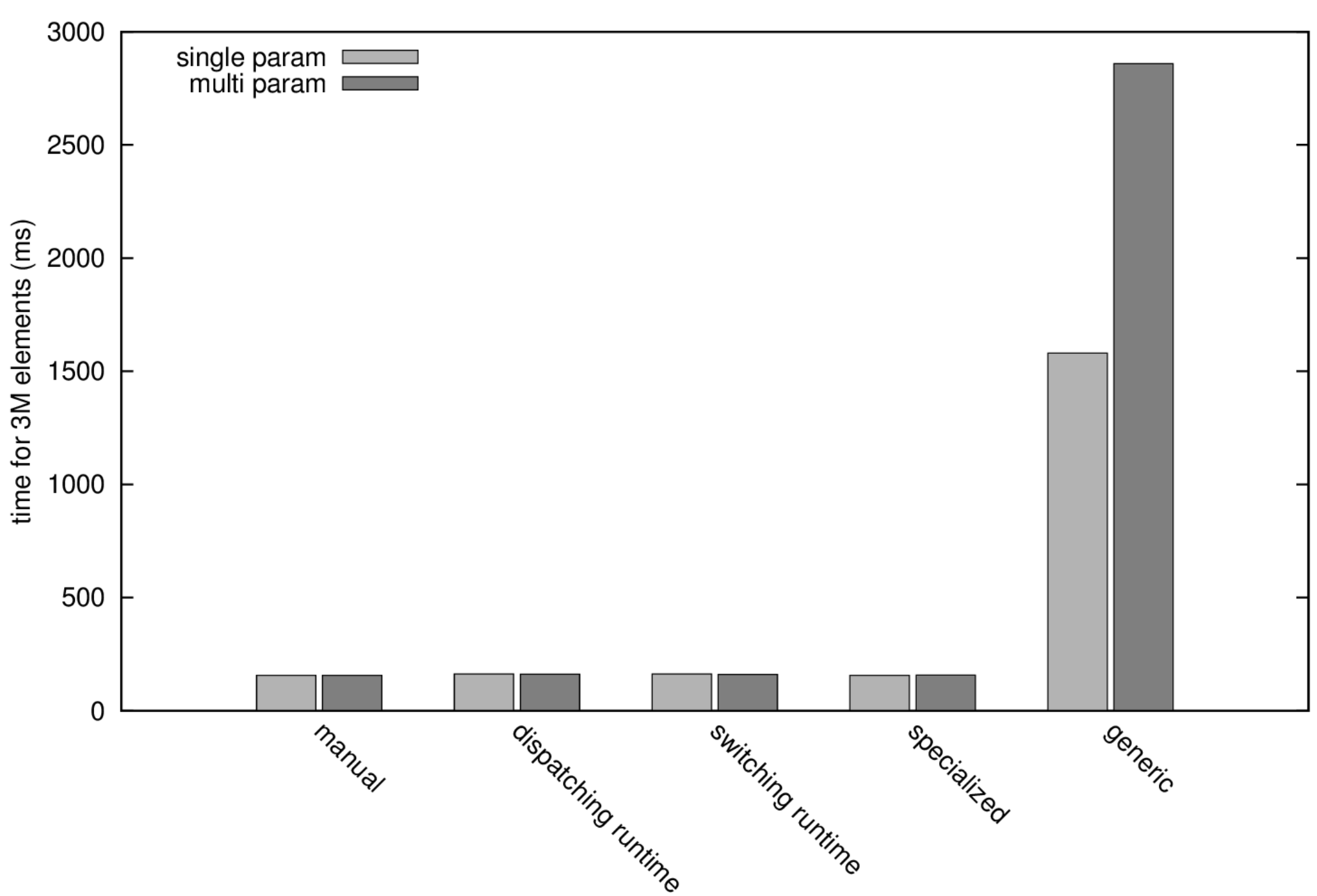


`ArrayBuffer.reverse()`

All benchmarks

- `ArrayBuffer.contains()`
- == operator performance

```
def contains(elem: T): Boolean = {
  var pos = 0
  while (pos < elemCount){
    if (getElement(pos) == elem)
      return true
    pos += 1
  }
  return false
}
```

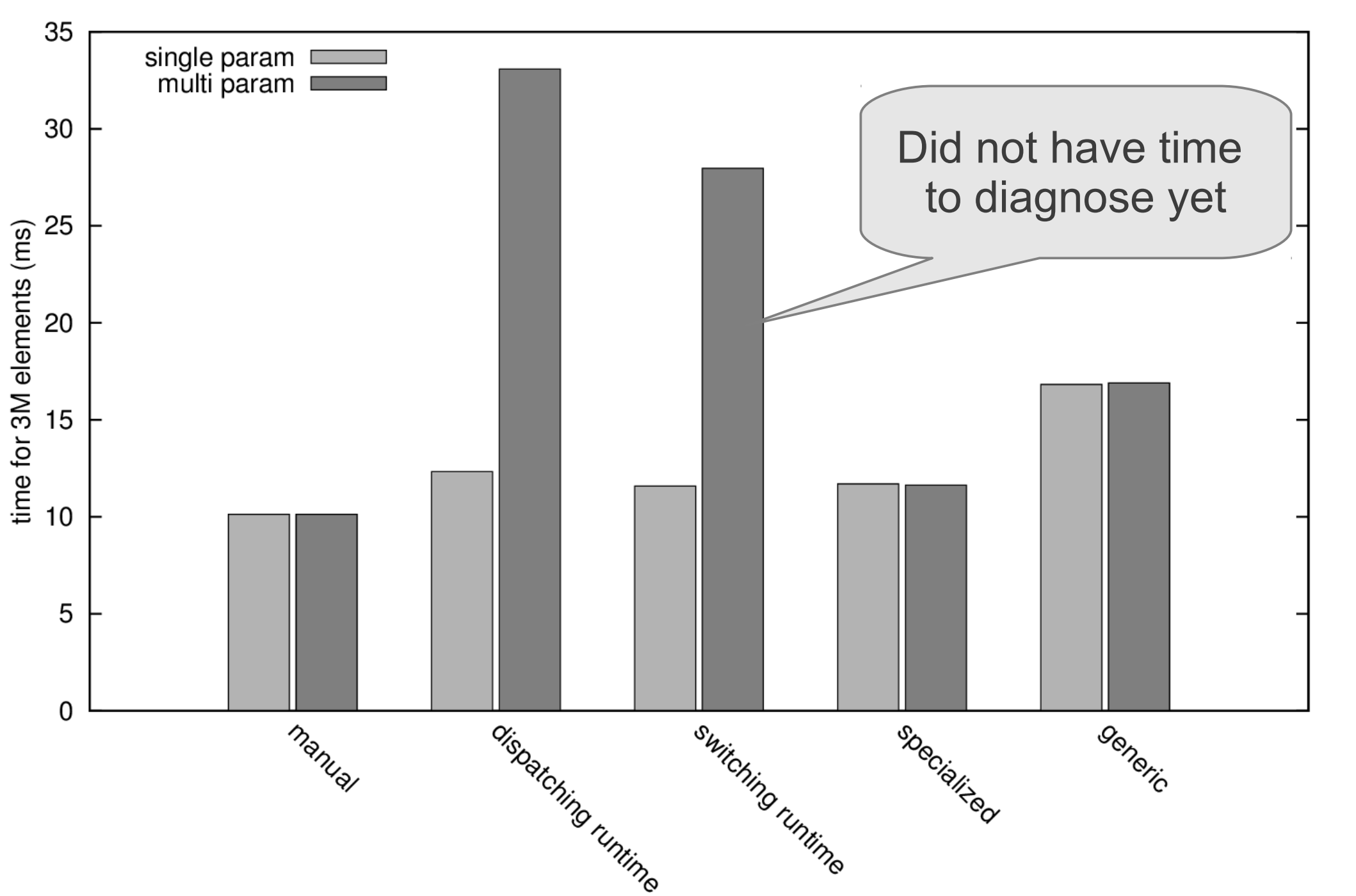


`ArrayBuffer.contains()`

All benchmarks

- `new LinkedList(...)`
  - new operator performance
- null tail ends a list

```
class LinkedList[T](val head: T, val tail: LinkedList[T])
```

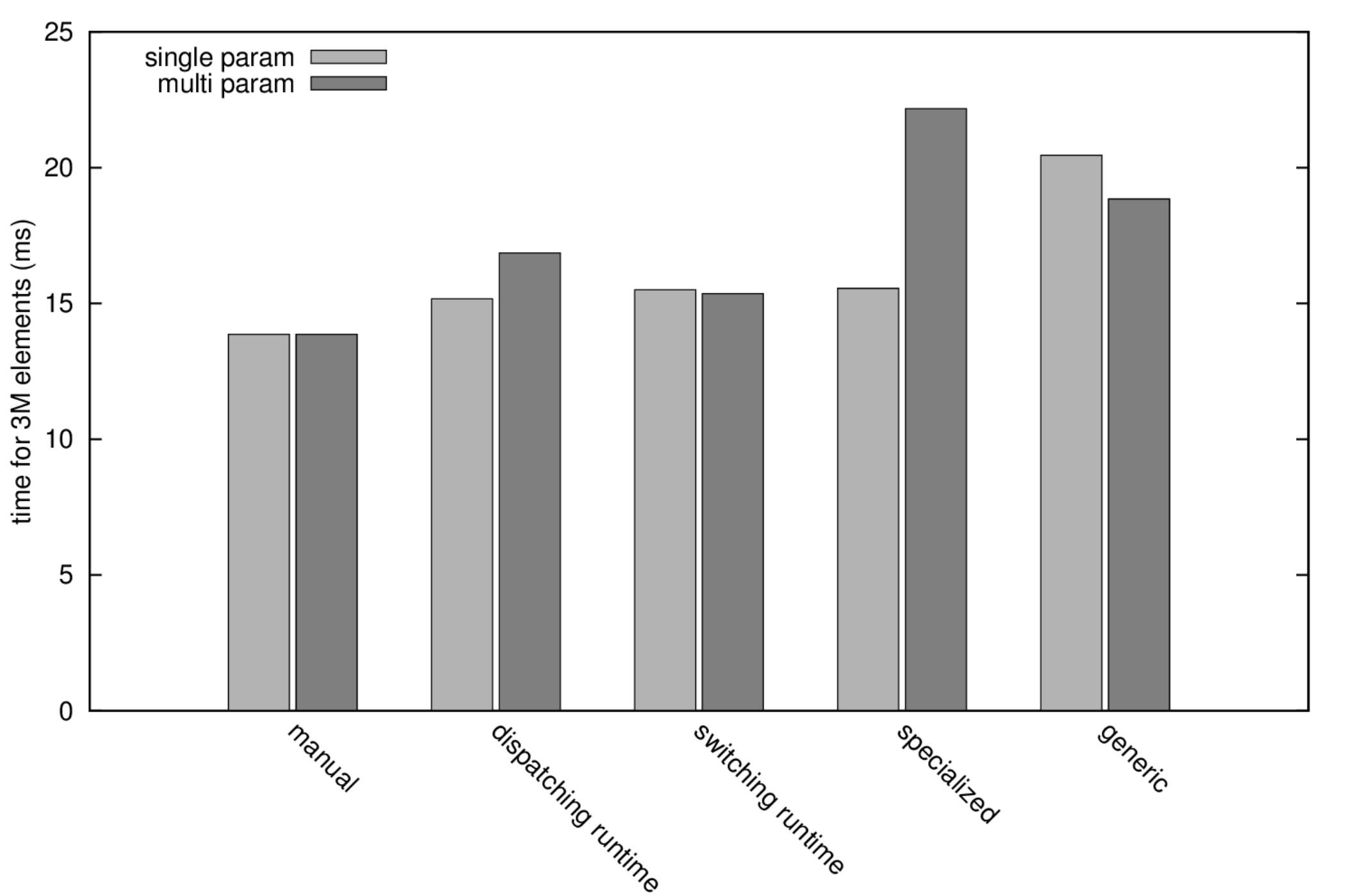



```
new LinkedList(...)
```

All benchmarks

- `LinkedList.hashCode()`
  - hashCode performance
  - traversal performance

```
  def contains(e: T): Boolean = {
    @annotation.tailrec def containsTail(list: List[T]):
Boolean =
      if (list.head == e)
        true
      else if (list.tail == null)
        false
      else
        containsTail(list.tail)

    containsTail(this)
  }
```

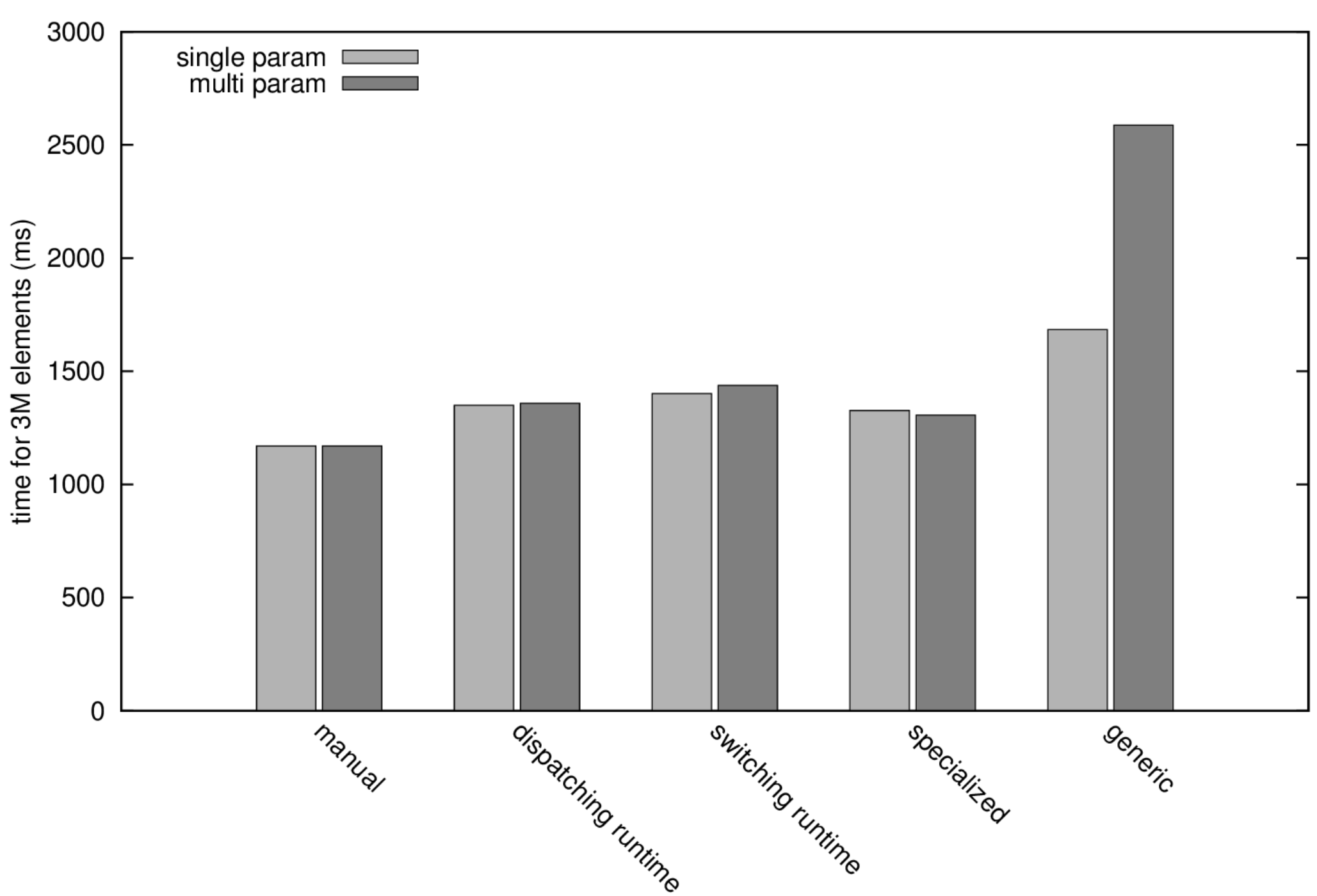


`LinkedList.hashCode()`

All benchmarks

- `LinkedList.contains()`
  - == operator performance
  - traversing performance

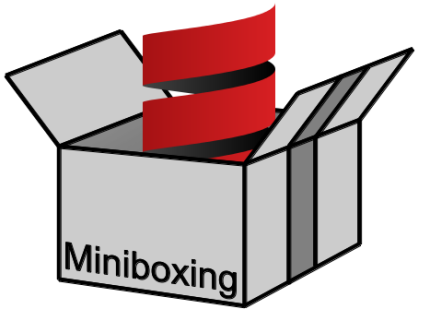
```
  def contains(e: T): Boolean = {
    @annotation.tailrec def containsTail(list: List[T]):
Boolean =
      if (list.head == e)
        true
      else if (list.tail == null)
        false
      else
        containsTail(list.tail)

    containsTail(this)
  }
```



`LinkedList.contains()`

Contributions

- Miniboxing itself, in an open-world assumption
- Exploration of the implementation space
  - Dispatchers
  - Runtime specialization
- Solution that works in practice



Thank you!

github.com/miniboxing

Vlad Ureche
vlad.ureche@epfl.ch