Fast 64-Bit Integers for the JavaScript Platform

John Q. Anon[1] and Joan R. Anon[2]

1 **Dummy University Computing Laboratory, Address/City, Country**
`open@dummyuniversity.org`
2 **Department of Informatics, Dummy College, Address/City, Country**
`access@dummycollege.org`

Abstract

The JavaScript platform is used as a target for many languages, and a number of them, such as Java and Scala, feature fixed-width integer data types. Although JavaScript as a language only features double-precision floating point numbers, there exists an efficient encoding of `Int`s, i.e., signed 32-bit integers, pioneered by asm.js. However, there is no such encoding for `Long`s, i.e., 64-bit integers. Languages compiling to JavaScript implement them in user-land, with disastrous run-time performance, or sacrifice correctness by compiling them to `Double`s. We present an efficient implementation of `Long`s for statically typed languages compiling to JavaScript, based on a combination of a) an efficient user-land implementation and b) an optimizing compiler capable of scalar replacement and simplification of operations on `Int`s. The implementation outperforms that of GWT, TeaVM and Kotlin by 3.6x to 60x and is only 3–5x slower than `Int`s. Using Predicate-Qualified Types, we provide mechanical proofs of correctness for comparators, bitwise operators and all arithmetic operators except `*`, `/`, and `%`. Our implementation shows that we no longer have to compromise between correctness and performance for 64-bit integers in JavaScript.

Keywords and phrases Performance, JavaScript, 64-Bit Integers, Predicate-Qualified Types

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

The list of languages compiling to JavaScript is growing every day. Some of them, such as Java and Scala, feature fixed-width integer data types, including `Int`s and `Long`s, which are 32 and 64 bits wide, respectively. However, the JavaScript language only features `Double`s, i.e., 64-bit floating-point numbers, which means that compilers need to encode the semantics of integer data types. Since the advent of asm.js [1], there exists a well-known, efficient encoding of signed 32-bit integers with wrapping operations. For example, the operation `a + b` for `Int`s can be encoded as `(a + b) | 0`. The encoding, which we describe in Section 2.1, has become so widespread that JavaScript VMs optimize it away.

There is no such encoding for 64-bit integers, however. The state-of-the-art technique consists in storing `Long`s as instances of a class provided by the runtime of the language. Each primitive operation is then implemented in a method and allocates a new instance for the result. For example, a possible implementation of bitwise `|` in ECMAScript 2015 would be the following:

```
class RuntimeLong {
  constructor(lo, hi) {
    this.lo = lo;
    this.hi = hi;
  }
```

```
  or(b) {
    return new RuntimeLong(this.lo | b.lo, this.hi | b.hi);
  }
}
```

Even though a bitwise `|` is easy to implement with this representation, the same cannot be said of most other operations, including simple arithmetic operations such as addition. Moreover, new instances of `RuntimeLong` are allocated for every primitive operation. Note that textbook algorithms for multi-precision arithmetics, such as described in [9], cannot be used, since they assume that single-precision instructions provide certain "services" to multi-precision arithmetics (e.g, `adc`–add with carry–can reuse the carry of a previous addition).

Such encodings cause significant performance issues on JavaScript engines [12, 20]. `Long` operations are typically one to two orders of magnitude slower than their `Int` counterparts, as we show in Section 4. These prohibitive numbers have even led some languages to entirely disregard correctness, and compile their `Long`s as simple `Double`s, which only provide 53 out of the 64 bits of precision for integer values. This includes languages specifically designed to cross-compile to JavaScript, such as Ceylon [15]. This is a drastic trade-off to make, as primitive values do not behave consistently across platforms, potentially leading to bugs that are difficult to track.

The Scala.js language and compiler have always chosen semantics first, leading to a slow implementation of `Long`s. They have been responsible for most—if not all—major performance issues in Scala.js applications, where users reported a surprisingly inefficient behavior (compared to the JVM version, or to subjective expectations, for example). Anecdotally, rewriting the implementation of `java.util.Random` from the specified algorithms using `Long`s to a specialized, hand-optimized one using `Int`s and `Double`s (though observably equivalent) turned a user's program from being unusably slow to being very responsive. The implementations of `BigInteger` and `BigDecimal`, which use `Long`s under the hood, have also been repeatedly reported to be excessively slow. Those issues were strong motivation for providing efficient `Long`s in Scala.js, which we address in this paper.

Contributions

In this paper, we present an efficient implementation of `Long`s for statically typed languages compiling to JavaScript. Our evaluation shows that the resulting `Long` operations are only 3–5x slower than their `Int` counterparts. We achieve this in several steps:

First, we survey in Section 2 the existing implementations of `Long`s in GWT [6], TeaVM [18] and Kotlin [10] (the only three open-source ahead-of-time compilers to JavaScript with correct `Long`s). Through a set of micro benchmarks, we identify what is the best implementation for each operation, and we combine those into a best-of-breed implementation of `RuntimeLong` for Scala.js.

Second, using stepwise refinements, we improve upon this implementation in Section 3, yielding measurable improvements on virtually all operations, including an order of magnitude improvement on division, modulo and `toString`. The implementation obtained at this point can be readily reused by GWT, TeaVM, Kotlin and Doppio.

Thirdly, in Section 4 we demonstrate how one can achieve even better performance when using an optimizing compiler along with a statically typed language. We exploit the fact that the implementations of all operations except `/`, `%` and `toString` (which we do not improve any further) are very compact. Our implementation of `RuntimeLong` greatly benefits from standard optimization techniques, i.e., inlining, scalar replacement (with a twist) and integer

rewrite rules. This brings an order of magnitude improvement to most operations as well as to an implementation of SHA-512.

We evaluate our implementation against that of GWT, TeaVM and Kotlin. Results show speedups from 3.6x to 60x on a SHA-512 benchmark, and 3x to 20x on individual operations. Furthermore, we demonstrate that, barring division and remainder, operations on `Long`s are only 3x slower than those on `Int`s on Chrome, and 5x slower on Firefox. The latter result has an important consequence: it is now possible for statically typed languages compiling to JavaScript to have both fast and correct `Long`s, without having to sacrifice one property to the other.

Finally, in Section 5, we mechanically verify the validity of user-land operations using Predicate-Qualified Types: comparisons, bitwise operators, as well as all arithmetic operators except `*`, `/` and `%`.

2 Survey of existing implementations

We begin our quest for fast 64-bit integers on JavaScript with a survey of the existing implementations. To the best of our knowledge, there are four open-source compilers providing correct `Long`s besides that of Scala.js: those of Doppio [20], GWT [6], TeaVM [18] and Kotlin [10]. Doppio and Kotlin share a common implementation derived from the Google Closure Library [5]. Since Doppio stands apart as a JVM implementation in JavaScript rather than an ahead-of-time compiler to JavaScript, we discard it in favor of Kotlin. We therefore focus our analysis on GWT, TeaVM and Kotlin, whose `Long` implementations can be found at [4, 19, 11].

All benchmarks and discussions of TeaVM include a simple performance "fix" that we have applied to the multiplication algorithm. The original algorithm normalizes inputs to positive values, which is superfluous, as multiplication in 2's complement produces the same results whether the bit pattern is interpreted as signed or unsigned. Removing the normalization brings a 22% performance improvement to the TeaVM multiplication.

2.1 Background: JavaScript and its numbers

Before looking at the various implementations, it is important to understand what JavaScript gives us. Specification-wise, JavaScript only has 64-bit double precision floating point numbers, i.e., `Double`s. All arithmetic operations (`+ - * / %`) have `Double` semantics. However, JavaScript also provides 32-bit integer bitwise operators. Their operands are technically `Double`s, but before applying the operation, the operands are coerced to signed 32-bit integers (wrapping, rather than capping, their values modulo 2^{32}). The result is then converted back to a `Double`. The available bitwise operations are: and `&`, or `|`, xor `^`, shift left `<<`, arithmetic shift right `>>` and logical shift right `>>>`[1].

We can use these operators, and in particular `|`, to concisely implement signed 32-bit *arithmetic* operations. The fact that `x | 0` forces wrapping `x` around signed 32 bits allows us to implement `+ - / %` easily as `(a+b)|0`, `(a-b)|0`, `(a/b)|0` and `(a%b)|0`, respectively. Signed 32-bit multiplication is harder. `(a*b)|0` does not work because the intermediate result `a*b` might already loose precision in the 11 least significant bits. Fortunately, ECMAScript

[1] Technically, `>>>` works on *unsigned* 32-bit integers, but we will ignore this detail as we can always "fix" it by using `(a >>> b)|0` instead of `a >>> b`.

2015 provides signed 32-bit multiplication under the builtin `Math.imul(a, b)`, and there exists a polyfill for older versions of JavaScript.

The above encoding of signed 32-bit arithmetic operations was made popular by `asm.js`. Since then, JavaScript VMs have started optimizing the `(a+b)|0` idiom so that they essentially amount to "primitive" signed 32-bit operations, paradoxically making them slightly faster than the corresponding double operations such as `a+b`. This allows to efficiently implement `Int`s and their operations in JavaScript.

For all intents and purposes, JavaScript therefore gives us *two* efficient numeric types: `Double`s and `Int`s. We can build on them to implement `Long`s. It is also worth mentioning that `Double`s, by their nature, provide accurate integer values up to 2^{53} (in absolute value). They have an effective precision of 53 bits in addition to a sign bit.

2.2 Overview of the design choices

We briefly survey the design decisions made in three prior implementations.

2.2.1 GWT

Unlike the other implementations, which use 2 `Int`s, GWT represents its `Long`s using three `Int`s, `l`, `m` and `h`, storing bits 0–21, 22–43 and 44–63, respectively. The motivation for this design choice is that components can be manipulated without incurring overflow in intermediate results. For example, the implementation of addition (in Java) is:

```
public static LongEmul add(LongEmul a, LongEmul b) {
  int sum0 = a.l + b.l;
  int sum1 = a.m + b.m + (sum0 >> BITS);
  int sum2 = a.h + b.h + (sum1 >> BITS);
  return create(sum0 & MASK, sum1 & MASK, sum2 & MASK_2);
}
```

The carries are part of the intermediate results `sum0` and `sum1`, and can easily be propagated by shifting them. A direct benefit is that `add` is branchless, at least at the JavaScript source code level.

This design decision must be placed in the context in which GWT was first developed. The implementation of `Long`s in GWT is ancient, and predates the `asm.js` encoding of 32-bit integers. As a matter of fact, GWT does not correctly implement `Int`s, but rather as plain JavaScript numbers, which basically means `Double`s. Therefore, the implementation of `Long`s cannot rely on the 32-bit wrapping semantics of primitive `Int`s. Besides additive operators, however, the encoding with 3 numbers is counterproductive. For example, bitwise and comparison operators need to handle 3 pairs of operands rather than 2. In a modern world with the `asm.js` encoding for `Int`s, the GWT design seems outdated.

2.2.2 TeaVM and Kotlin

Unlike GWT, which implements its `LongEmul` class in Java, TeaVM and Kotlin manually implement their `Long` class in JavaScript. They use a more natural encoding of `Long`s, using a pair of `Int`s each holding 32 bits. This design tries to better leverage the optimizations of contemporary JavaScript engines, which are good at dealing with 32-bit integers. For example, TeaVM exploits the `asm.js` encoding to efficiently implement `inc` (+1) on `Long`s.

```
function Long_inc(a) {
  var lo = (a.lo + 1) | 0;
  var hi = a.hi;
```

Name	Description
Nop	Baseline for micro-benchmarks, without any specific `Long` operation
Xor	`long64 ^ long64`
Add	`long64 + long64`
Mul	`long64 * long64`
Div N/M	`longN / longM`
Div N/pow2	`longN / longPow2`
ToString N	`Long.toString(longN).length()`

Table 1 List of the micro-benchmarks

```
  if (lo === 0)
    hi = (hi + 1) | 0;
  return new Long(lo, hi);
}
```

Note the two occurrences of `(x+1)|0`, which implement wrapping 32-bit additions.

This leads to more complicated implementations of additive operators, however, as operands need to be decomposed in chunks of 16 bits to prevent overflows in intermediate operations to lose carries. To compensate, TeaVM chooses to use fast paths when operands actually fit in 32 bits or 53 bits, leveraging efficient primitive operations on integers and doubles. For example, the addition in TeaVM is as follows:

```
function Long_add(a, b) {
  // Fast paths
  if (a.hi === (a.lo >> 31) && b.hi === (b.lo >> 31))
    return Long_fromNumber(a.lo + b.lo);
  if (Math.abs(a.hi) < Long_MAX_NORMAL && Math.abs(b.hi) < Long_MAX_NORMAL)
    return Long_fromNumber(Long_toNumber(a) + Long_toNumber(b));

  // Slow path (omitted)
  return new Long(...);
}
```

Interestingly, Kotlin has a similar-looking implementation of addition, but does not include fast-paths for small values.

2.3 Evaluation of existing implementations

In order to determine which implementation is best for each operation, we performed a set of micro-benchmarks on representative operations. Each operation is performed on a dataset of 100x100 operands, for a total of 10,000 iterations. The datasets have been randomly generated using a uniform distribution of values fitting in a certain number of bits in 2's complement representation.

Table 1 lists all our micro-benchmarks. `longN` represents a randomly generated N-bit value, sign-extended to a `Long`. N and M can be 32, 53 or 64. Some implementations take shortcuts when their operands fit in 32 or 53 bits. M can also be 8, providing test cases where the divisor is significantly smaller than the dividend, which typically trigger worst-case behaviors of the division algorithms. Similarly, `longPow2` is a dataset of randomly generated `Long` values `v = 1L << n` for `n` $\in [0, 63]$, for which some division algorithms use `>>>` as an optimization. In all cases, values are retrieved dynamically from an array, which means optimizers cannot perform ahead-of-time simplifications. The results of all operations are

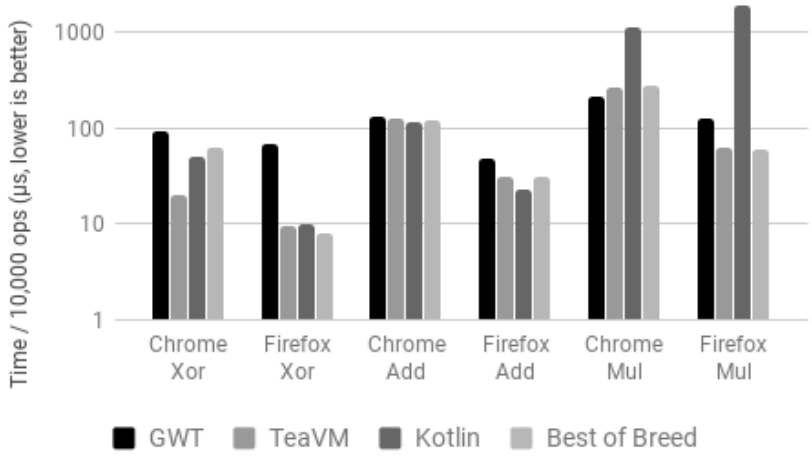

Figure 1 Micro-benchmarks of existing implementations for Xor, Addition and Multiplication

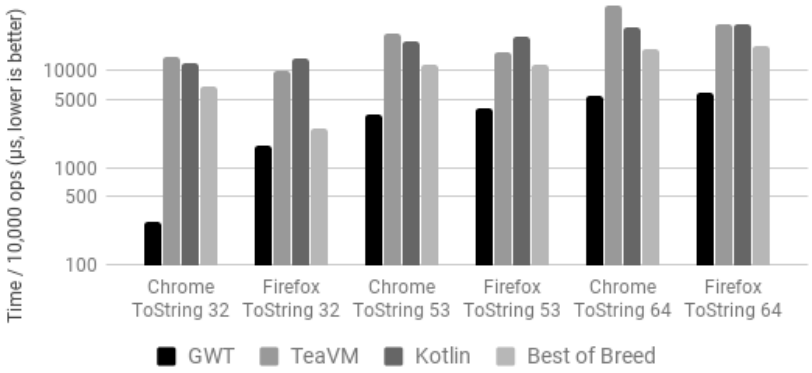

Figure 2 Micro-benchmarks of existing implementations for `toString`

xor'ed together, and the final result is checked for correctness to prevent optimizers from eliminating the computations as dead code.

We run the benchmarks on Linux 4.4.0-93 on an Intel® Core™ i7-4790 CPU clocked at 3.60GHz, with two browsers: Chrome 57.0.2987.110 and Firefox 55.0.2. The versions of TeaVM and Kotlin are 0.5.1 and 1.1.4-3, respectively. We use the "ADVANCED" optimization level of TeaVM rather than "FULL", because the latter yielded significantly worse results on Firefox. For GWT, we considered versions 2.7.0 and 2.8.x, but chose the former because it was overall faster for those benchmarks. Figures 1, 2 and 3 show the results. They include a fourth implementation, Scala.js "best of breed", which we discuss in Section 2.4. Note that all benchmark charts use a logarithmic scale, due to the large differences between implementations. Since the SEMs (Standard Error of the Mean) of all our measurements are at least 2 orders of magnitude smaller than their respective means, we do not clutter the graphs with error bars.

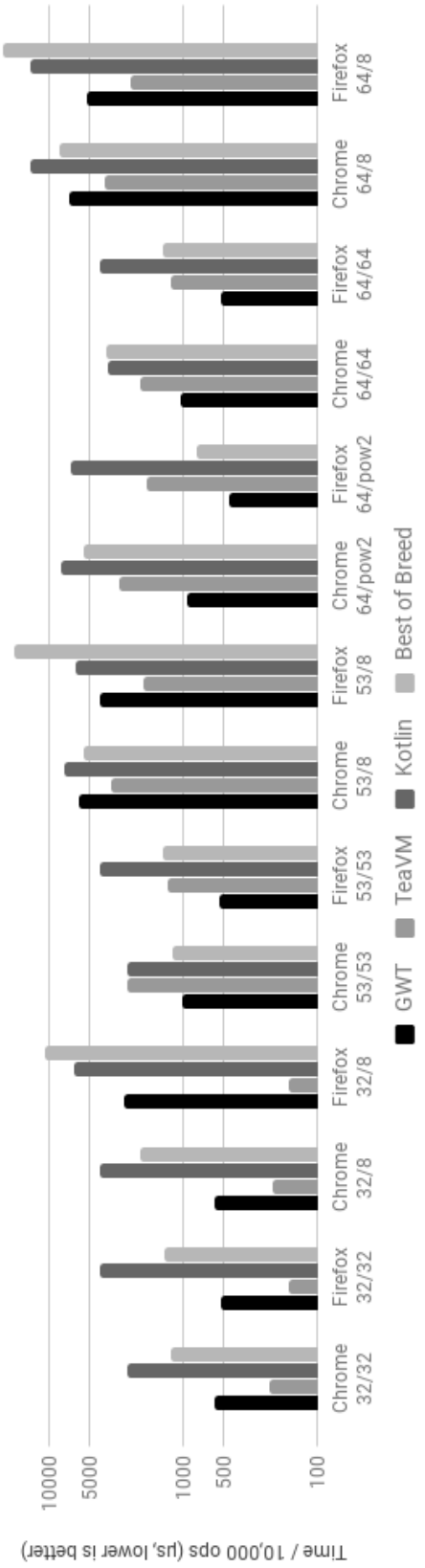

Figure 3 Micro-benchmarks of existing implementations for division

2.3.1 Xor

TeaVM is by far the best implementation of `xor` on Chrome, and on par with Kotlin on Firefox. GWT is unsurprisingly the slowest, having to deal with 3 fields rather than 2. In general, GWT performs poorly on all bitwise operations, because its data representation does not provide any shortcuts in these cases.

2.3.2 Addition

We would have expected GWT to shine in the addition case, given that its design allows for a branchless and compact implementation of addition. However, it happens to be the slowest. The fact that Kotlin wins over TeaVM on this benchmark is due to its lack of fast-path for small values. Indeed, we use random 64-bit operands, most of which should not fit in a double. In that sense, this benchmark is not fair to TeaVM. If we remove the fast-paths from TeaVM's algorithm, the two implementations become indistinguishable.

2.3.3 Multiplication

GWT exhibits the best implementation of multiplication on Chrome, with TeaVM a close second. The latter is however twice as fast as GWT on Firefox.

2.3.4 Division

TeaVM and GWT share the podium for division. GWT gives better results when the divisor has the same magnitude as the dividend, whereas TeaVM shines with small divisors. GWT also has a special optimization when the divisor is an exact power of 2, which gives it excellent results in those cases.

2.3.5 Conversion to string

GWT's `toString` outshines the other implementations. This is easily explained by inspection of their algorithm. They divide by 10^9 at each step, leveraging JavaScript's primitive conversion of `Int` to string, whereas TeaVM and Kotlin use steps of 10^6 and 10, respectively.

2.4 Best-of-breed implementation for Scala.js

Using the knowledge gathered in the previous section, we can write an initial implementation of `RuntimeLong` for Scala.js, with the best implementation of each operation. We choose the data representation with two integers, since TeaVM won on most non-division operations, and GWT's division algorithm does not directly depend on the data representation. The operations are picked from GWT and TeaVM:

- We obviously use GWT's implementation of `toString`.
- For division, there is no best implementation. We choose GWT's on a leap of faith that we will be able to improve its behavior with small divisors. TeaVM's implementation relies on a separate implementation of unsigned 80-bit integers, which we would like to avoid for code size reasons.
- For the other operations, we use TeaVM's implementations, without the fast-paths for small values. We will make the fast-paths irrelevant in Section 3.2 anyway.

The graphs of the previous section compare this implementation with those of GWT, TeaVM and Kotlin. We would expect it to be better than all the alternatives, but this is not the case. For xor, addition and multiplication, it exhibits similar performance characteristics to the best existing implementations, with a notable exception for xor on Chrome. Our (unproved) hypothesis for our implementation being slower is that Scala.js uses instance methods (e.g., `a.xor(b)`) rather than top-level functions (e.g., `Long_xor(a, b)`), and that JavaScript JITs need more expensive deoptimization guards for instance methods. We could improve this in Scala.js, but the issue will become moot in Section 4, once we have our own optimizer inline those functions. Another possibility is that the Scala.js compiler is overall of lesser quality than the others, although our biased mind is reluctant to consider it.

Division is overall 3x slower than GWT, our chosen reference. This is easily explained because we took some shortcuts when porting GWT's division algorithm. More specifically, we kept `RuntimeLong` as an immutable class, forcing us to allocate more instances than GWT, which mutates intermediate values in-place. Consequently, `toString`, which builds on division, is also overall 3x slower than GWT. The `toString 32` benchmark on Chrome exhibits an abnormal 25x slowdown which we did not manage to explain.

3 Improvements to `RuntimeLong`

Now that we have a baseline implementation in Scala.js, built from the best-in-class existing implementations, we can start optimizing it. These optimizations are in general independent of the source language and its compiler, which means that they can be readily ported to GWT, TeaVM and Kotlin to improve the performance of their `Long`s. There are two general themes that drive our optimizations:

- Reducing the number of branches, which is a typical trick in micro-optimization both for performance and code size, and
- Exploiting the properties of the 2's complement representation, in particular the relationships between the signed and unsigned interpretations of bit patterns.

Figures 4, 5 and 6 show the results of benchmarks for the various refinements developed in this paper. The improved algorithms derived in this section are shown as "User-land". "With scalar replacement" and "With integer simplifications" refer to further optimizations that we discuss in Section 4.

3.1 Removing branches: shifts

As a simple illustration of the theme of removing branches, we show the transformation we apply on <<, i.e., shift left. It is worth pointing out that in Java, Scala and JavaScript, shift operators mask their right-hand-side to the 5 least significant bits for `Int`s (`rhs & 31`) and the 6 least significant bits for `Long`s (`rhs & 63`). The base implementation coming from TeaVM is as follows:

```
def <<(n: Int): RuntimeLong = {
  val n1 = n & 63
  if (n1 == 0)
    this
  else if (n1 < 32)
    new RuntimeLong(lo << n1, (lo >>> (32 - n1)) | (hi << n1))
  else if (n1 == 32)
    new RuntimeLong(0, lo)
```

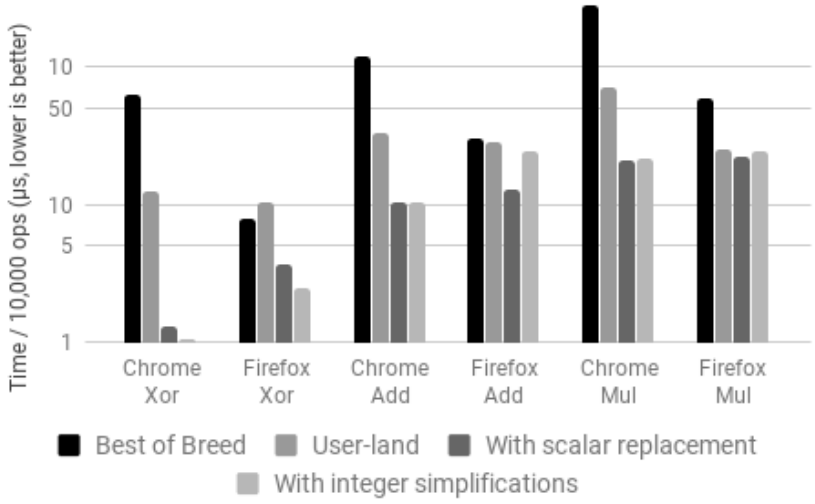

Figure 4 Micro-benchmarks of our improvements to Xor, Addition and Multiplication

```
  else
    new RuntimeLong(0, lo << (n1 - 32))
}
```

which contains 3 branches on the longest path. Using clever but otherwise mundane rewritings (which are detailed as comments in the source code, available in the supplementary material), we simplify it down to

```
def <<(n: Int): RuntimeLong = {
  if ((n & 32) == 0)
    new RuntimeLong(lo << n, (lo >>> 1 >>> (31-n)) | (hi << n))
  else
    new RuntimeLong(0, lo << n)
}
```

which has only one branch (and does not need an explicit `n & 63`). The skeptical reader will wonder whether the resulting operation is still correct. In Section 5, we mechanically verify the correctness of the optimized implementation of <<.

3.2 Addition

Recall from Section 2.2 that GWT had a significantly shorter implementation, because it could let the carry flow across the intermediate additions. The slow path for addition coming from TeaVM is pretty verbose. Textbook instruction sequences for multi-precision additions use the very convenient `adc` instruction (add with carry). What if we could do the same? An optimal algorithm would look like

```
def +(b: RuntimeLong): RuntimeLong = {
  val lo = this.lo + b.lo
  val hi = adc(this.hi, b.hi)
  new RuntimeLong(lo, hi)
}
```

Essentially, `adc` amounts to:

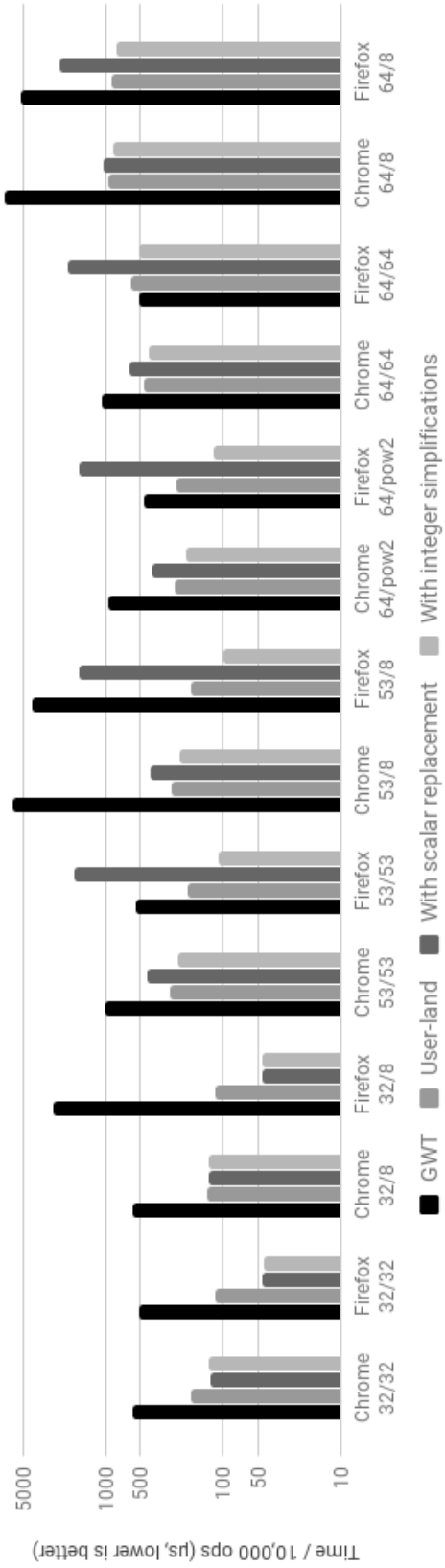

Figure 5 Micro-benchmarks of our improvements to division

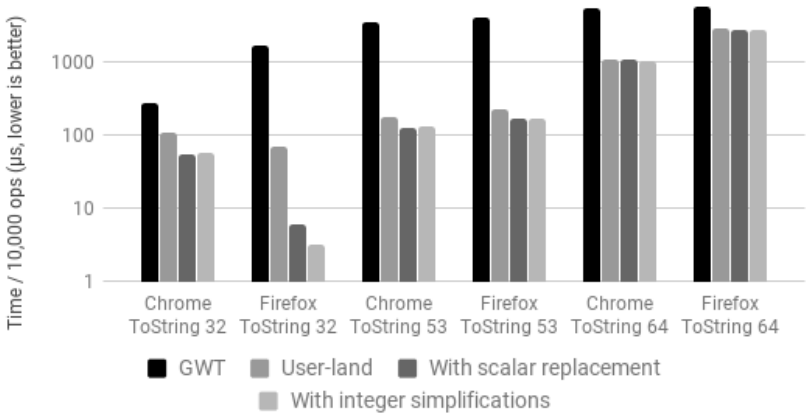

Figure 6 Micro-benchmarks of our improvements to `toString`

```
val hi =
  if (overflow_occurred_in_lo_+)
    this.hi + b.hi + 1
  else
    this.hi + b.hi
```

We can detect whether an overflow occurred by testing if `lo < this.lo`, interpreted as an *unsigned* comparison. This is easily done by flipping the sign bit of the operands to `<` :

```
def +(b: RuntimeLong): RuntimeLong = {
  val lo = this.lo + b.lo
  val hi =
    if (lo ^ 0x80000000 < this.lo ^ 0x80000000)
      this.hi + b.hi + 1
    else
      this.hi + b.hi
  new RuntimeLong(lo, hi)
}
```

The resulting implementation contains less operations than the fast-path of TeaVM, making the latter irrelevant.

As shown in Figure 4, this new implementation yields a 5x improvement to the addition on Chrome, bringing it on par with Firefox (for which it seems to have a slightly negative effect).

3.3 Multiplication

Both TeaVM and Kotlin have a multiplication algorithm that decomposes the product into 16-bit components. They do so in order to avoid the loss of bits due to wrapping around 2^{32}. Both algorithms require a total of 10 primitive multiplications, and a fair amount of additions and bitwise operations.

We can improve on this by decomposing some parts of the product into 32-bit components instead. We do lose some bits in the process, but this allows to accumulate the results of three 16-bit multiplications with a single 32-bit multiplication, if we can make use of the result. The derivation of our multiplication algorithm is unfortunately too long to fit here—it spans

200 lines of comments in the source code of `RuntimeLong`, available in the supplementary material—but we include the final result, which contains only 6 primitive multiplications:

```
def *(b: RuntimeLong): RuntimeLong = {
  val alo = this.lo
  val blo = b.lo
  val a0 = alo & 0xffff
  val a1 = alo >>> 16
  val b0 = blo & 0xffff
  val b1 = blo >>> 16

  val a0b0 = a0 * b0
  val a1b0 = a1 * b0
  val a0b1 = a0 * b1
  val lo = a0b0 + ((a1b0 + a0b1) << 16)
  val c1part = (a0b0 >>> 16) + a0b1
  val hi = {
    alo*b.hi + a.hi*blo + a1 * b1 +
    (c1part >>> 16) + (((c1part & 0xffff) + a1b0) >>> 16)
  }
  new RuntimeLong(lo, hi)
}
```

To the best of our knowledge, this algorithm and its derivation are novel. Figure 4 shows that it brings a 4x speedup on Chrome and 2.3x on Firefox.

3.4 Division and remainder

The core of the division algorithm of GWT is the good old restoring division algorithm [21], adapted to compare before subtraction. This algorithm only works for unsigned divisions, which is why GWT first needs to normalize the operands to be positive. Extra care must be taken for operands equal to `Long.MinValue`, which does not have an opposite in 2's complement.

We can however improve on GWT's algorithm with two major changes: first, we get rid of the `MinValue` special case by interpreting the 64 bits as unsigned after normalization; second, we short-circuit the loop as soon as the partial remainder fits in 53 bits.

For the first part, note that the bit pattern of `-MinValue`, interpreted as unsigned, is the correct mathematical value for the opposite of `MinValue`, i.e., 2^{63}. This means that, if the core division loop treats the 64 bits as unsigned, there is no need for any special case for `MinValue`, which streamlines the implementation and reduces code size. Doing so requires only one adaptation to the main loop: it needs to perform an *unsigned* `>=` rather than a signed one. Similarly to what we did in Section 3.2, we can flip the sign bit and perform a signed comparison to achieve that result.

For the second part, we can easily by-pass the loop entirely when the dividend fits in 32 or 53 bits, as we can reuse primitive int or double division. TeaVM does this, which reduces the cases where the loop is needed to large values of the dividend. The major improvement that we contribute is to take advantage of the primitive double division even for dividends larger than 2^{53}. Let us look at the core loop and its invariant:

```
def divModHelper(a: RuntimeLong, b: RuntimeLong,
    isDivide: Boolean): RuntimeLong = {
  var shift = numberOfLeadingZeros(b) - numberOfLeadingZeros(a)
  var bShift = b << shift
  var r = a
```

```
  var q = new RuntimeLong(0, 0)

  /* Invariants:
   *   q >= 0
   *   If shift >= 0:
   *     bShift == b << shift == b * 2^shift
   *     0 <= r < 2 * bShift
   *   Else:
   *     0 <= r < b
   *   q * b + r == a
   */
  while (shift >= 0 && r != 0) {
    if (unsigned_>=(r, bShift)) {
      r -= bShift
      q |= (1L << shift)
    }
    shift -= 1
    bShift >>>= 1
  }
  if (isDivide) q
  else r
}
```

The idea of this loop is that it maintains the invariant $q \cdot b + r = a$. Eventually, we want q to hold the quotient `a/b`, and r to hold the remainder `a%b` (interpreted as unsigned integer operations). This is true if $0 \leq r < b$, which is indeed the case after the loop, because either $r = 0$, or `shift` < 0, which implies $0 \leq r < b$.

We can instead short-cut the loop as soon as $r < 2^{53}$ (what we call an "unsigned safe double", because we can safely convert it to a double without loss of precision). Note that this happens after at most 11 iterations of the loop instead of 64, a worthwhile improvement. Once $r < 2^{53}$, we can finish off the algorithm with a double operation:

```
while (shift >= 0 && !isUnsignedSafeDouble(r)) {
  ...
}
if (unsigned_>=(r, b)) {
  val rDbl = asUnsignedSafeDouble(r)
  val bDbl = asUnsignedSafeDouble(b)
  if (isDivide)
    q + fromUnsignedSafeDouble(rDbl / bDbl)
  else
    fromUnsignedSafeDouble(rDbl % bDbl)
} else {
  if (isDivide) q
  else r
}
```

If $r < b$, then we already have $0 \leq r < b$ and nothing needs to be done. Otherwise, since $r < 2^{53}$, so is b, and we can perform the division on doubles, computing $q' = r/b$ and $r' = r\%b$. Since we know that $q' \cdot b + r' = r$ (by definition of unsigned `/` and `%`), and from the invariant, we have that

$$q \cdot b + (q' \cdot b + r') = a$$

and by associativity of the addition, we have

$$(q + q') \cdot b + r' = a$$

Observe that this is the shape of the definition of quotient and remainder, with the quotient being $(q + q') = (q + r/b)$ and the remainder being $r' = r\%b$. This is what we return from the algorithm.

Finally, as a last optimization, we manually replace all the intermediate `RuntimeLong`s by their `Int` components, to reduce allocations and heap accesses.

Figure 5 compares the performance of our resulting division to GWT's original division (recall that our "best of breed" implementation was consistently 3x slower than GWT). Altogether, these optimizations bring up to an order of magnitude improvement.

3.5 Conversion to string

The last user-land implementation that deserves some remarks is the conversion to string in base 10. We have already mentioned that GWT processes the number in chunks of 10^9, leveraging the primitive conversion from `Int` to string. We can do even better by also leveraging the conversion of `Double` to string. Once we have normalized the input to a positive value a, if $a < 2^{53}$, we immediately short-cut to a double-to-string conversion. If $a \geq 2^{53}$, we perform *exactly one* long `divMod` operation by 10^9 to obtain $q = a/10^9$ and $r = a\%10^9$. Since we know that $2^{53} \geq a < 2^{64}$, we have that $2^{53}/10^9 \geq q \leq 2^{64}/10^9$, which implies that $0 < q < 2^{53}$. This means that q necessarily fits in a double, and r in an int. We can therefore delegate to two primitive conversions to string and a concatenation to conclude.

Figure 6 shows that this optimization brings a 5x improvement to the implementation of `toString` in the worst case, and up to 20x for the best case.

This concludes our tour of the various algorithms we have developed for the user-land implementation of `RuntimeLong`. Overall, improvements from 3x to 20x can be observed, compared to the best existing implementations. This improved implementation of `RuntimeLong` can be readily adopted by any language implementing `Long`s in JavaScript, in particular GWT, TeaVM and Kotlin.

4 Using an optimizing compiler

In the previous section, we have developed a user-land implementation of `Long`s targeted for JavaScript, which can be readily reused by other languages compiling to JavaScript. If we put some more restrictions on the language and its implementation, namely that it is statically typed and features an optimizing compiler, we can go further. To the best of our knowledge, Scala.js is the first implementation to apply an optimizing compiler to its user-land implementation of `Long`s. It would be possible for GWT, TeaVM and/or Kotlin to follow suit if they have a powerful enough optimizing compiler.

We apply optimizations in two phases, to show their relative benefits. In addition to the micro-benchmarks shown in Figures 4, 5 and 6, Figure 7 shows the results of a macro-benchmark: an implementation of SHA-512, straightforwardly ported from the C implementation in mbed TLS [13]. This hashing function makes heavy use of `Long`s, with a core loop mainly consisting of bitwise operations, shifts and additions. A macro-benchmark is better suited to highlight the improvements of this section, which have a bigger effect on sequences of operations in the same method. The figure also shows benchmarks for "SHA-512-Int", which is a clone of "SHA-512" where every `Long` has been summarily replaced by an `Int`. The resulting algorithm is obviously wrong—it does not compute a correct SHA-512 hash anymore—but serves as an indication of the overhead imposed by `Long` operations relative to the primitive `Int`s.

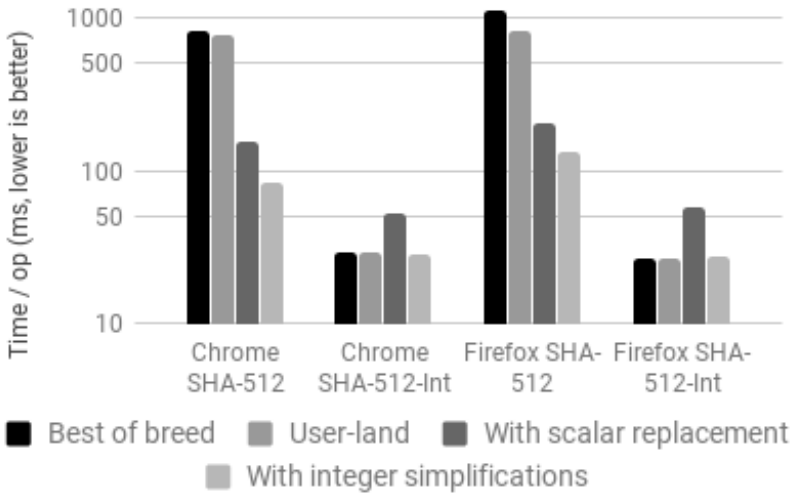

Figure 7 The SHA-512 macro-benchmark on the various refinements of Section 3 and 4

First, we apply scalar replacement (also known as object inlining or stack allocation) to all local `RuntimeLong` values. For scalar replacement to be useful, we also need to inline the methods of `RuntimeLong`. The implementations of `/`, `%` and `toString` are much too large to be inlined, but all the other operations can be considered for inlining. Scalar replacement is a major optimization, well-known for its dramatic effects on performance. Not only does it avoid allocations and heap accesses, it also puts less pressure on the garbage collector.

Although our implementation of scalar replacement is generic and applies to other classes besides `RuntimeLong`, the latter is special-cased in the optimizer for additional effect, adding a little twist to this otherwise standard optimization. In general, when a reference escapes to another method, an optimizer has to backtrack and cancel the scalar replacement of the escaping value. This is necessary to preserve mutability and object identity. However, we know that `RuntimeLong` is immutable, and that its identity is never observable (since it is only an artifact of compiling primitive `Long`s). This means that, when a `RuntimeLong` escapes, we can create a new instance on the spot rather than backtracking. Likewise, as soon as we fetch a `RuntimeLong` from an unknown scope and store it in a local variable, we force its stack allocation as two primitive `Int`s. Figure 7 shows that scalar replacement of `RuntimeLong` brings a 5x improvement on Chrome, and 4x on Firefox.

In a second step, we add rewrite rules to simplify operations on `Int`s. This includes rules as simple and generic as `x & 0 == 0`, up to rewrite rules designed specifically after the code of `RuntimeLong`, such as `(x & 0xffff) >>> 16 == 0` (a pattern we find when multiplying by a small constant). We would expect JavaScript VMs to be able to perform this kind of optimizations on their own, yet our evaluation shows that this still brings an additional 2x speedup on top of the scalar replacement, both on Chrome and Firefox.

If we compare SHA-512 versus SHA-512-Int, we can see that `Long`s are overall 4.9x slower than `Int`s on Firefox, and only 2.9x slower on Chrome. This a very important result, as it shows that Scala.js' `Long`s are not prohibitively slower than `Int`s anymore. We can therefore have our cake and eat it too: correct 64-bit semantics and decent performance.

To put things in perspective, Figure 8 also compares the final implementation in Scala.js–with all the optimizations–against GWT, TeaVM and Kotlin. Scala.js is around 60x faster

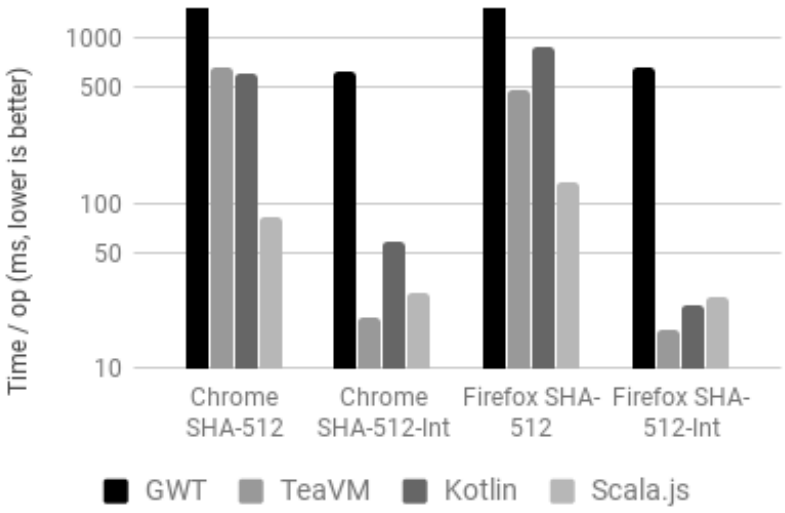

Figure 8 The SHA-512 macro-benchmark on existing implementations–GWT, TeaVM and Kotlin–and the final Scala.js implementation with optimizations

than GWT both on Chrome and Firefox. On Chrome, it is 8x faster than TeaVM, and 3.6x faster on Firefox.

5 Correctness

In Section 3 we outlined the various manual optimizations that shaped our user-land implementation. While these optimizations improve performance and code size, they also come at the cost of readability and result in considerably more complex code. To gain confidence that our implementation is correct, we mechanically verified most operations of `RuntimeLong` and provided a test suite for the rest.

5.1 Mechanical verification

Using Predicate-Qualified Types [17], a variant of refinement types currently under development for Scala, we proved the correctness of comparators, bitwise operators and arithmetic operators with the exception of `*`, `/` and `%`. That is to say, we verified that `RuntimeLong` faithfully implements the JVM's semantics of `Long`. To do so we extracted the relevant parts of the Scala.js implementation and added more precise type signatures which ensure the correspondence of semantics. The full specification—available in the supplementary material—was then run through a Scala compiler supporting Predicate-Qualified Types[2]. Below we give a brief overview of how we used Predicate-Qualified Types for this verification task.

The purpose of `RuntimeLong` is to provide a drop-in replacement for Scala's `Long` type, which in turn corresponds to the JVM's primitive `long` type. We first define a conversion from `RuntimeLong` to `Long`:

[2] The current version of the Predicate-Qualified Type system is available at `https://github.com/gsps/dotty/tree/liquidtyper`.

```
val toLong: Long =
  (this.lo.toLong & 0xffffffffL) | (this.hi.toLong << 32L)
```

Using this conversion, we can straightforwardly specify correct behavior of an operation in `RuntimeLong` by requiring it to return the same result as the corresponding operation on `Long` when given the same inputs. For instance, in the case of the left-shift operator seen in Section 3.1, we write:

```
def <<(n: Int): {v: RuntimeLong =>
    v.toLong == (this.toLong << n.toLong)} = {
  if ((n & 32) == 0)
    new RuntimeLong(lo << n, (lo >>> 1 >>> (31-n)) | (hi << n))
  else
    new RuntimeLong(0, lo << n)
}
```

Note that we extracted the body of `<<` from our user-land implementation and added only a qualified (result) type to the method:

```
{v: RuntimeLong =>
  v.toLong == (this.toLong << n.toLong)}
```

The *qualifier* `v.toLong == ...` constrains what values belong to the type and hence may be returned. In our specific case it prescribes that performing a left-shift by `n` bits using the implementation in `RuntimeLong` and converting the result to a `Long` (`v.toLong`) must give the equivalent result as converting the original `RuntimeLong` to `Long` first and performing the left-shift only then (`this.toLong << n.toLong`).

The compiler will then try to prove that the implementation adheres to this specification by inferring precise qualified types for the possible return values in both branches of `<<`. The resulting subtyping checks are then translated to validity queries in a logic over bitvectors and uninterpreted functions. The system discharges such proof obligations using SMT solvers such as CVC4 [2] or Z3 [3].

5.2 Limitations

Unfortunately we have not yet succeeded in verifying the implementation of all the operations of `RuntimeLong`. Two obstacles for a fully mechanical proof remain: currently the Predicate-Qualified Type system translates the multiplication and division methods to SMT queries that neither CVC4 nor Z3 can solve in a reasonable amount of time. We hope to remedy this in the future by providing adequate hints to the solver. Moreover, our fine-tuned implementation of division relies on conversions to and from `Doubles`, which further complicate the SMT queries.

5.3 Test suite

To mitigate these limitations we also built a test suite comprised of hundreds of randomly generated test cases. The expected results of operations are computed on a JVM, then stored as unit tests in the Scala.js test suite. This is essentially a manual and static variant of differential testing [14]. The test suite also includes manually constructed tests for known corner cases of `RuntimeLong`'s implementation (typically around overflow cases).

In addition to filling the gaps left by our verification tool on multiplication, division and `toString`, the end-to-end test suite gives confidence about the optimizer's rewritings described in Section 4.

Finally, the `Long`s are implicitly tested by the larger test suite of the entire Scala.js platform, which includes an implementation of big numbers relying on `Long`s.

6 Related Work

The art of compiling multi-precision arithmetic data types on top of single-precision ones is decades-old [9, 21]. However, all well-known algorithms assume that the primitive operations for single-precision arithmetics offer some support to build multi-precision operations on top of them. For example, they assume that addition sets a carry flag that can be recovered without branching (e.g., via `adc` on x86); or that multiplication outputs two single-precision registers for the lo and hi parts of the results. When compiling to a higher-level language such as JavaScript, the target does not offer such facilities, and we need to compute the carry ourselves.

One area we could explore, but as of yet have not, is the ahead-of-time optimization of divisions by a constant, using techniques ranging from using simple shifts to multiplication by the inverse [7]. The latter technique is not directly applicable, though, as it also assumes that multiplication gives access to the high half of the result. It might be worth implementing this multiplication in software, though, if it means avoiding the 10x slowdown of division over other arithmetic operations.

On a different note, compilation to JavaScript is commonplace, nowadays. When it comes to compiling down 64-bit integers from the source language, there have been two different strategies. One is to emulate them in software, in the tradition of GWT [6]. When GWT was first released, `Long`s were already known to be extremely slow, and GWT's documentation warns against using them too liberally. This strategy has been followed by surprisingly few implementations, e.g., TeaVM [18] and Kotlin [10]. On the more academic side, we find two other compilers doing so: Doppio by Vilk and Berger [20] and the Graal AOT JS compiler by Leopoldseder et al. [12]. Both papers mention `Long`s as a significant performance bottleneck. Emscripten [22] initially had incorrect 64-bit integers, but they later added compilation modes to choose between fast-but-incorrect and correct-but-slow. Remarkably, industrial and academic compilers alike seem to accept the terrible performance of `Long`s as a fact of life. In Scala.js, we were never satisfied with the performance of our `Long`s, and have always tried to improve them, eventually achieving the results presented in this paper. After we pointed out our implementation to Doppio's authors, they decided to look into adopting it[3].

The other and sadly more popular strategy adopted by implementers has been to disregard correctness. Instead, `Long`s are typically compiled down to JavaScript numbers, i.e., `Double`s, effectively dropping the precision down to 53 bits. Even languages specifically designed to cross-compile to JavaScript have been making that trade-off, such as Ceylon [15]. This has been justified by the fact that correctly compiling 64-bit integers yields code that is much too slow. With an implementation such as the one we demonstrate in this paper, this should not be a concern anymore, obviating the need for such a compromise on correctness.

A radical alternative would be to compile to WebAssembly [8] instead of JavaScript. This seems like a no-brainer, since WebAssembly features native 64-bit integers. However, it does not (yet) have any support for garbage-collected languages, nor for interoperating with arbitrary JavaScript values, which are features that Scala.js, along with GWT, TeaVM and Kotlin, all need.

[3] https://github.com/plasma-umass/doppio/issues/444

7 Conclusion

We have presented a fast implementation of 64-bit integers, aka `Long`s, for the JavaScript platform. The implementation relies on two key components: an efficient user-land implementation—derived as refinements of the state-of-the-art implementations—, and an optimizing compiler capable of inlining and scalar replacement at the least. We have implemented this strategy in the production-quality Scala.js compiler and optimizer [16]. The implementation is 3.6x to 60x faster than existing approaches, and, crucially, is only about 3–5x slower than native 32-bit integers.

To ensure correctness of our solution, we have mechanically verified parts of the user-land implementation using Predicate-Qualified Types for Scala. The mechanical proofs are complemented by a test suite based on differential testing.

Our fast implementation of `Long`s solves the years-old tension between correctness and performance for compilers targeting JavaScript. At last, we can have our cake and eat it too: the correct behavior of wrapping 64-bit integers, and the performance of the resulting code.

References

1 `http://asmjs.org/`, 2016.

2 Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, Lecture Notes in Computer Science, pages 171–177. Springer, July 2011. Snowbird, Utah.

3 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

4 Google. GWT Long implementation. `https://github.com/gwtproject/gwt/tree/2.7.0/dev/core/super/com/google/gwt/lang`, 2014.

5 Google. Closure Library. `https://developers.google.com/closure/library/`, 2015.

6 Google. Google Web Toolkit. `http://www.gwtproject.org/`, 2015.

7 Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 61–72, 1994.

8 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, 2017.

9 Randall Hyde. *The Art of Assembly Language*. No Starch Press, 2nd edition, 2010.

10 JetBrains. Kotlin. `https://kotlinlang.org/`, 2016.

11 JetBrains. Kotlin Long implementation. `https://github.com/JetBrains/kotlin/blob/v1.1.4-3/js/js.libraries/src/js/long.js`, 2017.

12 David Leopoldseder, Lukas Stadler, Christian Wimmer, and Hanspeter Mössenböck. Java-to-javascript translation via structured control flow reconstruction of compiler ir. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, pages 91–103, 2015.

13 mbedTLS. `https://github.com/ARMmbed/mbedtls`, 2015.

14 William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10:100–107, 1998.

15 Red Hat. Ceylon. `https://ceylon-lang.org/`, 2016.

16 http://www.scala-js.org/, 2015.

17 Georg Stefan Schmid and Viktor Kuncak. Smt-based checking of predicate-qualified types for scala. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, SCALA 2016, pages 31–40. ACM, 2016.

18 TeaVM. http://teavm.org/, 2016.

19 TeaVM Long implementation. https://github.com/konsoletyper/teavm/blob/0.5.1/core/src/main/resources/org/teavm/backend/javascript/runtime.js#L623, 2016.

20 John Vilk and Emery D. Berger. Doppio: Breaking the browser language barrier. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 508–518, 2014.

21 Shlomo Waser and Michael J. Flynn. *Introduction to Arithmetic for Digital Systems Designers*. Harcourt Brace College Publishers, 1995.

22 Alon Zakai. Emscripten: An llvm-to-javascript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 301–312, 2011.