Float Self-Tagging

OLIVIER MELANÇON, Université de Montréal, Canada
MANUEL SERRANO, Inria, France
MARC FEELEY, Université de Montréal, Canada

Dynamic and polymorphic languages attach information, such as types, to run time objects, and therefore adapt the memory layout of values to include space for this information. This makes it difficult to efficiently implement IEEE754 floating-point numbers as this format does not leave an easily accessible space to store type information. The three main floating-point number encodings in use today, tagged pointers, NaN-boxing, and NuN-boxing, have drawbacks. Tagged pointers entail a heap allocation of all float objects, and NaN/NuN-boxing puts additional run time costs on type checks and the handling of other objects.

This paper introduces self-tagging, a new approach to object tagging that uses an invertible bitwise transformation to map floating-point numbers to tagged values that contain the correct type information at the correct position in their bit pattern, superimposing both their value and type information in a single machine word. Such a transformation can only map a subset of all floats to correctly typed tagged values, hence self-tagging takes advantage of the non-uniform distribution of floating point numbers used in practice to avoid heap allocation of the most frequently encountered floats.

Variants of self-tagging were implemented in two distinct Scheme compilers and evaluated on four microarchitectures to assess their performance and compare them to tagged pointers, NaN-boxing, and NuN-boxing. Experiments demonstrate that, in practice, the approach eliminates heap allocation of nearly all floating-point numbers and provides good execution speed of float-intensive benchmarks in Scheme with a negligible performance impact on other benchmarks, making it an attractive alternative to tagged pointers, alongside NaN-boxing and NuN-boxing.

CCS Concepts: • **Software and its engineering** → **Source code generation**; **Polymorphism**.

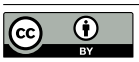
Additional Key Words and Phrases: Dynamic Languages, Polymorphic Languages, Floating-Point Numbers, NaN-Boxing, Compiler Optimization, Scheme

ACM Reference Format:
Olivier Melançon, Manuel Serrano, and Marc Feeley. 2025. Float Self-Tagging. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 330 (October 2025), 27 pages. https://doi.org/10.1145/3763108

1 Introduction

In dynamic and other polymorphic languages, efficiently supporting floating-point numbers (floats) remains a challenging issue. These languages require attaching type information to values at run time, including numeric types like integers and floats. This requirement conflicts with the IEEE754 standard [1] for encoding floats, which uses the entire 64-bit word to represent a double-precision float, leaving no space for additional type information. A straightforward but suboptimal solution is to represent floats as heap allocated objects. To reduce the overhead of heap allocating floats, several alternative representations have been proposed, each offering different trade-offs.

Authors' Contact Information: Olivier Melançon, Université de Montréal, Montréal, Canada, olivier.melancon.1@umontreal.ca; Manuel Serrano, Inria, Sophia Antipolis, France, manuel.serrano@inria.fr; Marc Feeley, Université de Montréal, Montréal, Canada, feeley@iro.umontreal.ca.



ACM 2475-1421/2025/10-ART330
https://doi.org/10.1145/3763108

This paper presents a novel technique for encoding floats, called *self-tagging*, that avoids most float allocations without adversely affecting the cost of checking types and accessing objects, which are frequent operations of dynamic languages.

Self-tagging exploits the fact that any invertible bitwise transformation on an IEEE754 float will map some floats to a bit arrangement that contains the correct type information at the correct location, thus requiring no memory allocation for these floats. Using the fact that some ranges of floats appear more frequently in practical applications [4, 10, 28, 45], it is possible to define a transformation that avoids heap allocating almost all floats encountered in practice.

This technique does not require any program static analysis and could be integrated in most runtime systems of languages that need run time types. It is developed in the context of 64-bit runtime systems but is also applicable to 32-bit systems. The rest of this section presents the main encodings in-use as well as their trade-offs.

1.1 Tagged Objects

The classical and popular solution to preserve type information is to attach a tag to all objects [5, 16, 21, 25]. An N-bit machine word is used to encode an *object reference* (either a value or a pointer to a heap allocated value) and a small bit field is reserved in the word to store type information. For the rest of the paper the term *object* is used interchangeably with object reference. Tagging allows low-cost type checks and object access at the cost of losing a few bits for data. Aligning all heap allocated values to 64-bit machine words conveniently frees the low bits of pointers to store a 3-bit tag. On many architectures, accessing the fields of the object can be done at no additional cost by using an offset in the memory dereference instruction to take into account the tag for that type of object. Objects can thus be encoded using one of the three following representations.

- **Tagged values** store type information with a tag, and the object's value in the remaining bits. This representation does not require heap allocation.
- **Tagged pointers** also store type information with a tag, but values are stored in the heap. The objects are represented by pointers with a tag in their low bits.
- **Generic pointers** represent all remaining objects as pointers with a generic tag. The generic tag is associated with multiple types, and type information is stored in a header, in the heap.

Hence, a tag indicates a type, but also a memory representation. A tagged value representation is the most efficient as it needs no heap allocation and no memory read. A tagged pointer still allows efficient type checks, but requires heap allocation. This introduces the overhead of dereferencing the pointer, and adds a strain on the garbage collector. A generic pointer is the least efficient representation as it involves storing type information in the heap, which increases space usage and the cost of type checks and memory management. Consequently, specific tags are a precious resource that must be carefully assigned to the most frequent types observed in programs, such as small integers or floats. Figure 1 shows a typical memory layout of each representation.

As an example, consider an implementation using 3-bit tags. Due to the frequent use of small integers, a tagged value representation is appropriate for small integers and the choice of the tag `000` allows direct addition/subtraction. Since 3 bits are occupied by the tag, small integers are limited to 61 bits. This is generally considered an acceptable trade-off since a two's complement representation of signed integers can accommodate any number of bits, and 61 bits still offers a wide range of values. For larger values, generic pointers to heap allocated big integers can be used [25]. This provides a hybrid representation where the most common integers appear as efficient tagged values and less-frequent big integers are represented with more costly generic pointers.

Unfortunately, when it comes to the representation of floats, none of the aforementioned options are well-suited. Double-precision floats cannot be represented as tagged values since the IEEE754

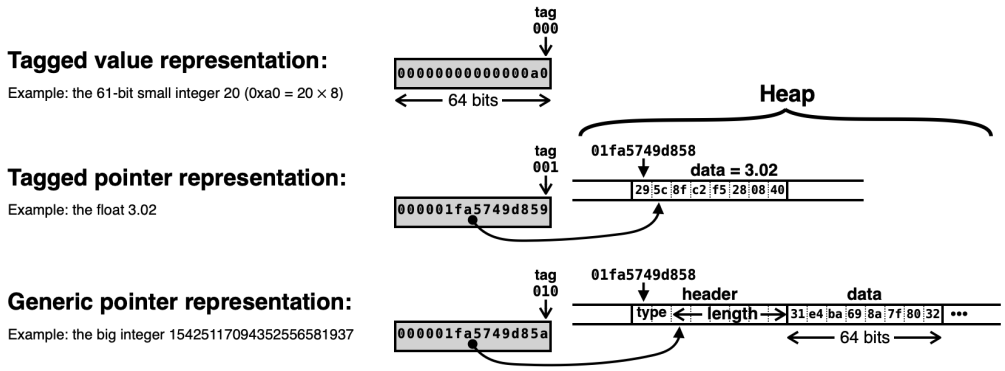


Fig. 1. The three representations in a tagged object system (here shown on a little-endian machine).

standard enforces the use of 1 sign bit, 11 exponent bits and 52 mantissa bits (see Figure 2a), totaling 64 bits and leaving no space for a tag [1]. Parting from this standard is impracticable due to the set of instructions offered by x86 [12] and ARM [26] architectures being specific to 32 or 64-bit floats.

Yet, tagged pointers introduce two major inefficiencies for floats. First, all uses of the value require fetching it from memory. Second, the heap allocation of all floats increases memory usage and the cost of computing float results. This is especially disconcerting considering that allocated floats are often short-lived intermediate results that add a strain on garbage collectors [22, 40]. This has costly implications for programs performing extensive numerical computations or languages such as JavaScript where floats are the only available primitive number type [20].

Still, tagged objects are straightforward to implement and are thus found in numerous compilers such as V8 [42], QuickJS [7], and Hopc [39] (JavaScript), the Lua interpreter [38] (Lua), CRuby [41] (Ruby), SBCL [35] (Lisp), and Gambit [30] and Bigloo [29] (Scheme). Recently, CPython also moved toward tagged pointers in the process of removing its global interpreter lock [15].

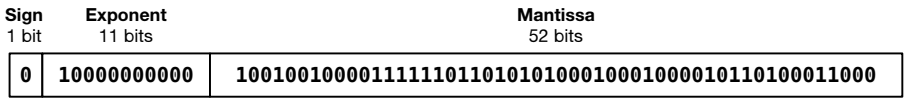
1.2 NaN-Boxing

NaN-boxing circumvents the drawbacks of tagged pointer floats by reclaiming unused bits in the encodings of floating-point NaN values to store data [16, 25]. As per the IEEE754 standard, a NaN value is represented by setting all 11 bits of the exponent to 1 and a non-zero mantissa (zero is used to encode `Infinity`). The mantissa's highest bit distinguishes between a quiet and signaling NaN, which determines whether the NaN should signal an exception or fall through operations.

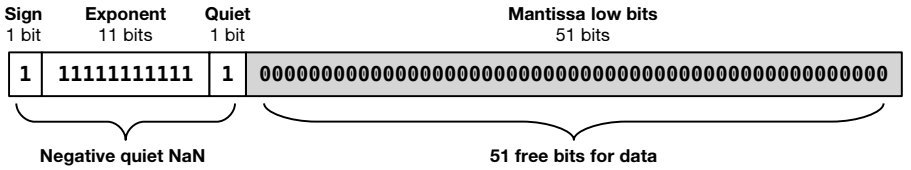
A subset of NaNs can thus be reserved to encode non-float objects. A convenient subset is that of negative, quiet NaNs, illustrated in Figure 2b, which correspond to the interval from `0xfff8_0000_0000_0000` to `0xffff_ffff_ffff_ffff`. This partitions floating-point numbers in two intervals. Bit patterns above `0xfff8_0000_0000_0000` are reserved for non-float objects. The bit patterns below `0xfff8_0000_0000_0000` are reserved for floats (including signaling NaN but not quiet NaN) and negative quiet NaN are mapped to the bit pattern `0xfff8_0000_0000_0000`.

On current hardware, memory addresses typically fit in 48 bits. The 51 bits uncovered by NaN-boxing are thus sufficient for storing tagged pointers with 3-bit tags, and 32-bit small integers. It offers the advantage of unboxed floats, but impacts the performance of other, non-float objects due to higher-cost machine instructions to check the type and dereference NaN-boxed pointers.

This reliance on hardware specific details also interferes with other optimizations and portability. Moreover, on 32-bit architectures, NaN-boxing would cripple memory management because 32-bit float NaNs leave space for only 22-bit pointers, which only allows addressing 4 MiB.



(a) IEEE754 floating-point representation of π.



(b) NaN-boxing uses the 51 free bits on negative, quiet NaN values to store data.

Fig. 2. Floating-point representations of numbers and NaN.

NaN-boxing is used in a few language implementations, including SpiderMonkey [14] (JavaScript), tinylisp [13] (Lisp), LuaJIT [36] (Lua), and Zag [31] (Smalltalk).

1.3 Pointer-Biased NaN-Boxing

NuN-boxing is a variant of NaN-boxing that alleviates the cost of dereferencing NaN-boxed pointers [25]. It relies on the (currently valid) fact that common hardware never returns negative, quiet NaNs greater than or equal to `0xfffe_0000_0000_0000`.

Instead of taking advantage of the unused NaN value range from `0xfffe_0000_0000_0000` to `0xffff_ffff_ffff_ffff`, which would result in an encoding similar to standard NaN-boxing, NuN-boxing biases all floats by adding `0x0001_0000_0000_0000`. This frees up the lowest and highest ranges for 48-bit tagged objects as shown in Figure 3. It however reintroduces a cost for encoding and decoding floats (adding/subtracting the bias) to operate on their value.

NuN-boxing is used by JavaScriptCore [19] (JavaScript).

1.4 Contribution

The contribution of this paper is a thorough analysis of a new approach to object tagging, named *self-tagging*, that allows attaching type information to *some* 64-bit objects by applying a bitwise transformation that preserves their value without heap allocation. This approach is applied to implement double-precision floats as tagged values instead of tagged pointers. Contrarily to NaN-boxing, it does not impact the performance of encoding and decoding other tagged objects and is applicable in the context of 32-bit architectures. Self-tagging cannot represent all floats and thus requires an hybrid representation for fast self-tagged and slow heap allocated floats. Yet, the conducted experiments show that the set of numbers covered by self-tagging is large enough to remove nearly all heap allocations of floats in practice. Hence, it provides another alternative to allocated floats with different trade-offs than NaN/NuN-boxing.

1.5 Paper Structure

This paper is structured as follows. Section 2 describes the general idea of self-tagging with a focus on its application to floats. Section 3 presents technical details for portable implementations that could be used by compilers that generate either C or assembly code. Section 4 shows experimental results in the context of Scheme with the Bigloo and Gambit compilers. Section 5 discusses the adaptation of self-tagging to 32-bit architectures. Finally, Section 6 presents related work.

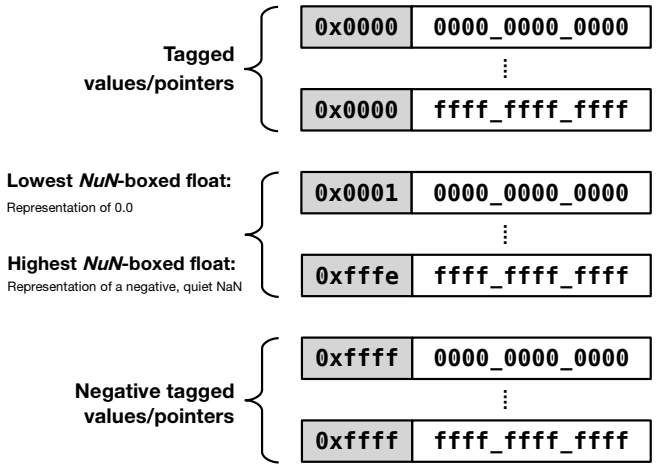


Fig. 3. Encoding of NuN-boxed values, after having applied the `0x0001_0000_0000_0000` bias to floats. The low and high ranges (whose 16 highest bits are `0x0000` or `0xffff`) are used for non-float objects that can be represented as tagged values or pointers.

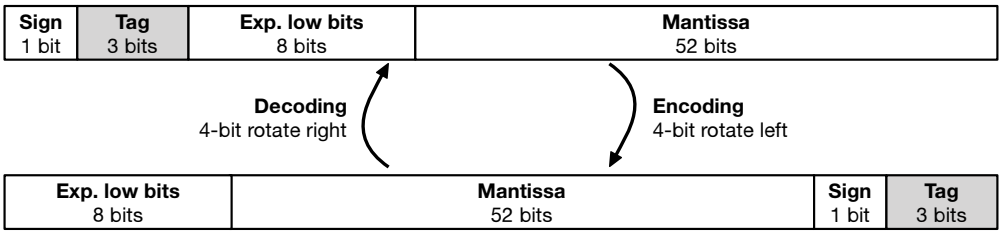


Fig. 4. A float self-tagging representation where the tag corresponds to the high bits of the exponent. The top bit sequence is a float where the tag is superimposed with the exponent's three most significant bits. The bottom sequence is the tagged value representation of the float where a 4-bit left rotation is applied to place the tag on low bits.

2 Self-Tagging

This section describes *self-tagging*, a tagging technique that exploits the fact that some values can be encoded to a tagged value by a bitwise transformation containing the tag corresponding to their type. Such self-tagged objects avoid the cost of heap allocation. In this paper, self-tagging is applied to the specific case of IEEE754 floats. Unless stated otherwise, double-precision floats and a 64-bit architecture are implied. However, self-tagging also pertains to 32-bit architectures, which is discussed in Section 5.

Consider the common case of tagged objects with 3-bit tags (8 available tags). Given tag T for floats, any invertible bitwise transformation will map $1/8$ of possible floats to a bit arrangement that contains T on its low bits. These floats can thus be encoded to tagged values with this transformation and decoded back to a IEEE754 representation by applying the inverse transformation. The set of floats that are mapped to a correctly tagged value by a given transformation are said to be *self-tagged* and require no memory allocation. Since only a subset of all floats can be self-tagged, the remaining floats must be heap allocated with tagged or generic pointers. Yet, the proportion of

self-tagged floats can be increased by assigning more than one tag to self-tagged floats. In general, using n tags for self-tagging allows avoiding heap allocation of $n/8$ of all possible floats.

As an example, consider the case where the encoding transformation is a 4-bit rotation to the left, which has the effect of using the 3 high bits of an IEEE754 float's exponent as tag for self-tagged floats (Figure 4). This transformation has the interesting property that each tag captures a contiguous range of floats. For instance, assigning the tags `000`, `011` and `100` to self-tagged floats, avoids heap allocating all floats in the ranges $0.0 \mathinner{..} 7\times10^{-251}$ and $1.7\times10^{-77} \mathinner{..} 2.3\times10^{77}$ as well as the corresponding negative ranges (detailed calculations of these ranges are provided in Section 2.1).

This suggests that, while many transformations can be chosen to implement self-tagging, some choices are better than others. Three properties make for a useful self-tagging scheme. It must:

- **Capture intervals of common floats**: most values computed in practice are either ±0.0 or centered around ±1.0 [4, 10, 28, 45]. These should be prioritized to reduce memory allocation.
- **Have an efficient encoding/decoding**: the cost of the transformation must not outweigh the benefit of decreased memory allocations and strain on the garbage collector.
- **Use few tags**: it should reserve as few tags as possible to avoid having to defer to generic pointers for other important types. Ideally, a single tag should be used for all common floats, and uncommon floats should be represented either with tagged or generic pointers.

Section 2.1 presents self-tagging variants that offer different tradeoffs between captured intervals and number of reserved tags. Section 3 describes efficient implementations for each variant.

2.1 Float Coverage

Self-tagging introduces the notion of tags that indicate which values of a type are not heap allocated. This contrasts with standard object tagging that invites assigning tags with efficient handling of frequent types in mind, regardless of values. Consequently, self-tagging should be used for the *most common* floats to maximize the likelihood that a float will be self-tagged. Floats whose magnitude is concentrated near 1.0 have been observed to be more common [4, 10, 28, 45], with a relative occurrence that decreases sharply away from 1.0 [45]. Zeros are common, but subnormal numbers are rarely used [45].

These observations from [4, 10, 28, 45] are reproduced by instrumenting the R7RS benchmarks, the standard benchmark suite for Scheme [2], to create a profile of computed floats. All benchmarks labelled as *float* benchmarks were profiled (this includes `fibfp`=recursive Fibonacci with floats, `fft`=1024 points Fast Fourier Transform, `mbrot`=mandelbrot set, `nucleic`=3D structure determination of a nucleic acid, `pnpoly`=determine if a 2D point is in a 20 sided polygon, `ray`=raytrace a scene of 33 spheres, `simplex`=linear programming using simplex method, `sumfp`=add the floats 0 to 10^6, and `sum1`=add a file of floats).

The results are shown in Figure 5. The table contains 32 rows, one for each combination of the 5 most significant bits of the float's 11-bit exponent field, which is indicated in the first column (for now, ignore the second column). Each of the right columns indicates what proportion of computed floats have a specific combination of most significant exponent bits. The computed floats are clearly clustered around 1.0, specifically in the range $1.1 \times 10^{-19} \mathinner{..} 3.7 \times 10^{19}$, and the value 0.0 also occurs frequently for some programs. In fact, R7RS benchmarks have a distribution of floats that appears even more tightly concentrated near 1.0 than that reported by [45, Fig. 7], shown in the rightmost column. Hence, other scientific programs may exhibit profiles that use slightly wider ranges of floats. It is apparent that many programs will operate with floats that are predominantly those around 1.0 on a logarithmic scale, and also 0.0.

5 most signif. expo. bits	5 most signif. expo. bits + 1	Value range	fft	fibfp	mbrot	nucleic	pnpoly	ray	simplex	sumfp	sum1	[45]
00000	00001	0.0	79%	9%	0%	1%	11%	7%	11%	13%	0%	0%
		5e-324 .. 2.1e-289	-	-	-	-	-	-	-	-	-	0%
00001	00010	2.1e-289 .. 3.8e-270	-	-	-	-	-	-	-	-	-	0%
00010	00011	3.8e-270 .. 7e-251	-	-	-	-	-	-	-	-	-	0%
00011	00100	7e-251 .. 1.3e-231	-	-	-	-	-	-	-	-	-	0%
00100	00101	1.3e-231 .. 2.4e-212	-	-	-	-	-	-	-	-	-	0%
00101	00110	2.4e-212 .. 4.4e-193	-	-	-	-	-	-	-	-	-	0%
00110	00111	4.4e-193 .. 8.1e-174	-	-	-	-	-	-	-	-	-	0%
00111	01000	8.1e-174 .. 1.5e-154	-	-	-	-	-	-	-	-	-	0%
01000	01001	1.5e-154 .. 2.8e-135	-	-	-	-	-	-	-	-	-	0%
01001	01010	2.8e-135 .. 5.1e-116	-	-	-	-	-	-	-	-	-	0%
01010	01011	5.1e-116 .. 9.4e-97	-	-	-	-	-	-	-	-	-	0%
01011	01100	9.4e-97 .. 1.7e-77	-	-	-	-	-	-	-	-	-	0%
01100	01101	1.7e-77 .. 3.2e-58	-	-	-	-	-	-	-	-	-	0%
01101	01110	3.2e-58 .. 5.9e-39	-	-	-	-	-	-	-	-	-	1%
01110	01111	5.9e-39 .. 1.1e-19	0%	-	-	-	-	0%	-	-	-	3%
01111	10000	1.1e-19 .. 2	21%	28%	89%	61%	63%	22%	64%	13%	0%	61%
10000	10001	2 .. 3.7e19	0%	63%	11%	38%	26%	70%	25%	75%	100%	28%
10001	10010	3.7e19 .. 6.8e38	-	-	-	-	-	-	-	-	-	2%
10010	10011	6.8e38 .. 1.3e58	-	-	-	-	-	-	-	-	-	1%
10011	10100	1.3e58 .. 2.3e77	-	-	-	-	-	-	-	-	-	0%
10100	10101	2.3e77 .. 4.3e96	-	-	-	-	-	-	-	-	-	0%
10101	10110	4.3e96 .. 7.9e115	-	-	-	-	-	-	-	-	-	0%
10110	10111	7.9e115 .. 1.5e135	-	-	-	-	-	-	-	-	-	0%
10111	11000	1.5e135 .. 2.7e154	-	-	-	-	-	-	-	-	-	0%
11000	11001	2.7e154 .. 4.9e173	-	-	-	-	-	-	-	-	-	0%
11001	11010	4.9e173 .. 9.1e192	-	-	-	-	-	-	-	-	-	0%
11010	11011	9.1e192 .. 1.7e212	-	-	-	-	-	-	-	-	-	0%
11011	11100	1.7e212 .. 3.1e231	-	-	-	-	-	-	-	-	-	0%
11100	11101	3.1e231 .. 5.7e250	-	-	-	-	-	-	-	-	-	0%
11101	11110	5.7e250 .. 1.1e270	-	-	-	-	-	-	-	-	-	0%
11110	11111	1.1e270 .. 1.9e289	-	-	-	-	-	-	-	-	-	0%
11111	00000	1.9e289 .. 1.8e308	-	-	-	-	-	0%	-	-	-	0%
		Infinity/NaN	-	-	-	-	-	-	-	-	-	N/A

Fig. 5. The absolute value ranges captured by each combination of the 5 most significant exponent bits and the proportion of all floats computed by float-intensive R7RS benchmarks. Intervals of interest are highlighted. Empty entries indicate no float in this range, whereas 0% means that very few floats were generated (less than 0.5%). The last column shows the relative occurence of float for each interval reported by [45, Fig. 7].

2.2 Self-Tagging with 3 Tags and 4 Tags

The float profile of Figure 5 has the property that almost all floats computed have the 3 most significant bits of the exponent field that are either `000` (green rows), `011` (dark blue rows), or `100` (light blue rows). Moreover, among the values captured by `000`, only `±0.0` are used.

These 3 exponent bits can be shifted using a bit rotation to align them with the lower 3 tag bits as shown in Figure 4. Thus a float is encoded to a tagged value by a 4-bit rotation to the left (or 60-bit to the right). A tagged value is decoded to a float with the inverse rotation. This corresponds to the encoding discussed in the previous section (Figure 4).

Test programs did not compute the float values ±Infinity and NaN, but those values would be assigned the tag 111 (gray rows). Assigning this fourth tag to self-tagged floats might be useful to avoid heap allocated floats in programs frequently computing these values.

2.3 Self-Tagging with 2 Tags and Preallocated Zeros

As is shown in the float profile of Figure 5 the tag `000` is only used to capture the values `±0.0`. It can be freed for other uses by preallocating the values `±0.0` using tagged pointer or generic pointer representation. When a float must be encoded as an object it is compared to zero, in which case one of the preallocated zeroes is returned, thus avoiding a new allocation. If this is done then only 2 tags are needed to cover the most frequent cases: `011` and `100`. Unfortunately, this approach may raise the number of failed branch predictions due to the test for `±0.0` (this is further discussed in Section 4.6).

2.4 Self-Tagging with 1 Tag

Self-tagging can also be achieved with a single tag by a different transformation of the most significant exponent bits. The 5 most significant bits of the exponent are added to 1 and the middle 3 bits are used as the tag (using a rotation 5 places to the left). This corresponds to the second column of Figure 5. Note that the tag `000` (bold pink rows) covers all the useful ranges, including ±Infinity and NaN that are not covered by the 2-tag and 3-tag variants.

It is noteworthy that there is nothing special about the tag `000`. If for some external reason some other tag is more convenient it can easily be assigned to 1-tag self-tagged floats by adding $1 + 2 \times tag$ rather than 1 to the 5 most significant exponent bits.

Putting all of this together, the self-tagged encoding of a float whose IEEE754 64-bit representation is the integer n can be computed as $(n \oplus ((1 + 2 \times tag) << 58)) <<_{rot} 5$, where $\oplus$ is addition modulo 2^{64} and $<<_{rot}$ is the left rotation operator. The operations are reversed to perform the decoding of a self-tagged float to its IEEE754 64-bit representation.

This variant has the advantages of covering all the useful ranges of floats computed by R7RS benchmarks and about 90% of floats reported by [45], while using a single tag, which leaves more tags available for other types.

2.5 Self-Tagging with a Different Exponent Bias

A small change in the exponent bias of the 1-tag variant can be used to get another self-tagging variant. If $2 \times tag_1$ is added to the highest 5 bits and the middle 3 bits are examined, then when they are equal to tag_1 or $tag_2 = tag_1 - 1$ they are self-tagged floats. This effectively doubles coverage by including both the light pink and bold pink rows of the second column of Figure 5. The following ranges will be self-tagged: $0..3.8 \times 10^{-270} \cup 5.9 \times 10^{-39}..6.8 \times 10^{38} \cup 1.1 \times 10^{270}..Infinity/NaN$. To put this into perspective, this is a superset of the IEEE754 32-bit floats, so any floating point value that can be represented as a IEEE754 32-bit float can be self-tagged with this variant which uses 2

of the 8 3-bit tags. This is likely a better use of 2 tags than the 2-tag variant of Section 2.3, which does not cover the ranges containing 0, Infinity, and NaN.

When compared to the 1-tag variant, there will be almost no difference in heap allocations for programs with similar float profiles as Figure 5. However, as will be explained in the next section, it yields more compact code, because the bias can be added after the rotation, and the need for a bias disappears when tag_1=`000`.

3 Implementation

This section discusses implementation details for float-related operations in dynamic languages. The types `f64` and `f32` will refer to IEEE754 64-bit and 32-bit doubles and the types `i64` and `i32` will refer to 64-bit and 32-bit integers. Moreover, the type `object` will refer to values in the language (the sum type of floats, integers, booleans, and other objects). To illustrate the low-level implications of the implementation and give a sense of the execution time performance, assembly code for the Intel x86 architecture is provided. This section focusses on a 64-bit implementation where `object` is an `i64` and floats are `f64`, but it is straightforward to implement the operations similarly on a 32-bit implementation where the type `object` is an `i32` and floats are `f32`.

Testing the low tag bits of an `object` value is a basic need for float-related operations to convert `f64` to and from `object` and for dynamic type checks. In particular it is possible to determine if a `f64` can be self-tagged by going through the encoding process and checking that the resulting tag is one of the tags (or *the* tag) appropriate for the chosen self-tagging variant.

3.1 Single Tag Testing

Checking that the N low bits are equal to a specific tag can be achieved by masking those bits followed by a comparison. Assuming the `object` value is in the x86 64-bit register `rax`, then the following sequence will branch to label `tag_matches` when the 3 low bits are equal to *tag*:[1]

```
and al, 7
cmp al, tag
jz  tag_matches
```

Note that by using `al` only the lower 8 bits of `rax` are accessed, leading to smaller constants and instruction encodings. However, this sequence modifies `rax` so if the value is needed after the tag test, as often will be the case, then an additional instruction and register are required to keep a copy. In the special case of *tag*=0, the two first instructions can be replaced with "`test al, 7`", which is a *bitwise-and* that tests if all 3 lowest bits of `rax` are zero without modifying `rax`. In the general case when *tag*≠0, the `test` instruction can be combined with a `lea` (*Load Effective Address*) instruction to copy the source register, plus a constant to cancel *tag* (mod 8), to a temporary register:

```
lea  ebx, [eax+(8-tag)] # ebx = eax + (8-tag)
test bl, 7
jz   tag_matches
```

3.2 Multiple Tag Testing

When it is necessary to check if an object's tag is one of a set of tags it is often possible to use a single branch instruction rather than a sequence of single tag checks. This is particularly the case when the implementer has leeway in the assignments of tags, as is often the case.

Checking that $tag \in \{tag_1, tag_2\}$ where $tag_2 = tag_1 + 2^n \pmod 8$ can be done just as efficiently as the single tag test by using the `lea` instruction to map tag_1 to 0 and tag_2 to 2^n and then using a

[1]Intel assembler syntax is used throughout.

mask of $7 - 2^n$ in the `test` instruction to ignore the bit that may be 0 or 1. For example, for 2-tag self-tagging (Section 2.5) with the tags `010` and `110` the check is:

```
lea  ebx, [eax+(8-2)]  # 010 -> 000 and 110 -> 100
test bl, 3             # 3 = 7-4
jz   tag_matches_010_or_110
```

4-tag self-tagging (Section 2.2) with the tags `010`, `011`, `110`, and `111` is even simpler:

```
test al, 2
jnz  tag_matches_010_or_011_or_110_or_111
```

The general case of checking that *tag* is in a set of tags can be done efficiently with the x86 *bit-test* instruction "`bt` *n*, *i*" that reads the bit at index *i* of *n* and puts it in the carry flag. The `bt` instruction comes in 16, 32, and 64 bit variants, but not 8 bit. For example, for 3-tag self-tagging (Section 2.2) with the tags `000`, `011`, and `100` the check is:

```
mov bx, 0x1919  # set of tags
bt  bx, ax      # test bit at index ax of bx
jc  tag_matches_000_or_011_or_100
```

The bit index in `ax` is obtained *mod 16*, so the byte `0b00011001=0x19` (all 0's except at bit indices 0, 3, and 4) is repeated twice in register `bx`. Using the `bt` instruction has the advantage of neither modifying the `rax` register nor the register holding the set of tags, amortizing the cost of `mov` over multiple checks of that set of tags (a possibly near zero cost if that register is globally reserved).

On architectures without a bit test instruction, a dynamic count shift of the tag set register can achieve the bit indexing. If done by modifying the tag set register, the initialization can't be shared by multiple tag tests. On most 3-address RISC architectures the shift can be done non-destructively. On ARM A64, this instruction sequence tests the 3 low bits of 64-bit register `x1` for a tags 0, 3, or 4:

```
lslv w3, w2, w1
cmp  w3, 0
bmi  matching_tag
```

It assumes that the 32-bit register `w2` has been preloaded with 0x98989898, the bit reversed tag set that aligns the bit for tag 0 with the sign bit.

The cost of the check does not depend on the number of tags tested. This can be advantageous if a tagged pointer is used for heap allocated floats, say with tag *h*. In that case, a check for a float (either self-tagged or heap allocated) can be done by adding *h* to the tag set. Once it is known that an object is a float, it is easy to check for the single tag *h* to discriminate between the self-tagged and heap allocated representations.

3.3 Boxing: Conversion `f64`→`object`

To convert a `f64` to an `object`, it is converted to an `i64`, added to a constant *bias* modulo 2^{64}, and bit-rotated left by *r*. For the 1-tag variant, $bias = (1 + 2 \times tag) \times 2^{58}$ and $r = 5$. For the 2-tag, 3-tag, and 4-tag variants, $bias = 0$ (no bias) and $r = 4$. After this there is a check to verify if the resulting object's tag is in the set of self-tagged tags. If this is the case the conversion is done, otherwise an out of line routine can be called to heap allocate a float and return an appropriately tagged pointer. As an example, if the `f64` value is in the 64-bit float register `xmm0`, then the following x86 code will set register `rax` to the corresponding `object` when 3-tag self-tagging is used:

```
  mov  rax, xmm0   # rax ← xmm0
  rol  rax, 4      # rotate left 4 places
  bt   bx, ax      # assumes bx initialized to 0x1919 elsewhere
  jnc  heap_alloc_float # tag is not one of 000, 011, or 100?
done:
```

Assuming the initialization of the tag mask register (bx) is amortized over multiple float-related operations, the cost for the hot path is low: 3 register-to-register operations and an easily predictable conditional jump.

The following x86 code is appropriate for 1-tag self-tagging using 000 as the tag:

```
  mov  rax, xmm0    # rax ← xmm0
  add  rax, rbx     # assumes rbx initialized to 1<<58 elsewhere
  rol  rax, 5       # rotate left 5 places
  test al, 7
  jnz  heap_alloc_float # tag is not 000?
done:
```

Here also, the cost for the hot path is just a bit more due to the bias: 4 register-to-register operations and an easily predictable conditional jump.

For the 2-tag variant with no special handling of zeroes (Section 2.5), the rotation can be done before adding the bias to use a compact addition instruction (because then the bias is the small constant tag_1). In the special case of tag_1=000 the bias vanishes. For example, here is the code when tag_1=011:

```
  mov  rax, xmm0    # rax ← xmm0
  rol  rax, 5       # rotate left 5 places
  add  al, 3        # sufficient to modify the lower 8 bits of rax
  bt   bx, ax       # assumes bx initialized to 0x0c0c elsewhere
  jnc  heap_alloc_float # tag is not one of 010 or 011?
done:
```

The add instruction uses a small constant as a bias, contrary to the code for the 1-tag version (large constants typically need to be setup in a register, which is additional work even if it is amortized). Moreover the 1-tag version cannot eliminate the bias, but here the add instruction can be removed when tag_1=000 which makes it slighly faster than the 1-tag variant and as efficient as the 3-tag variant: 3 register-to-register operations and an easily predictable conditional jump.

3.4 Unboxing: Conversion object→f64

To convert an object (that is known to be a float) to a f64, the tag must be tested to see if it is a self-tagged float. If it is, the boxing operations are done in reverse order and inverted, replacing the rol by a ror (rotate-right), and if there is a bias, replacing the add by a sub (subtract). If it is not a self-tagged float, a memory read gets the f64 result out of the heap allocated float. As an example using the 3-tag variant, if the object value is in the 64-bit register rax, then the following x86 code will set register xmm0 to the corresponding f64:

```
  bt  bx, ax              # assumes bx initialized to 0x1919 elsewhere
  jc  self_tagged
  mov xmm0,[rax+offset]   # xmm0 ← value field of float
  jmp done
self_tagged:
  ror rax, 4              # rotate right 4 places
  mov xmm0, rax           # xmm0 ← rax
done:
```

The number and type of instructions on the hot path is the same as for the boxing operation.

In summary, the self-tagging operations require few machine instructions but there are small variations depending on the instruction set and also the assignment of tags which can often be optimized for self-tagging without impacting the speed of operations on objects or small integers.

3.5 C Implementation

This section explains how the above operations can be implemented in C, as this is a common implementation language, in particular it is used by the Bigloo and Gambit Scheme to C compilers used in the experiments of Section 4.

Testing for a specific tag or a pair of tags is fairly straightforward to express in C and follows the same principles as the assembly code, so details are skipped. Testing for a set of tags is more challenging in portable C code. It can be implemented with the following pure C function, which is easily inlined by a C compiler (here testing for the tags `000`, `011`, and `100`):

```
#define TAG_SET ((1<<0)|(1<<3)|(1<<4)) /* 0x19 for tags 000, 011, and 100 */

inline bool has_tag_0_or_3_or_4(int64_t n) {
  return (((uint32_t)1 << (n & 31)) & (~(uint32_t)0/0xff * TAG_SET)) != 0;
}
```

The expression `n & 31` computes a shift count *mod 32* and the bitwise-and therefore tests the bit at index `n` *mod 32* of its second operand. The code uses the compile-time constant `~(uint32_t)0/0xff * TAG_SET` that repeats the 8 bit mask `TAG_SET` 4 times to fill a 32 bit word, giving 0x19191919. This achieves the equivalent of testing the bit at index `n` *mod 8* of `TAG_SET`.

Using this pure C definition with gcc version 13.2.0 and clang version 18.1.3 on x86-64 results in the non-destructive approach based on the bit test instruction. The expression `n & 31` is optimized out by those compilers because the bit test instruction on x86 implicitly does this operation. On older versions of those compilers that would implement this with a destructive shift instruction, the bit test approach can be achieved using the following definition that uses an `asm` statement:

```
inline bool has_tag_0_or_3_or_4(int64_t n) {
  bool carry;
  __asm__("bt %%ax, %2;" /* get one bit of mask into carry (ax is index) */
          : "=@ccc"(carry)
          : "a"((uint16_t)n),
            "r"(~(uint16_t)0/0xff * TAG_SET)); /* 0x1919 */
  return carry;
}
```

Boxing and unboxing operations need to convert between a float value and its bit representation. This can be achieved portably with a `union` type whose fields are of type `f64` and `i64`. Moreover, boxing and unboxing need to rotate the bit representation of the float. Although C does not provide an operator for bit rotation, both gcc and clang recognize an equivalent pair of shifts and generate a single machine rotate instruction.

The implementation of boxing in C for the 3-tag variant is:

```
union di { double d; int64_t i; };

#define ROTL(n,s) ((int64_t)(((uint64_t)n << s) | ((uint64_t)n >> (64 - s))))

inline int64_t f64_to_object(double f) {
  int64_t result = ROTL(((union di)f).i, 4);
  if (has_tag_0_or_3_or_4(result)) return result;
  return heap_allocate_float(f);
}
```

The implementation of unboxing for the 3-tag variant is:

```
inline double object_to_f64(int64_t o) {
  int64_t result = ROTL(o, 60);
  if (has_tag_0_or_3_or_4(result)) return ((union di)result).d;
  return value_of_heap_allocated_float(o);
}
```

4 Experiments

This section evaluates float self-tagging and demonstrates that it presents an interesting performance trade-off between programs that perform frequent operations on objects but few floating-point operations, and float-intensive programs where this balance is reversed. This evaluation is conducted through experiments that measure performance metrics across different object representations (both self-tagging and existing ones), multiple machine architectures, and two state-of-the-art Scheme implementations: the Bigloo (commit `5b1118`) [29] and Gambit v4.9.7 (commit `768900`) [30] compilers. These offer a level of performance that is competitive with other high-performance Scheme implementations such as Chez Scheme [2, 3, 33]. Using two independently developed compilers helps to demonstrate that self-tagging can be adapted to preserve the different design philosophies and historical implementation choices of these systems.

The following self-tagging variants are evaluated:

- **3-tag self-tagging** uses 3 tags for self-tagging floats corresponding to the green, dark blue, and light blue ranges from Figure 5. Remaining floats are heap allocated. Bigloo uses the tags `000`, `011` and `100`, and heap allocated floats are represented with generic pointers. Gambit offsets the tags by 3 (i.e. it uses `011`, `110` and `111`) so that the tag `000` is kept for tagging small integers, and tagged pointers with the tag `010` are used for the heap allocated floats.
- **4-tag self-tagging** (Gambit only) is like the 3-tag variant but uses the tag `010` to self-tag floats in the gray range from Figure 5. Allocated floats are represented with generic pointers.
- **2-tag self-tagging with preallocated zeros** (Bigloo only) tests the impact of expending the tag `000` for self-tagging. It only uses the tags `011` and `100` for float self-tagging the dark blue and light blue ranges from Figure 5 and reclaims the tag `001` to represent all heap allocated floats as tagged pointers. The floats `±0.0` are preallocated and the float boxing operation contains an explicit zero check to return the preallocated zeros.
- **2-tag self-tagging** (Gambit only) tests the variant that uses the tags `010` and `110` for float self-tagging the bold pink and light pink ranges from Figure 5.
- **1-tag self-tagging** tests the variant that uses a single tag for self-tagging the bold pink ranges from Figure 5. Bigloo uses the tag `001` and Gambit uses the tag `110`. heap allocated floats are represented with tagged pointers.

Both compilers can also be configured to represent all floats with heap allocated tagged objects or NuN-boxing. Bigloo also supports NaN-boxing. This provides a flexible comparison environment.

In this paper the C back end of the Bigloo and Gambit compilers are used. This way the resulting machine code benefits from the C compiler's optimizations and in particular excellent architecture specific instruction selection. In both cases the same version of the gcc C compiler is used.

4.1 Overview of the Bigloo Compiler

The Bigloo compiler conforms to the R5RS Scheme specification and adds several extensions, including optional type annotations. Exact rational numbers and complex numbers, which are optional in the R5RS Scheme specification, are not supported.

The Bigloo compiler builds a typed abstract syntax tree from the compiled Scheme program. Several analysis and optimizations refine the initial optional type annotations contained in the source program, the main ones being occurrence-typing [43] and storage use analysis [40]. During compilation, primitive values have a dual representation: one for polymorphic contexts and one for specific contexts. For instance, when the compiler can establish that a function is always invoked with floats, the compiler assigns the C `double` type to that function and, when needed, it introduces cast operations from and to the generic representation of floats and double.

The result of this compilation technique is that many local variables and local functions are precisely typed and avoid polymorphic representations. This contributes to removing otherwise necessary heap allocations and casts for floats. The compiler also tracks cases where only floats are stored into vectors, which are then transformed into arrays of C doubles.

Bigloo uses the Boehm-Demers-Weiser conservative mark-and-sweep garbage collector [8]. The complexity of collecting dead objects is proportional to the sum of the live and dead objects, contrary to copying collectors. For such a collector, avoiding allocating short lived floats is crucial for performance.

4.2 Overview of the Gambit Compiler

The Gambit compiler conforms to the R7RS Scheme language. It features the full numerical tower including arbitrary-precision integers, exact rational numbers and complex numbers. Arithmetic operators are generic and the result type depends on the specific values, for example multiplying two complex numbers may yield a number of any type. Like Bigloo, the compiler inlines the small integer and float cases of the dispatch and handles other cases in an out-of-line function.

First-class continuations (`call/cc`) and tail-calls are fully supported, and stack-overflows are gracefully handled. With the C back end, this requires the use of a trampoline and the management of the stack through an explicit array, stack pointer and stack-overflow checks. This causes a small slowdown when compared to a machine code back end where the trampoline would not be required. Memory management is based on a fast bump-allocator and a stop-and-copy garbage collector.

The compiler front end performs several source-to-source optimizations, such as constant folding, lambda lifting, and function inlining. It does not perform type analysis or type inference. However, the back end uses a simple mechanism to optimize float calculations by tracking the use of their results within basic blocks. The result of a float operation is kept in raw form so that other operations within the same basic block that consume it do not need to unbox the value. A float value is boxed only if it remains live at the end of the basic block.

4.3 Experimental Setup

Self-tagging variants are implemented in Bigloo and Gambit, and tested using a subset of the R7RS benchmarks, which is the standard benchmark suite for Scheme [2]. The experiment includes all macro-benchmarks (more than 500 lines of code) and benchmarks whose calculations mainly involve floats (tagged as such in the R7RS benchmarks suite). For the purpose of analyzing results, it is useful to distinguish between benchmarks that use few or no floats and float-intensive benchmarks (`fibfp`, `fft`, `mbrot`, `nucleic`, `pnpoly`, `ray`, `simplex`, `sum1`, and `sumfp`).

Experiments were repeated on the following machines that operate on distinct microarchitectures:

- Intel - Intel(R) Xeon(R) W-2245, 32GB, Linux 6.12.31 x86_64, Debian 10.13, gcc 14.2.0
- Amd - AMD Ryzen 7955WX, 62GB, Linux 6.12.31 x86_64, Debian 10.13, gcc 14.2.0
- M2 - Apple M2 Max, 64GB, macOS 15.4.1, gcc 14.3.0
- Risc-V - riscv sifive, u74-mc, 8GB, Linux starfive 6.1.31-starfive riscv64, gcc 14.2.0

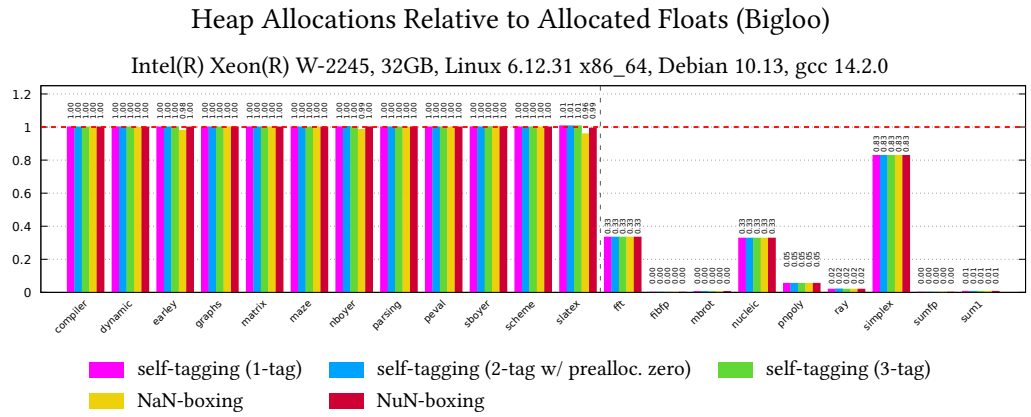


Fig. 6. Heap allocations of Bigloo with each variant of self-tagging, NaN-boxing, NuN-boxing, and allocated floats. The y-axis shows the number of allocated bytes relative to the version that heap allocates all floats (dashed horizontal red line). 1.00× indicates no change, while 0.00× indicates no heap allocation.

For each combination of machine, compiler and self-tagging variant, benchmarks are executed 10 times, and each repetition is configured to last at least 5 seconds. Repetitions of each variant and baseline are paired in order of execution to compute relative execution time for each pair. Upcoming sections report geometric means of these relative times, with geometric standard deviations.

To reduce variance on Intel and Amd, benchmarks are executed on the same CPU core with `taskset` and address space randomization is disabled with `setarch`. On M2, no equivalent tools are available, hence the higher variance. On Risc-V, variance remains low despite not using `taskset` and `setarch`.

For measurements that were similar across all machines, a single microarchitectures is shown. Omitted figures can be found in a companion artifact [32].

4.4 Memory Profiling and Execution Time

Figure 6 shows the memory allocations of Bigloo's self-tagging variants, NaN-boxing, and NuN-boxing compared to allocated floats. Since NaN/NuN-boxing are known to allocate no floats at all, these correspond to executions where heap usage is for non-float objects only. A few self-tagging variants allocate some floats on some benchmarks (see Figure 5), but these allocations are rare enough that memory profiling of all variants is nearly identical to that of NaN/NuN-boxing.

Detailed execution time comparisons with NuN-boxing are shown in Figures 7 and 8 (Bigloo and Gambit respectively). Figure 7 also shows time for NaN-boxing, which Bigloo implements. Figure 9 summarizes results by providing the geometric mean of execution times relative to NuN-boxing for each float encoding, compiler, and microarchitecture, for both float and non-float benchmarks.

As a general observation, none of the evaluated encodings is a better alternative overall; the best encoding depends on the nature of benchmarks, microarchitecture, and implementation. For instance with Bigloo, NaN-boxing is faster across microarchitectures on float-intensive benchmarks, but slowest on non-float benchmarks. Conversely, self-tagging generally fares better than NaN/NuN-boxing on non-float benchmarks since it introduces no overhead on non-float objects.

Bigloo's self-tagging does not consistently match the performance of NuN-boxing on float benchmarks across microarchitectures (but is faster than allocated floats, as will be discussed in Section 4.5). The 1-tag and 3-tag variants offer the highest performance overall for Bigloo, with the 3-tag variant being faster for float benchmarks and slightly slower for non-float benchmarks.

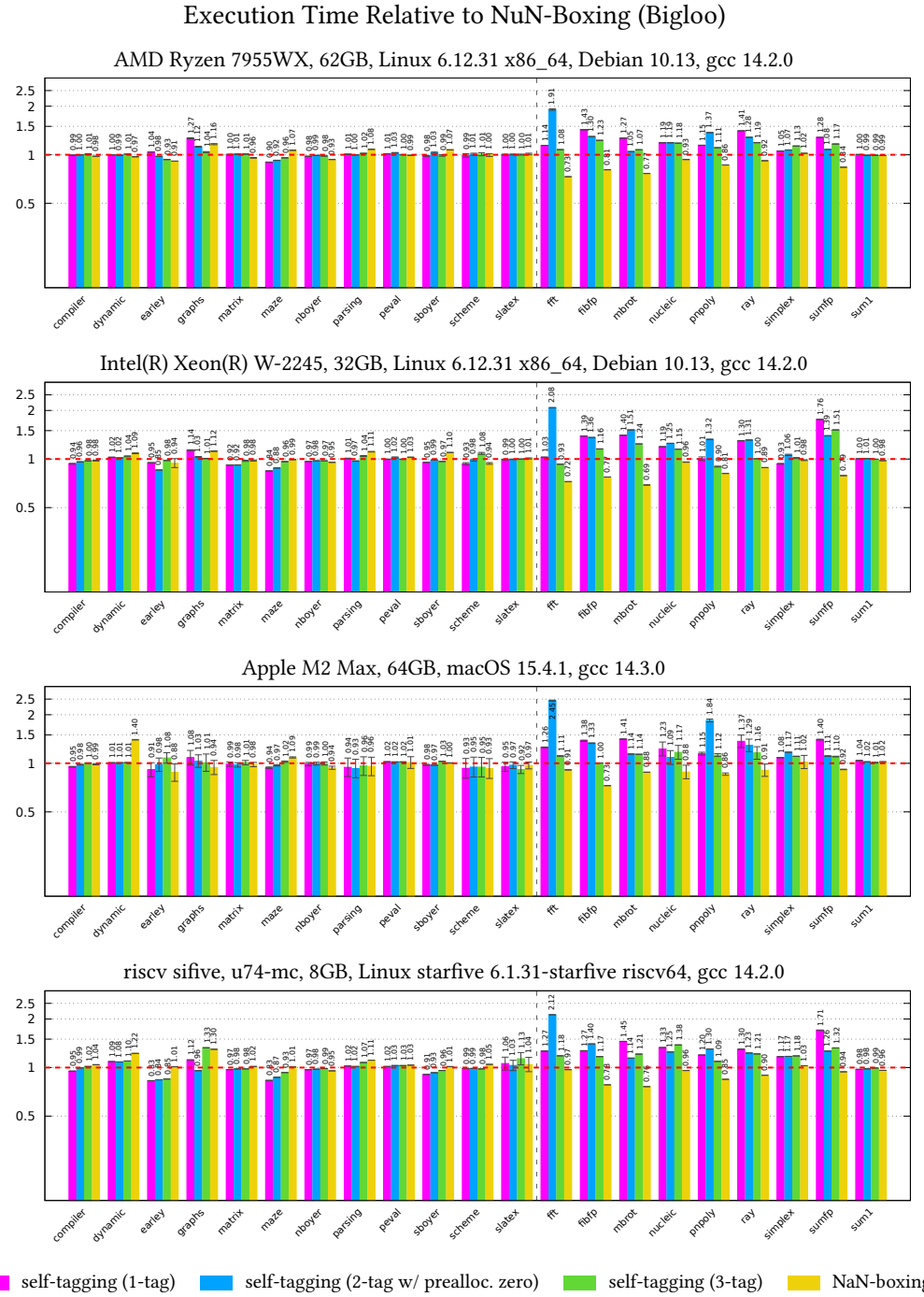


Fig. 7. Execution time of Bigloo with each variant of self-tagging and NaN-boxing relative to NuN-boxing on four distinct microarchitectures (from top to bottom: AMD, INTEL, M2, and RISC-V). The y-axis shows execution time relative to NuN-boxing (dashed horizontal red line) on a logarithmic scale. 1.00× indicates no change, lower means faster, and higher means slower execution than NuN-boxing.

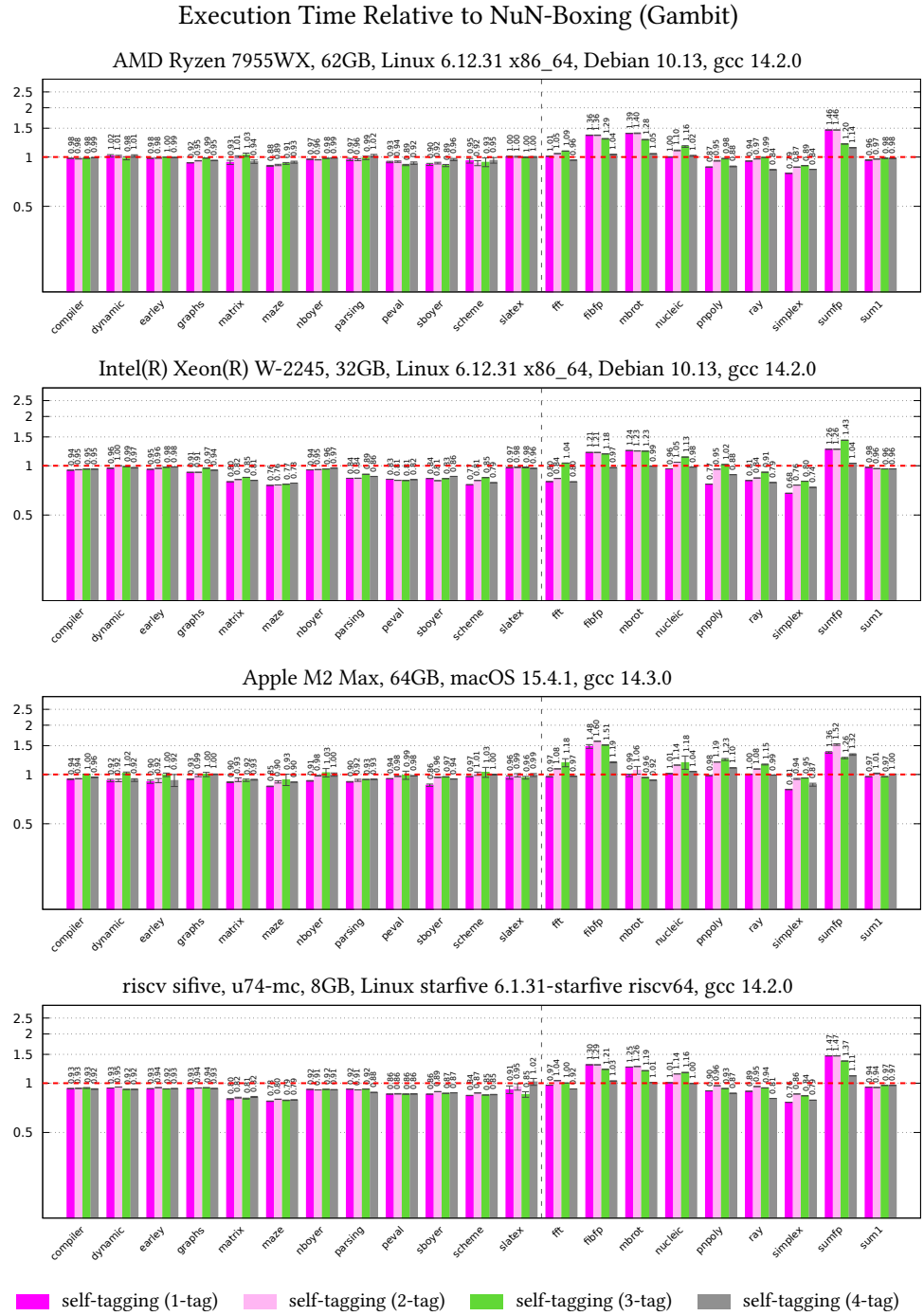


Fig. 8. Execution time of Gambit with each variant of self-tagging relative to NuN-boxing on four distinct microarchitectures (from top to bottom: AMD, INTEL, M2, and RISC-V). The y-axis shows execution time relative to NuN-boxing (dashed horizontal red line) on a logarithmic scale. 1.00× indicates no change, lower means faster, and higher means slower execution than NuN-boxing.

Summary of Execution Time Relative to NuN-Boxing

		Non-float benchmarks				Float benchmarks			
		Amd	Intel	M2	Risc-V	Amd	Intel	M2	Risc-V
Bigloo	1-tag	1.01×	.97×	.97×	.97×	1.20×	1.20×	1.25×	1.28×
	2-tag w/ prealloc. zero	1.01×	.96×	.98×	.97×	1.22×	1.34×	1.33×	1.29×
	3-tag	1.00×	1.00×	1.00×	1.02×	1.13×	1.09×	1.10×	1.19×
	NaN-boxing	1.01×	1.02×	1.00×	1.06×	.87×	.84×	.90×	.90×
Gambit	1-tag	.95×	.87×	.91×	.88×	1.06×	.94×	1.05×	1.03×
	2-tag	.96×	.88×	.95×	.89×	1.11×	1.00×	1.16×	1.09×
	3-tag	.96×	.90×	.98×	.88×	1.09×	1.06×	1.14×	1.06×
	4-tag	.97×	.89×	.96×	.89×	.97×	.90×	1.04×	.94×

Fig. 9. Geometric mean of execution times relative to NuN-boxing of float and non-float benchmarks for each combination of float encoding, compiler, and microarchitecture. 1.00× indicates no change, lower means faster, and higher means slower execution than NuN-boxing.

Bigloo's 2-tag with preallocated zeros has similar average performance as 1-tag, however it shows more variability across benchmarks (see fft and pnpoly in Figure 7) since it has to test for ±0.0 out-of-line. Hence, Gambit's 2-tag variant (Section 2.5) should be preferred to that of Bigloo.

Gambit's 4-tag variant provides consistently faster average execution times for both float and non-float benchmarks on all architectures except M2 where it is nearly the same speed as NuN-boxing. Gambit's 1-tag variant also fares well on all benchmarks on Intel and Risc-V.

Self-tagging and NaN/NuN-boxing have competitive performance and the best choice depends on the specific setting. When floating point computation performance is important but less so than operations on other objects, self-tagging is a good option to consider. Additionally, a significant portion of this study was spent adapting existing systems (Bigloo and Gambit) to new float encodings, and self-tagging was found to be qualitatively simpler to implement as it only affects the encoding of floats, while NaN/NuN-boxing has deeper object encoding implications.

4.5 Memory Management Considerations

Figures 10 and 11 show execution times of the best self-tagging variants on each implementation (Bigloo 1-tag, and Gambit 4-tag respectively) relative to allocated floats. For Bigloo, decreased memory allocations correlates with lower execution time on all architectures (see Figure 6) with the exception of `sum1`, an I/O bound benchmark.

Conversely, Gambit execution time slightly increases on many float benchmarks. This difference is caused by its use of a stop-and-copy garbage collector with bump allocation, while Bigloo uses the Boehm mark-and-sweep GC [8]. For programs with a nearly empty heap (default of R7RS benchmarks), the overhead of self-tagging's encoding/decoding outweighs that of bump allocation.

However, since self-tagging improves performance by allocating fewer floats, its impact is more pronounced for programs that spend more time in memory management tasks. This section presents an experiment to measure execution time in a more realistic setting where benchmarks are executed with live data on the heap, thus putting more pressure on the garbage collector.

For this experiment, the R7RS benchmark suite is modified to allocate a vector before the execution of each benchmark. The size of the allocated vector is given as a parameter of the benchmark. Each float benchmark is then executed with vectors of 10^4 to 10^8 fields, which correspond to about 80 kB and 800 MB respectively. An execution is also done with no preallocated vector.

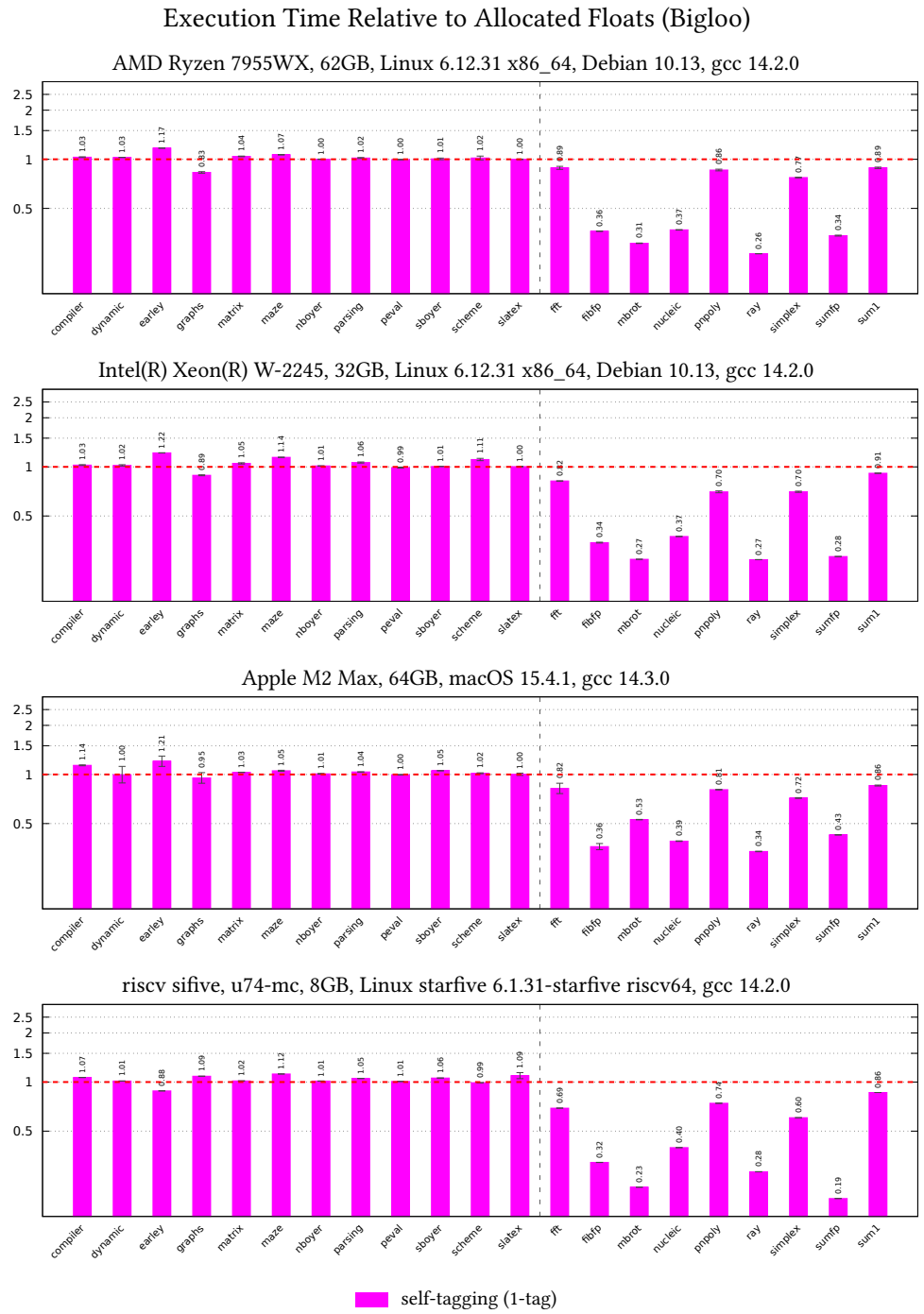


Fig. 10. Execution time of Bigloo with self-tagging (1-tag) relative to allocated floats on four distinct microarchitectures (from top to bottom: AMD, INTEL, M2, and RISC-V). The y-axis shows execution time relative to the version that heap allocates all floats (dashed horizontal red line) on a logarithmic scale. 1.00× indicates no change, lower means faster execution than allocated floats.

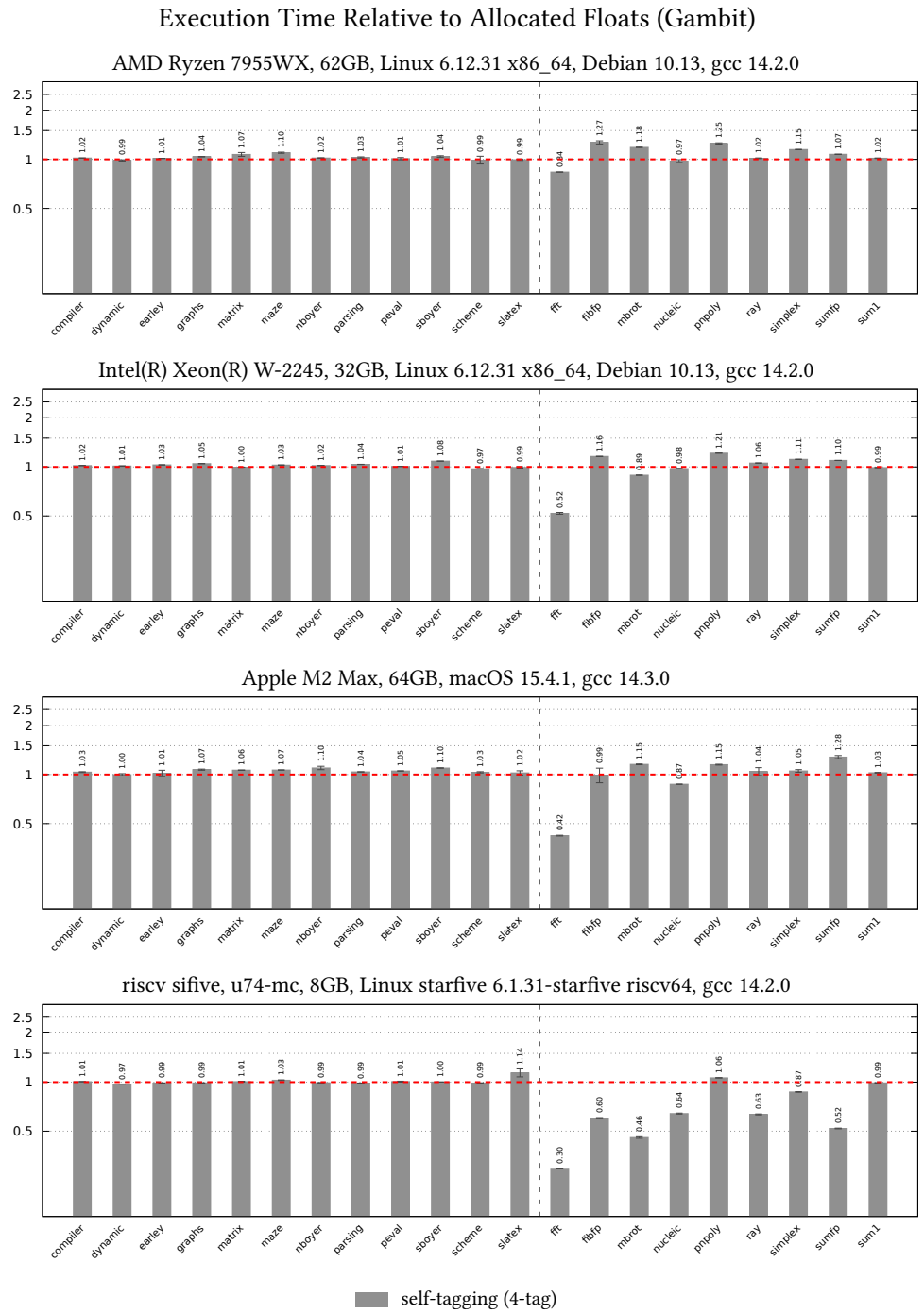


Fig. 11. Execution time of Gambit with self-tagging (4-tag) relative to allocated floats on four distinct microarchitectures (from top to bottom: AMD, INTEL, M2, and RISC-V). The y-axis shows execution time relative to the version that heap allocates all floats (dashed horizontal red line) on a logarithmic scale. 1.00× indicates no change, lower means faster execution than allocated floats.

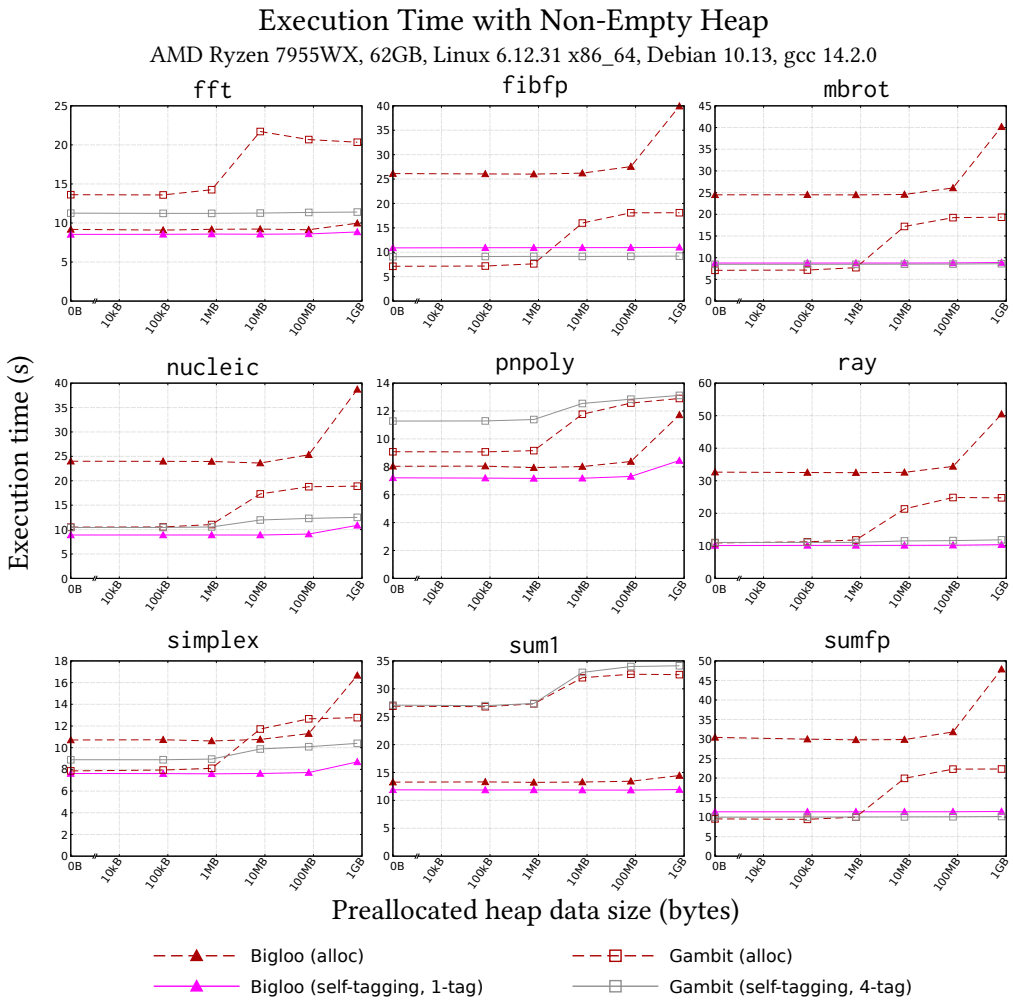


Fig. 12. Execution time of float benchmarks with extra data allocated on heap before execution with Bigloo and Gambit on AMD. The x-axis shows the size of preallocated data in bytes on a logarithmic scale. The y-axis represents execution time in seconds (mean of 10 repetitions). Execution time without preallocated data is added at $x = 0$. Dashed lines correspond to times with allocated floats. Solid lines are with self-tagging.

Figure 12 compares execution times for Bigloo and Gambit with self-tagging (1-tag and 4-tag respectively) and allocated floats with increasing vector sizes. With a nearly empty heap, and thus little strain on the garbage collector, the bump allocator of Gambit with allocated floats sometimes outperforms self-tagging since it does not incur the cost of encoding and decoding self-tagged floats. Once the heap is preloaded with about 1 MB (which is the size of the L2 cache per core on the AMD machine) the strain of the garbage collector starts to outweigh the cost of encoding and decoding self-tagged floats, thus self-tagging hereafter outperforms allocated floats.

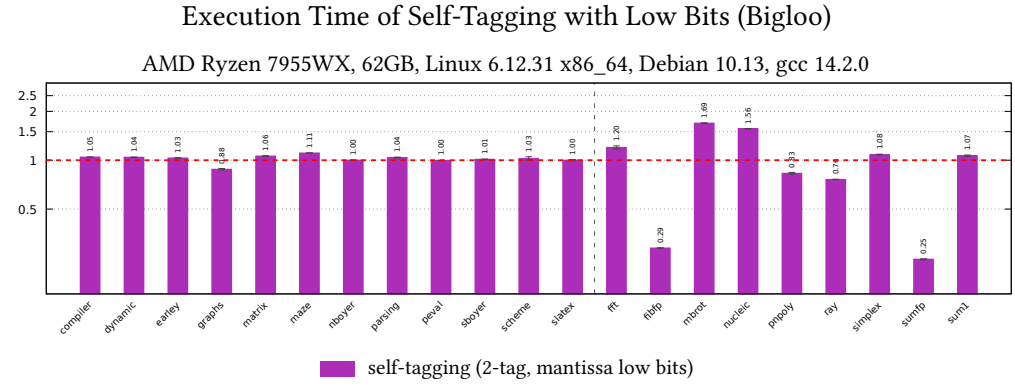


Fig. 13. Execution time of Bigloo with self-tagging using the mantissa low bits relative to allocated floats on the Amd microarchitecture. The y-axis shows execution time relative to the version that heap allocates all floats (dashed horizontal red line) on a logarithmic scale. 1.00× indicates no change, lower means faster, and higher means slower execution than allocated floats.

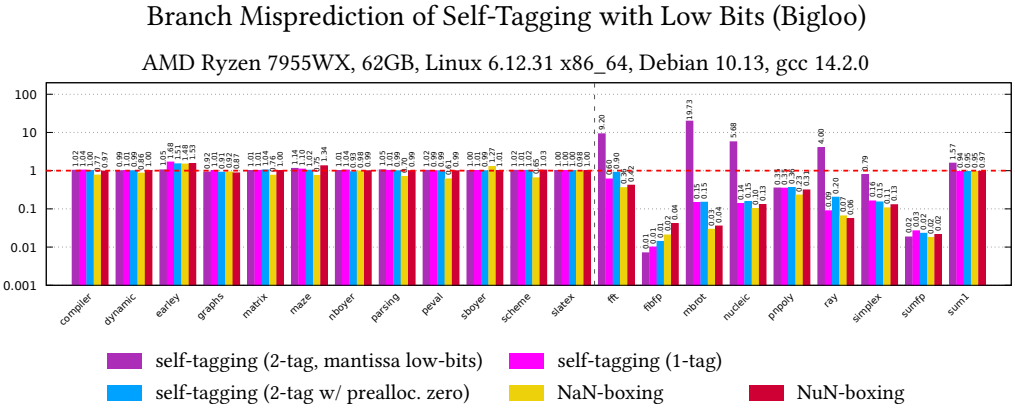


Fig. 14. Branch mispredictions of Bigloo with each variant of self-tagging, NaN-boxing, and NuN-boxing relative to allocated floats by the Amd microarchitecture. The y-axis shows branch mispredictions relative to allocated floats (dashed horizontal red line) on a logarithmic scale. 1.00× indicates no change, lower means less, and higher means more mispredictions.

4.6 Branch Prediction Considerations

This section discusses the impact of branch misprediction on the performance of self-tagging by presenting another variant of self-tagging that was developed based on intuitions that turned out to be wrong. This negative result is detailed here to guide implementers when adapting self-tagging.

Self-tagging variants from previous sections use high exponent bits of floats. This is convenient since it captures contiguous ranges of floats at the cost of a small encoding/decoding overhead. Alternatively, low bits of a float's mantissa can be used. For instance, by reserving the tag `000` for self-tagged floats and heap allocating all floats whose low mantissa bits are not `000`.

This encoding has no overhead since a self-tagged float corresponds exactly to its IEEE754 value. It comes as the cost of no longer capturing contiguous ranges of floats, but rather a set uniformly spread across all possible floats. As in previous techniques, the proportion of captured floats can be

increased by assigning more tags to self-tagging. This variant is implemented in Bigloo with tags `000` and `100`, thus self-tagging 1/4 of all floats, in particular all floats n that are integers $|n| \leq 2^{51}$.

Figure 13 compares execution times of this variant to allocated floats. As expected, benchmarks that extensively use floats corresponding to integers (`fibfp` and `sumfp`) execute faster, even faster than previous variants (see Figure 10), due to most floats avoiding heap allocation. However, other benchmarks (`mbrot` and `nucleic`) are slower despite allocating about 1/4 fewer floats.

This unpredictability is explainable by profiling the number of branch mispredictions caused by this variant. Figure 14 shows missed branch predictions of float encodings, including self-tagging with mantissa bits. This encoding causes a steep increase in missed branch predictions on some benchmarks, namely `mbrot` and `nucleic`. This stems from the fact that when tags occupy the low bits of the mantissa, even the smallest variations cause a float's representation to switch between self-tagged and tagged pointer in a hard to predict way. This is beyond the capabilities of the branch predictor and execution suffers costly misprediction penalties [11].

This result refutes the intuition that self-tagging with mantissa bits may be a good compromise due to its cheap encoding/decoding. It also hints at the necessity to avoid tests for special values in encoding/decoding, such as the test for ±0.0 in the 2-tag self-tagging variant with preallocated zeros, which has a higher occurrence of branch mispredictions than 1-tag self-tagging in Figure 14.

5 Self-Tagging on 32-bit Architectures

Although previous sections only considered double-precision floats on 64-bit architectures, self-tagging has a natural adaptation for 32-bit architectures. Indeed, the core idea, which is to superimpose a tag to a sequence of bits likely to appear in practice, can be used for any machine word size. A popular object representation on 32-bit architectures is to use 2-bit tags (4 possible tags) and heap allocated objects aligned to 32-bit words. Figure 15 shows the ranges covered by each combination of the top 4 bits of the exponent, which is 8 bits wide in the IEEE754 32-bit representation.

The 1-tag approach described previously can be adapted to the narrower exponent of the IEEE754 32-bit representation field: the self-tagged encoding of a float whose IEEE754 32-bit representation is the integer n can be computed as $(n \oplus ((1 + 2 \times tag) << 27)) <<_{rot} 4$, where $\oplus$ is addition modulo 2^{32} and $<<_{rot}$ is the left rotation operator. If the two lowest bits are equal to tag then the float is self-tagged. This will cover all float values in the ranges: $0.0..3.9 \times 10^{-34} \cup 3.1 \times 10^{-5}..1.3 \times 10^{5} \cup 1.0 \times 10^{34}..Infinity/NaN$ (bold pink rows of Figure 15 using tag=`00`).

The coverage can be doubled by using an adaptation of the 2-tag variant described in Section 2.5: $0.0..2.5 \times 10^{-29} \cup 4.7 \times 10^{-10}..8.6 \times 10^{9} \cup 1.6 \times 10^{29}..Infinity/NaN$ (light pink rows of Figure 15 using tag_1=`00` and tag_2=11). With this variant, it would be reasonable to reserve one tag for small signed integers in the range $-2^{29}..2^{29} - 1$ (-536870912 .. 536870911), and the remaining tag for heap allocated objects, including the floats that can't be self-tagged.

6 Related Work

Handling floats is a long-standing issue in dynamic language implementation. In the last decades, little progress has been made improving the encoding of floats, with most implementations either using heap allocated floats or suffering from the overhead of NaN-boxing on pointers. Thus, more general strategies that use data flow analysis [18, 23, 27, 37, 40] and partial evaluation [44] have been developed to find locations where the type of a value is known across its lifecycle. This allows the generation of specialized code that safely handles fully unboxed, untagged values. Such strategies are not specific to unboxing floats. Rather, they tackle the more general problem of inferring types in dynamic languages, which allows dropping type information in either tagged pointers or NaN-boxed pointers. While more limited in scope, self-tagging solves the problem of heap allocated floats with a new encoding that involves no program analysis.

4 most signif. expo. bits	4 most signif. expo. bits + 1	Value
0000	0001	0.0
		1.4e-45 .. 3.9e-34
0001	0010	3.9e-34 .. 2.5e-29
0010	0011	2.5e-29 .. 1.7e-24
0011	0100	1.7e-24 .. 1.1e-19
0100	0101	1.1e-19 .. 7.1e-15
0101	0110	7.1e-15 .. 4.7e-10
0110	0111	4.7e-10 .. 3.1e-5
0111	1000	3.1e-5 .. 2
1000	1001	2 .. 1.3e5
1001	1010	1.3e5 .. 8.6e9
1010	1011	8.6e9 .. 5.6e14
1011	1100	5.6e14 .. 3.7e19
1100	1101	3.7e19 .. 2.4e24
1101	1110	2.4e24 .. 1.6e29
1110	1111	1.6e29 .. 1.0e34
1111	0000	1.0e34 .. 3.4e38
		Infinity/NaN

Fig. 15. The positive value ranges captured by each combination of the 4 most significant exponent bits of the IEEE754 32-bit representation. The rows coloured in bold pink are covered by the 1-tag variant (tag `00`) and the rows coloured in light pink are covered by the 2-tag variant (tags `00` and 11).

Self-tagging is straightforward to add to compilers that already use object tagging. Therefore, implementations that represent floats as tagged pointers could benefit from it with minimal implementation effort. Such popular implementations include CPython [15] and Google's V8 [42]. Compilers that use NaN/NuN-boxing, such as Mozilla's SpiderMonkey [14] and Apples's JSC [19] can also benefit from the lower impact that self-tagging has on the performance of non-float types.

Due to the shortcomings of tagged pointers and NaN-boxing, additional strategies are used to avoid boxing. A straightforward approach is to provide homogeneous data structures such as `TypedArray` in JavaScript [20] and `numpy`'s `float64` arrays in Python [17]. Type homogeneity then allows unboxing values within the collection. More generally, compilers can detect data homogeneity at run time and use context-dependent storage strategies to store data in a way that prevents pointer chasing [9]. Since self-tagging avoids heap allocating most floats in practice, it effectively prevents such pointer chasing in the case of floats without static or run time analysis.

Even for general-purpose languages not specifically designed for numerical computation, ensuring reasonably fast float operations is a major concern. All *production-quality* implementations surveyed for this paper devote significant efforts to optimizing them. Unfortunately, the academic literature is scarce when it comes to techniques used by these efficient implementations.

In many cases, the source code itself has to be studied to understand the implementation and compare it to self-tagging. The NuN-boxing implementation used in experiments in this paper is based on that of JSC.[2] NuN-boxing has the benefits over self-tagging that it encodes all floats uniformly and does not reserve tags for them. Its drawbacks are that it negatively impacts the type checking and accesses of all objects, it can represent integers with at most 48 bits (usually 32 bits for practical reasons), and it can't be used on 32-bit architectures.

NaN/NuN-boxing are trade-offs between the performance of floats and pointers (NaN-boxing favors floats, NuN-boxing favors pointers). Self-tagging offers a different trade-off between the performance of frequently used floats and other floats. The fact that some ranges of floats are more commonly used has been studied in scientific computing where real world data typically fall into a narrow range of values [4, 10, 28, 45]. This is in accordance with the results presented in Figure 5.

[2] https://github.com/WebKit/WebKit/commit/74c9b9bbc2c7527144ac366c3f62f84aaa1a8a4e

The techniques closest to self-tagging discovered among surveyed implementations are those of CRuby,[3] OpenSmallTalk[4] and MonNom [34]. CRuby and OpenSmallTalk use a variant close to the 1-tag encoding of Section 2.4. The MonNom implementation from [34] uses a variant of the 2-tag encoding of Section 2.3 with 2-bit tags. Floats captured by these implementations all exclude ±0.0, and treat it as a special case. This imposes complex sequences of instructions for encoding/decoding that can hardly be inlined and risks increasing branch misses as discussed in Section 4.6.

To the best of the authors' knowledge, this is the first work exposing the principles of self-tagging, implementing it in optimizing compilers, and evaluating its benefits, drawbacks, and variants.

In this paper, self-tagging was discussed in the context of dynamic languages. Yet, polymorphic languages such as OCaml and Haskell also attach information to run time values and spend considerable effort to find efficient encodings for abstract data types [6, 24]. Self-tagging could be applied in these languages, either to floats or to the encoding of abstract data types.

7 Conclusion

Avoiding heap allocation of floating point numbers has always been a major concern for dynamic and polymorphic languages. This paper presents and evaluates the popular techniques deployed to that end (NaN and NuN-boxing) as well as a new approach, coined *self-tagging*, that allows representing certain floats without heap allocation.

The core idea is to apply a bitwise transformation that maps the most common floats used in practice to bit patterns that contain the required type information at the correct position. This allows encoding floats in specific ranges as tagged values instead of heap allocated objects, reducing the strain on the garbage collector and improving performance. Contrary to NaN-boxing and NuN-boxing, which are designed to optimize the handling of floats, self-tagging offers a different trade-off that adds no overhead to the handling of other types. Since it does not rely on the specificity of the IEEE754 NaN encoding, self-tagging is highly portable, including to 32-bit architectures.

It is also straightforward to retrofit in implementations that already use object tagging. The only required changes are to reserve one or more tags for float self-tagging and implement an encoding/decoding function for self-tagged floats. The rest of the runtime stays unchanged.

As shown by the experimental evidence presented, float self-tagging is a good candidate for general-purpose languages, such as JavaScript, whose specification relies heavily on floats, but where the performance of other types must not be sacrificed.

Given the purely bit-shuffling nature of float self-tagging boxing and unboxing, it may also be a good candidate for an essentially zero-cost implementation at the hardware level when floats are moved between float and integer registers.

Acknowledgment

The authors thank Erven Rohou for valuable insights and discussions that contributed to this work. This work was supported by the *Natural Sciences and Engineering Research Council of Canada* and the *Fonds de recherche du Québec*[5].

Data Availability Statement

Data generated and analyzed for this paper is available in a companion artifact [32] that includes results for all microarchitectures listed in Section 4. The artifact also contains all source code used to run the experiments.

[3] https://github.com/ruby/ruby/commit/b3b5e626ad69bf22be3228f847f94e1b68f40888
[4] https://clementbera.wordpress.com/2018/11/09/64-bits-immediate-floats
[5] https://doi.org/10.69777/366699

References

[1] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

[2] 2025. R7RS Benchmarks. https://github.com/ecraven/r7rs-benchmarks.

[3] 2025. R7RS Benchmarks. https://ecraven.github.io/r7rs-benchmarks/.

[4] Azim Afroozeh, Leonardo X. Kuffo, and Peter A. Boncz. 2023. ALP: Adaptive Lossless floating-Point Compression. *Proc. ACM Manag. Data* 1, 4 (2023), 230:1–230:26. https://doi.org/10.1145/3626717

[5] Andrew W. Appel. 1998. *Modern Compiler Implementation in C.* Cambridge University Press, Cambridge, UK, Chapter 16, 372–373.

[6] Thaïs Baudon, Gabriel Radanne, and Laure Gonnord. 2023. Bit-Stealing Made Legal: Compilation for Custom Memory Representations of Algebraic Data Types. *Proc. ACM Program. Lang.* 7, ICFP (2023), 813–846. https://doi.org/10.1145/3607858

[7] Fabrice Bellard and Charlie Gordon. 2024. QuickJS JavaScript Engine. https://bellard.org/quickjs/.

[8] Hans-Juergen Boehm and Mark D. Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exp.* 18, 9 (1988), 807–820. https://doi.org/10.1002/SPE.4380180902

[9] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2013. Storage strategies for collections in dynamically typed languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 167–182. https://doi.org/10.1145/2509136.2509531

[10] Ashley W. Brown, Paul H. J. Kelly, and Wayne Luk. 2007. Profiling floating point value ranges for reconfigurable implementation. In *Proceedings of the 1st HiPEAC Workshop on Reconfigurable Computing.* 6–16.

[11] Randal E. Bryant and David R. O'Hallaron. 2016. *Computer Systems - A Programmer's Perspective* (Third Global ed.). Pearson, Chapter 5, 577–579.

[12] Intel Corporation. 2024. *Intel® 64 and IA-32 Architectures Software Developer's Manual - Instruction Set Reference.* https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[13] Robert Van Engelen. 2024. Lisp in 99 lines of C and how to write one yourself. https://github.com/Robert-van-Engelen/tinylisp.

[14] Mozilla Foundation. 2024. SpiderMonkey 131.0 - Mozilla's JavaScript and WebAssembly Engine. https://spidermonkey.dev/.

[15] Sam Gross. 2023. PEP 703 - Making the Global Interpreter Lock Optional in CPython. https://peps.python.org/pep-0703.

[16] David A. Gudeman. 1993. *Representing Type Information in Dynamically Typed Languages.* Technical Report. University of Arizona. https://www.cs.arizona.edu/sites/default/files/TR93-27.pdf

[17] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[18] Fritz Henglein. 1992. Global Tagging Optimization by Type Inference. In *Proceedings of the Conference on Lisp and Functional Programming, LFP 1992, San Francisco, California, USA, 22-24 June 1992*, Jon L. White (Ed.). ACM, 205–215. https://doi.org/10.1145/141471.141542

[19] Apple Inc. 2024. JavaScriptCore 2.46.3 - JavaScript Engine for WebKit. https://webkit.org.

[20] ECMA International. 2015. *Standard ECMA-262 - ECMAScript Language Specification* (6th ed.). https://www.ecma-international.org/ecma-262/6.0/

[21] Richard Jones, Antony Hosking, and Eliot Moss. 2012. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman and Hall/CRC, Florida, USA, Chapter 11, 168–171.

[22] Richard Jones, Antony Hosking, and Eliot Moss. 2012. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman and Hall/CRC, Florida, USA, Chapter 9, 111–114.

[23] Shiro Kawai. 2008. Efficient floating-point number handling for dynamically typed scripting languages. In *Proceedings of the 2008 Symposium on Dynamic Languages, DLS 2008, July 8, 2008, Paphos, Cyprus*, Johan Brichau (Ed.). ACM, 6. https://doi.org/10.1145/1408681.1408687

[24] Andrew J. Kennedy and Dimitrios Vytiniotis. 2012. Every bit counts: The binary representation of typed data and programs. *J. Funct. Program.* 22, 4-5 (2012), 529–573. https://doi.org/10.1017/S0956796812000263

[25] Daan Leijen. 2022. *What About the Integer Numbers? Fast Arithmetic with Tagged Integers - A Plea for Hardware Support.* Technical Report MSR-TR-2022-17. Microsoft Research. https://www.microsoft.com/en-us/research/publication/what-about-the-integer-numbers-fast-arithmetic-with-tagged-integers-a-plea-for-hardware-support/

[26] Arm Limited. 2009. *Instruction Set Assembly Guide for Armv7 and earlier Arm® architectures Reference Guide.* https://developer.arm.com/documentation/100076/0200.

[27] Tobias Lindahl and Konstantinos Sagonas. 2003. Unboxed Compilation of Floating Point Arithmetic in a Dynamically Typed Language Environment. In *Implementation of Functional Languages*, Ricardo Peña and Thomas Arts (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 134–149. https://dx.doi.org/10.1007/3-540-44854-3_9
[28] Peter Lindstrom and Martin Isenburg. 2006. Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualization and Computer Graphics* 12, 5 (2006), 1245–1250. https://doi.org/10.1109/TVCG.2006.143
[29] Manuel Serrano. 2024. Bigloo. https://www-sop.inria.fr/indes/fp/Bigloo.
[30] Marc Feeley. 2025. Gambit. https://gambitscheme.org.
[31] Dave Mason. 2022. Design Principles for a High-Performance Smalltalk. In *Proceedings of the International Workshop on Smalltalk Technologies 2022 co-located with the 28th European Smalltalk User Group Conference (ESUG 2022), Novi Sad, Serbia, August 24th-26th, 2022 (CEUR Workshop Proceedings, Vol. 3325)*, Loïc Lagadec and Vincent Aranega (Eds.). CEUR-WS.org. https://ceur-ws.org/Vol-3325/regular2.pdf
[32] Olivier Melançon, Marc Feeley, and Manuel Serrano. 2025. *Float Self-Tagging Artifact.* https://doi.org/10.5281/zenodo.16356364
[33] Olivier Melançon, Marc Feeley, and Manuel Serrano. 2024. Static Basic Block Versioning. In *38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria (LIPIcs, Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:27. https://doi.org/10.4230/LIPICS.ECOOP.2024.28
[34] Fabian Muehlboeck and Ross Tate. 2021. Transitioning from structural to nominal code with efficient gradual typing. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29. https://doi.org/10.1145/3485504
[35] William Harold Newman, Christophe Rhodes, Nikodemus Siivola, Juho Snellman, Paul Khuong, Jan Moringen, and Douglas Katzman. 2024. Steel Bank Common Lisp 2.4. https://www.sbcl.org.
[36] Mike Pall. 2024. LuaJIT 2.1 - Just-In-Time Compiler for Lua. https://luajit.org.
[37] Leaf Petersen and Neal Glew. 2012. GC-Safe interprocedural unboxing. In *Proceedings of the 21st International Conference on Compiler Construction* (Tallinn, Estonia) *(CC'12)*. Springer-Verlag, Berlin, Heidelberg, 165–184. https://doi.org/10.1007/978-3-642-28652-0_9
[38] Roberto Ierusalimschy. 2024. Lua 5.4. https://www.lua.org.
[39] Manuel Serrano. 2021. Of JavaScript AOT compilation performance. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473575
[40] Manuel Serrano and Marc Feeley. 1996. Storage Use Analysis and its Applications. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, Robert Harper and Richard L. Wexelblat (Eds.). ACM, 50–61. https://doi.org/10.1145/232627.232635
[41] Pat Shaughnessy. 2013. *Ruby Under a Microscope: An Illustrated Guide to Ruby Internals.* No Starch Press, San Francisco, USA.
[42] Igor Sheludko and Santiago Aboy Solanes. 2020. Pointer Compression in V8. https://v8.dev/blog/pointer-compression.
[43] Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/1863543.1863561
[44] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 662–676. https://doi.org/10.1145/3062341.3062381
[45] Niko Zurstrassen, Lennart M. Reimann, Nils Bosbach, Lukas Juenger, and Rainer Leupers. 2023. Evaluation of the RISC-V Floating Point Extensions. In *DVCon Europe 2023; Design and Verification Conference and Exhibition Europe*. 57–64. https://ieeexplore.ieee.org/abstract/document/10461383

Received 2025-03-25; accepted 2025-08-12