Compact Representations for Arrays in Lua

Roberto Ierusalimschy
Noemi Rodriguez
roberto@inf.puc-rio.br
noemi@inf.puc-rio.br
Departamento de Informática
Catholic University of Rio de Janeiro — PUC-Rio
Rio de Janeiro, Brazil

Abstract

Several languages use a tagged representation for values, so that each value carries its own type during runtime. Lua, in particular, represents each value by a structure with two fields: A union for the values themselves and a byte with the tag. Despite its simplicity, this representation has a big drawback: Due to alignment restrictions, it typically wastes more than 40% of memory in padding. This waste is specially expensive for large arrays. In this work, we discuss alternative implementations for Lua arrays that eliminate this waste and evaluate them regarding performance, with a special focus on code overhead and memory locality. The presented data structures are quite generic, and can be used not only in the Lua interpreter, but in any program that needs arrays of tagged values.

CCS Concepts: • **Software and its engineering** → **Interpreters**; **Runtime environments**; **Software performance**.

Keywords: array representation, language implementation, Lua, memory layout

1 Introduction

Most languages with dynamic typing—and some with static typing, too—use a tagged representation for values, so that each value carries its own "type" (actually a tag) during runtime.

Some languages pack the tag information inside the value, often sacrificing some values. For instance, OCaml typically has 31 bits in its `int` type [7], while Haskell ensures only 30 bits for its `Int` type [2]. Lua uses a representation for values with explicit tags; we will call a value with this representation a *t-value*. Each t-value is represented by a C structure `TValue` with two fields, a union for the value itself and a byte with the explicit tag; see Figure 1. T-values live in the interpreter's stacks, tables, closures, etc. More often than not, Lua code passes around pointers to these structures, to avoid copying them.

This representation has several advantages: It allows a direct representation for numbers—both integers and floating-points—without using dynamic memory; it is simple and portable; and it is reasonably efficient. Its main negative point

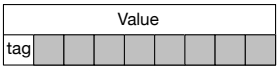

Figure 1. Layout of a `TValue` in a 64-bit memory

is memory waste: As we can see in Figure 1, a `TValue` structure typically wastes around 40% of its memory in padding due to alignment requirements.

For objects with a few t-values, that waste is usually not relevant. For instance, in a stack with 200 elements in a 64-bit system, it adds up to 1400 bytes. However, for tables representing arrays with a myriad of elements, that waste may translate to an increase of 40% in the total memory used by an application. A representation of arrays that could avoid that waste would bring a significant improvement in Lua's memory efficiency.

In this paper, we consider a few alternatives for the representation of arrays in Lua and evaluate them. For the evaluation, we focus on the following aspects:

Memory : The main goal of all alternatives is to reduce memory use. All alternatives we consider in this work eliminate all padding in the representation of arrays. However, as we will see, some of them add a little overhead, particularly for small arrays.

Performance : That is the second main goal: Ideally, an alternative implementation should add no CPU overhead compared to the current implementation. Although what matters in the end is overall performance, it is instructive to better understand what causes the differences in performance among the alternative implementations. To this end, we also consider the next aspects in our evaluation.

Code overhead : As we mentioned, one of the advantages of the current implementation is its simplicity. We can access a value with an expression like `a[i].val` and a tag with `a[i].tag`. Alternative representations may need more complex expressions to access the same information.

Memory locality : The current implementation has pros and cons regarding locality: On the good side, each tag is always next to the corresponding value. On the bad side, the extra space wasted on padding

spreads sequential elements. As we will see, the alternative implementations do not store tags and values together; on the other hand, they reduce the spreading by eliminating the wasted space.

All our alternative implementations are reasonably simple, but we have not seen any of them described in the literature before. All code described in this paper works for any architecture, given an ISO-C compliant compiler; our code works even for a machine with 5-byte pointers. Lua's codebase is very strict about its use of C [5]. The discussions, however, assume a "reasonable" architecture. A reasonable architecture for us is one where the size of pointers, floats, and integers is a power of two, typically 4 or 8 bytes. To simplify the exposition, we will assume a 64-bit architecture from now on.

The rest of this paper is organized as follows. The next section discusses related work. Section 3 reviews the implementation of arrays (actually tables) in Lua. Section 4 presents our three alternative implementations. Section 5 evaluates them. Finally, Section 6 draws some conclusions.

2 Related Work

Hugo Gualandi reported[1] that just adding the gcc attribute `__attribute__((packed))` to the definition of the structure `TValue` reduces its size from 16 to 9 bytes, without any sensible difference in performance. However, this attribute is a gcc extension not present in ISO C. Moreover, even in gcc it is not guaranteed to work [3]. As portability is a hallmark of Lua, this almost magical solution is a no-go.

As we have already mentioned, some languages like OCaml and Haskell sacrifice some bits in their native integer types to allow an unboxed representation for them. Because most machines align pointers to heap-allocated memory in word boundaries, the code can assume that any pointer to an object ends with two or three zeros. Any other value in these bits means something that is not a pointer, such as a primitive integer. This approach has the drawback of not representing faithfully C integers. For Lua, which is typically used in close integration with C code, this seems a grave issue. A second drawback is that this kind of bitwise manipulation of pointers is not expressible in ISO C [6].

Another technique to avoid explicit tags is the so called *NaN boxing* [8]. This technique assumes a double floating-point number as its basic type, and packs all other values in the 51 bits of payload of *Not-a-Number* (NaN) doubles. It is particularly well suited for JavaScript and older versions of Lua, where the only numeric type is a double. (LuaJIT, in particular, uses this technique.) But it makes little sense for a language with 64-bit integers. Moreover, it cannot directly represent pointers in 64-bit machines. (LuaJIT, for instance, limits its memory allocation to 1 GB.)

[1] Personal communication

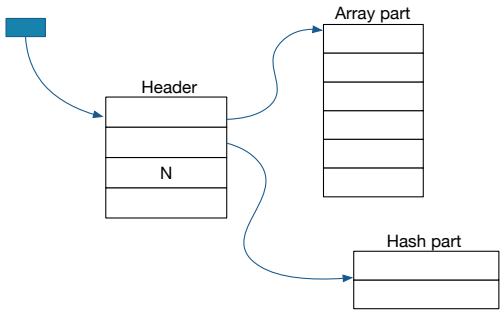

Figure 2. The structure of a table in Lua

Some dynamic language implementations have the concept of *homogeneous arrays*, arrays where all the elements have the same type. An homogeneous array needs to store just one tag for the whole array, instead of one tag per entry. NumPy, a Python library for multi-dimensional arrays, uses this concept, but the programmer must explicitly provide the type of an array when creating it. (This slightly betrays the character of a dynamic language.) To do that dynamically, the interpreter must type-check all assignments to the array and automatically deoptimize the array in case of an assignment with a value with a different type. For instance, a single integer may spoil the optimization of an array of floats. That seems too subtle for a language that is aimed at end-user programmers, which are often non-professional programmers [5].

3 Tables in Lua

In Lua, a *table* is an associative array that accepts keys and values of any type. Tables are the only data structure in Lua. Lua programs implement arrays simply by using a table with integer keys.

Since version 5.0, Lua implements tables with two data structures: A hash table and an array. Figure 2 shows this arrangement. Each table has an array size N: All positive integer keys up to N are stored in the array part; all other keys are stored in the hash part. The array size for each table is recomputed every time the table is resized and there is a rehash [4]. This dual data structure is completely transparent to programmers, except for performance. There is no way a program can detect whether a key is stored in the hash part or the array part of a table.

All entries in the hash part of a table are kept in a single array. Each element in this array stores a key, its corresponding value, and a `next` field, which is an integer used for collision resolution. Since version 5.4, Lua packs this structure by separating the key's components, as we can see in Figure 3. Unlike the key, the t-value in each entry is stored whole—a value followed by its tag; the field `i_val` in the union allows the code to access it as a proper `TValue`. Figure 4 shows the memory layout of this structure for a 64-bit architecture. As

```
typedef union Node {
  struct NodeKey {
    Value value_;  /* value of the value */
    unsigned char tt_;  /* tag of the value */
    unsigned char key_tt;  /* taf of the key */
    int next;  /* for chaining */
    Value key_val;  /* value of the key */
  } u;
  TValue i_val;  /* value as a 'TValue' */
} Node;
```

Figure 3. Type for the elements in the hash part of a table.

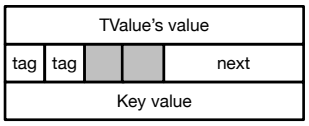

Figure 4. Layout of an element in the hash part of a table

we can see, it wastes only 2/24 in padding, so it is already reasonably efficient regarding memory.

The array part of a table, however, is stored directly as an array of `TValue`. As arrays can be quite large, the padding in `TValue` can result in significant memory waste for the whole program. So, it is worth considering alternative representations for this array. In the next section, we will explore some alternatives.

4 Alternative Implementations

In this section, we will consider three alternative implementations for the array part of a table. For each alternative, we will discuss its pros and cons, but we will postpone a direct comparison among them to the following section.

4.1 Reflected Arrays

A common technique to avoid padding in arrays is the use of *parallel arrays* [1]: Instead of a single array of structures, we use one individual array for each field in that structure. Figure 5 illustrates this technique.

A major drawback of parallel arrays for Lua is that it needs two pointers in the table header, instead of one. For large arrays, this overhead is irrelevant, but for programs that create a myriad of small arrays—or even tables with no array part—this increase in the base size of a table can be significant. Moreover, each access to an entry needs one extra memory access, to retrieve this extra array pointer.

To avoid these drawbacks, we propose an implementation that we call *reflected arrays*. The idea is to represent the array of `TValue` by an inverted array of values followed by an array of tags: Figure 6 clarifies this idea. A single pointer pointing to the junction of the two arrays allows the code

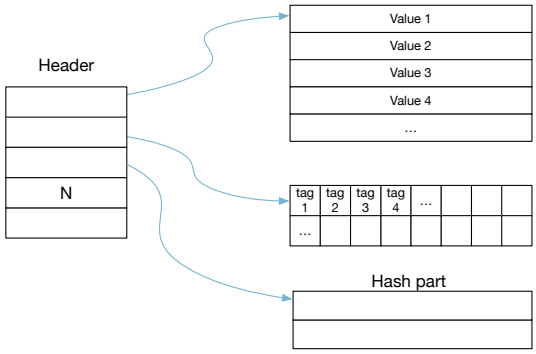

Figure 5. Use of parallel arrays to avoid padding

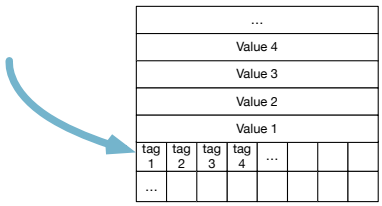

Figure 6. A reflected array

to access both, the tags with non-negative indices and the values with negative indices.

This layout completely eliminates padding without introducing any extra memory overhead, even for small arrays: An array of size 1 uses the same two words as the basic implementation; an array of size 2 already saves one word. Also, the code to access elements is quite simple. If we declare the array as a pointer to values, then the expression `arr[-idx - 1]` accesses the value at index `idx`, while the expression `((char*)arr)[idx]` accesses its corresponding tag.

A drawback of this layout is its bad locality for random access, as each value is stored quite apart from its respective tag. Another drawback concerns resizes. Unlike a regular array, we cannot simply reallocate a reflected array to resize it. If the array grows, we need to move up its elements to keep them centralized. If the array shrinks, we cannot use a reallocation at all, since it will always erase some data that should be kept in the final array. Instead, we must first allocate a new array, move the data to its new location, and then free the original.

Another peculiarity of this structure is its use of interior pointers, that is, pointers to the middle of heap-allocated blocks. That is correct regarding ISO-C, but it can be problematic for some tools, such as conservative garbage collectors and Valgrind. (If such a block is not deallocated, Valgrind may report it as a "possibly lost" block.)

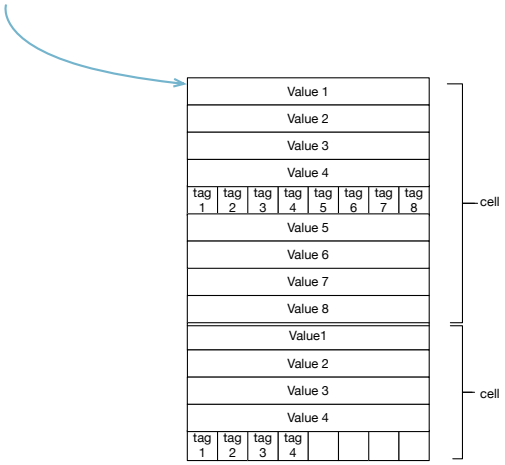

Figure 7. Cells in an array of cells

4.2 Cells

Our next implementation tries to avoid the drawbacks of reflected arrays. To improve locality, it interpolates the array of tags into the array of values. More specifically, for each block of `NM` values, where `NM` is the size of a value, it inserts a word with `NM` tags. Figure 7 shows this arrangement. (As we discussed, the figures assume that values have 8 bytes [64 bits], to simplify the exposition.) The block of tags can be inserted anywhere in the cell; different positions bring different costs, as we will discuss later.

To implement this structure in C, we declare the array as being composed by unions of values and arrays of tags:

```
union ArrayCell {
  char tag[NM];
  Value value;
};
```

We also define a constant `NMTag`, which can be any value between 0 and `NM` (both inclusive), to be the position in the cell where to insert the block of tags. Figure 7 uses `NMTag=4`.

To address the value at index `idx`, we can use the following expression:

```
arr[idx + ((idx + (NM - NMTag))/NM)].value
```

The first summand is the number of values before the one we are indexing, while the second is the number of tag words. If we simply divide `idx/NM`, that gives the number of complete cells behind `idx`; that is also the number of blocks of tags behind `idx` when `idx` is in the first part of a cell. However, if `idx` is in the second part—that is, after the block of tags of its own cell—we need to skip one more word. The addition of `(NM - NMTag)` to `idx` in the dividend does that correction.

Let us see some examples to make things clearer. We will assume that `NMTag` is 4. Suppose we want to access the value at index 10. That value lies in the first part of the second cell. ("First part" are the values before the block of tags.) Its index in the array is 11, as it lies in the third slot of the second cell. Note, in the second summand, how the dividend `(10 + 4)` does not add to 16, and so the division rounds to 1. Overall, we skip 10 values plus 1 block of tags to arrive at index 11.

Suppose now we want to access the value at index 22. That one lies in the second part of the third cell. Now we have to skip 22 values plus 3 blocks of tags. The division `(22 + 4)/8` correctly rounds to 3, thanks to the addition of 4 (`NM - NMTag`).

To address the tag of the value at index `idx`, we need a little more work. The following expression does the trick:

```
arr[(idx/NM * (NM + 1)) + NMTag].tag[idx%NM]
```

The division gives the cell where the value lives; the product translates that to array indices, as `NM+1` is the number of elements in each cell. The addition of `NMTag` moves the index from the beginning of a cell to the block of tags inside the cell. Once the correct block is selected, the remainder gets the specific tag inside the block.

In "reasonable" architectures, where `NM` is a power of two, all operations in both expressions are cheap: The divisions become shifts, the remainder becomes a bitwise-and, and the product becomes a shift followed by an add. Nevertheless, each access involves several more cheap instructions than were needed in the previous implementations.

If the size of the array is not a multiple of `NM`, the last cell in an array will be incomplete. In that case, we do not need to allocate the whole cell. However, we still need to allocate its block of tags. Therefore, the value of `NMTag` limits the minimum size for an incomplete cell. If we consider only locality, the ideal value for `NMTag` would be 4, putting the tag block right in the middle of its respective values. However, that value implies that an array of one or two elements would still need five entries. To avoid any waste in small arrays, `NMTag` should be at most one. That of course decreases locality: In the worst case, with `NMTag` equal to zero, a value can be 56 bytes (7 words) apart from its tag.

4.3 Cell-0

If we follow the route of avoiding memory waste, it is worth considering the special case of `NMTag` equal to zero, that is, the layout where each block of tags immediately precedes its corresponding values.

It seems reasonable to expect this special case to be somewhat simpler than the general case. In fact, this layout allows a simpler and, in our view, more elegant implementation. Regarding the C code, we can implement the entire cell as a single C structure:

```
struct ArrayCell {
  lu_byte tag[NM];
  Value value[NM];
};
```

Now, the array part is an array of `ArrayCell`. To address the tag of the value at index `idx`, we can use the following expression:

size	basic	refl.	cells	cell-0
8M	132M	75M	75M	75M
64M	1050M	591M	591M	591M
256M	4.19G	2.36G	2.36G	2.36G

Table 1. Memory use for arrays

```
array[idx / NM].tag[idx % NM]
```

The access to a value is similar, only changing the field name from `tag` to `value`. Once again, in "reasonable" architectures, the division becomes a shift and the remainder becomes a bitwise-and.

The memory layout is exactly the same as for generic cells when `NMTag` is zero, despite the differences in the C code. Each access still involves a multiplication by `NM+1`, but it is produced by the compiler, as this is the size (in words) of each array element. As we already mentioned, the main gains from this implementation are simplicity and elegance. A by-product of the simplicity is the easiness of verifying the correctness of the implementation.

As the reflected-array structure, this data structure never uses more memory than the basic implementation: An array of size 1 uses the same 2 words; an array of size 2 already saves one word.

5 Evaluation

In this section, we evaluate the different alternatives regarding the criteria presented in the introduction. We evaluate four implementation: the basic one, using an array of `TValue`, reflected arrays, cells, and cell-0.

For the evaluation, we implemented two very simple C programs, one that creates an array of a given size and then traverses it linearly, and another that creates an array and then does a long sequence of (pseudo)random accesses. These two tests present quite different behaviors regarding locality. We also modified a Lua interpreter to use each of the presented data structures, and tested each resulting interpreter in five standard Lua benchmarks.

The entire code used in these tests is available at the following link: www.inf.puc-rio.br/~roberto/docs/array24.zip.

5.1 C benchmarks

We ran the C tests in an Ubuntu machine with kernel 5.4.0-182-generic and an Intel Core i7-4790 CPU @ 3.60GHz, 8 GB, compiled with gcc 9.4.0, optimized with -O2.

Table 1 shows the memory use for the C tests, while Table 2 shows their run-times. Memory use is the maximum resident set size of the process, as reported by `time`. The times were collected with `perf`, with 10 repetitions for times bellow 1 sec. and 5 repetitions for longer runs. (The variance is given by the number of significant digits.)

The memory results only corroborate our analysis: All alternatives use approximately 9/16 bytes of the current implementation. The time results, on the other hand, reflect the tradeoffs between locality and simplicity of the various implementations.

For linear accesses, all alternative implementations outperform the current one, with the reverse array being a clear winner. When traversing the array linearly, the reverse array has two points of high locality, one traversing the values and the other traversing the tags. It has a small footprint, compared to the basic implementation, and a simple access code, compared both to cells and cell-0.

For random accesses, the performance of the reflected-array implementation worsens for large array sizes, but except for one point all alternative implementations still are on par or better than the basic one. (Differences up to 5% are inside the margin of error of our measures.)

To shed more light on these times, we used perf to count other events. Table 3 shows these counters for the linear-access benchmark, and Table 4 shows them for the random-access benchmark. For both benchmarks, we used a fixed size of 64M. The collected events are CPU cycles, instructions, page faults, cache references, and cache misses. CPU cycles and instructions allow us to evalute the code overhead of the alternatives; page faults, cache references, and cache misses allow us to evaluate the memory locality of the alternatives.

From the linear benchmark, we can see that both cells and cell-0 execute almost twice as instructions as the basic and the reflected implementations. In the random-access benchmark, this difference of ~1G instructions is still there, but diluted by the extra instructions from the pseudo-random generator. The page-fault counter shows the weight of memory spreading in the basic implementation, while cache misses shows its weight in the reflected-array implementation doing random accesses.

In a direct comparison between cells and cell-0, we can see that cell-0 is simpler—less instructions—but it also has worse locality—more cache misses.

5.2 Lua benchmarks

Now let us see some results for our alternative data structures applied to the Lua interpreter.

We compiled four Lua interpreters: One with the basic representation as an array of `TValue`, and one for each alternative data structure: reflected arrays, cells, and cell-0. The four interpreters were based on Lua commit 5edacafcf, available on GitHub. The changes we made to that code were extremely localized. More exactly, we changed the following macros/functions:

- Macro `getArrTag`: This macro is used in all accesses to the tag of a t-value in the array part of a table. Each macro was replaced by the corresponding code fragment we presented in the text.
- Macro `getArrVar`: This macro is used in all accesses to the value of a t-value in the array part of a table.

		basic	refl.	cells	cell-0
linear	8M	0.061 (100%)	0.038 (62%)	0.045 (74%)	0.042 (69%)
	64M	0.46 (100%)	0.28 (61%)	0.35 (76%)	0.32 (69%)
	256M	1.8 (100%)	1.1 (62%)	1.4 (78%)	1.3 (72%)
random	1M	0.032 (100%)	0.023 (72%)	0.025 (78%)	0.025 (78%)
	8M	0.30 (100%)	0.31 (103%)	0.29 (97%)	0.30 (100%)
	64M	2.5 (100%)	2.9 (116%)	2.5 (100%)	2.6 (104%)
	256M	13 (100%)	13 (100%)	11 (84%)	11 (84%)

Table 2. Runtime (in seconds and fraction of basic time) for array accesses

impl.	cycles	instr.	page-faults	cache-ref.	cache-mis.
basic	0.79G	1.1G	262K	17M	7.8M
refl.	0.53G	1.1G	147K	7.5M	3.3M
cells	0.78G	2.5G	147K	2.3M	0.89M
cell-0	0.68G	2.1G	147K	3.7M	1.4M

Table 3. perf results for linear access, size 64M

impl.	cycles	instr.	page-faults	cache-ref.	cache-mis.
basic	8.9G	5.6G	262K	131M	67M
refl.	10.9G	5.6G	147K	237M	131M
cells	9.4G	6.8G	147K	149M	87M
cell-0	9.6G	6.3G	147K	166M	103M

Table 4. perf results for random access, size 64M

Again, each macro was replaced by the corresponding code fragment we presented in the text.

- Function `resizearray`: This function resizes the array part of a table, including resizes from and to size zero, that is, creation and deletion of the array.
  In the basic (original) implementation, this function has only 4 lines of code.
  Its size grows to 16 lines, excluding comments, in the cells implementation and to 14 lines in the cell-0 implementation. In both cases, the extra complexity is due to the code for computing the size in bytes for the new array.
  In the reflected-array implementation, that function has 36 lines. As we already discussed, we cannot use `realloc` when shrinking an array; so, we opted to always allocate a new block and move the old contents to its correct place in the new block.

All interpreters were compiled with the makefile provided in the commit, using gcc 9.4.0.

With these four interpreters, we ran five benchmarks:

`matrix` does a matrix multiplication of two matrices 600x600.

`binsearch` does 10^7 binary searches over an array with 10^6 random integers.

`heapsort` sorts an array of 10^6 random numbers in the interval $[0, 1)$, using the heapsort algorithm, and then checks that the sort is correct. This workload is repeated five times.

`sieve` implements the Sieve of Eratosthenes, finding all primes up to $5x10^7$.

`n-body` is an instance of the n-body problem. It does not use arrays at all. We included it here as a "control group".

These benchmarks were selected for their simplicity, emphasis on array accesses (except for the "control group"), and different patterns of accesses.

Tables 5 and 6 show the benchmark results for the four versions of the Lua interpreters over these five programs. We run each test 20 times in an Ubuntu machine with kernel 5.15.0-117-generic with an Intel Core i7-1065G7 CPU @ 1.30GHz, 16 GB. Memory use is the maximum resident set size of the process, time is the number of CPU-seconds used directly by the process, both as reported by `time`.

The first thing we can check from the tables is our sanity check: As expected, the different interpreters show no significative differences, both in memory and in time, for the n-body benchmark.

The second important take is that, again not surprisingly, all alternatives bring ~40% reduction in the memory used by

impl.	matrix	binsearch	heapsort	sieve	n-body
basic	32M	19M	19M	1.0G	2.7M
refl.	19M	12M	12M	0.59G	2.7M
cells	19M	12M	12M	0.59G	2.7M
cell-0	19M	12M	12M	0.59G	2.7M

Table 5. Memory use for Lua interpreters

impl.	matrix	binsearch	heapsort	sieve	n-body
basic	5.8 (100%)	5.4 (100%)	5.8 (100%)	4.7 (100%)	2.2 (100%)
refl.	5.6 (97%)	5.4 (100%)	5.4 (93%)	3.6 (77%)	2.2 (100%)
cells	5.3 (91%)	5.7 (1.06%)	5.8 (100%)	4.7 (100%)	2.2 (100%)
cell-0	5.4 (93%)	5.5 (1.02%)	5.8 (100%)	4.6 (98%)	2.2 (100%)

Table 6. Time (in seconds and fraction of basic time) for Lua interpreters

each of the other benchmarks. Note that these benchmarks make heavy use of arrays. Programs with other characteristics may not show those gains.

The numbers presented by Table 6 are more nuanced. The first observation is that the overhead of Lua dilutes the larger differences we saw on the C benchmarks. There are some fluctuations, but in general the alternatives are not slower than the original implementation. The only exceptions are for cells and cell-0 in the binsearch benchmark. This benchmark is the one that exibits more randomness in its array accesses, due to the nature of the binary-search algorithm. Nevertheless, the overheads may be a small price—negligible for cell-0 (2%) and 6% for cells—for the savings in memory.

6 Conclusions

We have proposed three novel, simple data structures to improve the memory efficiency of arrays in Lua. According to our evaluations, all three options can reduce by ~40% the memory use of large arrays in the language, without significant impact on performance.

Among the alternatives, the extra complexity of cells seems to outweigh its slightly better locality when compared to cell-0. Similarly, the simplicity of reflected arrays seem to compensate for its worse locality, when used in the Lua interpreter. Other applications, and other benchmarks, may lead to different conclusions.

For the reflected-array structure, two peculiarities are worth noting: The first is its use of interior pointers. That is not an issue for Lua or other ISO-C code, but can be an issue for some tools, such as conservative garbage collectors or Valgrind. The second, a consequence of the first, is its extra complexity for resizing a block. Although Lua does resize its tables on demand, in most programs that operation is sparse. (Note that resizing is already a heavy operation in Lua, because of the computation of the array size.) Moreover, the two-step reallocation (using `malloc`–`free`, instead of `realloc`) is only necessary when shrinking an array, a rare event.

There is nothing particular about Lua in these data structures. Arrays of tagged unions are a quite common structure not only in interpreted languages, but in several other areas. The proposed data structures seem a good addition to the toolbox of any programmer, to be considered whenever alignment becomes a problem.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior—Brazil (CAPES)—Finance Code 001.

References

[1] Wikipedia contributors. 2022. Parallel array. https://en.wikipedia.org/w/index.php?title=Parallel_array&oldid=1109345767 [9 September 2022 09:40 UTC].

[2] Simon Marlow (editor). 2010. *Haskell 2010 Language Report.*

[3] Free Software Foundation [n.d.]. *GNU C Language Manual. 15.6 Packed Structures.* Free Software Foundation. https://www.gnu.org/software/c-intro-and-ref/manual/html_node/Packed-Structures.html

[4] Roberto Ierusalimschy, Luiz H. de Figueiredo, and Waldemar Celes. 2005. The Implementation of Lua 5.0. *Journal of Universal Computer Science* 11, 7 (2005), 1159–1176. https://doi.org/10.3217/jucs-011-07-1159 (SBLP 2005).

[5] Roberto Ierusalimschy, Luiz H. de Figueiredo, and Waldemar Celes. 2007. The Evolution of Lua. In *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, CA). ACM, New York, NY, 2.1–2.26. https://doi.org/10.1145/1238844.1238846

[6] ISO 2000. *International Standard: Programming languages — C.* ISO. ISO/IEC 9899:1999(E).

[7] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, KC Sivaramakrishnan, and Jérôme Vouillon. 2023. *The OCaml System, release 5.1.* INRIA.

[8] Robert Nystrom. 2021. *Crafting Interpreters.* Genever Benning, Chapter § 30.3 NaN Boxing, 590–601. ISBN 978-0-9905829-3-9.