Efficient “out of heap” pointers for multicore OCaml

Guillaume Munch-Maccagnoni*
INRIA

11th June 2022

1. Introduction

This paper reports an experiment with a large pages allocator for the OCaml runtime, with measured performance improvements. A large pages allocator (also known in the literature under other names: superpages, etc.) is a standard component of a memory allocator that stands between the OS and user-facing allocators (e.g. minor and major heaps) and which reserves and manages large chunks of contiguous memory. The OCaml runtime currently gives up a good amount of control by assigning this role to the system allocator. Other languages have a simple implementation from which practical lessons can be learnt (especially in terms of portability), such as the one from the Go runtime.

Control over large pages affect (traditionally) the components implementing virtual address translation (hardware-level page table, translation lookaside buffer), in particular by leveraging (hardware-level) *huge pages*. It also enables (more specifically) efficient implementations of the OCaml page table, a data structure used in various parts of the OCaml runtime to classify pointers, to distinguish for instance which blocks belong to the heaps and which ones are “out of heap” during the marking phase of the GC.

One goal was to evaluate the possible performance of a page table for multicore OCaml. Whilst I did not use original techniques, some of the results are unexpected *a priori* based on beliefs expressed in the OCaml community. In particular, the analysis shows that a page table can speed-up marking, a phenomenon which we reproduced in real-world programs.

In essence, this paper reports the good hypothetical performance, in rigorous practical terms, of embedding (borrowing) linearly-allocated values inside garbage-collected values. A companion submission to the ML workshop reports a symmetrical result: how to efficiently embed (own) garbage-collected values inside linearly-allocated values. Taken together, the broader motivation is to show the feasibility of basing *linear allocation with re-use* in languages that would still leverage state-of-art garbage collection for non-linear values.

Page table The page table in OCaml 4 stores information about the memory layout of the process at a granularity of 4kB OS pages. It has a fairly inefficient implementation on 64-bit systems using a hash table with linear probing. Its size grows linearly with the size of the heap and tends to suffer from collisions. It has for a long time been criticised for its performance overhead in 64-bit programs especially with large heaps, leading to the introduction of a compilation option called “no naked pointers” avoiding page table checks in several performance-critical locations, at the cost of stricter programming constraints. Furthermore, porting this data structure to multicore OCaml was considered impossible due to synchronisation costs.

Out of heap pointers To permit the complete removal of a page table, most forms of out-of-heap pointers have been deprecated starting from OCaml 4.11. I presented some time ago a position paper (Munch-Maccagnoni, 2020) defending opportunities for the alternative (of preserving out of heap pointers), and identifying and addressing four challenges for a more efficient page table design compatible with multicore OCaml.

One proposed application is for integrating *linear-allocation with re-use* in languages that use a GC for managing non-linear values. Linear allocation with re-use is a programming technique inspired by linear logic (Lafont, 1988; Baker, 1992), also more recently advocated and expanded under the name *“functional but in-place”* by Reinking, Xie, de Moura, and Leijen, 2021. A page table (or something to this effect to let the GC tell values apart) enables the embedding (*borrowing*) of linearly-allocated values inside GC-allocated values, in a way reminiscent of sharing from functional programming. More background on the motivations, and perspectives in link with linear allocation, are given in Appendix A.

Now, Munch-Maccagnoni (2020) did not experiment with an implementation, which is the subject of the present work.

Since, and independently, the marking loop of OCaml’s GC has been rewritten in order to leverage memory-level parallelism in the CPU using prefetching, and tightly optimised, with a main loop spanning a few CPU instructions.[1] This further challenged the prospects of an efficient page table, since it was not clear how to leverage memory-level parallelism with this additional lookup, nor whether the processor core had enough resources left during the optimized marking for any such lookup not to cause a noticeable slowdown.

In other words, doubts were raised, independently from OCaml’s own choices, that the theorised mixed memory management technique might fundamentally involve a performance trade-off; a claim that seemed to go beyond OCaml.

Other uses A large page allocator is also useful to implement support for hardware-level *huge pages,* for instance via Transparent Huge Pages (THP) in Linux. Hardware huge pages reduce the cost of cache misses by reducing the chances of TLB miss, and therefore are beneficial to both the garbage collector and the

*Guillaume.Munch-Maccagnoni@inria.fr

[1] Stephen Dolan (2021), *“Speed up GC by prefetching during marking”* ocaml/ocaml#10195

mutator in the presence of large heaps. OCaml currently has an option to use huge pages in Linux, but it is difficult to set up, and only affects the major heap, while users report large benefits of having also a very large minor heap in some use-cases. In addition, OCaml multicore has a new allocator that currently lacks this support. OCaml minor and major heaps are particularly suitable for huge pages, and the lack of proper support represented a missed optimisation opportunity.

In the future, a large page allocator or something to this effect could be useful for targets that only offer an `sbrk`-style interface to allocate memory (WebAssembly and bare metal). A page table can also let the layout of minor heaps be relaxed in OCaml 5 (they currently have to be allocated contiguously to implement an efficient *"is young?"* check); this could simplify for instance the implementation of adaptive sizing for the minor heaps (Jones, Hosking, and Moss, 2011, 7.7).

2. Experiment

The following is an abridged version of https://gitlab.com/gadmm/ocaml-large-pages-experiment, which describes the complete experiment and its results in more details.

Implementation I have implemented a best-fit allocator that manages large chunks of virtual memory (2^N bytes reserved, not allocated, per chunk), and subdivides it into smaller chunks of memory allocated on request (multiples of 2^M bytes where $M < N$). Virtual memory is reserved as needed, preferably contiguously to other chunks. I took $2^N = 256\,\text{MB}$ and $2^M = 2\,\text{MB}$ (the size of a huge page on x86-64) for the parameters in this experiment. The implementation reuses generic data structures of the OCaml runtime: it consists in two skip-lists storing the free chunks by address and by size (respectively), with coalescing. For a simple implementation, lookup and insertion have a logarithmic average time that is fitting for a data structure which remains small and is used infrequently (e.g. when the major heap is expanded).

The new page table is implemented with a 1-level BiBoP ("big bag of pages", like OCaml's 32-bit page table implementation), whose entries describe these 2^N bytes chunks; i.e. a table indexed by the $(48 - N)$ most significant bits of the addresses of the chunks. It is subject to a monotonicity constraint which is realistic for practical use-cases of the page table, as described in Munch-Maccagnoni (2020). Page protection tricks are used to allocate the table on demand (this turned out to be a secondary aspect not needed for efficiency, with several plausible alternatives). Some care and exploration were used to fit the assembly-optimised style of the marking loop.

The strategy to fit the prefetching algorithm is to simply consider that the page table entries are very likely to be in L1 cache. A back-of-the-envelope calculation indeed shows that the data needed by the marking loop (including memory being prefetched) fits on average a small portion of the L1d cache available on modern processors ($\geq$ 32KB). Now each cache line of the page table describes 2^{N+6} bytes of contiguous memory, when our allocator tries to allocate as contiguously as possible. So in practice the page table does not use more than a couple additional of lines of L1d cache. Hence the page table does not need to be prefetched itself, does not significantly affect the rate of false evictions, nor do its accesses compete with prefetching resources (e.g. line fill buffers). In this situation, depending on instruction-level parallelism, it is possible that the page-table test costs nothing.

Some auxiliary tables are used for certain kinds of entries that require a smaller granularity (e.g. static data). For these I re-used the data structures of my best-fit allocator. This improves over those which already exist in OCaml thanks to coalescing. These auxiliary tables, although efficient, are not visited inside the marking loop.

I have also implemented the synchronisation code that is needed inside the marking loop for an implementation in multicore OCaml. There are two options:

1. The synchronisation code tests if the page table entry is empty. If so, a `compare_exchange` is attempted in order to *taint* the page table entry with a special value, which in case of success indicates that the value was an actual out-of-heap pointer (otherwise the up-to-date value is loaded). Thanks to the monotonicity constraint, we know that this synchronisation path is taken very infrequently in the presence of multiple domains (at most once per domain and per 2^N-bytes page table entry; much less in practice because it is only executed in case of races). Therefore we can reason that the performance differences in the single-core benchmarks are representative of multicore. Notably, effects of code size and branch prediction are correctly taken into account.

2. More efficiently, by relying on dependency ordering on platforms with a very weak memory model (e.g. Arm, and thus stronger models such as x86), no extra synchronisation is needed. The downside is that no tainting is performed to detect when the monotonicity condition is broken. Since there is no synchronisation and no contention, the performance in single-core is again representative of multicore.

Unless indicated otherwise, this paper presents results for the first implementation.

Finally, I have implemented an option to use huge-page allocation with Linux's THP for OCaml's heaps, which I tested separately.

Benchmark The baseline of my experiment is Stephen Dolan's optimised GC implementation with prefetching (*"Speed up GC by prefetching during marking"* ocaml/ocaml#10195) rebased on top of OCaml 4.12. The new large page allocator and page table were implemented on top of it.[2] I compared the performance of the "no-naked-pointer" mode (which avoids a page table check during marking) with the legacy and the new optimised page table using synthetic benchmarks.

Care needed to be taken to compare equals for equals: even small changes in GC patterns (for instance due to changing minor heap size slightly) would have changed the results significantly due to threshold effects, and rendered the comparison meaningless.

The synthetic benchmark is inspired by the one proposed by Dolan in his experiment. It consists in allocating a large quantity of values (ranging from 800MB to 11GB) distributed randomly,

[2] At the time of the experiment, OCaml 4.12 was the latest stable version of OCaml, and the prefetching patch had not been merged yet. In addition, it was not possible to work directly with the multicore branch since it lacks prefetching (still at the time of writing this paper).

and so it constitutes a worst case in terms of cache locality. One measures the major GC duration and other statistics.

I have improved the synthetic benchmark by introducing variables such as the proportion of immediates, proportion of statically-allocated blocks (to measure the benefits of skipping during marking, an optimisation which a page table permits), randomness of the heap (to measure the effect of load stalls), and randomness of the block layout (to measure effects on branch prediction). I have instrumented the OCaml runtime to measure how fast mark slices run, how often statically-allocated blocks data was encountered, and how diverse (non cache-local) the static data visited during marking was.

With the synthetic benchmark, care was taken to control in various ways for measurement biases caused by code layout, but this effect was rarely important.

The impact of huge pages (THP) was also measured, separately from that of the page table. The two features are orthogonal from each other.

I have also obtained results on real-world OCaml and Coq workloads, however they show greater variability. Controlling for code layout effects as carefully as for the synthetic benchmark was not doable, but I obtained consistent results by running the benchmarks using different compilers and compiler options (which give different memory layouts).

The benchmarks were run using an Intel Core i7-6600U (Skylake) processor, similar to Dolan's experiment, which required some care to avoid unreliable results due to a processor bug, and later reproduced with AMD Ryzen 7 5850U and Intel Core i7-1185G7 (Tiger Lake). Unless indicated otherwise, the figures reported below are for i7-6600U. In the real-program tests (running on i7-6600U and i7-1185G7), care was also taken to eliminate CPU frequency changes and CPU thermal throttling (in the case of the synthetic benchmark, this was also considered, but actually not necessary in order to obtain stable results).

3. Results

Contrary to the expectation that a page table would have an extravagant cost with the prefetching GC, both the "no-naked-pointers" mode and the new optimised page table bring a similar speedup of about 26% in the synthetic benchmark, which measures major GC duration, compared to the legacy page table. Furthermore, the new page table consistently outperformed slightly the optimised GC in "no-naked-pointers" mode in the synthetic benchmark, albeit negligibly (by ~1% of the major GC duration, that is ~0.15% of the duration before the prefetching patch).

To understand this result, one should consider that even without a page table, a test is still performed to skip values in the minor heap. In a sense, multicore OCaml still needs a page table of some kind[3]. The new page table turned out to be more efficient than the previous check.[4]

This speedup is further improved by taking the skipping of static data into account. Surprisingly, static data was found to be encountered fairly frequently during marking in real-world OCaml and Coq workloads: between 3.7% and 7.2% of values on average were skipped thanks to the page table, with spikes at 40% during some marking slices. In terms of unique cache lines, the amount of static data skipped during each marking slice would most often be too large to remain in L1 (and often L2) cache (see Figure in Appendix C). At a ratio of 5% static data skipped, the synthetic benchmark shows a speedup of ~3% (a figure that does not take into account possible costs of additional branch mispredictions this causes).

The current page table scales poorly to large heaps, and to multicore. The new page table scales to large heaps both in theory and practice, as the figures remained consistent with very large heaps. Regarding scaling to multicore OCaml, the results concerns an implementation that already supports parallel accesses to the page table. I concluded that this single-core performance is representative of the multi-core performance by a theoretical argument.

THP brings various performance improvements, from about 10% of major GC time in the synthetic benchmark with a random heap to a negligible impact with a non-random heap. (Note that only the impact of THP during marking was measured.) It brings comparable improvements whether by using my large page allocator or by using the *jemalloc* allocator with the option "thp:always" (by preloading from the command line). So, the interested user can already get most benefits of THP by changing the command line invocation.[5] A built-in allocator has further advantages than a global system allocator setting (besides convenience): THP is only used for memory allocations that are well-suited for it. This can become relevant if the program uses non-OCaml libraries not well-suited for THP.

In a similar area, I measured noticeable benefits, in conditions of high TLB misses, of allocating memory as contiguously as possible, which could likely be attributed to improved behaviour for the MMU cache (TLB misses completing faster, see Barr, Cox, and Rixner, 2010).

With the OCaml and Coq workloads, I initially observed that the "no-naked-pointers" mode and the new page table have similar performance, indistinguishable in terms of total running time given the greater amount of variability. These real-program results are hard to exploit because they are more noisy, and micro-architectural effects might be more important, unlike our experience with the synthetic benchmark.

However more recent results with Core i7-1185G7 and using the second implementation technique for the page table check (the one using only dependency ordering for synchronisation), I could indeed observe a consistent speedup of using a page table probably due to the skipping of static data, from 1-2% (with gcc) to 4-5% (with clang) of the marking slice duration, depending on compiler options. (Remember that using different compilers and compiler options was meant as a way to control to some degree for code layout effects; this seems to confirm *some* speedup but it does not tell by how much.)

Limitations I used a synthetic benchmark that exerts either a lot of cache misses (random heap), or very few cache misses (non-random heap). Real-world programs are likely in the middle, but results between a random and a non-random heap are consistent, with the exception of the impact of THP. The synthetic

[3] One that limits the way minor heaps can be allocated in order to test quickly whether a value is young, and does an expensive test to distinguish code pointers in the bytecode interpreter.

[4] One hypothesis is that an optimisation turned out to be a pessimisation. Fixing this would-be pessimisation improved the performance comparably to the new page table. An analysis in terms of costs of branch mispredictions was consistent with the effect.

[5] This is unfortunately not the case with the OCaml 5.0 allocator, which does not use the system allocator in the same way.

benchmark also falls short of simulating real-world amounts of branch mispredictions. Now the various implementations branch in similar ways, except on static data. It is yet unclear whether the benefits of skipping static data reflect in practice as strongly as in the synthetic benchmarks; this could be explained by the cost of branch mispredictions this creates.

Regarding static data usage, I have only tested OCaml and Coq on intensive workloads; it would be interesting to see how other programs visit static data during marking.

The synthetic benchmark only measures the major GC duration, while huge pages are likely to benefit the whole program. Results on varied Coq workloads measuring the benefits of huge pages and the new page table are encouraging[6].

The friendliness towards THP is due among others to memory reclamation being performed via compaction, because it frees up memory in large chunks. Without compaction, one would encounter known issues of huge pages regarding memory reclamation (whereby low amounts of fragmentation can prevent reclamation). Thus one should expect multicore OCaml, which currently lacks compaction, to be less friendly towards THP once memory reclamation is implemented.

Thanks Thanks to Stephen Dolan for invaluable discussions, patient explanations and advice. (Any error or opinion left in this work is mine.)

A. Naked pointers & linear allocation with re-use: motivations and perspectives

The usage of out-of-heap (naked) pointers in OCaml 4 falls broadly into two categories.

Interoperability & backwards-compatibility The first usage is for interoperability, where the concern is mainly backwards-compatibility. When interacting with foreign code (for instance via the OCaml C FFI), using out-of-heap pointers is the most straightforward way to refer to foreign data inside OCaml data. This usage is now considered inferior to alternatives using some form of wrapping, essentially due to an unsoundness involving the recycling of memory by the system allocator.

For various reasons including unsoundness, most forms of naked pointers have been deprecated in OCaml 4.11. Starting with OCaml 5, the program can crash or do worse things if the GC encounters such a pointer. Compatibility risks have been evoked early on with the deprecation, according to a public comment dating back from 2015.

As we recently discovered, the use of naked pointers has been advocated for a long time in a tutorial[7] that has been recommended by several popular resources from the OCaml community over the years[8].

Searching the opam repository for one specific usage pattern suggested by this tutorial reveals a large sample of affected libraries (some of which have perhaps been fixed since the deprecation was announced). A careful evaluation to eliminate false-positives was necessary, and there is of course no guarantee of exhaustivity. (List in appendix B)

Without entering the details of why this usage of out-of-heap pointers was unsound, one can ask under which set of conditions it would materialise, given that it was apparently not an obstacle to the development of many libraries using this technique. The way the unsoundness manifests itself depends in fact on the implementation details of the system allocator; reports of this issue occurring in practice (which it does) involve specific usage patterns. As matter of fact, no practical or theoretical unsafety had been reported to the author of the tutorial during its existence.

One hypothesis is that this technique has been working reliably in many situations. This creates of course an interesting and delicate situation for backwards-compatibility. This is a usage which one reasonably wants to deprecate due to its unsoundness, yet:

1. It is possible that a good proportion of programs using these libraries is actually accidentally bug-free.
2. The existence and usage of workarounds have been reported for situations where the bug occurs (namely, forcing the GC to run before large amounts of foreign data structure are freed).
3. Some libraries that manage their own memory mapping (such as Ancient, described later) took other measures to avoid this bug.

For the sake of backwards-compatibility, a "naked pointer detector" has been proposed in OCaml 4.12, which warns when the GC encounters a naked pointer. But while this tool can prove that a program creates dangerous out-of-heap pointers, it does not prove that a program nor its dependencies do not create any. (We also do not know how well it works for users in practice.) In contrast, delicate backwards-compatibility situations in relationship to bugs are not unfamiliar to major operating systems and industrial programming languages,[9] where it can be preferred to keep a bug in place (along with the work-arounds existing in user code) over breaking user code.

But the new page table presents a better alternative, which is to make these previously-dangerous uses safe, even in the event where the choice was made to keep discouraging the use of such naked pointers for new programs. Concretely, this would take the form of an option to reserve the address space in advance for the remainder of the execution[10]. This option could be enabled anytime by either of the library, the final program, or even just the user of the program in case of legacy programs.

Prior to this tutorial and its publicity being brought to my attention, cases of high-profile code that must be changed were already known[11], including the innocuous usage of `NULL` as a special value by some libraries[12]. In another case, an indus-

[6] Thanks to Pierre-Marie Pédrot and Ralf Jung for running Coq benchmarks on my branches.

[7] Florent Monnier, *"How to wrap C functions to OCaml"*, 2007-2020, https://web.archive.org/web/20200223115730/http://www.linux-nantes.org:80/~fmonnier/OCaml/ocaml-wrapping-c.html (an updated version which no longer advocates the use of naked pointers is available at http://decapode314.free.fr/ocaml/ocaml-wrapping-c.html)

[8] At least until 2020 on the website http://ocaml.org and in the popular book Real World OCaml (https://www.realworldocaml.org/).

[9] Several sources advertise the OCaml language as an "industrial-strength" language, or even a "systems" programming language.

[10] For some practical reasons, this behaviour is not desirable as a default.

[11] Some whose fix is still pending as of September 2022, such as the LLVM bindings and *native_compute* in the Coq proof assistant.

[12] An interesting example of the usefulness of having a value different from every valid OCaml value has been given, but the examples actually found in the wild were much less interesting.

trial user has resigned to maintaining a patch to OCaml for an indefinite amount of time for a key performance use-case, after a proposed alternative solution was rejected from the main compiler branch.[13]

Eventually, the development of the "no-naked-pointers" mode, followed by the full removal of the page table, a data structure used pervasively in the OCaml runtime, the introduction of the "naked pointer checker", etc., have proved to require lots of efforts, some compromises, and created unexpected issues. It also caused a lot of adaptation work for some users.

The present work demonstrates that re-implementing the page table with standard and well-tested techniques (a BiBoP based on a large pages allocator, the latter of which is useful for other things as well) was a simpler, less risky and just as performant alternative.

Linear allocation The second usage of out of heap pointers is more involved than storing foreign pointers, and concerns experimental ways of mixing garbage-collected memory with dynamically-allocated memory, for instance excluding some memory from the GC. Either for performance and resource-usage reasons (the *Ancient* library, which implements a persistent heap outside of the major heap) or for incompatible ownership patterns such as sharing data between processes (*OCamlnet*, an approach which might be revisited for some use-cases once the limitations of a stop-the-world collector will be felt). Dynamically-allocated and -deallocated memory such as Ancient and OCamlnet are not supported without a page table or something to this effect[14].

I have in mind two experiments that would rely on the page table. The first one is to prototype a version of Ancient that performs hash-consing, specialised for the needs of the Coq proof assistant, inspired by an experiment by Pédrot which has previously shown promising gains.

The second one, where I believe my experiment is of general interest, is about linear allocation with re-use. It should now be possible to experiment with measuring benefits of this technique alongside a garbage collector to manage non-linear values.

Linear allocation with re-use is an old idea (Lafont, 1988; Baker, 1992), which has been found hard to apply; this is generally believed to be due to the expressiveness limitations of the linearity discipline. It is now well understood that a linearity discipline must be complemented with a way of relaxing it through some controlled form of copying (e.g. borrowing).

"Functional-but-in-place" (FBIP) programming (Reinking et al., 2021) is a variant of linear allocation with re-use, which achieves such mixing of linearity of copying by using reference counting, in a context that can accommodate the known drawbacks of the latter. (Linearity is detected dynamically using the reference count, rather than statically.) It should be possible instead to use regular tracing GC to express copying, which requires to understand how linear allocation interacts with a GC (at the level of types, but also at the machine level).

One crucial aspect seems to be the ability to treat the borrowing of linearly-allocated data structures as a homomorphism (the borrow of a list is isomorphic to a list of borrows, etc.). This polymorphism of borrowed objects can be seen at work with the Ancient library, in the way Ancient-allocated data structures can be conveniently manipulated using usual OCaml code, and mixed with usual OCaml data structures. Such isomorphisms are also suggested by an abstract model of ownership (Combette and Munch-Maccagnoni, 2018). It relies on leveraging the notion of sharing from functional programming in a new way, and for this reason it is not (yet) seen in languages such as Rust. This resource polymorphism, whereby code operate on data indifferently of the way the latter has been allocated, is, for me, what motivates the page table, prior to efficiency concerns, compared to any proposed alternative ways of handling out-of-heap pointers.

[13] See Leo White (2020), *"Add raw_data primitives to avoid naked pointers"* ocaml/ocaml#9910

[14] For static allocation, i.e. without deallocation, other tricks have been proposed to support it without a page table.

B. Some opam libraries using naked pointers (as of August 2018)

Source: https://gitlab.com/gadmm/stdlib-experiment/-/blob/master/other/async_audit/value (subject to analysis and transcription error).

- curses
- dssi
- ezsqlite
- flow_parser
- flowtype
- gles3
- glfw-ocaml
- glMLite
- glsurf
- grib
- hiredis
- lablgl
- lablgtk3
- lablgtk3-gtkspell
- ladspa
- lutin
- mm
- ocamlsdl
- odbc
- offheap
- rdbg
- spf
- sqlite3
- sundialsml
- tuntap
- utp
- vhd-tool
- xen-gnt

This was tedious and time-consuming, I stopped half-way through.

I have reported some to their author as time permitted, in which case they have been fixed since.

C. Example of static data usage

I instrumented the OCaml runtime to record data about the static data visited during marking. Figure 1 compares the amount of values skipped thanks to the page table to the total number of values seen by the GC when running OCaml (to compile the Coq proof assistant) and the Coq proof assistant (to compile its standard library and various libraries from the Mathcomp project).

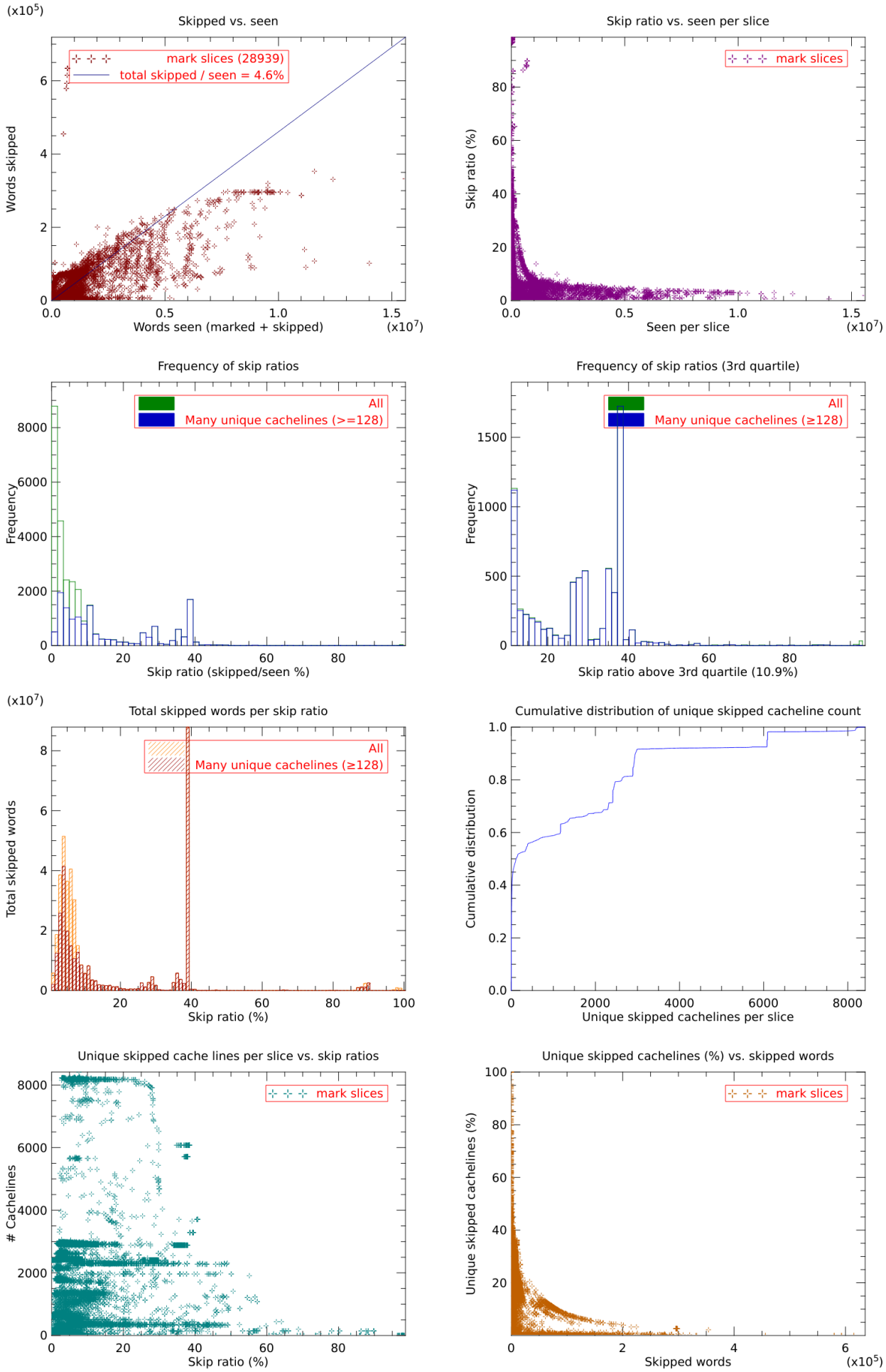

Figure 1: coq_mathcomp_no_native

References

Henry G Baker. 1992. Lively linear lisp: "Look ma, no garbage!". *ACM Sigplan notices* 27, 8 (1992), 89–98. 1, 5

Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don't Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. Association for Computing Machinery, New York, NY, USA, 48–59. https://doi.org/10.1145/1815961.1815970 3

Guillaume Combette and Guillaume Munch-Maccagnoni. 2018. A resource modality for RAII. In *LOLA 2018: Workshop on Syntax and Semantics of Low-Level Languages* (2018-04-16). https://hal.inria.fr/hal-01806634 5

Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook* (hardcover ed.). Routledge. 520 pages. 2

Yves Lafont. 1988. The linear abstract machine. *Theoretical computer science* 59, 1-2 (1988), 157–180. 1, 5

Guillaume Munch-Maccagnoni. 2020. Towards better systems programming in OCaml with out-of-heap allocation. In *ML Workshop 2020*. Jersey City, United States, 1–6. https://hal.inria.fr/hal-03142386 1, 2

Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage Free Reference Counting with Reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 96–111. https://doi.org/10.1145/3453483.3454032 1, 5