Exploring Perceus For OCaml

A Direct Comparison between Precise Reference Counting and Generational Garbage Collection.

Presented at the Higher-order, Typed, Inferred, Strict: ML Family Workshop 2023. Sep 8, 2023.

ELTON PINTO, Georgia Institute of Technology, USA
DAAN LEIJEN, Microsoft Research, USA

1 SUMMARY

The Perceus algorithm [Reinking, Xie et al. 2021] is a precise and garbage-free reference counting scheme which shows good performance in practice. However, the algorithm has only been compared against garbage-collection across different systems and languages. There is no *direct* comparison between Perceus and a garbage collector within the same system.

In this work, we take a step towards this goal. We have implemented a prototype of Perceus for OCaml 4.14.0 (which subsumes the standard garbage collector [Doligez and Leroy 1993]). Now we can directly compare the performance of programs compiled with the exact same compiler, where we only switch the backend: either using the standard generational collector, or using Perceus compilation with a reference counted runtime system. The initial performance results look quite promising, motivating futher exploration.

2 PERCEUS REFERENCE COUNTING

With Perceus style reference counting the compiler automatically inserts reference count instructions in a way that guarantees that the resulting execution is *garbage free*, where non-live objects are never retained. This is quite different from most other reference counting implementations that are *scope* based. Consider for example:

```
let () =
  let xs = biglist () in
  let ys = map inc xs in
  print_list ys
```

In scope based reference counting (like a C++ smart pointers or the Rust `Drop` trait), the references to `xs` and `ys` are dropped at the end of the scope. However, this means that the big `xs` list is not deallocated until the end of the scope – just before the `print_list` we now have two big lists in the heap. In contrast, Perceus style reference counting passes the ownership of the `xs` reference to `map`, where `map` will drop the `xs` list nodes as it iterates over it (and the same for the `ys` list in `print_list`) – we have effectively halved the memory usage. Perceus would generate no reference count instructions for `main` at all.

In contrast, as shown in Figure 1, the `map` function does the reference counting now. Here, the Perceus compilation phase inserts *dup* and *drop* instructions. In the `Cons` branch, the `rc_dup` function increments the reference count of the children `x` and `xx`, and then decrements the reference count of the parent `xs` with the `rc_drop` instruction. If the reference count drops to zero, the cell is freed and its children are recursively dropped (using a dynamic traversal at runtime).

We have implemented this transformation in the OCaml compiler and the result on `map` is shown in Figure 1b.

2.1 Drop Specialization

We can already see that the naive Perceus algorithm is a bit inefficient: if `xs` is unique (with a reference count of 1 at runtime), we first increment the reference count of its children, and then immediatedy decrement them again when `xs` is dropped. We can avoid this by inlining the drop operation, where

```
let rec map xs f =
  match xs with
  | Nil -> Nil
  | Cons(x,xx) -> Cons(f x,map f xx)

type 'a list = Nil | Cons of 'a * list
```
(a) plain map

```
let rec map xs f =
  match xs with
  | Nil -> rc_drop xs; rc_drop f; Nil
  | Cons (x, xx) ->
    rc_dup x; rc_dup xx; rc_drop xs;
    Cons ((rc_dup f) x, map f xx)
```
(b) with dup/drop insertion

```
let rec map xs f =
  match xs with
  | Nil -> rc_drop xs; rc_drop f; Nil
  | Cons (x, xx) ->
    if rc_is_unique xs then rc_free xs
    else (rc_dup x; rc_dup xx; rc_decref xs);
    Cons ((rc_dup f) x, map f xx)
```
(c) with drop specialization

Fig. 1. Perceus on the `map` function. Certain details (like those pertaining to fixing an evaluation order for correctness of reference counting in OCaml) have been omitted for clarity.

```
rc_dup x; rc_dup xx; rc_drop xs
```
becomes:

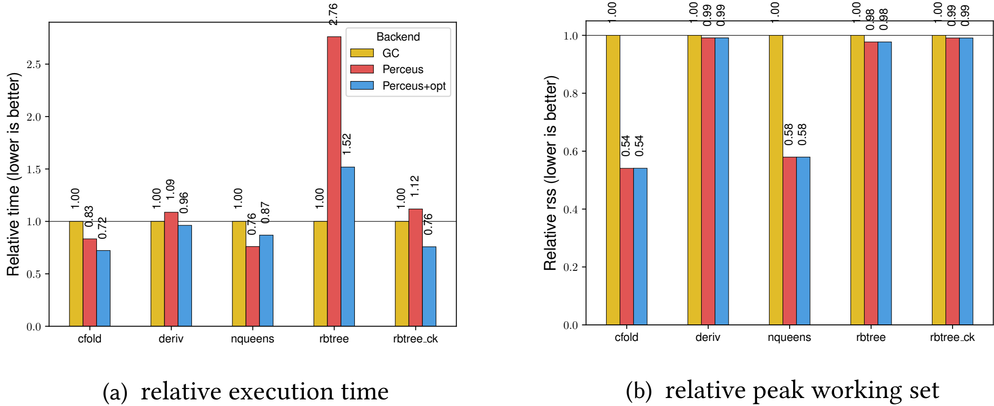
```
rc_dup x; rc_dup xx; if rc_is_unique xs then (rc_drop x; rc_drop xx; rc_free xs) else rc_decref xs;
```
We can now push down the `rc_dup` instructions into the branches and fuse them with the `rc_drop` instructions:

```
if rc_is_unique xs then rc_free xs else rc_dup x; rc_dup xx; rc_decref xs;
```
This is much better! This transformation is called *drop specialization* [Lorenzen and Leijen 2022; Reinking, Xie et al. 2021], which we have also implemented. Figure 1c shows the result of this optimization on the `map` example. There are further optimizations that can be done — in particular *reuse analysis* and *reuse specialization* can have a large performance impact [Lorenzen and Leijen 2022; Lorenzen et al. 2023b]. We plan to implement this in the future.

3 ADAPTING OCAML FOR REFERENCE COUNTING

The OCaml runtime is highly optimized for GC and fast allocation [Doligez and Leroy 1993; Sivaramakrishnan et al. 2020]. In particular, it uses a calling convention that is different from the regular C ABI, where all registers are caller-save (for fast exceptions and GC root scanning) and registers `r14` and `r15` are reserved for use by the runtime. The `r14` register contains the thread-local OCaml state (including the minor generation limit) while the `r15` register contains a pointer into the minor generation. This allows for very fast bump pointer allocation where `r15` is just incremented and compared to the minor generation limit to see if a GC is required.

However, this calling convention is not well suited for a reference counting scheme like Perceus, where we rely on standard `malloc` and `free` to allocate and free objects. Since these routines are typically implemented in C, we would have to save and restore all live registers (including `r14` and `r15`) on each call which could make the operation quite expensive. We work around this by explicitly linking with the *mimalloc* allocator [Leijen et al. 2019] – a highly performant and scalable allocator used by the Koka and Lean languages (and at the same time also at some very large services at various companies) that is relatively small (~8k loc) and well suited for runtime integration. The mimalloc allocator has the notion of a very small fast-path for allocation and freeing which we can inline those directly into the generated code, resulting in allocation needing only about 8 to 10 assembly instructions containing a single test instruction. In our prototype, we use `r15` to point to the mimalloc thread local heap to avoid thread local storage. In these specialized fast paths, we carefully use just 2 or 3 fixed registers, and only save the full register state on slow paths for generic allocation and freeing, amortizing the cost over many allocations. This approach may also work well for any future transition to OCaml 5 where we can execute the fast path on a split-stack directly [Sivaramakrishnan et al. 2020].



(a) relative execution time

(b) relative peak working set

Fig. 2. Relative execution time and peak working set with respect to the OCaml GC. 'Perceus' is baseline Perceus reference counting while 'Perceus-opt' applies drop specialization. Reuse analysis is not applied. Using a 64-bit Core i7 @2.5GHz with 16GiB memory, macOS Ventura 13.2.1. Each benchmark was compiled using the Clambda middle-end with `-O2` optimizations.

4 INITIAL BENCHMARKS

The prototype was evaluated on the same benchmark suite used in the original Perceus paper [Reinking, Xie et al. 2021] which consists of five medium sized examples that are allocation intensive. The systems compared are the standard generational GC (OCaml 4.14.0) (*GC*), plain Perceus with no optimizations (*Perceus*), and Perceus with drop specialization (*Perceus+opt*). The relative execution time and peak working set as the mean over five runs is given in Figure 2.

Even in our prototype without reuse optimizations, the Perceus backend performs close to the GC backend and is even a bit faster in four out of the five benchmarks. The `rbtree` benchmark does 42 million balanced red-black tree insertions and folds the tree at the end. The `rbtree-ck` version keeps a reference to every 10th tree in a list and thus shares many of the subtrees. For both of these, drop-specialization proves to be important since each uses complex pattern matches which benefit from the the dup-drop fusion. The balanced insertion creates many short-lived objects while rebalancing back up the tree – an ideal case for the copying collector of the minor generation, explaining the good performance of the GC variant on the `rbtree` programs. We see higher overheads in `rbtree-ck` as more trees need to be promoted due to sharing. Balanced insertion lends itself well to *reuse* of those short-lived objects [Lorenzen and Leijen 2022] and we anticipate that a future reuse specialization will improve the performance of `rbtree` further for Perceus.

Since Perceus is *garbage free*, the Perceus backend uses less memory than the OCaml garbage collector on all benchmarks, with a 40%+ reduction in the `cfold` and `deriv` programs. However, for the other three benchmarks the working set is suprisingly close which is a testament to how well the OCaml GC is able to manage its memory.

5 CONCLUSION AND FUTURE WORK

The benchmarks used in the evaluation are quite limited – too limited to draw any firm conclusions. We do believe these benchmarks represent memory intensive workloads that are quite typical in functional programming. Despite seemingly having "more" instructions to run, the Perceus backend turns out to be competitive with OCaml's GC for our benchmarks. As such, the results presented here are encouraging and motivate further investigation to arrive at a stronger conclusion.

The prototype currently supports only a limited subset of OCaml and for example does not consider exceptions and mutable references. In future work we hope to add support for these and evaluate the system on a larger set of standard OCaml benchmarks. We also plan to implement reuse analysis and specialization to further improve performance.

REFERENCES

Damien Doligez, and Xavier Leroy. 1993. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 113–123.

Daan Leijen, Zorn Ben, and Leo de Moura. 2019. Mimalloc: Free List Sharding in Action. *Programming Languages and Systems*, LNCS, 11893. Springer International Publishing. doi:https://doi.org/10.1007/978-3-030-34175-6_13. APLAS'19.

Anton Lorenzen, and Daan Leijen. Sep. 2022. Reference Counting with Frame Limited Reuse. In *Proceedings of the 27th ACM SIGPLAN International Conference on Functional Programming (ICFP'2022)*. ICFP'22. Ljubljana, Slovenia.

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. May 2023. *FP^2: Fully in-Place Functional Programming*. MSR-TR-2023-19. Microsoft Research.

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. Sep. 2023. FP^2: Fully in-Place Functional Programming. In *Proceedings of the 28th ACM SIGPLAN International Conference on Functional Programming (ICFP'2023)*. ICFP'23. Seattle,USA. Under submission. See [Lorenzen et al. 2023a] for the extended technical report.

Reinking, Xie, de Moura, and Leijen. 2021. Perceus: Garbage Free Reference Counting with Reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 96–111. PLDI 2021. ACK, New York, NY, USA. doi:https://doi.org/10.1145/3453483.3454032.

KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. Aug. 2020. Retrofitting Parallelism onto OCaml. *Proc. ACM Program. Lang.* 4 (ICFP). Association for Computing Machinery, New York, NY, USA. doi:https://doi.org/10.1145/3408995.

A EXAMPLE OF THE GENERATED CODE

As an example of the fast paths, here is the assembly generated on x64 for a part of the `Cons` branch of the `map` function:

```
                                ; enter the Cons branch, %rax = xs
    movq    (%rax), %rdi            ; store x and xx in %rdi and %rsi
    movq    8(%rax), %rsi
    movq    %rsi, (%rsp)

  ; if unique xs then free xs else (dup x; dup xx; decref xs)
    cmpl    $0, -4(%rax)            ; xs.refcount == 0 ? (0 is unique)
    jne      .L101                  ; if not unique, use slow path to dup x and xx
    leaq    -8(%rax), %r11          ; free xs (using our fast-path free which expects the arg in r11)
    call    caml_rc_asm_mi_free_hp

  ; dup f
.L100:                              ; (the slow path L101 comes back here)
    cmpl    $0, -4(%rbx)            ; if f.refcount is negative,
    jl      .L112                   ;   then use slow path for an atomic dup
    incl    -4(%rbx)                ;   else just increment the refcount

  ; call f(x)
.L111:                              ; (the slow path L112 comes back here)
    movq    (%rbx), %rsi            ; load environment of f
    movq    %rdi, %rax              ; load argument (x)
    call    *%rsi                   ; call f
  ...                           ; tailcall map
```

Here we see how the `drop` is specialized to where the fast path `if rc_unique xs then rc_free xs else ...` is inlined. Even though we do inline allocation, the freeing code is a bit too large (~20 instructions) and we use a call instead. For the `rc_dup f` we inline the common case of a non-concurrent increment of the reference count. If a reference count is negative we need to an atomic increment which is done in a slow path. (OCaml 4 does not have such concurrency but we already generate code that can handle these situations in order to compare fairly since we hope to port this to OCaml 5 in the future.)