JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Low Overhead Allocation Sampling in a Garbage Collected Virtual Machine

Christoph Jung and CF Bolz-Tereick
Heinrich-Heine-Universität Düsseldorf, Germany

ABSTRACT
Compared to the more commonly used time-based profiling, allocation profiling provides an alternate view of the execution of allocation heavy dynamically typed languages. However, profiling every single allocation in a program is very inefficient. We present a sampling allocation profiler that is deeply integrated into the garbage collector of PyPy, a Python virtual machine. This integration ensures tunable low overhead for the allocation profiler, which we measure and quantify. Enabling allocation sampling profiling with a sampling period of 4 MB leads to a maximum time overhead of 25% in our benchmarks, over un-profiled regular execution.

KEYWORDS Sampling Profiler, Allocation Profiler, Python, PyPy, Garbage Collection

1. Introduction

There are many time-based statistical profilers for various programming languages. They allow programmers to gain insights into where their software is spending time, without causing too much overhead. Examples include perf (for full-system and kernel profiling on Linux) and many language-specific tools such as py-spy[1] and VMProf[2] for Python.

On the other hand, there are memory profilers such as Memray[3] for Python and VisualVM[4] for Java. They can be handy for finding leaks or for discovering functions that allocate a lot of memory. Memory profilers typically profile every single allocation done. This results in precise profiling but larger overhead.

In this paper we present our experimental approach to low-overhead statistical memory profiling. Instead of instrumenting every allocation, we take a sample every n-th allocated byte. This allows programmers and VM implementers to gain information about where allocations happen. We have tightly integrated VMProf (PyPy's time-based sampling profiler) and the PyPy[5] garbage collector (GC) to achieve this.

Our main technical insight is that the check whether an allocation should be sampled can be made free. This is done by folding it into the bump-pointer allocator check that PyPy's GC uses to find out if it should start a minor collection. In this way the fast path with and without memory sampling is exactly the same.

This approach's overhead will be measured and quantified in Section 5. A comparison with other profilers is not done, as other profilers employ different sampling techniques, stack walk implementations or even work on different languages or language implementations, making a direct comparison difficult. Instead we compare allocation sampling with vanilla execution and time sampling.

Performing sampling on the level of the GC is also important because it gives more accurate numbers compared to source- or bytecode-level instrumentation. PyPy's just-in-time compiler (JIT) does escape analysis (Bolz et al. 2011) to remove a lot of short-lived allocations. This means, for example, that instrumenting Python bytecode to profile allocations would give the

JOT reference format:
Christoph Jung and CF Bolz-Tereick. *Low Overhead Allocation Sampling in a Garbage Collected Virtual Machine.* Journal of Object Technology. Vol. 25, No. 1, 2026.
http://dx.doi.org/10.5381/jot.2026.25.1.a16

[1] https://github.com/benfred/py-spy
[2] https://github.com/Cskorpion/vmprof-python/tree/pypy_gc_allocation_sampling_obj_info
[3] https://github.com/bloomberg/memray
[4] https://github.com/oracle/visualvm
[5] https://github.com/Cskorpion/pypy/tree/gc_allocation_sampling_obj_info_u_2.7

wrong picture. On the other hand, the JIT also introduces a lot of new allocations for itself, both during its optimization phase and also at run-time, because it uses heap-allocated frames. All these effects are correctly captured by profiling on the level of the GC.

The allocation profiler captures a call stack of a sampled allocation site. Additionally, it stores the type of a sampled object and whether it got tenured to the old generation after the next minor collection. We also store general heap statistics like total live heap size of the GC and total RSS of the process. The resulting profile can be visualized using the Firefox Profiler user interface by converting our internal format into a JSON file that the Firefox Profiler UI can present. This gives a convenient way to consume and analyze the resulting data. An example profile can be seen in Figure 1.

The contributions of this paper are:

- We integrate a sampling memory profiler tightly into PyPy's GC.
- We enrich the captured profiles by also storing the RPython-level types of the sampled objects, as well as their survival length.
- We convert[6] VMProf's internal format to that of the Firefox Profiler.
- We measure the overhead of sampling allocations.

All the work is open source, and we hope to contribute it to upstream PyPy and VMProf in the future.

The paper is structured as follows: we give some background information about the involved technologies in Section 2, then explain our technical approach in Section 3. In Section 4 we present our conversion tool that can turn the data our profiler produces into a format that can be visualized with the Firefox Profiler UI. We evaluate the overhead of sampling allocations in Section 5 and also discuss a small example problem we identified with the memory profiler. Then we discuss future ideas, related work, and finally conclude.

2. Background

In this section we explain the details necessary to understand the technical contributions of our paper to make the writing self-contained. For more general information about garbage collection we refer the reader to the relevant literature (Wilson 1992; Jones et al. 2023).

2.1. PyPy and its GC

PyPy[7] is an alternative Python implementation (Rigo & Pedroni 2006). Written in RPython (Ancona et al. 2007), it features a meta-tracing just-in-time compiler (Bolz et al. 2009) that can give significant performance gains for pure Python programs over the default CPython implementation.

Another major difference to CPython is that PyPy does not use reference counting for memory management but a generational incremental collector. That means there are two spaces for allocated objects, the nursery and the old-space. Freshly allocated objects will be bump-pointer allocated into the nursery. When the nursery is full at some point, a minor collection is performed. Then all surviving objects will be tenured, i.e., moved into the old-space. The old-space is much larger than the nursery and is collected less frequently and incrementally, using a mark-and-sweep approach.

2.2. Bump-Pointer Allocation in the Nursery

The nursery is a small continuous memory area (typically a few megabytes in size) that utilizes two pointers to keep track of the start and end of the free space available for allocation in it. They are called `nursery_free` and `nursery_limit`. When memory is allocated, the GC checks if there is enough space in the nursery left. If there is, the `nursery_free` pointer will be returned as the start address for the newly allocated memory, and `nursery_free` will be moved forward by the amount of allocated memory. This is the fast path of allocation in a nursery, and it is very efficient: if there is space in the nursery, allocation takes only a couple of instructions. Note that there is only one nursery, into which every thread allocates.

At some point there won't be enough space left in the nursery to fulfill an allocation request. Then `collect_and_reserve` is called to start a minor collection and allocate afterwards. A nursery collection will move all surviving objects into the old-space so that the nursery is empty, and the requested allocation can be made. Figure 2 (there is enough space in the nursery) and Figure 3 (not enough space, need to collect) illustrate this process. Nursery allocation is shown as pseudocode in Listing 1.

Pseudo-code for `collect_and_reserve` is shown in Listing 2. It triggers a minor collection and sets the `nursery_free` according to the allocation size and returns the start of the nursery afterwards.

2.3. Large-Object Allocations

The nursery is usually no more than a few megabytes in size. If an allocation size exceeds a certain threshold, it will therefore be allocated in a separate large-object space (which is collected with a mark-sweep approach). Those objects are allocated into a separate memory area outside of the nursery and old-space, thus don't use the nursery bump-pointer logic for allocation.

2.4. VMProf

VMProf[8] is a statistical time-based profiler for PyPy. VMProf samples the call stack of the running Python code a user-configured number of times per second. By adjusting this number, the overhead of profiling can be modified to pick a trade-off between overhead and precision of the profile. In the resulting profile, functions that run for a long time stand out the most; functions with shorter run time less so. VMProf was developed by PyPy developers for profiling PyPy as 'normal' CPython profilers don't work well with PyPy or have a lot of overhead.

[6] https://github.com/Cskorpion/vmprof-firefox-converter/tree/allocation_sampling_obj_info
[7] https://pypy.org
[8] https://vmprof.readthedocs.io/en/latest/

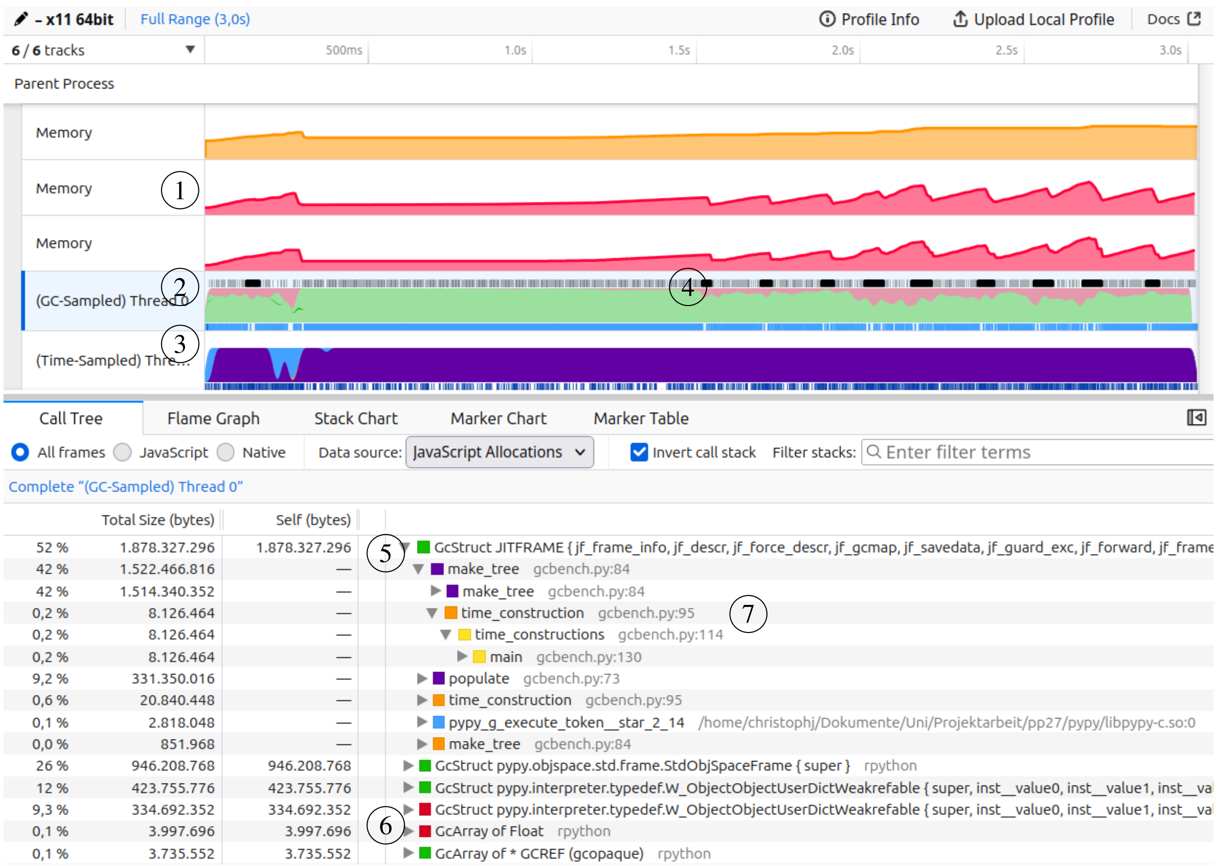

Figure 1 Firefox Profiler Call Tree View showing VMProf data. The three memory tracks (1) show different kinds of memory statistics. There are two tracks with stack samples, both allocation samples (2) and time-based samples (3). The thin black vertical marks (4) above the allocation sample track are the points where minor collections happen, the wider rectangles are (incremental) major collections. In the lower half, the call-tree pane is active, showing inverted call stacks. The top stack frames (5) and (6) show the type of object that triggered the corresponding allocation samples. The color of each type frame indicates how quickly the object died. If the type name has a green square (5) the object died before it was tenured; if it's red (6) it survived at least one minor collection. The frames below the top frame show (7) show the stack of Python frames that caused the allocation.

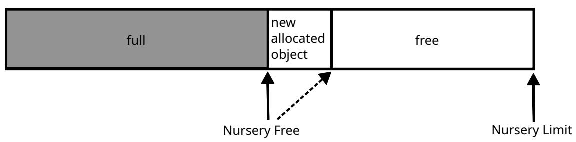

Figure 2 Nursery allocation, fast path taken

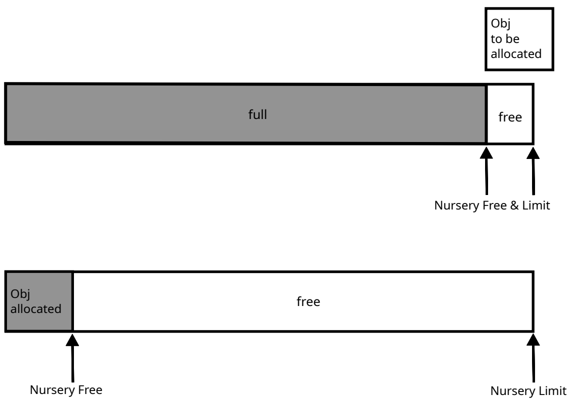

Figure 3 Nursery allocation, but the nursery is full (upper picture). Therefore, we perform a minor collection, which evacuates the nursery, and the newly allocated object can be placed at the beginning (lower picture).

```
def allocate(size):
    # result is a pointer into the nursery, obj will be allocated there
    result = gc.nursery_free
    # Move nursery_free pointer forward by size
    gc.nursery_free = result + size
    # Check if this allocation exceeds the nursery
    if gc.nursery_free > gc.nursery_limit:
        # If it does => collect the nursery and allocate afterwards
        result = collect_and_reserve(size)
    return result
```

Listing 1 Pseudo-code for allocation function of PyPy's GC

```
def collect_and_reserve(size):
    gc.minor_collection()
    result = gc.nursery_free
    gc.nursery_free += size
    return result
```

Listing 2 Pseudo-code for collect_and_reserve

```
def allocate_with_sample_slow(size):
    # result is a pointer into the nursery, obj will be allocated there
    result = gc.nursery_free
    # Move nursery_free pointer forward by size
    gc.nursery_free = result + size
    # Check if this allocation exceeds the nursery
    if gc.nursery_free > gc.nursery_limit:
        # If it does => collect the nursery and allocate afterwards
        result = collect_and_reserve(size)
    # new code:
    gc.allocated += size
    if gc.allocated >= gc.sample_n_bytes:
        vmprof.sample_now()
        gc.allocated = 0
    return result
```

Listing 3 Pseudo-code for an inefficient way to implement allocation sampling

2.5. Firefox Profiler UI

The Firefox Profiler[9] is a tool for analyzing the performance of Web-based applications. Profiles from other profilers can be imported into the Firefox Profiler. The UI features multiple ways of visualizing profiled information, like call trees, flame graphs, stack charts, assembly- and source-code view, a zoomable timeline, and more. By using the vmprof-firefox-converter, it is possible to view VMProf profiles with the Firefox Profiler's user interface.

3. Technical Details of our Approach

3.1. Sampling from a conceptual point of view

We want to sample every n-th byte that is being allocated, which is configured by the parameter `sample_n_bytes` by the user. This parameter is called the *sampling period*, the inverse of the sampling frequency. To do that, we want to keep track of how many bytes have been allocated since the last sample in a new variable `allocated` in the GC. Every time we allocate, we add the number of bytes allocated to that number. If the number `allocated` exceeds `sample_n_bytes`, we want to sample the allocation. To achieve this, we could change the allocation logic to the one in Listing 3.

However, this approach has a big downside. Specifically, it will make the fast path of allocation roughly twice as expensive, no matter how large `sample_n_bytes` is. Given that allocation is an extremely common operation in PyPy and given that it is currently very fast, we wanted to avoid that. PyPy's GC manages to achieve a peak allocation rate of about 11 GB/s on the benchmark machine (see Section 5.1).

To improve this, we can observe that there is already an addition of `size` in the `allocate` function and a comparison with a limit, the `nursery_limit`. We want to reuse these for also keeping track of whether we need to sample or not.

[9] https://profiler.firefox.com/

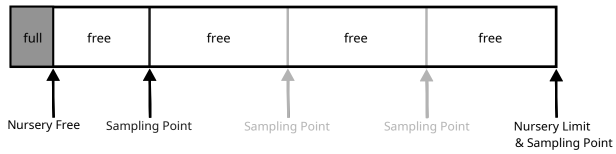

Figure 4 Nursery with Sample Points

3.2. Sampling efficiently

We want to reuse the check for whether the nursery is full in the allocation fast path to perform a dual-function. The check is then used both to check whether we should start a minor collection or whether we should take an allocation sample.

Usually, when there is not enough space left in the nursery to fulfill an allocation request, the nursery will be collected and the allocation will be done afterwards. We reuse that mechanism for sampling by introducing a new pointer called `sample_point`, which points into the middle of the nursery. It is initialized as:

$$\texttt{sample\_point} = \texttt{nursery\_free} + \texttt{sample\_n\_bytes}$$

At this point there are 0 bytes allocated since the last sample, so we can rewrite this to:

$$\texttt{sample\_point} - \texttt{nursery\_free} = \texttt{sample\_n\_bytes} - 0$$

Because `nursery_free` gets incremented every time an allocation is performed, the following *invariant* is maintained by `allocate`:

$$\texttt{sample\_point} - \texttt{nursery\_free} = \texttt{sample\_n\_bytes} - \texttt{allocated}$$

From now on, we call that number '`bytes_until_sample`'. If `bytes_until_sample` becomes negative, a sample needs to happen. If we now set the `nursery_limit` to the (lower) sample point instead of the real `nursery_limit` (which we will call `nursery_top` from now on), we can use the check `gc.nursery_free > gc.nursery_limit` to check whether a sample needs to be taken. Thus, we have achieved the goal of leaving `allocate` in its original efficient state.

Figure 4 is an illustration of what the nursery with a `sample_point` may look like. It is a conceptual simplification as only one `sample_point` exists at any given time. After we created the `sample_point`, it will be used as `nursery_limit` as opposed to the actual end of the nursery.

There are now two different situations in which `nursery_free` can exceed `nursery_limit` in the bump-pointer mechanism in `allocate`. Therefore, `collect_and_reserve` must check whether we're really out of nursery space and must collect the nursery or if it was a `sample_point`, and we need to take an allocation sample instead. If the latter, `collect_and_reserve` will call VMProf directly to trigger a stack sample. Then it computes the next `sample_point` by adding `sample_n_bytes` and sets it as `nursery_limit` if it fits into the nursery. Otherwise, it sets the `nursery_limit` to `nursery_top`. Listing 4 shows pseudocode for these mechanisms.[10]

```
def collect_and_reserve(size):
    # Check if we exceeded a sample point
    if gc.nursery_limit == gc.sample_point:
        # Sample and move sample_point forward
        vmprof.sample_now()
        gc.sample_point += sample_n_bytes

        # Set sample point as new nursery_limit
    if it fits into the nursery
        gc.nursery_limit = min(gc.sample_point,
    gc.nursery_top)

        # Is there enough memory left inside the
    nursery
        if gc.nursery_free <= gc.nursery_limit:
            # nursery_free was already
    incremented in allocate, thus we need to
    substract the object's size
            result = gc.nursery_free - size
            return result
    # Normal collect_and_reserve from here on
    ....
```

Listing 4 Pseudocode for modified `collect_and_reserve` function

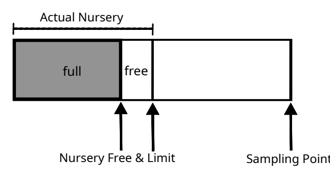

Figure 5 Sample Point outside of Nursery

3.3. Sampling period bigger than the nursery size

If we want to profile long-running applications, we might want to set the `sample_n_bytes` parameter to values that are bigger than the size of the nursery. In this subsection we will work through what needs to happen when setting `sample_point` to a point outside the nursery. If that is the case, it can't be used as `nursery_limit`, because otherwise `allocate` wouldn't notice that the nursery is full. But we still need to track somehow after how many minor collections the samples should happen. Figure 5 illustrates that scenario; `sample_point` is outside the nursery and is therefore not used as `nursery_limit`.

To track when the next sample should happen, we need to make sure our `bytes_until_sample` invariant from Section 3.2 is maintained. Since a minor collection moves the `nursery_free` after evacuating the nursery, we need to move the `sample_point` by the difference of the old value of `nursery_free` and the new value before finishing the minor collection. Figure 6 shows how this effectively moves the `sample_point` to the left. If that point is now inside the

[10] The pseudocode in Listing 4 is somewhat simplified. It does not correctly deal with the situation where an object is bigger than the sampling period. Such an object needs to be sampled more than once.

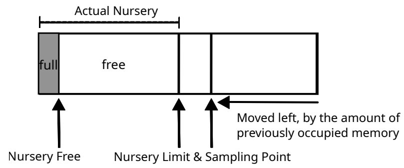

Figure 6 Sample Point moved left after Minor GC

nursery, we will use it as the `nursery_limit`. If not, the next sample happens after at least another minor collection.

3.4. Sampling Out-of-Nursery Allocations

In Section 2.3, we introduced large-object allocations. Objects that are too big for the nursery will be allocated in a separate large-object space, meaning they don't pass the nursery bump-pointer logic and thus don't get sampled there. Big objects are a crucial part of most programs allocation behavior, therefore it's necessary to sample them too.

We can reason about them by taking a look at the `bytes_until_sample` invariant again. Since `nursery_free` doesn't move for an out-of-nursery allocation, we can make sure that the invariant is maintained by changing the `sample_point`, and moving that to the left by the allocated size. Then we can check whether `sample_point` is less than `nursery_free` to find out whether we need to sample the out-of-nursery allocation. If yes, we sample and then add `sample_n_bytes` to `sample_point`. In this way, the invariant is maintained, and we don't need a separate mechanism for deciding whether to sample out-of-nursery allocations.

Listing 5 shows pseudocode for the sampling logic in `allocate_out_of_nursery`. It is slightly simplified compared to the real code, because the real code needs to take the possibility into account that the allocated object is so huge, that it needs to be sampled more than once.

Changing `allocate_out_of_nursery` in this way means that out-of-nursery allocations get slightly more expensive. However, they are rarer and anyway significantly slower than nursery allocations, because they call a much more complicated allocation function and do extra bookkeeping.[11]

3.5. Type and Survival Profiling

Every sampled allocation stores the current call stack into the profile, to find out where in the Python program the allocation was performed. We would also like to store some extra information to know what kind of object was being allocated. In addition, we want to know whether the object survived at least a single minor collection, or whether it died as a young object.

Both of these pieces of information are not available at the point where the stack sample is taken. The type of the object is stored only after the allocation as part of the initialization of the object header. And the survival information is only available at the next minor collection.

[11] Just as above, the pseudocode is somewhat simplified and does not deal with individual allocations that are larger than the sampling period correctly.

```
def allocate_out_of_nursery(size):
    gc.sample_point -= size
    if gc.sample_point < gc.nursery_free:
        vmprof.sample_now()
        gc.sample_point += sample_n_bytes
        # Set sample point as new nursery_limit if it fits into the nursery
        gc.nursery_limit = min(gc.sample_point, gc.nursery_top)
    # now perform the allocation as usual
    ...
    return ...
```

Listing 5 Pseudocode for `allocate_out_of_nursery` function

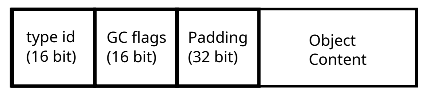

Figure 7 Object with Header and Padding

Every object (Figure 7) allocated by the GC has a header, composed of a 16-bit type ID and 16 bits for GC flags. (The padding is only on 64-bit platforms and omitted on 32-bit PyPy.)

In order to store the type ID and whether the object survived one minor collection into the profile, we maintain a list of addresses of the sampled objects that were allocated since the last minor collection (a small subset of all the allocated objects). This list will grow up to `nursery_size / sample_n_bytes` word-size entries.

At the end of the next minor collection, we can walk this list of addresses and find out whether the corresponding object survived and what its type is. Those pieces of information are then also stored into the profile.

There are no real alternatives for the address list, as we need to access every recorded address from inside the GC on the next minor collection to check if the corresponding objects survived, etc.

To be able to map type IDs of RPython types from numbers to something human-readable, we also dump a mapping of RPython type IDs to their respective names into the profile so that a UI tool like the vmprof-firefox-converter may use that to display the actual type names.

Unfortunately, the type IDs correspond to the types of objects at the RPython level, not necessarily the types of objects at the Python level. This makes understanding the profile harder for programmers that are not knowledgeable about the internals of PyPy. We plan to also sample the Python-level types in the future, or to at least give more understandable names to the RPython-level type names.

3.6. Storing GC Statistics

In addition to information about the sampled objects, we also store some general information about the state of the GC heap into the profile at every minor collection. That information is the following: The `total_size_of_arenas` tells us how much space the GC actually has to allocate tenured objects, while `total_memory_used` tells us how much of that is already occupied. But there is more to a VM than just the memory the GC manages, thus the `vmRSS` tells us how much memory PyPy consumes from the point of view of the operating system. Finally, the `GC state` stores the current major collection phase, which is one of: `scanning, marking, sweeping, finalizing`.

3.7. Correctness

While developing allocation sampling, we used various techniques for ensuring its correctness and robustness. We started with writing simple unit tests for the allocation sampling logic. However, PyPy's GC and the allocation sampling logic are entangled and have complex interactions, thus we didn't trust handwritten unit tests to sufficiently cover these interactions. Therefore, we added support for allocation sampling into the already existing randomized testing facility (fuzzer)[12] for PyPy's GC. This randomized test uses Hypothesis[13] (MacIver et al. 2019; MacIver & Donaldson 2020), a property-based testing framework for Python.

Fuzzing PyPy's GC with Hypothesis has two phases. The first phase is generating random action sequences. Those actions consist of object-, string-, or array allocations, freeing allocated objects, accessing an object, and manually forcing a minor collection. We added new actions, which are for activating and deactivating allocation sampling with a random sampling period. In the second phase, these actions are executed against the GC implementation and their intermediate results asserted. If there is a bug in the GC, e.g., freeing an object too early, the fuzzer could produce a random action sequence that leads to an error when accessing an already freed object.

When generating these actions, we also keep track of how much memory will be allocated when they are executed. With this information, we can decide if each generated allocation action should trigger a sample. When these actions are executed in the second phase, we can check for each allocation if it should trigger a sample and if it actually did. For a failing check, we then get the sequence of actions that led to the failed check, so we can trace the bug down.

Fuzzing was very helpful for getting rid of many bugs inside the allocation sampling logic, because it demonstrated interactions between sampling and garbage collection that we hadn't foreseen properly.

To give a simple example, at some point in the development process, it was possible to disable allocation sampling without enabling it first. However, doing so led to segfaults. On disabling allocation sampling, the `nursery_limit` was set to `nursery_top`. But since allocation sampling hadn't been enabled before, `nursery_top` was a non-initialized pointer. An action sequence that showed the problem was generated by the fuzzer and we then fixed the bug.

[12] https://pypy.org/posts/2024/03/fixing-bug-incremental-gc.html
[13] https://github.com/HypothesisWorks/hypothesis

4. vmprof-firefox-converter

The vmprof-firefox-converter is a tool for converting VMProf profiles into a format, which can be imported and displayed in the Firefox-Profiler. Since the Firefox Profiler is a tool for analyzing the performance of Web-based applications, some of its UI elements are JavaScript- or Web-specific. One example is the frame-filter radio buttons named 'All Frames', 'JavaScript' and 'Native' shown in Figure 1. Here we re-use the Firefox-Profiler as a UI for VMProf; unfortunately we cannot (or at least are not aware of how to) change the name or description of some UI elements. The converter was adapted throughout the development of allocation-sampling for PyPy, to work with all the new information that can be extracted from PyPy's GC. Notably, we arrange to display the allocated object type on top of the sampled call stacks. We also show additional memory timelines for the heap statistics discussed in Section 3.5, and we add markers for GC major collection phases.

Figure 1 shows an example profile. The profile consists of two separate sample timelines, one for time-sampling and the other for allocation-sampling.

The colorization[14] of those timelines represents the density of categories of the top-level frames. The top-level frames in the 'GC-Sampled'-timeline represent the sampled objects. Green colored objects were collected, while red indicates tenured objects.

The 'Memory' timelines can be identified by hovering over them. Unfortunately, we currently are not aware of how to change their names to something other than 'Memory'. [15]

5. Evaluation

5.1. Overhead

Our most important goal of introducing the allocation sampling mechanism was to make the overhead of sampling allocations configurable. In this section we evaluate the overhead of sampling allocations in dependency on the sampling period. This evaluation is preliminary, with a limited benchmark set.

To accomplish that, we are going to focus on the following questions:

- Is the overhead completely configurable via the sampling period?
- How does allocation sampling perform in comparison to time sampling?
- How large do the files written by the allocation profiler get on disc?

To answer the first question, we take a look at the following plots that show sampling period (i.e., rate) versus the overhead.

[14] The colors yellow, blue, purple, and orange specify the different types of code execution. Those code execution categories are yellow for interpreted, blue for native code (i.e., execution of a C-extension), and finally, orange and purple for jitted code (orange hinting the transition from interpreter to JIT).
[15] Note that this example profile is just for introducing the Firefox-Profiler UI and vmprof-firefox-converter and has nothing to do with the evaluation.

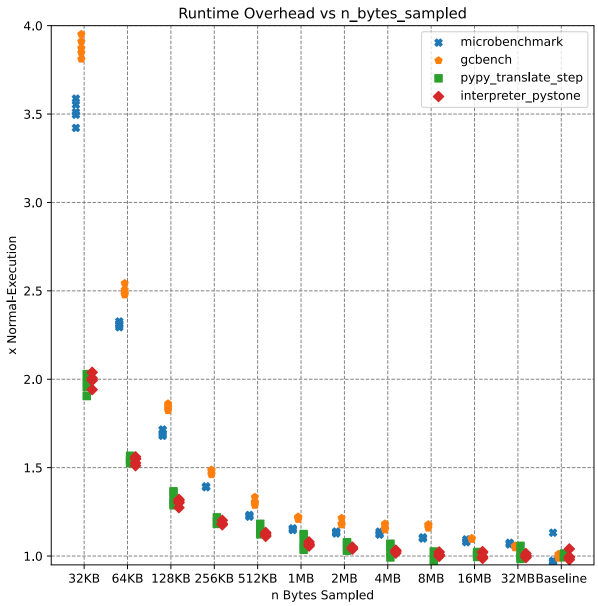

Figure 8 Allocation Sampling Period vs. Overhead

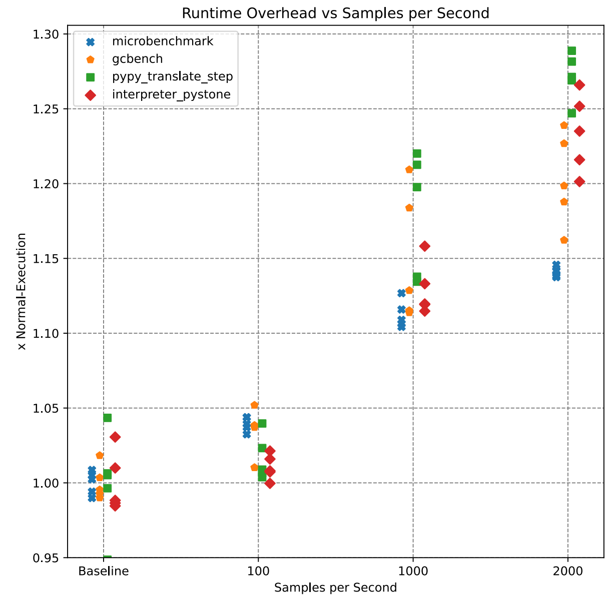

Figure 9 Time Sampling Rate vs. Overhead

Figure 8 shows our benchmark set, profiled with allocation sampling. We also measured the overhead of the previously existing time-based sampling in Figure 9 as a point of comparison. All benchmarks were executed five times on our self-built, modified PyPy with JIT and native profiling enabled. Every dot in the plot is one run of a benchmark. [16] The overhead is computed for every run of a benchmark as

$runtime\_with\_sampling / runtime\_without\_sampling$.

In Figure 8 one can see the overhead dropping when increasing the sampling period, this leads to the conclusion that the overhead is indeed (mainly) configurable by the sampling period.

Both with allocation and time sampling, it is possible to reach any amount of overhead and any level of profiling precision desired. In practice, the best approach is to experiment with different sampling periods and pick one that produces the right trade-off between precision and overhead.

To answer our second research question, whether allocation sampling performs better, worse, or the same as time sampling, we calculate the 1000 samples per second overhead for allocation sampling and compare that to time sampling at 1000 samples per second.

The average sampling rate in samples per second, overhead, and the overhead normalized to 1000 samples per second for 32KB allocation sampling are shown in Figure 10. The average sampling rate is computed as

$num\_samples / runtime\_in\_seconds$

and the 1000 samples per second normalized overhead is computed as

$1 + (overhead - 1) / num\_samples * 1000$.

(The minus one results from 1 being the baseline in overhead calculation, i.e., overhead minus 1 is the additional overhead)

It's visible that the seemingly high overhead at 32KB allocation sampling in Figure 8 is a result of the big number of samples per second. Comparing the normalized 1000 samples per second allocation sampling overhead to the time-sampling values in Figure 9, one can see that allocation sampling performs slightly better. `microbenchmark` and `gcbench` have a normalized 1000 samples/s overhead of 1.05 and 1.07, while both their time sampling overhead in Figure 9 is above 1.1. The same applies for `pypy_translate_step` and `interpreter_pystone`, their (1000 samples/s) time sampling overhead is slightly worse than the allocation sampling 1000 samples/s overhead.

This leads us to the conclusion that allocation sampling does perform slightly better than time sampling. We conjecture that this is due to using Unix interrupts for time sampling and the associated context switching is disruptive to CPU pipelining and caching. With allocation sampling, there is less disruption as samples are not triggered via timer interrupts but by a direct call. We plan to investigate these effects more thoroughly in the future.

Penultimately, run time, general memory usage, and memory usage per second are shown in Figure 11. Our benchmarks display different memory characteristics; `pypy_translate_step` is long-running but has the second-lowest allocation rate of only 0.580 GBs. On the other hand, gcbench, interpreter_pystone and the microbenchmark are quite short-running, but two of them (microbenchmark and gcbench) have the highest allocation rates of 6.063 GB/s and 5.222 GB/s.

The very last important thing to keep in mind, is the size of the resulting profile. Sampling allocations in long-running code with a small sampling period `sample_n_bytes` can quickly lead to a big number of samples. The resulting profile could consume a great amount of disk space.

For example, the `pypy_translate_step` benchmark with

[16] Note that the allocation sampling and time sampling graphs are results from different executions. Even though it is technically possible to run allocation and time sampling at the same time, that was not done for the evaluation.

Name	Samples/s	Overhead	Norm. Overhead
microbenchmark	52658	3.51	1.05
gcbench	41081	3.88	1.07
pypy_translate_step	8987	1.97	1.11
interpreter_pystone	8483	1.99	1.12

Figure 10 32KB Allocation Sampling statistics

Name	Run time (s)	GB Allocated	GB/s Allocated
microbenchmark	4.44	26.937	6.063
gcbench	0.69	3.606	5.222
pypy_translate_step	250.06	144.917	0.580
interpreter_pystone	8.10	4.494	0.555

Figure 11 Benchmark Memory statistics without Sampling

allocation sampling with a 32KB sampling rate runs for about 8 minutes, resulting in a profile of about 4GB in size in VMProf's binary format. Converting it to the Firefox Profiler JSON would make it smaller, because it is more efficiently organized but harder to write. For profiling long-running programs, it is crucial to choose a sampling period that gives the right trade-off, not just between overhead and profiling precision, but also disk space consumption.

Huge profiles could lead to the assumption that disk speed has a relevant impact on profiling overhead. We assume that the most overhead arises from walking the Python call stack and especially from walking the C stack with `libunwind`[17] (native profiling). Therefore, we cannot preclude a significant correlation between call stack depth and stack walking overhead, but those are questions for further research.

All benchmarks (see Section 9) were executed on:

- Kubuntu 24.04 (Linux Kernel 6.8.0-60)
- AMD Ryzen 7 5700U
- 24gb DDR4 3200MHz (dual channel)
- SSD benchmarking at read: 1965 MB/s, write: 227 MB/s (Sequential 1MB 1 Thread 8 Queues)
- Modified PyPy[18]
- Modified VMProf[19]
- Modified vmprof-firefox-converter[20]

[17] https://github.com/libunwind/libunwind
[18] https://github.com/Cskorpion/pypy/tree/gc_allocation_sampling_obj_info_u_2.7
[19] https://github.com/Cskorpion/vmprof-python/tree/pypy_gc_allocation_sampling_obj_info
[20] https://github.com/Cskorpion/vmprof-firefox-converter/tree/allocation_sampling_obj_info

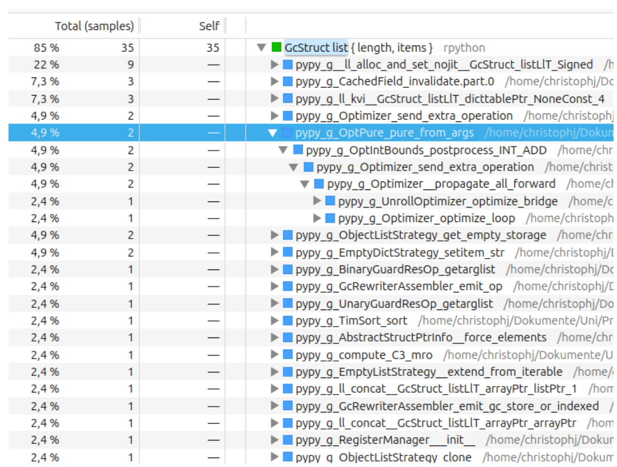

Figure 12 Firefox Profiler: Call Tree View

5.2. Case Study: Removing Unnecessary Allocations in the PyPy JIT

We didn't perform a serious attempt at doing a user study for the VMProf allocation sampling tooling. However, we found some unnecessary allocations in PyPy's JIT compiler with the tool, which we want to present here as a case study.

We profiled some SymPy[21] functions with allocation sampling. Figure 12 shows the resulting profile opened with the Firefox Profiler's call tree view filtered for functions allocating (RPython) lists.

There we encountered two functions from the optimizer of PyPy's JIT, `pure_from_args` and `postprocess_INT_ADD`. Both those functions take part in optimizing pure integer operations.

[21] https://www.sympy.org

If PyPy's JIT encounters an `INT_ADD` operation that cannot be optimized away[22], `postprocess_INT_ADD` will be called to also cache some arithmetic rewrites of that addition. E.g., if the JIT emits an operation $x = a + b$ it will remember that $x - a$ can be optimized to b from then on. This is done by calling an API `pure_from_args(INT_SUB, [arg0, arg1])`. Similar logic exists for other integer operations like multiplication or xor.

The reason why these `postprocess_...` functions of the JIT appear in the memory profile is that they all allocate a list. This list is extremely short-lived. One call level deeper, its elements are read out again, and the list is then discarded. Then, one level deeper, yet another list is built, which is discarded one function call further. Additionally, in all of these `postprocess_...` methods, there are never more than two arguments passed to `pure_from_args` inside that list.

After seeing `pure_from_args` in the memory allocation profile, we decided to rewrite it to make the list allocations unnecessary. To achieve this, we split it up into two functions, `pure_from_args1` and `pure_from_args2`, which take the elements of the list as extra arguments directly, foregoing the allocation. Then all call sites were adapted[23] to either call `pure_from_args1` or `pure_from_args2` directly, and thus saving two list allocations per `pure_from_args` call.

On its own, this optimization does not yield a measurable improvement to the warmup time of PyPy's JIT. We plan to improve the allocations in PyPy's interpreter and JIT compiler more systematically using the new profiling tools in the near future and hope that this work will lead to measurable results.

6. Future Work

In this section we want to discuss our ideas to extend and improve our work.

While the check for whether a memory sample should be taken is free, since it is folded into the GC allocation logic, actually doing stack samples is relatively expensive. We inherited this logic from the existing VMProf time-based sampling and did not try to optimize it for this paper. Reducing the overhead per sample could be done by not walking the entire stack every time, but mark walked stack frames so that for the next sample, we only need to walk up to an already marked stack frame. This could reduce the overhead because stacks typically do not change completely from sample to sample, and there is indeed a significant correlation between stack depth and overhead.

Other directions of improvement would involve moving other sources of information in the PyPy virtual machine into the profiles. PyPy has a logging interface, which is used to log GC and JIT events with a timestamp. Unfortunately, those timestamps are the clock counts read from the CPU's `TSC` (Time Stamp Counter, number of cycles since last reset) register (at least on x86/x86_64), which are not perfectly suitable for measuring time. Our modified version of VMProf on the other hand uses timestamps retrieved with Unix' `CLOCK_MONOTONIC`. This means we cannot exactly associate log events with VMProf samples. An easy fix would be to use the same timestamps for the logging facility as we do for VMProf, but this might introduce too much overhead. A better way of associating them could be to record the `TSC` with each sample so we'd get an approximate alignment of logged events and samples.

Another potential idea for future improvements would be using `ptrace` to trace system calls. That could give an insight on where/when/how much the executed code spent time opening files, reading from files, waiting for subprocesses to finish, etc.

We are also hoping to transfer some of the techniques used here for profiling PyPy to profile other RPython languages, such as the CPU simulators generated by Pydrofoil (Bolz-Tereick et al. 2025).

7. Related Work

Finally, we want to discuss related approaches for memory profiling, allocation- and object lifetime sampling.

Jump et al. (Jump et al. 2004) employ a similar bump-pointer sampling logic but use that for sampling allocation-site lifetime behavior for dynamic pretenuring in a Java VM. We sample the call stack for general performance analysis, but also record (a more primitive) object lifetime measure. Their goal is to reduce GC load through analyzing allocation-site lifetimes and directly allocating objects from long-living allocation sites to the old-space. Their results show a reduction of GC time of up to two times in some benchmarks, but also a performance degradation in other benchmarks. We implement a mechanic for analyzing performance that does not directly increase performance, furthermore introduces (fully adjustable) overhead to PyPy when used.

Harris (Harris 2000) uses a slightly different approach. He samples for allocation site and class lifetime statistics for dynamic pretenuring in Java. In contrast to our approach, Harris takes a sample every time a thread's LAB (small memory area for one single thread) overflows. Then on every first generation collection, it is recorded if the objects got tenured or died. Another key difference is that he also keeps track of when an object died inside the second generation heap, while we don't care for objects after they left the nursery.

Poireau (backtrace labs 2020) follows a more low-level approach. They intercept calls to `malloc/calloc/free/realloc` and use perf to take a sample with a certain probability for each allocated byte. Their target is to use the collected samples for debugging and leak detection, more than memory housekeeping optimizations and general performance analysis. A key difference to our work is that intercepting calls to `malloc`, etc. works with any language using malloc, etc., while our approach is limited to PyPy and (theoretically) any other RPython language with VMProf support that uses the same GC implementation as PyPy.

Memray (bloomberg 2022) on the other hand, is an instrumenting memory profiler for CPython. That means instead of sampling, Memray records every allocation; thus, it introduces non-adjustable overhead. Memray has the same use cases as our approach to allocation sampling for PyPy. That is to help developers understand where their code allocates memory, find

[22] https://pypy.org/posts/2024/10/jit-peephole-dsl.html
[23] https://github.com/pypy/pypy/commit/ef590f639e529ebe319c7d5ff8f5e03e31bcc304

leaks, and discover optimization potential. Memray has very carefully optimized its stack walking and profiling format to lower the profiling overhead as much as possible.

Guile, the GNU Scheme implementation, has a statistical memory profiler. It works by taking a stack sample every time the garbage collector runs.[24] Since a GC run is likely to be triggered by functions that allocate a lot, this effectively samples functions with high allocation rates. This approach can be seen as a special case of what we did in this paper, it corresponds to setting `sample_n_bytes` to the size of the nursery so that minor collection and sampling always happen together.

Similarly, Ocaml has a statistical memory profiling facility.[25] Unlike the work presented in this paper, it is not a full profiler on its own, but library functionality that can be used to then implement a complete profiler. The user can register a callback that gets invoked on a sampled object allocation. One big difference between their work and our is that they randomize their samples with a pseudo-random number generator. They use a geometric distribution in order to simulate a low probability for sampling every individual allocated word, instead of using a fixed rate. We plan to adopt this more careful statistical approach in our implementation in the future.

In contrast to Memray and Poireau, Sciagraph (Turner-Trauring 2024) is a commercial Python profiler and not open source. Sciagraph does both performance and memory profiling and introduces low overhead (below 5%) by taking about 20 stack samples per second and sampling calls to `malloc`. As allocation sampling for PyPy with VMProf can also be combined with time sampling, its use case is the same as Sciagraph's. However, Sciagraph's main use case is to find memory leaks and to find out which allocation sites contribute to an application's peak memory usage.[26]

Finally, Burchell et al. (Burchell et al. 2024) use late compiler phase instrumentation to profile the Java GraalVM. They insert instrumentation code only at the latest possible point in time during JIT-compilation; thus, only code that is jitted will be profiled. In contrast, our approach is sampling everything that allocates memory, which the JIT itself does a lot. Burchell et al. emphasize the problem of inlining in profiling, that is, functions not appearing in the profile because they've been inlined into the caller. Their approach is to estimate the amount of run time that is spent in an inlined function inside its caller. VMProf on the other hand, handles this by maintaining detailed information about inlining for every point in the generated machine code, such that Python stacks can be precisely reconstructed by the profiler.

8. Conclusion

In this paper, we introduced allocation sampling in PyPy's GC with VMProf. Using this tool to simultaneously do allocation and time sampling can give insight into where the program spends time, and what functions allocate much memory, leading to garbage collections. This tool is aimed at both PyPy developers and non-PyPy developers, with the target of being easy to use while introducing little overhead. Right now, the tool is still in development; we hope to merge and ship it with a PyPy release in the future.

9. Benchmarks

- microbenchmark
  - github [27]
  - *pypy microbench.py 65536*
- gcbench
  - github[28]
  - print statements removed
  - *pypy gcbench.py 1*
- `pypy_translate_step`
  - first step of the pypy translation (annotation step)
  - *pypy path/to/rpython –opt=0 –cc=gcc –dont-write-c-files –gc=incminimark –annotate path/to/pypy/goal/-targetpypystandalone.py*
- `interpreter_pystone`
  - pystone benchmark on top of an interpreted pypy on top of a translated pypy
  - *pypy path/to/pypy/bin/pyinteractive.py -c "import test.pystone; test.pystone.main(1)"*

Acknowledgments

We'd like to thank Matti Picus for his work on the PyPy and VMProf projects. Also, Max Bernstein and Andy Wingo have provided helpful comments for earlier drafts of this paper. Additionally we'd like to thank the anonymous reviewers for helping us improve the paper.

References

Ancona, D., Ancona, M., Cuni, A., & Matsakis, N. D. (2007). RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 symposium on dynamic languages* (p. 53–64). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1297081.1297091 doi: 10.1145/1297081.1297091

backtrace labs. (2020). *Poireau github.* Retrieved 2025-05-22, from https://github.com/backtrace-labs/poireau

bloomberg. (2022). *memray github.* Retrieved 2025-05-22, from https://github.com/bloomberg/memray

Bolz, C. F., Cuni, A., Fijałkowski, M., Leuschel, M., Pedroni, S., & Rigo, A. (2011). Allocation removal by partial evaluation in a tracing JIT. In *Proceedings of the 20th acm sigplan workshop on partial evaluation and program manipulation* (p. 43–52). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1929501.1929508 doi: 10.1145/1929501.1929508

[24] https://www.gnu.org/software/guile/manual/html_node/Statprof.html#index-gcprof
[25] https://ocaml.org/manual/5.3/api/Gc.Memprof.html
[26] Itamar Turner-Trauring, personal communication: https://hachyderm.io/@itamarst/114546578271860025
[27] https://github.com/Cskorpion/microbenchmark
[28] https://github.com/pypy/pypy/blob/main/rpython/translator/goal/gcbench.py

Bolz, C. F., Cuni, A., Fijałkowski, M., & Rigo, A. (2009). Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the implementation, compilation, optimization of object-oriented languages and programming systems* (p. 18–25). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1565824.1565827 doi: 10.1145/1565824.1565827

Bolz-Tereick, C. F., Panayi, L., McKeogh, F., Spink, T., & Berger, M. (2025). Pydrofoil: Accelerating Sail-Based Instruction Set Simulators. In J. Aldrich & A. Silva (Eds.), *39th european conference on object-oriented programming (ecoop 2025)* (Vol. 333, pp. 3:1–3:31). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik. Retrieved from https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2025.3 doi: 10.4230/LIPIcs.ECOOP.2025.3

Burchell, H., Larose, O., & Marr, S. (2024). Towards realistic results for instrumentation-based profilers for jit-compiled systems. In *Proceedings of the 21st acm sigplan international conference on managed programming languages and runtimes* (p. 82–89). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3679007.3685058 doi: 10.1145/3679007.3685058

Harris, T. L. (2000). Dynamic adaptive pre-tenuring. In *Proceedings of the 2nd international symposium on memory management* (p. 127–136). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/362422.362476 doi: 10.1145/362422.362476

Jones, R., Hosking, A., & Moss, E. (2023). *The Garbage Collection Handbook: The Art of Automatic Memory Management* (2nd edition ed.). Boca Raton: Chapman & Hall/CRC.

Jump, M., Blackburn, S. M., & McKinley, K. S. (2004). Dynamic object sampling for pretenuring. In *Proceedings of the 4th international symposium on memory management* (p. 152–162). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1029873.1029892 doi: 10.1145/1029873.1029892

MacIver, D. R., & Donaldson, A. F. (2020). Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer. In R. Hirschfeld & T. Pape (Eds.), *34th european conference on object-oriented programming (ecoop 2020)* (Vol. 166, pp. 13:1–13:27). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik. Retrieved from https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2020.13 doi: 10.4230/LIPIcs.ECOOP.2020.13

MacIver, D. R., Hatfield-Dodds, Z., & many other contributors. (2019, November). Hypothesis: A new approach to property-based testing.
doi: 10.21105/joss.01891

Rigo, A., & Pedroni, S. (2006). PyPy's approach to virtual machine construction. In *DLS.* Portland, Oregon, USA: ACM. Retrieved 2008-06-16, from http://portal.acm.org/citation.cfm?id=1176753 doi: 10.1145/1176617.1176753

Turner-Trauring, I. (2024). *Sciagraph homepage.* Retrieved 2025-05-22, from https://www.sciagraph.com/docs/reference/limitations/

Wilson, P. R. (1992). Uniprocessor Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management* (pp. 1–42). Springer-Verlag. Retrieved 2010-03-17, from http://portal.acm.org/citation.cfm?id=664824

About the authors

Christoph Jung is a graduate student at the Heinrich-Heine-Universität Düsseldorf (Germany). You can contact the author at Christoph.Jung@hhu.de.

CF Bolz-Tereick is a researcher at the Heinrich-Heine-Universität Düsseldorf (Germany). You can contact the author at cfbolz@gmx.de or visit https://cfbolz.de.