Deepcopy Report

Introduction

The deepcopy operation in Python, provided by the standard copy module, is a fundamental utility for object duplication. Unlike a shallow copy, which only replicates the outer container while keeping references to the same nested objects, deepcopy creates a fully independent copy of an object and all objects it references recursively. This ensures that changes to one copy have no effect on the other, making it essential in scenarios where object independence is required (situations include simulations, cloning of data structures, safe manipulation of complex nested containers).

Because of Python's dynamic object model, implementing a generic deepcopy routine requires traversing arbitrary object graphs, supporting both built-in and user-defined types, and correctly handling edge cases such as shared references and recursive structures. To achieve this, the standard deepcopy relies on memoization, type-specific dispatch, and object reconstruction. While this design ensures correctness and broad applicability, it also introduces significant interpreter overhead.

We chose to focus on the deepcopy benchmark because it directly reflects Python's handling of memory and object management. Optimizing it provides insight into interpreter-level performance, exposes inefficiencies in recursive traversal and object reconstruction, and can yield practical improvements for applications that frequently duplicate structured data.

In this project, we introduce an optimized version of deepcopy that reduces redundant work and adds targeted shortcuts for common cases. Our changes focus on memo optimization to skip unnecessary lookups for atomic objects, and the addition of two fastpaths that accelerate common patterns in object reconstruction and tuple handling. The optimized design preserves correctness while improving runtime efficiency, making the benchmark run faster across different workloads.

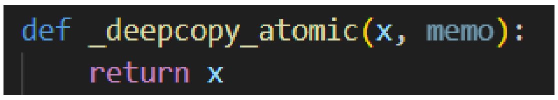
Overview

The deepcopy function in Python operates by recursively copying every element of an object graph, ensuring that the new object is entirely independent of the original. To achieve this, it relies on several mechanisms: atomic handling for immutable types, memoization to prevent cycles and preserve shared references, a dispatch table for built-in objects, and the reduction protocol for user-defined classes. Together, these mechanisms allow deepcopy to handle arbitrary Python objects correctly but also introduce overhead that can make it slow for large or frequently duplicated structures.

Atomic Objects

Certain objects in Python are immutable by design - such as int, float, str, bytes, bool, complex, and None. Since these objects cannot be modified after creation, they can be safely reused instead of duplicated.

In the implementation, these types are registered in the _deepcopy_dispatch dictionary with the function _deepcopy_atomic, which simply returns the original object:



```python
def _deepcopy_atomic(x, memo):
    return x
```

image 1: Deepcopy Atomic

By doing this, deepcopy avoids unnecessary allocations and ensures efficiency when copying data structures filled with scalars. For example, in a list of one million integers, only the list itself and its references are duplicated - the integers remain shared, since creating new copies would waste both time and memory without any benefit.

Memoization

Memoization is one of the most important features of deepcopy. It ensures that each object in the input graph is copied at most once, even if it appears multiple times or refers to itself.

This is implemented with a dictionary called memo, where keys are the unique IDs of objects and values are their corresponding copies. The id() function in Python returns a unique identifier for the object's lifetime, typically its memory address in CPython. Using this ID as a key allows deepcopy to recognize when an object has already been processed.

Internally, the lookup is performed using Python's built-in dictionary mechanism, which stores keys in a hash table. A lookup first computes the hash of the object ID and then checks the table for a match. On average, this is an O(1) operation, but each lookup still has a cost because it involves hashing, probing the table, and resolving collisions if they occur.

Dispatch Table

To efficiently handle built-in types, deepcopy uses a dispatch table (_deepcopy_dispatch) that maps each type to its dedicated copier function. When deepcopy encounters an object, it looks up its type in this table and calls the corresponding function.

- Atomic types such as int, float, str, bytes, bool, and None are all mapped to _deepcopy_atomic, a trivial function that simply returns the object itself. This avoids both memoization and unnecessary allocations, since these objects are immutable and safe to reuse.
- Container types like lists, dictionaries, sets, and tuples are mapped to specialized copier functions (_deepcopy_list, _deepcopy_dict, _deepcopy_tuple, etc.) that know how to iterate over elements and recursively apply deepcopy where needed.

This design ensures that common types are handled efficiently by avoiding the slower, generic reconstruction path. It also provides extensibility: new types can be registered with their own copy logic if needed, and user-defined classes fall back on the reduction protocol when no specialized copier is available.

Reduction Protocol

For user-defined classes and objects not covered by the dispatch table, deepcopy falls back on the reduction protocol, which is the same mechanism used by the pickle module.

An object can define either:

- __reduce__ – returns a tuple describing how to reconstruct the object.
- __reduce_ex__ – a version-aware alternative, called with a protocol number.

A typical reduce tuple has the form:

(cls, args, state_dict)

Where:

- cls is the constructor,
- args is a tuple of arguments to pass to the constructor,
- state_dict is the object's internal state (usually __dict__).

Deepcopy uses this tuple to rebuild the object without invoking its full __init__ method, ensuring the new instance has the same state as the original. This mechanism provides flexibility for arbitrary Python objects.

Benchmark Analysis

The Benchmark

The deepcopy benchmark from the pyperformance suite evaluates the performance of deepcopy. This suite includes three scenarios that exercise different aspects of deepcopy:

1. General benchmark – Copies a mix of standard Python types, including dictionaries, lists, tuples, strings, and a dataclass instance. This reflects typical usage of deepcopy in real applications, where a variety of built-in and user-defined structures appear together.

2. Reduce benchmark – Copies instances of a custom class that implements __reduce__ and __setstate__. This directly stresses the reconstruction logic of deepcopy, since the object must be rebuilt from its reduce tuple.

3. Memo benchmark – Copies a large data structure with shared references. For example, the same list appears multiple times inside tuples and dictionaries. This stresses the memoization logic, which must detect repeated references and avoid duplicating objects unnecessarily.

Each benchmark repeatedly performs deep copies inside timed loops, measuring throughput and latency. The results provide a clear view of how different internal mechanisms — memoization, reduction, and dispatch — impact performance.

Benchmark Baseline Performance

deepcopy: Mean +- std dev: 665 us +- 4 us

deepcopy_reduce: Mean +- std dev: 5.80 us +- 0.06 us

deepcopy_memo: Mean +- std dev: 79.7 us +- 0.4 us

Perf Report + Stats

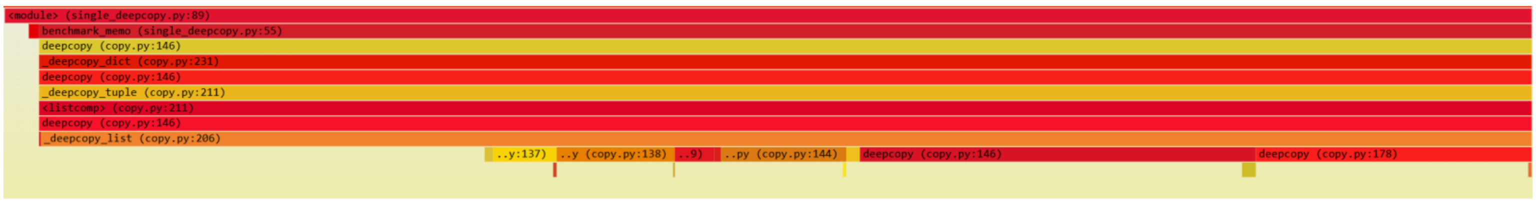
The perf report showing the distribution of the function calls during a single deepcopy operation can be seen at res/dc/perf_report_dc_baseline.txt.
The profile shows that most CPU time is spent deep inside the Python interpreter's frame execution and call machinery -_PyEval_EvalFrameDefault, call_function, and the vectorcall chain - rather than in any single user-space function body. In other words, the cost is dominated by lots of Python-level calls/dispatch and object protocol steps (type checks, method lookups, refcount ops). That aligns with deepcopy's job: it repeatedly consults dispatch tables, tests special methods (e.g., custom __deepcopy_ / __reduce_ex__), and recurses into containers - each step paying interpreter overhead.

Here are the extracted perf stat which will be compared later on with the stats extracted from the optimized benchmark run.

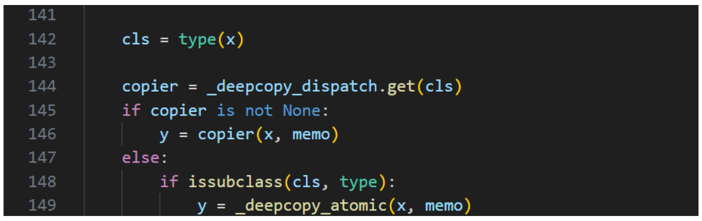
Metric	Baseline Value
L1-dcache-load-misses	1.906E+09
L1-dcache-loads	1.060E+11
LLC-load-misses	9.787E+06
LLC-loads	2.018E+08
branch-misses	2.915E+08
branches	6.704E+10
context-switches	4.205E+03
cpu-migrations	1.080E+02
cycles	1.381E+11
dTLB-load-misses	2.589E+07
dTLB-loads	1.047E+11
iTLB-load-misses	1.141E+07
iTLB-loads	2.402E+07
instructions	3.603E+11
page-faults	2.350E+05
task-clock	5.403E+04
time-elapsed	5.621E+01

Flamegraph

The Flamegraph of the single baseline deepcopy operation can be seen at res/dc/flamegraph_dc_baseline.svg



In the baseline flamegraph of deepcopy, the hottest path is centered around lines 144–146 of copy.py, where the dispatcher retrieves and calls the type-specific copier.



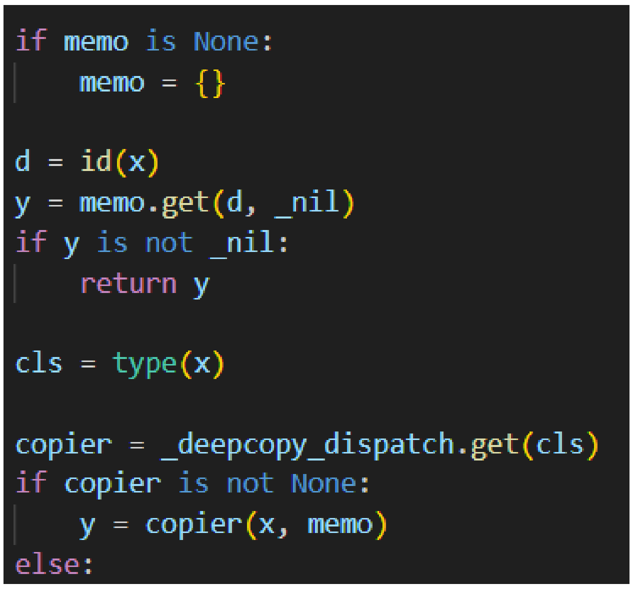
```
141
142     cls = type(x)
143
144     copier = _deepcopy_dispatch.get(cls)
145     if copier is not None:
146         y = copier(x, memo)
147     else:
148         if issubclass(cls, type):
149             y = _deepcopy_atomic(x, memo)
```

This is expected because every object passed to deepcopy pays this cost: computing type(x), performing a dictionary lookup in _deepcopy_dispatch, and then calling the copier function. When copying nested structures like tuples, dicts, or lists, this dispatcher path is hit repeatedly for each element, amplifying the overhead. The “heat” visible on the final return y line is not because return itself is slow, but because profilers attribute the accumulated time of the function to its exit point. Overall, the flamegraph highlights how much of the deepcopy cost comes from repeatedly dispatching into copier functions and recursing into container elements, which explains why this area became the natural target for our optimizations.

Optimization Strategies

Memo Optimization

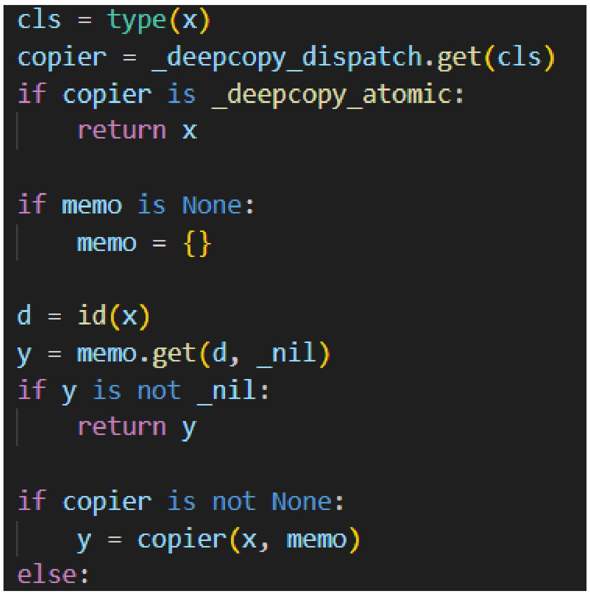
In the original implementation of deepcopy, the algorithm first checks the memo dictionary to see if an object has already been copied. This ensures that shared references and recursive structures are preserved correctly. Only afterwards does it check the object's type to decide whether the object can be returned directly.



```
if memo is None:
    memo = {}

d = id(x)
y = memo.get(d, _nil)
if y is not _nil:
    return y

cls = type(x)

copier = _deepcopy_dispatch.get(cls)
if copier is not None:
    y = copier(x, memo)
else:
```

image 2: Original library code first checks the memo and then the type

In our optimized version, we reversed this order: we first check whether the object's copier function is registered as _deepcopy_atomic. If so, we immediately return the object without consulting memo. If not, only then do we fall back to the usual memo lookup.



```
cls = type(x)
copier = _deepcopy_dispatch.get(cls)
if copier is _deepcopy_atomic:
    return x

if memo is None:
    memo = {}

d = id(x)
y = memo.get(d, _nil)
if y is not _nil:
    return y

if copier is not None:
    y = copier(x, memo)
else:
```

image 3: Optimized code first checks if the type is atomic

This change is correct because atomic immutables (such as int, str, float, None) can never participate in cycles or shared mutable references, so skipping memoization is safe.

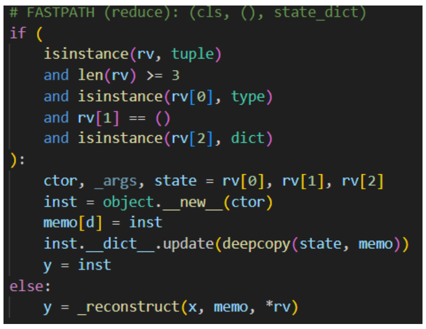
The benefit is that we avoid unnecessary dictionary lookups in workloads dominated by scalars. While dictionary lookups in Python are O(1) on average, they still involve hashing the object ID, probing the table, and checking for matches. By reordering the logic, we remove the lookup entirely for atomic types.

Importantly, this optimization doesn't add extra code paths or complexity - it simply changes the order of existing checks. The logic remains equivalent, but the new order yields better performance because of how frequently atomic objects appear in real data structures.

Reduce Fastpath

When an object defines __reduce__, deepcopy receives a tuple describing how to reconstruct it. A very common case is: (cls, (), obj.__dict__). This means - create a new instance of cls with no constructor arguments, then populate it with the object's dictionary. In the original code, this simple case is handled by the generic _reconstruct function, which performs additional checks for __setstate__, slot states, iterators, and more.

We introduced a fastpath for this simple reduce pattern. The fastpath avoids unnecessary branching and directly builds the new object.



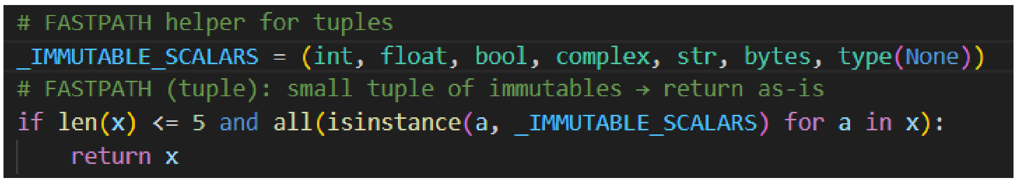
```python
# FASTPATH (reduce): (cls, (), state_dict)
if (
    isinstance(rv, tuple)
    and len(rv) >= 3
    and isinstance(rv[0], type)
    and rv[1] == ()
    and isinstance(rv[2], dict)
):
    ctor, _args, state = rv[0], rv[1], rv[2]
    inst = object.__new__(ctor)
    memo[d] = inst
    inst.__dict__.update(deepcopy(state, memo))
    y = inst
else:
    y = _reconstruct(x, memo, *rv)
```

image 4: Optimized code using reduce fastpath

Tuple Fastpath

Tuples are frequently encountered in Python, often as small containers for scalars (1, 2, 3) or ('x', 'y')). In the original implementation, tuples are always traversed element by element and rebuilt, even if all elements are atomic immutable.

We added a fastpath: if a tuple has length ≤ 5 and all its elements are immutable scalars, it is returned directly without traversal. This is particularly effective in the general benchmark, where small tuples appear often inside larger structures.



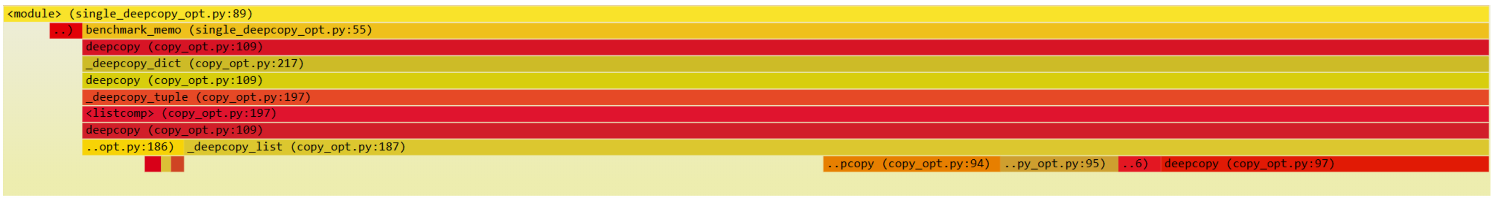
```python
# FASTPATH helper for tuples
_IMMUTABLE_SCALARS = (int, float, bool, complex, str, bytes, type(None))
# FASTPATH (tuple): small tuple of immutables → return as-is
if len(x) <= 5 and all(isinstance(a, _IMMUTABLE_SCALARS) for a in x):
    return x
```

image 5: Optemized code using tuple fastpath

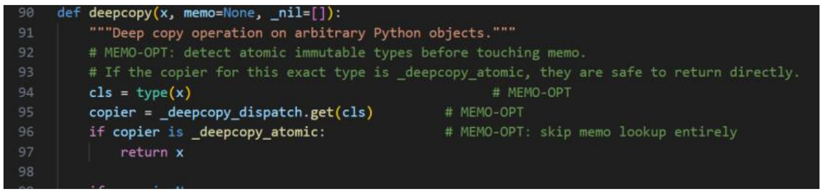
Performance Comparison

Flamegraph

The Flamegraph of the single opt dc operation can be seen at res/dc/flamegraph_dc_opt.svg



In the optimized flamegraph, the hottest site shifts to lines 94–97 of our patched copy.py - the MEMO-OPT early check: cls = type(x), lookup in _deepcopy_dispatch, and the fast "atomic → return x" path. This hotspot is expected, because every object still flows through this gateway. The difference is how quickly we bail out for immutables and simple cases. While this block concentrates heat (it's the new front door), its share of total time drops from >50% to ~34%, showing that fewer calls proceed into deeper, more expensive paths (memo probes, reducer lookups, _reconstruct, recursive element copies). In short, we didn't eliminate the dispatcher but rather moved work earlier and made more of it trivial, so the flame narrows at the top and fades below. See lines 94–97 in the optimized file for the MEMO-OPT block:



```python
90  def deepcopy(x, memo=None, _nil=[]):
91      """Deep copy operation on arbitrary Python objects."""
92      # MEMO-OPT: detect atomic immutable types before touching memo.
93      # If the copier for this exact type is _deepcopy_atomic, they are safe to return directly.
94      cls = type(x)                                   # MEMO-OPT
95      copier = _deepcopy_dispatch.get(cls)        # MEMO-OPT
96      if copier is _deepcopy_atomic:              # MEMO-OPT: skip memo lookup entirely
97          return x
98
```

Perf Report

The perf report showing the distribution of the function calls during the single optimized dc operation can be seen at res/dc/perf_report_dc_opt.txt

We can see from the report that less % of time is spent on python's interpreter - _PyEval_EvalFrameDefault, (58% -> 52%) so we can deduce that our optimizations led to less python function calls and dispatch and object handling, which is expected with our changes to avoid un-needed calls and memory handles with our fastpath and memo-opt changes.

Performance & Stats Comparison

Benchmark	Baseline Mean (us)	Baseline Std (us)	Optimized Mean (us)	Optimized Std (us)	Delta (us) (Opt - Base)	Improvement (%)
deepcopy	665	4	496	5	-169	25.41
deepcopy_memo	79.7	0.4	67.5	0.5	-12.2	15.31
deepcopy_reduce	5.8	0.06	4.51	0.04	-1.29	22.24

We observed a strong overall performance gain across all benchmarks, with the classic deepcopy benchmark showing the largest improvement of ~25% (665 µs → 496 µs). The deepcopy_reduce benchmark also improved significantly, by ~22%, reflecting the effectiveness of our fast-path optimization for __reduce__-based objects, which avoids unnecessary overhead when reconstructing objects with a simple reduce protocol. The deepcopy_memo benchmark showed a smaller but still substantial gain of ~15%, which can be explained by its heavy reliance on memoization: here the relative share of atomic copies is lower and memo operations dominate, reducing the effectiveness of our memo fast-path compared to the classic case. In short, the results align with our expectations: optimizations that skip redundant dispatch and fast-path common cases yield the highest impact in the generic benchmark, while still delivering measurable improvements in specialized reduce and memo scenarios.

Metric	Baseline Value	Optimized Value	Delta (Opt - Base)	%
L1-dcache-load-misses	1.906E+09	1.674E+09	-2.313E+08	-12%
L1-dcache-loads	1.060E+11	8.228E+10	-2.374E+10	-22%
LLC-load-misses	9.787E+06	5.846E+06	-3.941E+06	-40%
LLC-loads	2.018E+08	2.251E+08	2.333E+07	+12%
branch-misses	2.915E+08	2.695E+08	-2.207E+07	-8%
branches	6.704E+10	5.322E+10	-1.382E+10	-21%
context-switches	4.205E+03	4.271E+03	6.600E+01	+2%
cpu-migrations	1.080E+02	4.300E+01	-6.500E+01	-60%
cycles	1.381E+11	1.106E+11	-2.745E+10	-20%
dTLB-load-misses	2.589E+07	2.318E+07	-2.705E+06	-10%
dTLB-loads	1.047E+11	8.386E+10	-2.088E+10	-20%
iTLB-load-misses	1.141E+07	9.338E+06	-2.070E+06	-18%
iTLB-loads	2.402E+07	2.146E+07	-2.559E+06	-11%
instructions	3.603E+11	2.823E+11	-7.797E+10	-22%
page-faults	2.350E+05	2.352E+05	1.680E+02	0%
task-clock	5.403E+04	4.385E+04	-1.018E+04	-19%
time-elapsed	5.621E+01	4.586E+01	-1.036E+01	-18%

The optimized deepcopy shows a clear reduction in overall work: both the instruction count and cycle count dropped by about 20–22%, which matches the performance gain we measured in the benchmark runtime. This validates that our optimizations reduced unnecessary Python interpreter calls and code execution. Alongside this, we see fewer L1 data-cache loads (-22%), L1 misses (-12%), and TLB misses (-10 to -20%), all of which are natural byproducts of executing fewer operations and traversing fewer Python objects at the interpreter level. One interesting stat is that LLC loads increased by +12%, even while LLC misses decreased by 40%. This might be explained by improved prefetching behavior: with less interpreter overhead and fewer irregular jumps, the workload becomes more sequential and predictable (e.g., scanning through containers and copying bytes). Hardware prefetchers therefore stage more cache lines into the LLC, raising the absolute number of LLC load events, while at the same time reducing the number of misses because those lines are available when requested. Overall, the statistics confirm that the optimized version makes memory access patterns more cache-friendly and reduces wasted cycles in the interpreter, leading to a consistent performance uplift.

Performance Compare for Middle Versions (partial opts)

Ver	deepcopy Mean (us)	deepcopy Std (us)	deepcopy % vs Baseline	deepcopy_reduce Mean (us)	deepcopy_reduce Std (us)	deepcopy_reduce % vs Baseline	deepcopy_memo Mean (us)	deepcopy_memo Std (us)	deepcopy_memo % vs Baseline
BASELINE	672	4	0%	5.77	0.06	0%	79.6	0.4	0%
FASTPATH	625	4	7.0%	5.29	0.05	8.30%	81.6	1.6	-2.5%
MEMO ONLY	520	3	22.6%	5.03	0.05	12.8%	65.3	0.5	18.0%
MERGED	497	5	26.1%	4.55	0.07	21.2%	67.1	0.5	15.7%

We can look at the performance gains of each of the optimisations separately to understand the impact of each of them. In the FASTPATH version, we optimized the entry path of deepcopy by adding direct checks for simple cases. This shows up most strongly in the benchmark_reduce, where skipping overhead around __reduce__ handling cut time by ~8.3%. On the other hand, benchmark_memo sees almost no improvement (slight regression of –2.5%) since the fast path doesn't apply once shared references dominate. The MEMO ONLY optimization specifically targeted the cost of repeatedly looking up objects in the memo table. That directly benefits benchmark_memo, showing a big 18% gain, while also reducing overhead in benchmark_reduce by ~12.8%. Finally, the MERGED version combines both strategies, giving consistent improvements across the board: ~26% for the general deepcopy case, ~21% for benchmark_reduce, and ~15.7% for benchmark_memo. This confirms that our optimizations map cleanly onto the bottlenecks revealed in the flamegraphs -fastpathing improves trivial objects, and memo-lookup streamlining helps when shared references are present.

Hardware Acceleration Proposal

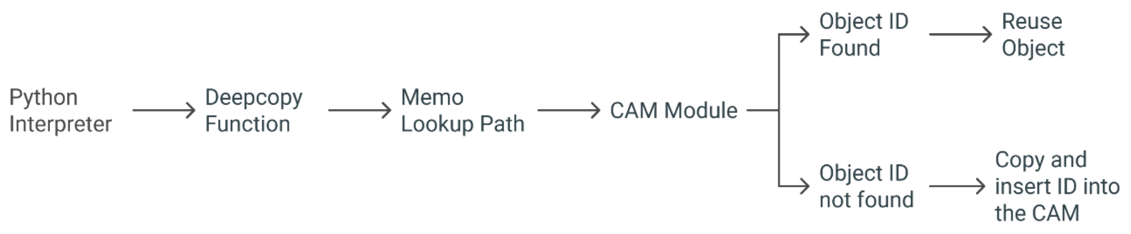
When we analyzed Python's deepcopy, it became clear that a large fraction of the cost comes from dictionary lookups. The most important one is the memo dictionary, which tracks for each object ID whether a copy already exists, ensuring cycles and shared references are handled correctly. Another repeated lookup happens in the deepcopy_dispatch table, which determines which copier function to use for each type. Since these lookups occur constantly during a deep copy, they form a natural target for optimization.

The proposed solution is to accelerate these dictionary lookups with a CAM (Content-Addressable Memory). Instead of relying on Python's software hash tables, we store object IDs directly in a hardware CAM and query them in constant time. Whenever deepcopy processes an object, it asks the CAM: if the ID is already there, the copy is reused; if not, a new copy is created and the ID inserted. Because CAM checks in parallel, this bypasses the hashing and probing overhead of Python dictionaries and delivers consistently fast responses.

Inputs: object IDs generated during the deepcopy process, together with lookup requests from Python.
Outputs: immediate lookup results - either a "found" signal returning the associated reference, or a "not found" signal that triggers the normal copy path and then updates the CAM. In practice, this means the CAM becomes a transparent replacement for the memo dictionary's search logic.

Trade-offs: On the positive side, it offers faster lookups, reduced interpreter overhead, and better scalability when handling large or repetitive object graphs, while requiring minimal software changes - only the memo lookup path in deepcopy needs to be redirected to the CAM. On the other hand, the CAM block introduces hardware cost, design complexity, and potential integration challenges (e.g., memory controller interactions, cache coherence). Moreover, since the bulk of deepcopy time still lies in copying object contents, CAM acceleration does not eliminate the main bottleneck.

Overall, the CAM proposal shows how targeted hardware support can complement software optimizations: by offloading repeated dictionary lookups into a specialized memory unit, we reduce wasted cycles and make Python's deepcopy measurably more efficient, even though copying itself still dominates runtime.



Conclusion

Through a combination of software optimizations and a hardware acceleration proposal, we demonstrated that Python’s deepcopy can be made significantly faster without sacrificing correctness. The reordering of memo checks and the addition of targeted fastpaths reduced redundant lookups and unnecessary branching, yielding performance gains of 15–25% across different benchmarks. These improvements were confirmed not only by runtime measurements but also by hardware counters, which showed fewer instructions, reduced cache misses, and more efficient memory access patterns. Beyond software changes, the CAM-based hardware proposal highlights how dictionary-heavy operations like memo lookups can be offloaded to specialized memory structures, providing constant-time responses and lowering interpreter overhead. While object copying itself remains the dominant cost, these optimizations and accelerations together illustrate a clear path toward more efficient object management in Python: trimming away unnecessary overhead, streamlining common cases, and complementing the interpreter with lightweight hardware support when needed.