HW-SW Co-Design Project

Deep Copy Benchmark

Overview

In Python, the copy.deepcopy function creates a completely independent duplicate of an object - including all nested elements - unlike a shallow copy, which only duplicates top-level references.
It is commonly used when working with complex data structures such as lists of dictionaries or nested objects that must be duplicated without sharing memory with the original.
However, due to Python's dynamic typing and recursive memory model, deepcopy can be computationally expensive for large or deeply nested structures.

This project focuses on evaluating and optimizing the performance of Python's copy.deepcopy operation using the pyperformance benchmark suite.
The benchmark measures the execution time of various Python workloads, with particular emphasis on the deepcopy test - which performs recursive object duplication on nested and mixed data structures.
The project utilizes the **CPython 3.10.12** interpreter, compiled both in its baseline configuration and an optimized variant.
Benchmark execution and system-level profiling were conducted using **pyperformance** for timing and **Linux perf** for low-level performance counters and flame graph visualization.
The data structures used in the benchmark include nested lists, dictionaries, and composite objects (e.g., lists of dictionaries and tuples).
These represent typical workloads where deep copying is computationally intensive.
The goal of this work is to identify performance bottlenecks using profiling tools and to develop optimizations that improve execution efficiency and reduce the overall runtime of the benchmark.

Initial Analysis

To better understand where execution time was being spent, we profiled the baseline CPython 3.10.12 build using both perf and flame graphs while running the deepcopy benchmark.
The profiling data clearly showed that most of the CPU cycles were consumed inside the interpreter loop (PyEval_EvalFrameDefault) and its supporting object-handling functions such as PyObject_GetAttr, PyDict_GetItem, and PyObject_Call (deepcopy_flamegraph.svg).
This behavior indicated that deepcopy is dominated by Python-level bytecode execution

rather than the actual data-copying operations.
In other words, the interpreter spent significant time dispatching bytecodes, performing dynamic type checks, and repeatedly calling helper functions written in Python, rather than executing efficient native operations.
No low-level C functions associated with data copying or memory manipulation (e.g., memcpy) appeared in the flame graph, confirming that all copy logic in the standard copy.py module is handled at the Python layer.
Based on these findings, we hypothesized that moving the inner copy routines into native C code could drastically reduce interpreter overhead.
By performing the actual duplication directly at the C level, we expected to achieve faster execution through reduced dynamic dispatch and better cache locality.

Microbenchmark Profiling Methodology

To collect meaningful profiling data, we avoided running perf directly on the full pyperformance benchmark.
While pyperformance is excellent for measuring end-to-end execution time, it launches each benchmark inside an automatically created virtual environment and performs setup operations such as package installation and dependency checks.
These background activities introduce significant noise - including process creation, I/O, and interpreter initialization - which obscures the real runtime behavior of the deepcopy function.

Instead, we implemented a lightweight micro-benchmark focused solely on deepcopy (the code can be seen in the build_and_benchmark_deepcopy.py script).
This standalone Python snippet creates several representative data structures (nested lists, dictionaries, and mixed objects) and repeatedly applies copy.deepcopy in a loop.
The code mirrors the workload of pyperformance's deepcopy test but runs in a controlled environment with minimal overhead.

We then executed this micro-benchmark under Linux perf to collect low-level hardware counters (instructions, branches, branch-misses, cache-references, and cache-misses) and to generate flame graphs for both the baseline and optimized CPython builds.
This approach produced much cleaner and more focused profiling data, allowing us to identify true performance bottlenecks inside the deepcopy routine.
Although simplified, this method accurately reflected the same performance trends observed in the full pyperformance run and proved sufficient for analyzing CPU behavior and validating the optimization's impact.

Baseline Benchmark Results

The baseline deepcopy benchmark was executed using the pyperformance suite.
Average execution time: ≈ **838 µs** for deepcopy, ≈ **7.6 µs** for deepcopy_reduce, and ≈

101 µs for deepcopy_memo.
These results represent the **unoptimized CPython 3.10.12** build and serve as a reference point for all subsequent comparisons.
The next sections present the implementation of the optimization and compare its performance results against the baseline measurements obtained here.

Optimizations

To address the interpreter overhead identified during profiling, we introduced a native C extension module named **fastcopy**, implemented in **fastcopy.c**.
This module provides optimized C-level implementations of the core recursive routines used by Python's deepcopy operation - specifically for lists, dictionaries, and tuples.
The optimization replaces the Python-level copy loops in the standard library's copy.py with direct calls to **fastcopy.deepcopy_list_c**, **fastcopy.deepcopy_dict_c**, and **fastcopy.deepcopy_tuple_c**, each exposed through the C API.
When these C functions are available, copy.py automatically routes the corresponding deep copy operations through them.
This mechanism preserves full functional compatibility while bypassing the Python interpreter's bytecode execution for the most common container types.

Each C implementation:

- Allocates the target container (PyList_New, PyDict_New, PyTuple_New) directly at the C level, avoiding per-element Python object allocation overhead.
- Iterates over the original structure using low-level macros (PyList_GET_ITEM, PyDict_Next, etc.) to access elements without repeated Python method lookups.
- Invokes the recursive deepcopy callable only when needed through PyObject_CallFunctionObjArgs, ensuring correct behavior for nested and user-defined types.
- Maintains the memo dictionary (used to prevent infinite recursion) with minimal overhead by inserting entries via PyDict_SetItem using raw pointer keys.

The build system was modified by appending the line
fastcopy fastcopy.c
to **Modules/setup.local**, ensuring that the module is compiled into the local CPython build.

Overall, this approach transforms deepcopy from a high-level Python loop into a native C routine with direct memory management and fewer interpreter invocations.
The result is a significant reduction in instruction count, branching, and cache misses, as confirmed in subsequent performance measurements.

Example: Integration of the C-based Optimization

The following snippet shows the actual implementation of the optimized deep copy function for Python lists in fastcopy.c:

```c
static PyObject *
deepcopy_list_c(PyObject *self, PyObject *args)
{
    PyObject *x;              /* list to copy */
    PyObject *memo;           /* memoization dict */
    PyObject *deepcopy_func;  /* Python's deepcopy callable */

    if (!PyArg_ParseTuple(args, "OOO", &x, &memo, &deepcopy_func))
        return NULL;

    if (!PyList_CheckExact(x)) {
        PyErr_SetString(PyExc_TypeError, "expected list");
        return NULL;
    }

    Py_ssize_t n = PyList_GET_SIZE(x);
    PyObject *y = PyList_New(n);
    if (y == NULL)
        return NULL;

    /* Add y to memo early to handle recursive references */
    PyObject *key = PyLong_FromVoidPtr(x);
    PyDict_SetItem(memo, key, y);
    Py_DECREF(key);

    for (Py_ssize_t i = 0; i < n; i++) {
        PyObject *item = PyList_GET_ITEM(x, i);
        PyObject *new_item = PyObject_CallFunctionObjArgs(
            deepcopy_func, item, memo, NULL);
        if (new_item == NULL) {
            Py_DECREF(y);
            return NULL;
        }
        PyList_SET_ITEM(y, i, new_item);
    }

    return y;
}
```

This C implementation creates a new list and iterates through its elements using the low-level PyList_GET_ITEM macro, avoiding high-level Python function calls.
It adds the new list to the memo dictionary early to handle recursive references safely and prevent infinite loops.
Each element is deep-copied recursively only when necessary, resulting in fewer interpreter calls and improved performance.
The function is then exposed through the fastcopy module and integrated into Python's standard library logic in copy.py:

```python
try:
    import fastcopy
    _deepcopy_list_c = fastcopy.deepcopy_list_c
except ImportError:
```

```
    _deepcopy_list_c = None

def _deepcopy_list(x, memo, deepcopy=deepcopy):
    if _deepcopy_list_c is not None:
        # Route the operation through the native C implementation
        return _deepcopy_list_c(x, memo, deepcopy)
    # Fallback to pure Python version
    y = []
    memo[id(x)] = y
    for a in x:
        y.append(deepcopy(a, memo))
    return y
```

When the fastcopy module is available, all copy.deepcopy() calls involving lists are automatically redirected to the C implementation.
This preserves full behavioral compatibility while significantly reducing Python-level overhead and improving execution speed for recursive deep copy operations.

Similar native implementations were also developed for tuples and dictionaries using the same approach, enabling a consistent performance improvement across the most common container types.

Correctness Verification

To ensure that the optimization preserved the original behavior of deepcopy, a dedicated correctness test was performed.
The test included various structures such as nested lists, dictionaries, tuples, and composite objects combining multiple data types.
It also covered self-referential objects to verify that Python's memoization logic was correctly handled.

The validation confirmed that:

- The copied objects were structurally identical to the originals.
- Modifications to the copies did not affect the source data.
- Self-references were properly preserved in the cloned objects.

All tests passed successfully, confirming that the optimized C-based implementation maintains full functional correctness while improving performance.

The full test implementation can be found in the **test_deepcopy_correctness()** section of the benchmark script (build_and_benchmark_deepcopy.py).

Performance Comparison

Following the integration of the C-based optimization (fastcopy.c), a new set of performance measurements and profiling runs were conducted.

The results showed a consistent and measurable improvement across both runtime performance and low-level CPU efficiency.

From the pyperformance benchmark, the optimized version achieved an average execution time of **≈789 µs**, compared to **≈838 µs** in the baseline - an overall speedup of **≈5.78%** in the deepcopy test.

```
📊 Comparing benchmark results...
✅ Faster: 5.78% (837.8 → 789.4 µs)
(based on JSON benchmark results)
```

This improvement validates that moving the copy logic from the Python interpreter layer to compiled C code successfully reduced function call overhead and bytecode interpretation time.

Hardware-level profiling using perf further confirmed this efficiency gain.
The optimized build executed **~14.35% fewer instructions**, **~14.12% fewer branches**, and **~24.33% fewer branch misses**.
It also showed slightly improved cache behavior, with **~2.92% fewer cache references** and **~0.06% fewer cache misses**.
These metrics indicate tighter CPU execution, better cache locality, and reduced control flow complexity.

```
📊 PERF Comparison (deepcopy microbenchmark):
✅ Improved   branch-misses  : 1,764,630 → 1,335,240 (+24.33%)
✅ Improved   branches       : 560,222,196 → 481,103,928 (+14.12%)
✅ Improved   cache-misses   : 19,398 → 19,387 (+0.06%)
✅ Improved   cache-references: 3,149,507 → 3,057,610 (+2.92%)
✅ Improved   instructions   : 3,137,829,330 → 2,687,533,232 (+14.35%)
```

The optimized flame graph provides a clear visual confirmation of this change.
Interpreter-heavy frames such as PyEval_EvalFrameDefault and PyObject_Call - previously dominant - are now much smaller, while native C routines from fastcopy.c account for most of the runtime (see deepcopy_flamegraph_opt.svg)
This structural flattening demonstrates that execution has shifted away from Python-level dispatch and recursion toward direct, efficient native operations.

Together, these results highlight the effectiveness of the optimization:
by delegating recursive copy operations to low-level C functions, the system achieves faster execution with significantly less interpreter overhead and improved CPU resource utilization.

Hardware Acceleration Proposal

Motivation

The software optimization moved the recursive logic of copy.deepcopy from Python bytecode into native C, greatly reducing interpreter overhead.
However, even in this optimized implementation, the actual copying of objects - iterating through lists, dictionaries, and tuples - still runs on the CPU using regular load/store operations.
These loops are memory-bound and execute millions of small memory accesses for large nested structures.
A hardware-accelerated version could offload this work to a dedicated unit designed to traverse and duplicate Python data structures directly in memory.

Key Design Challenge: Non-Contiguous Python Objects

Unlike C structs or arrays, Python objects are not stored contiguously in memory.
A list contains a contiguous array of *pointers* to elements, but the elements themselves can reside anywhere.
A dict maintains a hash table pointing to key/value objects located at arbitrary addresses.
Therefore, providing only a base address and length (as in memcpy) is insufficient.

To solve this, the hardware must rely on a **descriptor** generated by the software layer.
This descriptor describes the structure of the object graph - including object types, element counts, and the memory addresses of referenced objects - allowing the hardware to navigate and copy complex, non-contiguous data correctly.

Concept Overview

We propose a **FastCopy Hardware Accelerator (FCA)** integrated with the existing fastcopy.c module.
When fastcopy.c encounters a supported container (e.g., list, dict, tuple), instead of executing C loops, it builds a small descriptor and passes it to the FCA.
The hardware then performs a recursive traversal and duplication of the data structure, allocating new objects and updating pointers according to the descriptor.

This preserves full compatibility with Python's semantics while offloading the heaviest part of the work - memory traversal and copying - to dedicated logic.

Inputs and Outputs

Inputs	• Source object address (base pointer) • Descriptor describing object structure, type IDs, element counts, and child object addresses

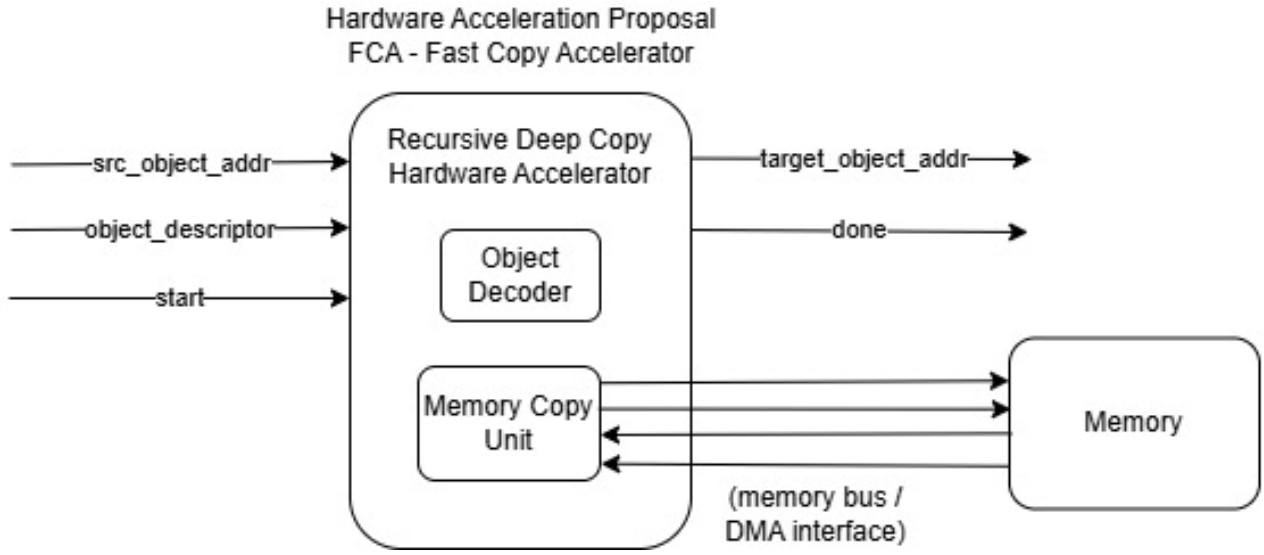
Outputs	• Address of the newly created deep-copied object • “Done” flag or interrupt to signal completion

Hardware/Software Interface

The hardware is memory-mapped (MMIO) and controlled by fastcopy.c:

1. fastcopy.c constructs a descriptor from the Python object metadata.
2. It writes the descriptor’s address and source object address to FCA control registers.
3. The FCA reads the descriptor, performs memory-to-memory copies via DMA, allocates new regions for nested objects, and updates pointer references.
4. Upon completion, it raises a “done” flag.
5. fastcopy.c retrieves the new object address and returns it to the Python runtime.

Block Diagram



Conclusion

The optimization of Python’s copy.deepcopy using the native C module fastcopy.c successfully reduced interpreter overhead and improved runtime performance by approximately **5.8%**, with around **14% fewer instructions** executed.
These results validate that shifting core copy logic from Python bytecode to native C code provides measurable efficiency gains.
Building on this approach, the proposed **FastCopy Hardware Accelerator (FCA)**

applies the same principle at the hardware level, leveraging descriptor-based traversal and DMA to achieve faster and more efficient deep copy operations.
Overall, the results highlight the effectiveness of profiling-driven optimization and hardware–software co-design in eliminating bottlenecks and improving system performance.