Specializing Adaptive Interpreter without the GIL

Author: Ken Jin
kjooi@quansight.com / kenjin4096@gmail.com

CPython 3.11 introduced PEP 659: Specializing Adaptive Interpreter (SAI), significantly boosting single-threaded performance. That work however, implicitly depends on the GIL being enabled. In CPython 3.13, an option to remove the GIL was added to CPython, with the SAI disabled. Users thus have to choose between better single threaded performance, or better multi-threaded performance.

This document proposes the first step towards reconciling that difference – making the specializing adaptive interpreter usable without the GIL. Included are design considerations, ideas considered, and possible tradeoffs. I target a 25% performance gain in single-threaded code from this effort for free-threaded builds.

Ideally, we should not diverge how we treat specialized bytecode formats between free-threaded and GIL-builds. Having separate behavior for both would increase maintainer burden. We can ifdef the usual atomics and friends but more egregious changes like changing the bytecode format should be maintained across both versions.

A maximum 1% performance loss on pyperformance on the default build's (with GIL) SAI should be acceptable, and I target no memory increase for single-threaded applications, though multi-threaded applications may use more memory. I expect that this loss on performance will be offset by the Tier 2 JIT compiler work anyways, so overall there should still be a net speedup. Note that some of the free-threaded work also overlaps with, and lays the foundation for the Tier 2 optimizer work. For example, PEP 703's deferred reference counting's setup can also be used for unboxed integers in the Tier 2 optimizer in CPython 3.14.

Design

The main design idea is as follows: Instead of locks, do our best to layout our data in a thread safe manner and use atomics. For objects that are not shared across multiple threads, preserve a happy fast path, otherwise if the objects are shared, use locks. This takes inspiration from multiple papers, such PEP 703. This document also uses some terminology from "Efficient and Thread-Safe Objects for Dynamically-Typed Languages" (Daloze, Marr, Bonetta & Mössenböck, 2016).

Keeping caches alive

Inline caches mostly hold things that are safe on their own – like version numbers. However, they also hold borrowed references to method descriptors. Additionally, heap types store caches to __init__ and __getitem__ in the type object itself.

For inline cache method descriptors, we shall mark all inline-cached method descriptors as deferred (see deferred reference counting in PEP 703). This should already be happening since method descriptors ought to be deferred anyways. Code objects need to support GC (they already do in the free threaded build), and we must teach the GC to traverse the code object's inline caches. For this, Stefan Marr suggested keeping a bitmap of all method references in the bytecode and quickly traversing that. This will keep the borrowed references alive. The equivalence in literature is that the bytecode array now becomes a "zero count table" of sorts. Since GC becomes more expensive, we should probably switch to the incremental GC.

For heap types, we can simply make them per-thread or copy-on-write.

Alternatives: making inline caches strong references. This is untenable, as we incur refcounting overhead during specialization, and we also complicate deoptimizations. Current deoptimizations are essentially side-effect free with a few notable exceptions (like materializing a dictionary).

Defer GC to CHECK_EVAL_BREAKER / GC for deferred method is already safe

To make this effective, we must mark every method of a type as deferred (or at least, anything that could be in _PyType_Lookup). The CHECK_EVAL_BREAKER must also only do a sweep if all threads have signaled the eval breaker at least once. This ensures we don't free a method while a thread is executing in the middle of a bytecode instruction. I think this is already done.

Note: the difference with how things are right now is that any Py_DECREF call could call a destructor, which runs arbitrary code, like potentially running the GC. So we have no clue if a borrowed method will be safe to access, even if it is deferred. Consider the following scenario:

1. Two threads, A and B.
2. Thread A grabs a deferred method from inline cache, preparing to read it.
3. Thread B overwrites that inline cache with a new method, also calling Py_DECREF and triggering GC.
4. Despite A's method being deferred, the GC does not see it as it's no longer reachable from any roots (not in code object, not in stack).
5. GC thus cleans up A's deferred method.
6. A tries accessing the deferred method. Segfault.

Sam pointed out that: “Thread B cannot start a GC without Thread A's cooperation. The GC has a stop-the-world pause so every thread is either in the eval breaker or some other state that could have released the GIL. In those cases, it's already not safe to have unprotected borrowed references.”

Addendum:
Mark Shannon planned to do this already, but I’m going to re-propose its value here anyways. Not having to care about whether a GC could run in between specializations is a significant simplification.

Mark’s current plans of deferring GC altogether has some folks concerned. The main concern is that there could be libraries that depend on the prompt finalization would otherwise run out of memory. That would break existing Python code.

However, this approach preserves that (implementation-dependent) behavior, by only deferring GC sweep (stop the world) of methods to cooperative safepoints (CHECK_EVAL_BREAKER).

Handling Bytecode Rewriting

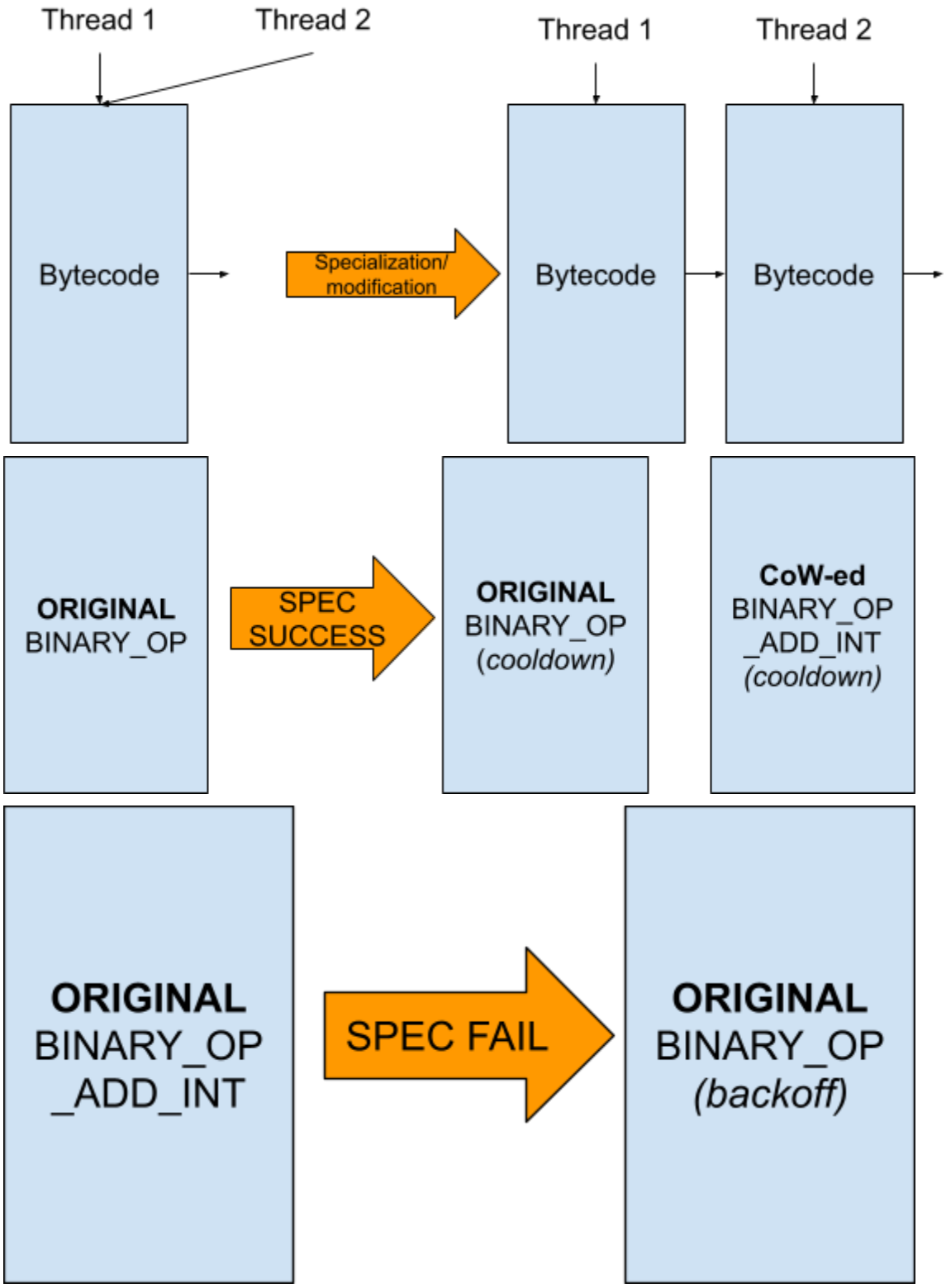
A key design feature of the SAI is bytecode rewriting – ie specialization or quickening. This allows the SAI to adapt to new behaviors observed in user code. Self-modifying bytecode brings race conditions and Python semantic concerns.

I suggest we should handle bytecode rewriting without any locking, and defer most of the complexity to the specializing infrastructure. This will ensure good multi-threaded performance and good single-thread performance. There are two options. A. or B.

Option A: Make bytecode per-thread

For this, we shall duplicate bytecode objects when they are being executed by the non-owning thread. This is a simple scheme, that also makes reasoning about inline cache consistency simpler, but it comes with significant memory cost, and also some runtime cost considering every single function that needs specialization needs bytecode copying.

Note: Donghee Na and Matt Page both suggested some form of RCU (Read-copy-update). Perhaps we could duplicate bytecode on the first specialization? The added benefit of that is no cache layout change required on the default build. The other benefit is that we no longer need atomic reads of the bytecode or locked writes with such a scheme. Perhaps performing better as well.



An additional optimization would be to keep updated bytecode interned and re-use them for other threads whenever threads start and die.

The bytecode duplication scheme would keep a linked list of bytecode per code object. Each bytecode also keeps a counter on the number of “users” (ie frames). Upon first specialization of a code object that has multiple users, the code object is duplicated. Thus the scheme is lazy.

To reduce refcounting overhead, the linked list of bytecode will be marked as deferred. The GC shall scan the bytecode linked list to see if they should be kept alive.

For recursive calls, the bytecode may wrongly detect there are multiple users because there are multiple frames on the stack. There are two solutions I can think of:

1. There needs to be a set of thread IDs that have been seen in a py_hashtable of sorts. Duplication will thus look at this table to determine if copying is indeed required. This means that **only for recursive calls**, a lookup is incurred on every successful specialization attempt. We may need a better solution to reduce the cost. Note that this should be an optimization for recursive calls. Ie. the cost of checking for a thread state ID should be < the cost of checking for duplication. Assuming a high-end 128 core server with 256 threads, this means a linear scan through such an array should at worst still be cheaper than copying the whole bytecode. This doesn’t seem scalable. OR
2. Before every escaping call, decrement the number of users on the frame, then after the call, increment it again. This seems scalable, but would hurt single-threaded perf a slight bit. On the other hand, most specializations are non-escaping. So if a program specializes well/optimizes in tier 2 well, this should be okay.

Option B: Make bytecode modification multi-thread compatible

For this, we do not need to differentiate between shared and unshared objects, because specialization has an exponential backoff – it should naturally be a once in a blue moon type of operation. Reads from cache need to use acquire atomic loads. Writes to cache need to use mutexes on the code object. Specialization itself (changing the bytecode instruction) needs release atomic writes.

The key observation is that specialization only exists within a bytecode family, and bytecode families are all (to the user at least), semantically equivalent to each other. Even if a thread reads a wrong bytecode specialization and executes it, it should safely deoptimize to the original bytecode. Thus I propose the following design:

1. For specialization and de-opt, we need to atomically write instructions and their inline caches to the bytecode object. We can write this in sequence, without any locks, only if step 2. Is implemented.
2. We must ensure equivalent cache layout among an entire family. This will increase memory usage slightly. What I mean by equivalent layout is this: take for example the LOAD_ATTR (method) variant’s inline cache entry:

```
typedef struct {
```

```
    _Py_BackoffCounter counter;
    uint16_t type_version[2];
    union {
        uint16_t keys_version[2];
        uint16_t dict_offset;
    };
    uint16_t descr[4];
} _PyLoadMethodCache;
```

To save space, the cache uses a union of keys version and dictionary offsets. In the proposed schema, it is unsafe to do this because the guards might wrongly interpret something as safe and let the unsafe operation execute. What should happen is the layout should look like this:

```
typedef struct {
    _Py_BackoffCounter counter;
    uint16_t type_version[2];
    uint16_t keys_version[2];
    uint16_t dict_offset;
    uint16_t descr[4];
} _PyLoadMethodCache;
```

We waste some space, but instead gain lock-free reads and writes.
Note: Sam pointed out that the fields which are multi-entry cannot be read atomically because they are not aligned. We either need a seqlock (hopefully not, because it's slow) or manually align it ourselves (more wasted space).

Note that if 1 and 2 are implemented, bytecode rewriting becomes thread safe (with some further caveats).

Consider if the following case:

1. Two Thread A and Thread B threads are executing the same bytecode instruction.
2. Thread A rewrites _LOAD_ATTR_METHOD_WITH_VALUES to _LOAD_ATTR_METHOD_NO_DICT
3. Thread B still sees the old _LOAD_ATTR_METHOD_WITH_VALUES and executes that.
4. Thread B's guards will automatically deoptimize and then execute the right instruction.

This however, does not consider the case of if thread B is in the midst of executing something after it has already executed its guards. For that, we need a consistent snapshot of inline caches. That is the next section.

Consistent Snapshot of Inline Caches

One problem with C is the concept of a dangling pointer. A parallel runtime in something like Java, according to my interview with Benoit, makes things easier because once you have a reference to something, it will remain valid.

The main goal: we do not want to segfault. It is also not okay to return wrong inline cached values. Note that however, we can avoid segfault, and return stale inline cache values if the Python code is inherently race-condition inducing, and the semantics are undefined. Also because the current status quo already would return stale cache values from something like _PyType_Lookup.

Method descriptors

Consider the following code:

```
def call(obj):
   obj.meth()
```

The following scenario is acceptable, because it is not against Python semantics. The user probably just forgot to put a lock:

1. Suppose we have two threads, A and B.
2. A passes guards for LOAD_ATTR.
3. B modifies obj's MRO and overrides meth with meth1.
4. Thread A now loads the old obj.meth().

In the current free-threading build, this will also likely segfault if the SAI is enabled. Once step 2 happens, A now has a dangling reference to `meth` in its own LOAD_ATTR. However, thanks to the previous section on "Keeping caches alive" and "Delay GC of deferred objects to safepoints", that problem is now fixed. Instead, the user then gets a stale cache value. This is acceptable because in this case, it is on the user to lock calling and writing to `obj.meth`. Python makes no guarantees that a change in one thread is atomically updated and shown in another thread (or does it previously with the GIL..? needs more research). At the very least, even without inline caching, this is the behavior on the free-threaded build with _PyType_Lookup.

Note: you might consider what happens if a method descriptor is invalidated by another thread, written to the inline cache, then a GC run happens, so the old cache entry disappears. Well recall the we are delaying GC of deferred objects to safepoints, so a GC run like that won't happen.

Namespaces and inlined values

Handling namespaces or dictionaries (__dict__) is tougher than method descriptors. Consider the following scenario.

1. Two threads, A and B, operate on a same object `obj`.
2. Thread A is in the midst of LOAD_ATTR from `obj`, and passes the guards
3. Thread B deletes the attribute A is accessing.

At worst, this scenario segfaults, and at best, a stale value is obtained (ie, an invalid value of a new object that has taken the old object's place).

We do not want segfaults. However, in CPython's specific implementation, a stale value should be safe to read (though not necessarily safe to dereference!), because the new value that is modified should reside at the address of the old location. This should also be the correct value to load, assuming the values' layout has not changed (this assumption does not always hold, see the section on `changing the object's values layout`).

Thus the question becomes not "how do I ensure the index is valid?" but "how do I ensure the object I'm pointing to is valid?" Ie. it's another dangling pointer problem. This time, however, we can't use the same solution as before. We cannot defer namespace values as that might make them live unnecessarily long and break existing Python code, causing them to OOM. See the next section.

Note: Sam pointed out that for most LOAD_ATTR specializations, _Py_TryIncrefCompare is enough for them.

Invalidating when PyObject's layout change

Python objects have a few ways of storing values — either a values array directly inline in the object itself, a "lazy" dictionary, a key-sharing dictionary, or a full-blown dictionary. For ease of explanation, we shall just distinguish the inline values array and dictionary.

Values array is thread-safe as long as we read and write atomically to it. However, it is not thread-safe in between specializations, as bytecode might change the "shape" of the dictionary.

Consider the following example:

1. Thread A's LOAD_ATTR might have passed its guards and is preparing to load from a values array.
2. Thread B executes STORE_ATTR which then cause the values array to be invalidated, and create a lazy dictionary instead. Such a materialization requires copying the inline values array to the lazy dictionary. This also implies the old address that the old entry was at in the values array will still hold valid data. So loading a stale entry from there will not segfault when reading the values array, though it might return a stale value.
3. However, we might still segfault when operating on the object as a stale cache value may be a dead reference

Notice that for method lookups loading stale data is fine — stale methods will never be dead because we defer them to GC safepoints. So LOAD_ATTR_*_METHOD variants are mostly safe thanks to the previous sections. In current free-threaded builds, we already return stale data due to _PyType_Lookup anyways, so there's no behavioral change there.

For globals, builtins, and attribute lookups, dk_version and tp_version_tag is not a strong enough guarantee, because a stale cache value might be dead (no longer pointing to valid data) but the tag wouldn't change. Similar to PEP 703, an optimistic fast path is to just read without lock. The key difference is we need to insert a few more guards to check our optimistic fast path is valid, and deoptimize otherwise (see PEP 703 for how this could be done). A _Py_TryIncrefCompare should be enough for most cases.

Alternatives:

1. Consider using dictionary watchers?

Lists

Same as PEP 703. For mutation: use critical sections. For reads, use _Py_TryXGetRef and no overhead if the object is not shared, fall back to slow path and lock on failure.

Implementation Sequence

I propose we start by implementing the simplest bytecode specializations and "handling bytecode rewriting" partially.

So we start with BINARY_OP and friends.

Next we implement delay GC of deferred objects to safepoints. That will allow us to implement some forms of LOAD_ATTR specialization: specifically, most of LOAD_ATTR_*_METHOD.

LOAD_GLOBAL, and the rest of LOAD_ATTR and BINARY_SUBSCR then finally need the full "consistent snapshot of inline caches".

The goal is to all this working by CPython 3.14. We might leave out some of the LOAD_ATTR specializations that are more complex.

Expected Performance

Expected performance loss on default build (with GIL):

- 0-2% pyperformance, from slightly increased inline cache sizes. The more branchy code is guarded by the usual ifdefs, so there should be no loss there. This is an upper bound I hope. Last I recall, shrinking inline caches brought not much performance gain anyways.
- Maximum additional 5% memory usage, inline cache sizes.
- OR
- No change, due to RCU scheme on free-threaded builds.

For free-threaded build: SAI boosted single-threaded performance on GIL builds by ~30%. I'm targeting a 25% single-threaded speedup for free-threaded builds from the SAI. The rough estimation is based on my memory of when I was working on the SAI. The following are the losses:

- 0-2% pyperformance from slightly increased inline cache sizes.
- 1-2% pyperformance from more branches in the happy path.
- 1% miscellany
- OR
- 1-3% perf loss, due to RCU scheme on free-threaded builds.

For multi-threaded scalability, most of the operations follow PEP 703 so the same arguments apply from there. Writing to bytecode will be slower, but the SAI has exponential backoff for specializations last I recall, so this should be fine.

Open Questions

1. Instrumentation?
2. How to handle code explosion in tier 2 JIT if each thread can trigger its own trace compilation?

Acknowledgements

Thanks to the following people who contributed. In no particular order:

- Stefan Marr
- Sam Gross
- Matt Page
- Donghee Na
- Guido van Rossum
- Mark Shannon