UNIVERSITÀ DI TRENTO

Department of Information Engineering and Computer Science

Master's Degree in Computer Science

Final Dissertation

A Concurrent Hash Table for CPython

Supervisor	Student
Prof. Paola Quaglia	Daniele Parmeggiani

Academic year 2023/2024

Alla memoria di Lidia e Norma.

Acknowledgements

I would like to thank Sam Gross and Eric Snow for taking the time to read this Thesis and provide valuable feedback.

Abstract

The Python community is increasingly interested in parallelization. One of the biggest changes to the CPython interpreter, the most common implementation of the language, is set to be released this October 2024. In it, the greatest long-standing bottleneck to multithreading performance scaling, the Global Interpreter Lock (GIL), is removed. We can expect more Python programmers to start choosing multithreading concurrency models in the future. We can also expect concurrent data structures to gather increasing interest in the coming future.

Concurrent hash tables are among the most important concurrent data structures, and the built-in sequential hash table of CPython is ubiquitously used within the interpreter and throughout the community. We find that the research for efficient, concurrent hash tables has been very active in recent years, and many implementations are available for C programs.

We propose here an efficient concurrent hash table that can be used from Python code directly, albeit implemented in C. We introduced it to the Python community.

Contents

Chapter 1

Introduction

The focus of this Thesis is the development of an efficient concurrent hash table that can be used directly from Python code. Given the ubiquitous use of hash tables in Python programs, the research aims to contribute to the Python community by proposing a data structure that meets the performance and scalability demands of a post-GIL Python environment.

This Thesis is organized into several chapters, beginning with an introduction to hash tables, concurrent hash tables, and their applications, followed by an exploration of the challenges and solutions related to free-threading CPython. We then review the state of the art in concurrent hash table design, detail the design and implementation of the proposed data structure, and conclude with an evaluation of its performance.

Overall, this work provides a significant contribution to the field of concurrent data structures for Python and offers practical insights for its community as it prepares for new possibilities in parallel computing with the upcoming changes to CPython.

This Chapter begins by recalling hash tables and their many applications in computing. It then delves into the relevance of concurrent hash tables. The Chapter also touches on CPython's built-in hash table, `dict`, which is widely used within the CPython interpreter and in the outer Python ecosystem. The final Section introduces "free-threading Python," an experimental build of CPython that removes the GIL in the upcoming 3.13 release.

1.1 Hash tables and their applications

A hash table is a data structure that stores a set of distinct elements, called *keys*, and associates with each of them a *value*. A *hash function* $h(k)$ is used to determine a unique position for each key k in an array of *buckets* (or *slots*), in which the desired key-value association can be found. An *ideal* hash function assigns each key to a unique slot, but such functions are very hard to find. The occurrence of multiple keys having the same hash is called a *collision*. The mechanism with which a hash table handles collisions is called *collision resolution*.

Hash tables support common operations, which are executed in expected constant time:

1. an *insert* creates a new (non previously existing) association of a key to a value;
2. an *update* replaces the value associated with a certain key with a new value;
3. a *delete* removes a key-value association; and
4. a *lookup* returns the value associated with a certain key; in particular, a lookup is described as *successful* if it in fact returns said value, or alternatively it is described as *failing*.

Implementations of hash tables are generally distinguished by their choice of hash function and of collision resolution. Common implementations of hash tables include:

- open addressing, where items are placed into an array of slots, and collisions are resolved by displacing an item at slot x to the next available slot $f(x)$, which may be computed by:
  - linear probing, i.e. f is a linear function;
  - double hashing, i.e. f is another hash function, distinct from h; or
- separate chaining, also known as closed addressing, where items are placed into linked lists called buckets and collisions are resolved by appending to the buckets.

Applications of Hash Tables. Hash tables are ubiquitous in computing. They are taught in entry level Computer Science courses, and form the basis of many programs. Common applications include: map-reduce systems, databases (notably, their indexing and aggregation capabilities), caching systems, and dynamic programming, among others. A characteristic that is common to these applications is that hash table access can dominate their execution time.

1.1.1 Concurrent Hash Tables

In a shared-memory concurrency environment, several threads can share information in a flexible and efficient way by sharing access to a common hash table. In fact, hash tables can provide a form of *natural parallelism*, as worded in the introduction to [10, Chapter 13]:

> [...] We studied how to extract parallelism from data structures [...] that seemed to provide few opportunities for parallelism. In this chapter we take the opposite approach. We study *concurrent hashing*, a problem that seems to be “naturally parallelizable” or, using a more technical term, *disjoint–access–parallel*, meaning that concurrent method calls are likely to access disjoint locations, implying that there is little need for synchronization.

In other words, two threads accessing a shared hash table have little or no need for synchronization when accessing two distinct keys stored therein.

In the following paragraphs of this Section, we provide some informal definitions for important properties of concurrent algorithms and data structures, drawing them from [10].

Linearizability. It is an important property of concurrent data structures: a method call carried out by a thread t, should be observable by any thread other than t, to have modified the state of the data structure instantaneously, at some moment between its invocation and response.

Lock-freedom and wait-freedom.

> A method is *lock-free* if it guarantees that infinitely often *some* method call finishes in a finite number of steps.

Wait-freedom is a more stringent property:

> A method is *wait-free* if it guarantees that *every* call terminates in a finite number of steps.

These two are very desirable properties of concurrent data structures: they imply that threads don't need to wait for any other thread's work to complete, before completing their own work (wait-freedom), or that at least one thread can always make progress (lock-freedom).

In terms of Amdahl's Law, the implication is that the sequential part of a program is very small, so that the speedup from parallelization scales almost linearly with the number of threads. That is, for S being the speedup gained by parallelizing a program, p being the parallelizable fraction of the program, and n being the number of parallel processors:

$$S = \frac{1}{1 - p + \frac{p}{n}}, \text{ with desired } (1-p) \ll p.$$

The hash table design proposed in this Thesis is almost entirely lock-free. Some well-defined relaxations of the lock-freedom property are taken in this design, and elaborated in Section 4.2. These relaxations are not a novel contribution to the field of concurrent hash tables.

1.2 Python and its Built-in Hash Table

Python is a popular, high-level programming language. It is dynamically typed and garbage-collected, and it chiefly supports object-oriented programming.

The CPython interpreter is the reference implementation of the Python language, and it is often confused for the language itself. Other implementations of Python exist, but are less popular. They are often specialized implementations that improve the performance of certain workloads.

Python has a built-in hash table, named `dict`, which is considered an integral part of the language. It is a standard type, used in virtually every Python program, either explicitly, by directly creating a `dict` instance, or implicitly. The CPython interpreter may create dictionaries in order to create an instance of a class, a very frequent operation in an object-oriented language. The attributes of an object are entirely stored in a hash table, which can also be accessed directly. In fact, the set of attributes of a Python object may be altered at runtime, thus differing from the one specified in its class. (This was true up to version 3.12, released October 2023, where instances were optimized to avoid allocating this attributes dictionary. The dictionary may still be created to view all the attributes of an object, upon a program's request.)

An exploration on the usages of the built-in hash table has been carried out and it is presented in Section 6.2. For a discussion on `dict`'s design, refer to Section 3.1. For a more thorough explanation of the common usages of CPython's `dict`, refer to [5, §Principal Use Cases for Dictionaries].

1.3 Free-threading Python

A known limitation of the CPython interpreter (not the Python language) is the presence of a Global Interpreter Lock (GIL). This lock is used to protect the interpreter's internal state from concurrent mutations. It has simplified significantly the implementation of the CPython interpreter over the years, at the cost of Python programs being often unable to efficiently use the multiple processors available in the system. The limitation does not affect *all* CPython-interpreted programs, in the sense that several programming paradigms do not make use of multi-threading regardless of the GIL, and are thus not actually impacted by its presence.

Examples of these paradigms, except from obviously single-threaded programs, include:

1. co-routines-based concurrency, where a set of co-routines are executed concurrently and scheduled by the interpreter instead of the OS; and

2. multiprocessing-based concurrency, where the separate strands of execution are offloaded from a main process to a set of child processes, each having their own GIL.

Many other programs, instead, cannot effectively cope with the presence of the GIL. Various times in the past, GIL removals have been attempted: see [2], and [9, §Related Work]. The rationale behind them is always a variation of the same theme, quoting from [9]:

> CPython's [GIL] prevents multiple threads from executing Python code at the same time. The GIL is an obstacle to using multicore CPUs from Python efficiently. [...] For scientific computing tasks, this lack of concurrency is often a bigger issue than speed of executing Python code, since most of the processor cycles are spent in optimized CPU or GPU kernels. The GIL introduces a global bottleneck that can prevent other threads from making progress if they call any Python code.

It should be noted at this point that the addition of the GIL predates the popularity of multiprocessor systems. (Although not yet called "GIL," references to an interpreter lock can be found in source distributions of Python 1.0.1, released January 1994; available online at `https://www.python.org/ftp/python/src/python1.0.1.tar.gz`.[1]) At the time, it made a lot more sense to simplify the implementation of the interpreter (and still support multi-threading), rather than being capable of more performance, but almost never enjoy it.

[1] Last accessed October 4, 2024.

Free-threading Python (formerly known as nogil) is the latest attempt at removing the GIL. Initially carried out by Sam Gross, it is now in CPython's official codebase. Python 3.13, the next release, is scheduled for October 2024 (the year of writing), and will feature an experimental build mode for free-threading Python. The build of Python 3.13 that will be commonly distributed, is referenced to as the *default* build.

The work presented later in this Thesis is based on the original reference implementation, nogil, itself based on Python 3.9. This choice was made because free-threading Python had not been implemented when the work for this thesis began in October 2023. Porting this work to Python 3.13 is planned.

In the next Chapter, Free-Threading Python is described in more detail. The changes described in this Thesis seem to be fully implemented, with the remaining work for the next release being primarily focused on resolving numerous minor issues.

Overall, free-threading is anticipated to result in a 6% performance overhead for single-threaded executions and an 8% overhead for multithreaded executions, with the higher overhead primarily due to the absence of optimizations available in single-threaded executions. (See the article on Gross' presentation at the 2023 Python Language Summit [27].)

Chapter 2

Free-Threading Python

In this Chapter we delve into the modifications necessary to enable free-threading in CPython. This is a combination of several different optimizations primarily focused on maintaining single-threaded performance, in a concurrent context.

We start by examining concurrent, biased reference counting, which optimizes performance by assuming most objects are accessed primarily by the thread that created them. Deferred reference counting is introduced for objects where consistent reference counting is impractical. These objects defer computing their true reference count until Garbage Collection (GC) pauses.

Quiescent State-Based Reclamation (QSBR) is presented as a mechanism to support lock-free read accesses to built-in data structures, optimizing those accesses which are the most frequent. QSBR delays freeing objects until no thread can be in the process of accessing them, preventing use-after-free bugs. To support this change, it was also necessary to change CPython's memory allocator to mimalloc, a thread-safe alternative. Adjustments were also necessary for CPython's GC, which so becomes a "stop-the-world" collector.

Finally, we also detail the thread-safety of built-in data structures. Per-object mutexes maintain the atomic operations previously ensured by the GIL, similarly to Java's model, with some read-only operations optimized to avoid locking, relying on some modifications to mimalloc for consistency.

2.1 Concurrent, biased reference counting

Concurrent reference counting is an implementation of reference counting that maintains correctness in multithreaded environments. Free-threading Python implements a design of concurrent reference counting based on the assumption that most objects are only accessed by the thread that created them. This bias towards the owner's reference counting does in fact turn out to be justified. This design is backed by the relevant literature [4]. Overall, the result of biased reference counting is that the impact of concurrent reference counting is greatly reduced; that is, most reference counting operations use the biased, faster path.

An object is modified to store an owner thread identifier, an owner's reference count, and a shared reference count. The owner thread may use normal reads and writes when modifying its reference count, while the other threads must use atomic operations to modify the shared reference count, thus resulting in a slower path for reference counting.

The shared reference count also stores the reference counting state, using two bits of the 4-byte counter. An object may be in one of the following states:

1. default;
2. weakrefs;
3. queued; or
4. merged.

Their meanings are described in [9, §Biased Reference Counting]:

> The states form a progression: during their lifecycle, objects may transition to any numerically higher state. Objects can only be deallocated from the "default" and "merged" states. Other states must transition to the "merged" state before deallocation. Transitioning states requires an atomic compare-and-swap on the [shared reference count] field.

The "default" state. Objects are initially created in this state. (Logically, an object cannot be shared among multiple threads before and during its instantiation.) If an object, during its lifetime, was in fact only accessed by its owner, it will have remained in the "default" state, and can thus enjoy the quick deallocation path: upon reaching a reference count of 0, the object's memory is immediately freed (this is what happens to Python objects in the default build of CPython).

The "weakrefs," "queued," and "merged" states. Objects that are actually shared between threads during their lifecycle cannot use the quick deallocation path, as [9] states:

> [...] The first time a non-owning thread attempts to retrieve an object in the "default" state it falls back to the slower locking code path and transitions the object to the "weakrefs" state.

An object enters the "queued" state when a non-owning thread wishes to merge the two reference counts; that is, when the shared reference count becomes negative. The object is then enqueued in the owner's queue of objects to be merged. When the object enters the "merged" state, its owner thread field is reset, indicating that the object is not owned by any thread, and the owner's reference count is no longer used. When the shared reference count reaches 0, the object may be deallocated, when a quiescent state is reached (see Section 2.3).

Note that these transitions rarely happen, because (1) most objects are accessed by a single thread, and (2) it is rare for an object to have a negative shared reference count.

Thread identifiers. It logically follows from the above discussion that threads accessing CPython objects must be uniquely identified. Such identifiers are computed by calling CPython's new `_Py_ThreadId` internal API. Internally, it uses various hardware- and platform-dependent calls to generate the number. For instance, on x86-64 hardware running Linux, the identifier is stored in the FS register. It is a pointer to the thread's `pthread` struct, used primarily for fast access to its Thread Local Storage (TLS). Refer also to the Linux Kernel Documentation, §29.8.1, Common FS and GS usage. Available online at `https://www.kernel.org/doc/html/v6.9/arch/x86/x86_64/fsgs.html`.[1] This API is also used in this thesis to elect the leader thread for a hash table migration. (See Section 5.4.2.)

2.2 Deferred reference counting

For some particular objects, it does not make sense to keep a consistent reference count. The owner's reference count also stores two flags, used to indicate that the object is immortal, or that it can use deferred reference counting. Immortal objects are objects whose lifetime equals the lifetime of the program, such as the `True`, `False`, and `None` objects.

Reference counts for some non-immortalized, contended objects may be deferred. Namely, for "top-level functions, code objects, modules, and methods," because "these objects don't necessarily live for the lifetime of the program, so immortalization is not a good fit."

In [9, §Deferred Reference Counting] Gross details an approach where instead of letting the interpreter increment and decrement those objects' reference counts as they are pushed to and popped from the evaluation stack, it instead skips those reference counting operations. It follows that their "reference count fields no longer reflect their true number of references." Instead, their true reference counts can only be safely computed during Garbage Collection (GC) pauses, and thus such objects can only be deallocated during a GC cycle. CPython's GC was designed to only collect objects that are part of reference cycles. It has thus been modified in order to also compute true deferred reference counts, and to collect those objects as well.

This is not a general approach to deferred reference counting, and thus departs from the existing literature; cf. [1].

2.3 Quiescent State-Based Reclamation

An additional change to CPython for free-threading is QSBR. Although not directly referred to as QSBR, it was initially described in [9, §Mimalloc Page Reuse], and later discussed in greater detail in the accompanying GitHub issue [8]. CPython's implementation of QSBR is derived from FreeBSD's implementation of Globally Unbounded Sequences, as noted by Gross in the referred issue.

This mechanism serves to support lock-free read accesses to built-in data structures. For example, when a thread t_r optimistically avoids locking a shared `dict` H's keys array, due to a read-only access, an unsafe circumstance may

[1] Last accessed October 4, 2024.

arise. If H is resized (migrated) by another thread t_m while t_r is operating its read, it may be that t_m frees H's keys object before t_r gets a chance to read it. This circumstance may arise while H's reference count remains > 0, thus this problem does not pertain concurrent reference counting.

In the above example, t_m is not allowed to free the keys immediately, instead the following happens:

1. there exist a global, monotonically increasing sequence number S, accessible to every thread $t \in T$;

2. t_r is endowed with a local sequence number $S(t_r)$, which records the most recently observed value of the global S, say that in this execution $S(t_r) = S$;

3. t_r begins the read-only access over H (until the access ends, t_r is not allowed to change $S(t_r)$);

4. t_m begins the resize (migration) of H;

5. t_r and t_m read the keys object k (a pointer) associated with H;

6. t_m creates a new keys object k', and completes the resizing (migration of keys in k to k');

7. t_m substitutes the key object associated with H, from k to k';

8. t_m calls for a *delayed* free of k;

9. t_m observes the value of S;

10. t_m adds a pointer to k in a shared QSBR queue Q, and sets the QSBR goal g for its deferred free to $g(k) = S + 1$;

11. t_r completes its read;

12. t_r increments its local sequence number, $S'(t_r) = S(t_r) + 1 = S + 1$;

13. ...*eventually*...

14. a thread t_q (it may be that $t_q \equiv t_r \vee t_m \equiv t_m$) is in a quiescent state, because it is running a GC cycle;

15. t_q observes that there exist a lower bound σ to the sequence number of every thread $\min_{t \in T} S(t) \geq \sigma$;

16. if $g(k) > \sigma$, k will not be freed;

17. otherwise, k will be freed, and removed from Q.

This is a generic mechanism that can help implement lock-free data structures. Note that thread-local (i.e. not shared) data structures are not affected by QSBR, and are instead immediately freed upon reaching a reference count of 0, because they have remained in the "default" state.

2.4 Changes to the memory allocator

The described QSBR mechanism was located in the "Mimalloc Page Reuse" section of PEP 703 because it requires changes to CPython's memory allocator in order to work. In fact, the memory allocator employed so far in CPython, is not thread-safe. To allow allocations of objects without locking the entire interpreter, it was decided to change memory allocator entirely.

The new allocator of choice is mimalloc, a general purpose allocator with good performance and thread-safety. It is described in detail in [16].

Some changes to mimalloc were necessary in order to support QSBR, essentially around restricting page reuse. When a mimalloc "page" (not equiv. to an OS memory page) is empty, mimalloc may decide to reuse it for incoming allocation requests. Though, it may be that an empty page contains objects whose free operation was deferred, due to QSBR. Therefore, it would be unsafe to reuse the page for new objects. Instead, the following happens:

1. a thread deletes an object, which happened to reside in mimalloc page p;
2. p is now empty;
3. p gets tagged with the global sequence number S (described earlier in Section 2.3), s.t. $\tau(p) = S$;
4. p may be reused when a later global sequence number $S' > \tau(p)$ is observed.

Crucially, this modification to mimalloc ensures that threads which may still read an object in p don't read unrelated objects, resulting in undefined behavior, and instead necessarily read a reference count of 0, because the freed object is still stored in the same mimalloc page, which is not reused until a quiescent state is reached.

2.5 Garbage collection

CPython's GC was also changed for the free-threading build to work. Foremost, it was important to run the GC during "stop-the-world" pauses, to restore thread-safety guarantees that were previously provided by the GIL. Additionally, it was also necessary to integrate it with deferred and biased reference counting.

To support "stop-the-world" pauses a status is assigned to each thread; one of "attached," "detached," or "gc". The "attached" and "detached" states follow the semantics that were previously assigned to the acquisition and release of the GIL. A thread that would previously acquire the GIL, transitions to the "attached" state; conversely, a thread that would previously release the GIL, transitions to the "detached" state.

When a GC cycle begins, the GC thread transitions all other threads that were in the "detached" state into the "gc" state. Threads in the "attached" state are requested to pause themselves and transition to the "gc" state. When all threads are in the "gc" state, the GC thread starts the collection cycle. Once the collection finishes, it transitions all threads in the "gc" state to the "detached" state, and notifies them. If a thread was in the "detached" state before entering the "gc" state, it ignores the notification.

Note that "stop-the-world" pauses also implicitly create quiescent states (when all threads are in the "gc" state). The deterministic presence of quiescent states is required for QSBR to function properly.

2.6 Thread-safety of built-in data structures

Most operations in Python are not atomic, but historically some were in fact made implicitly atomic due to the presence of the GIL. For instance, calling `list(set)` atomically creates a list from a set, because the GIL is held for the duration of the call. The thread safety guarantees of built-in objects that were implicitly provided by the GIL, are retained, as detailed in [9, §Container Thread-Safety].

Note that this does not mean that all operations on built-in data structures are thread safe. (In fact, an operation like `my_dict[key] += 1` cannot be made thread-safe, in the same way a C instruction `my_counter++` cannot be made thread-safe, without external synchronization.) Instead, Gross proposes to simply preserve the guarantees that were in place before the GIL was removed.

To achieve this goal, CPython objects are endowed with per-object mutexes. When an operation that was previously thread-safe because of the GIL is requested, the object involved is locked and the operation performed. Some operations require that two objects are locked, e.g. `list(set)`. This is done by locking first the object whose memory address is lower, and then the other, to avoid deadlocks.

Optimistically avoiding locking. To improve performance, a few operations have been selected to be executed without any locking. They are frequently used, read-only operations on `list` and `dict` objects (e.g. `my_list[0]`). Such operations may fall back to a slower path which acquires the involved objects' mutexes, in case the container objects were concurrently modified during the read operation. (This optimistic fast-path requires the changes to mimalloc described in Section 2.4. See also [9, §Optimistically Avoiding Locking], for further details.)

Chapter 3

A Review of the State of the Art

This Chapter provides a detailed examination of various hash table implementations and optimizations, focusing on their design principles, performance implications, and practical applications. It is structured into several sections, each addressing different designs. This Chapter is not an exhaustive list of hash table designs: we avoid detailing the designs which are generally considered superseded, and instead focus on the most relevant ones.

We begin by exploring CPython's built-in sequential hash table, already in wide use in the Python community. Its core design principles, and the limitations imposed by the tradeoffs of free-threading CPython are described. There are also included some general remarks about hash computation in Python.

We then explore concurrent hash table designs, beginning with Maier's hash table which was the main source of design inspiration for `AtomicDict`. Important differences with the hashing techniques used in CPython's `dict` are highlighted. We then continue exploring novel hash table designs which are all based on Maier's design: it has been found to be an important milestone in the literature. Two designs are explored: Bolt and DRAMHiT, both of which contributed some design decisions in `AtomicDict`, albeit less prominently.

We conclude with some remarks on the feasibility of completely lock-free hash tables, finding that they are in fact not practical due to the limitations of currently available hardware.

3.1 Python's Sequential Hash Table

Python's built-in hash table, already introduced in Section 1.2, is a sequential hash table with important ramifications inside the CPython interpreter. This Section provides an in-depth examination of CPython's `dict`, highlighting its design principles and operational mechanisms.

One notable optimization is key sharing, which significantly reduces memory consumption by storing the set of keys separately from their associated values. Furthermore, the compact dictionary design, proposed by Raymond Hettinger, has significantly improved the previous implementation by optimizing the overall layout for enhanced performance and space utilization.

We find some relevance of these design choices for `AtomicDict` as well. Later, we also highlight the changes that have been made to `dict` in order to support free-threading, but here we conclude that those changes overly favor the single-threaded use-cases, which are less relevant for `AtomicDict`.

3.1.1 Key Sharing

Recall that CPython uses hash tables to store the attributes of objects. It is immediate to see that different instances of the same class will generally have the same keys in their respective hash tables. A relatively simple optimization is thus to store the set of keys separately.

This optimization is only applied for those hash tables that are used for class instances. When creating a `dict` directly in Python code, CPython will not create a hash table with key sharing.

This creates slight performance overhead, compared to using a combined table for keys and values. Indeed, data locality is slightly diminished, an important factor for cache access optimization. It was measured, though, that the overall decrease in memory usage, more than compensated for this loss of locality. Further details are available in Shannon's proposal [23].

This optimization was not interesting for use in `AtomicDict`, because it was deemed unlikely that the hash tables shared among multiple threads share the same set of keys.

3.1.2 Compact Dictionary

In late 2012, Raymond Hettinger proposed a compact, ordered implementation of CPython's `dict`, in [11]. The design, which is still `dict`'s main design as of the time of writing, is based on the observation that the previous one was memory-inefficient. Hettinger noted that the table in which the items were stored, was unnecessarily wide: it required 24 bytes for storing the key pointer, the value pointer, and the hash value. In combination with the necessity of keeping the table sufficiently sparse (in order to cope with collisions), it generated a large amount of unused space. He instead proposed having two separate tables: one sparse table, called the index, and another, dense table, called the entries table, which the index table refers to.

Insertion order is preserved because iterations read the entries table, and new items are appended to that table. The actual hash table is the sparse index, which can fit into two cache lines for tables of capacity $m \leq 128$.

To further reduce memory usage, the size of index slots will vary depending on the size of the key set: when it is small there is no need to reference the entries table with 64-bit numbers. Instead, the slot size is chosen dynamically, from a lower bound of 1-byte, when $m \leq 128$, to a maximum of 8 bytes for very large hash tables.

In order to accomodate key sharing, the entries table may be further split into a key table, and a values table.

Splitting the sparse index table from the dense entries table, improves on both the storage space and the iteration speed. In fact, if we call α the load factor of the dictionary (i.e. the number of occupied slots over the total number of slots), there not need be $(1-\alpha)m$ empty 24-bytes entries for the unused slots in the hash table. Instead, there will be $(1-\alpha)m$ empty $O(8)$-bytes slots in the index. That is, the entries table will have a load factor ≈100%: entirely filled, except for deleted entries. When iterating over the keys in the dictionary, the dense entries table will serve as the basis for the iteration, instead of the sparse index. Thus, reading unoccupied slots is avoided, and the memory bandwidth required for an iteration is reduced.

As the set of keys becomes larger than the maximum allowed load factor $\alpha_{\max} = 2/3$, the resizing process is initiated, which does not modify the entries table, but only the smaller index. (Resizing is also termed *migrating* in the literature.) CPython's `dict` does not reduce its size after a large-enough sequence of deletions, that is, $\nexists\alpha_{\min}$.

3.1.3 Python Hashing

CPython takes the following approach to hashing that is grounded in the concrete usage of their hash tables. Quoting from [6]:

> Most hash schemes depend on having a "good" hash function, in the sense of simulating randomness. Python doesn't: its most important hash functions (for ints) are very regular in common cases:
>
> ```
> >>>[hash(i) for i in range(4)]
>    [0, 1, 2, 3]
> ```
>
> This isn't necessarily bad! To the contrary, in a table of size 2^i, taking the low-order i bits as the initial table index is extremely fast, and there are no collisions at all for dicts indexed by a contiguous range of ints. So this gives better-than-random behavior in common cases, and that's very desirable.

The hash functions employed for built-in CPython data types enjoy the following properties:

1. objects which compare equal have the same hash value;
2. one object always produces the same hash value;
3. the computed hashes distribution is uniform, for a uniformly distributed set of values to be hashed;
4. their running time complexity is reasonable, generally $O(1)$, and $O(n)$ for strings and tuples;
5. they don't necessitate holding locks, nor doing I/O, or other operations which may be generally considered improper for a hash function.

The reader will probably have noticed how these properties are generally true for all hash functions, with properties (1) and (2) being fundamental for any hashing scheme to work, properties (4), and (5) being trivial for all hash functions known in the literature, and property (3) being what is usually the concern behind the implementation of hash functions.

With hash functions being so foundational to the implementation of hash tables, it is important to make one explicit remark. There is no guarantee that the hash functions we indirectly employ (and that CPython's `dict` employs, too) enjoy the above properties. That is to say, a hash function written in Python code may perform any arbitrary operation. For instance, it may acquire a lock, initiate a request to a remote machine on the network, read a large file from disk, or run an $O(n!)$ algorithm to produce the result.

Of course, such behaviors would be completely unexpected of a hash function. We will therefore resort to *assume* that those properties hold, as we have no control over those functions. Most operations detailed in Chapters 4 and 5 are lock-free and $E\left[O(1)\right]$ only under this assumption: that user-code hash functions are reasonable and enjoy the above properties. Furthermore, it turns out that most real Python programs that use dictionaries also use strings as keys. Thus, in the common cases, these properties hold.

3.1.4 With the Free-Threading Changes

Without the GIL, concurrent accesses to a sequential hash table such as `dict` may yield inconsistent program states, and lead to interpreter crashes. Free-threading CPython requires that concurrent accesses to `dict` must be synchronized through a mutex.

Size retrieval of a `dict` is exempt from this requirement, since the size field itself is maintained correct with the `dict`'s mutex, and is thus easily available for concurrent, wait-free reading.

Two operations are also partially exempt. They are:

1. `dict[key]`, i.e. a lookup operation; and

2. reading the next value from a `dict` iterator, which is also implemented with a lookup operation.

These two operations may optimistically run without locking the `dict`'s mutex, but are sometimes required to fall back to a slow path that requires locking. The slow path is executed when some mutation on the `dict` was performed concurrently.

Let us consider two cases involving a lookup (or eq. an iteration), and another concurrent mutation. Note that concurrent mutations are always obstructed by other concurrent mutations.

Lookup and Update. Suppose two threads t_l and t_u are respectively reading a key k and updating its associated value v, in `dict` H. The following can happen:

1. t_u acquires the lock of H;

2. t_l and t_u find the position of k in H;

3. t_l reads v;

4. t_u updates the value associated with k, from v to v';

5. t_u decrements the reference count of v;

6. t_u observes that v's reference count is 0, and proceeds with a delayed free of v;

7. t_l tries to increment v's reference count, but fails because it is 0;
8. t_l falls back to the slow path and is obstructed in the acquisition of H's lock, which t_u still holds;
9. *…eventually…*
10. t_u releases H's lock;
11. w.l.o.g. assume that t_l acquires the lock (it may be that another thread acquires it first, but later releases it, giving t_l another chance to acquire it, infinitely often);
12. t_l restarts the lookup operation, and finds the position of k in H;
13. t_l reads v';
14. t_l successfully increments the reference count of v' (in fact, the count could not be 0, because v' is still referenced by H, and no mutations can happen in H while t_l is holding its lock);
15. t_l releases the lock of H, and returns v'.

Lookup and Resize. Suppose two threads t_l and t_i are respectively reading a key k and inserting another key k', in `dict` H. The following can happen:

1. t_i acquires the lock of H;
2. t_l finds the position of k in H;
3. t_l reads v;
4. t_i notices that the current load factor α of H is too high to perform the insertion;
5. t_i resizes H;
6. t_l notices that a concurrent resize has happened (let us avoid explicating how this check is performed);
7. t_l falls back to the slow path and is obstructed in the acquisition of H's lock, which t_i still holds;
8. *…eventually…*
9. t_i inserts k';
10. t_i releases H's lock;
11. w.l.o.g. assume that t_l acquires the lock;
12. t_l restarts the lookup operation, and finds the position of k in H;
13. t_l reads v;
14. t_l successfully increments the reference count of v;
15. t_l releases the lock of H, and returns v.

As it can be seen from the above correct executions, CPython's `dict` does not exhibit the aforementioned natural parallelism: operations on distinct keys generate contention. Not just contention actually: they generate obstruction. Furthermore, this applies to *any* two or more keys, differently from a fine-grained locking scheme like the one described in [10, §13.2.2].

3.2 Maier's Concurrent Hash Table

In [17], Maier et al. describe a table migration process that crucially eliminates most of the contention that can arise from a migration. In contrast with the standard literature (see [10, Figure 13.30]) the migration process described does not necessitate the acquisition of one or multiple locks by a single thread that then takes care of performing the resizing. In fact, in the entire Chapter 13 of [10], resizing (eq. migrating) is described only in terms of a "stop-the-world" operation.

Maier's hash table implementation has an important difference when compared with Python's dictionary. In the latter, the least-significant bits of the hash are used to determine the position in the linear array; that is, $d_0(x) = x \bmod 2^s$, with s being the logarithm base-2 of the dictionary size. While in Maier's, the most-significant bits are used: $d_0(x) = x \gg (64 - s)$, where $\gg$ is a right-shift operation. The reason why Maier chose the right-shift based scheme is so that the relative position of two keys which are in the same probe doesn't change as much as in the alternative, when the array is resized.

That is, consider a key x s.t. $h(x)\&2^{s+1} = 0$, and a key y s.t. $h(y)\&2^{s+1} = 1$, where & is a bitwise-and operation. If they were colliding before resizing from s to $s+1$ (i.e. $h(x) \,\&\, (2^s - 1) = h(y) \,\&\, (2^s - 1)$), they will not collide in the corresponding hash table of size $s+1$. But this creates an unexpected problem. Consider the distance-0 position d_0 after the migration $d'_0(x) = d_0(x)$. The d_0 position of y will have changed from $d_0(y) = h(y) \,\&\, (2^s - 1)$ to $d'_0(y) = h(y) \,\&\, (2^{s+1} - 1)$. In other words, $d'_0(y) = d_0(y) + 2^s$. The problem with this behavior is that the position in the new array cannot be safely determined by looking at the prior array, without reading the actual hashes of each key in a probe.

This problem is avoided by choosing the most-significant bits scheme: $d_0(y) = h(y) \gg (64 - s)$, and $d'_0(y) = h(y) \gg (64 - s - 1)$. In other words, $d'_0(y) = d_0(y) \cdot 2 + 1$. While $d_0(x) = h(x) \gg (64 - s)$, and $d'_0(x) = h(x) \gg (64 - s - 1) \Rightarrow d'_0(x) = d_0(x) \cdot 2$.

It follows that the nodes of a cluster in the prior array will "stay close to each other" in the new array. This behavior enables avoiding the necessity to exercise care in the moving of nodes from one generation of the hash table to the next, resulting in much less synchronization overhead.

3.2.1 Complex Keys and Values

The hash table of [17] is limited to integer keys and values. This is too severe a limitation for a Python hash table: it would in fact not be possible to directly handle C integers from Python code, and it would be very desirable to use other types of objects as well.

In order for Python code to interact with this limited hash table, the keys and values would need to be cast to Python `int` types. While possible in principle, it also would entail a performance decrease, since the `int` objects themselves would be contended, even between lookups. Therefore, it would make no difference to store the `int` object pointers themselves in the hash table.

In [17, §5.7], Maier et al. note that it is possible to generalize their hash table to complex keys and values, albeit with some impact on performance. The publicly available implementation advertises the availability of this functionality, thus some work followed the publication of the paper. Notwithstanding, the feature seems not to be functioning. The necessary compiler flag was explicitly disabled by default, and enabling it causes errors. The authors of the library have not responded to inquiries by this author pertaining said functionality. We have resolved to assume that it is not available. We did not have the possibility to assess its performance impact.

The remarks made on memory management are resembling of mimalloc's approach to sharded, thread-free lists, described in [16] and employed in free-threading CPython.

3.3 Bolt

Building on the work described in the previous Section, E. Kahssay presented in his Master Thesis a concurrent hash table named Bolt that allegedly has even better performance. It is based on the application of Robin Hood hashing, a modified linear probing scheme first described by Celis et al. in [3] where already inserted items can be displaced by newly inserted items so as to reduce the variance in probe length. The technique enables the pruning of searches in linear probing.

This leads to a more complex insertion routine. Consider the general situation in which a new item is being inserted. Paraphrasing from [3]:

> Suppose that element A is at location l, at distance d_a from its d_0 slot, and B is to be inserted. Suppose B has been denied allocation of its first $d_b - 1$ choices, and we are now considering l, its d_bth choice.

Then, the Robin Hood invariant maintains that:

if $d_a \geq d_b$: A retains l, the insertion of B continues by considering its $(d_b + 1)$st choice;

if $d_a < d_b$: B displaces A from l, the insertion procedure for A is applied starting with its next $(d_a + 1)$st choice;

The name Robin Hood hashing is derived from an analogy with the popular character:

> This procedure of "taking from the rich and giving to the poor" gives rise to the name Robin Hood hashing. Like Robin Hood, it does not change the average wealth (mean probe length), only its distribution. It is clear that the principal effect is to substantially reduce the variance.

This insertion scheme can involve more than one write to the hash table in order to insert one item: a non-trivial problem for a concurrent hash table.

Consider the linearizability property of concurrent data structures, where it is desirable that the mutation of one method is instantaneously applied. This generally requires to have exactly one write to the shared data structure. For instance, in Maier's implementation the insertion is carried out by a single atomic write into the hash table, and this satisfies the linearizability property. Bolt satisfies this property by locking sections of the hash table, a technique known as fine-grained locking. A thread inserting a new item first acquires all the relevant locks, and then proceeds with the Robin Hood insertion described.

Clearly, the lock-freedom property is lost in this process. The performance characteristics presented in Kahssay's Thesis supposedly compensate for this. Notwithstanding, an implementation was not provided, neither in the Thesis, nor upon request to Kahssay. The author has thus been unable to confirm the results presented.

The Robin Hood scheme was considered interesting nonetheless, because it especially enhances the performance of lookups that fail. In fact, with regular linear probing it is necessary to visit the entire probe before returning a failure, while in Robin Hood hashing, it is possible to prune the search by observing the distance of items in the probe. The approach taken to Robin Hood hashing by this author is presented in Section 4.5.

Performance under contention. Contentious accesses are a common pattern in real-world usages of concurrent hash tables. Bolt's scheme of using fine-grained locking would seem especially subject to degradation in high-contention scenarios, as also noted by Kahssay himself. Let us compare the approaches of Maier et al. [17, §8.4, "Performance under contention"] and of Kahssay [13, §7.4] in evaluating these scenarios. Differently from Maier, Kahssay only evaluates a Zipf distribution with a contention parameter $\theta = 0.75$. In a Zipf distribution it can be that $\theta \in [0, 2]$, which are the bounds considered by Maier. In fact, as noted by Maier, a little contention can help performance. The experiment evaluated by Kahssay comprises a workload of 90% lookups and 10% updates. In such a scenario, it happens that contention actually increases performance due to caching for 90% of the workload. The chosen scenario makes it unfeasible to properly compare Kahssay's and Maier's hash tables under contention. It would have been much more interesting to review a scenario with 100% updates and varying contention parameter θ, as shown in Maier's work. Without any distribution of Bolt, it is not possible to properly review its performance under contention.

3.4 DRAMHiT

In [20] a hash table that allegedly reaches the throughput of main memory is presented. We raise here various doubts on this work and its underlying implementation.

The abuse of empiricism makes it difficult to verify their results, rendering them unreliable. For instance, the observation that:

> [...] on a fill factor of 75–80%, lookup and insertion operations require only 1.3 cache line accesses per request on average. [...] This is critical for reducing pressure on the memory subsystem.

is not backed by any evidence, nor it is ever detailed how such measurements were made. Thus, the measurements are found to be unreproducible.

The obsession with maintaining the highest attainable performance, has induced the authors of DRAMHiT to directly employ assembly code in their implementation, making it substantially less portable. Given that the architectures supported by CPython are numerous, the present author necessitates code portability between different platforms and architectures; it was hard to draw inspiration from this implementation. Also, considering the added burden of supporting several assembly code paths in order to support several architectures, this was considered a poor implementation choice. In addition to the observations on portability, the employment of assembly code also renders the code much less readable and maintainable.

DRAMHiT operations design is largely based on the design of Maier's hash table. Never throughout the paper the lock-freedom property of DRAMHiT is expressly mentioned. We find that it is in fact not lock-free, albeit largely based on lock-free designs. This is chiefly because of the design of [20, Algorithm 1], where the prefetch engine is presented. In fact, it can be easily seen that if one thread t performs a lookup operation, but there are currently not enough pending operations to fill the prefetch window, t is arbitrarily delayed. Furthermore, if there were no running threads other than t, the prefetch window would never be filled, and t would be blocked for an unbounded amount of time; even if it is the only thread running. By the operations presented in the paper, it seems as though a single thread performing a single key lookup is not an effectively supported use-case, which is absurd.

Finally, the choice that made the author most reluctant to accept the design of DRAMHiT was its asynchronous interface. Only accessing prefetched memory is a very desirable characteristic, but in order to do so DRAMHiT chose to radically change the external interface of hash tables. In fact, the common operations enumerated in Section 1.1 are not directly available. Instead, those operations only entail a *request* to the hash table. The program needs to also check for the availability of the response before reading the result. That is, a program as simple as `v = H.lookup(k)` has to become something along the lines of:

```
response = H.lookup(k);
while (!response.is_ready()) {}
v = response.get_result();
```

Such a radical change cannot be well-received in any computing community.

In conclusion, we cannot accept most design choices of DRAMHiT, but it is important to highlight their observation that main memory has become the limiting factor of concurrent hash table designs. This problem is partially addressed in `AtomicDict`'s `batch_lookup` and `reduce` operations.

3.5 Completely Lock-Free Hash Tables

Most operations on Maier's hash table are lock-free. This, crucially, does not comprise the migration operation as well. That a fully lock-free migration process is possible has been shown by Hesselink et al. in [7, §3.5]. Notwithstanding the theoretical achievement, the practicality of such an approach can be subject to debate.

A migration process, as described in [7], comprises $\Omega(n)$ atomic memory writes. An atomic write can be considered to be an order of magnitude more expensive than a normal write. For instance, in the measurements cited in Chapter 6, the throughput of an atomic store operation in the best case is 377MOps/s, while a normal store would generally have a throughput close to the clock frequency of the machine in the best case (constantly writing to uncontended cache, as in the atomic store measurement). In the case of the machine examined, that would be an order of magnitude higher throughput.

Because of the cost of atomic writes, it is easy to see how the cost of migrations quickly becomes prohibitive, as noted in [17] as well:

> While [...] lock-free dynamic linear probing hash tables are possible, there is no result on their practical feasibility. Our focus is geared more toward engineering the fastest migration possible; therefore, we are fine with small amounts of locking, as long as it improves the overall performance.

The present author has also taken the same stance. At the current time, it appears unfeasible to use completely lock-free hash tables in practice. Furthermore, it is hard to believe that such limitations on the performance of atomic operations will ever be lifted by the hardware of the future.

Chapter 4

Design

> There are two ways of
> constructing a software design:
> One way is to make it so simple
> that there are *obviously* no
> deficiencies, and the other way is
> to make it so complicated that
> there are no *obvious* deficiencies.
>
> C. A. R. Hoare,
> The Emperor's Old Clothes.

In this Chapter, we describe the design that we chose for `AtomicDict`, a concurrent hash table for the CPython interpreter. It is a linear-probing, resizable hash table, with double-hashing, and split index and data tables. Linear probing is found to be effective on modern hardware due to its excellent cache behavior. The resizing scheme is drawn from Maier's hash table, and the split index and data tables from CPython's design. We enhance the hash table with other useful features for a concurrent context: synchronous operations, reservation buffers, and iteration partitioning. Additionally, we include methods for batching operations so as to amortize the cost of memory accesses.

We were required to choose a double-hashing scheme because of the differences in hash computations in CPython and in Maier's hash table. That is, if we did not re-hash CPython-generated hashes most keys would collide.

To drive our design choices, we hold the following claims, which we confirm later in Section 6.2:

1. the common usage of a hash table consists of 90% lookups, 9% insertions, and 1% deletes; and

2. most lookups fail.

The first claim should not be surprising. (See for instance [10, §13.1].) The second one may be explained by the commonality of iterations; that is, of operations that return the entire key-value set. Instead of looking up all the individual keys which *are known* to be in the table, it may be that Python programs prefer to iterate over all keys. And furthermore, it may be that when a program looks up one specific key, in most cases it really *isn't known* whether the key is present or not.

Based on this claim, we enhance the table hashing with the Robin Hood scheme [3, 13], which is especially effective in pruning searches for keys which are not in the table. Our approach, detailed later in Section 4.5, differs from [13] in that we don't necessarily constrain the hash table to this scheme: we maintain the Robin Hood invariants as long as we can do so in a lock-free manner. When the maintenance of those invariants requires an increasing number of atomic operations to be carried out in the index, we instead fall back to regular linear probing.

Synchronous operations extend the obstructing nature of Maier's migration design for other operations: consistent size-retrival and sequentially consistent iterations. Adding other sequentially consistent operations in the future remains possible, based on this framework. We therefore maintain like [17], but we extend to synchronous operations, that:

> [...] The hash table works lock-free as long as no [synchronous operation] is triggered.

Finally, we comment on the choice of using CPython's GC to track the internal structures of `AtomicDict` and the impact of this choice regarding lock-freedom.

4.1 Split Index and Data Tables

The hash table itself is an index over another table, called the data table. In the latter is where the pointers to the keys and values objects reside. The index table is sparse, so as to cope with collisions and their resolution, and the data table is dense, so as to provide better performance for iterations and reduce memory usage. Collision resolution only addresses the index. We call *slots* the elements of the index table, and *entries* the elements of the data table. Slots may be occupied by *nodes*.

The size of a node depends on the capacity of the hash table, and it ranges from 1 to 8 bytes. The node contains the location of the entry it points to, the distance from its distance-0 slot, and an optional tag. Depending on the capacity of the hash table, the node and distance fields are rescaled as well, according to Table 4.1.

The tag contains a portion of a key's hash. When the tag is not the same as the equivalent portion of the hash of the key looked up, the relevant entry in the data table is not visited. This reduces the cost of having split index and data tables. That is, when a lookup visits the index, it reads the distance-0 slot of the searched key: a random address in memory. Then, it linearly reads the nodes in a probe, which are likely to be in the same cache line of the distance-0 slot. Each time a tag indicates that the node may refer to an entry containing the searched key, the entry must be read, at a different, random address in memory. The tag serves to amortize the cost of the second random read. It is possible to observe from Table 4.1 that there are definite minimum and maximum sizes, respectively of $2^6 = 64$ and $2^{56} \approxeq 7.2 \times 10^{16}$.

$\log m$	Node	Distance	Tag
6	8	2	0
7	16	3	6
8	16	3	5
9	16	4	3
10	16	4	2
11	32	4	17
12	32	4	16
13	32	4	15
14	32	4	14
15	32	4	13
16	32	4	12
17	32	5	10
18	32	5	9
19	32	5	8
20	32	5	7
21	32	5	6
22	32	5	5
23	32	5	4
24	32	5	3
25	32	5	2
26	64	5	33
27	64	5	32
28	64	5	31
29	64	5	30
30	64	5	29
31	64	5	28
32	64	5	27
33	64	6	25
34	64	6	24
35	64	6	23
36	64	6	22
37	64	6	21
38	64	6	20
39	64	6	19
40	64	6	18
41	64	6	17
42	64	6	16
43	64	6	15
44	64	6	14
45	64	6	13
46	64	6	12
47	64	6	11
48	64	6	10
49	64	6	9
50	64	6	8
51	64	6	7
52	64	6	6
53	64	6	5
54	64	6	4
55	64	6	3
56	64	6	2

Table 4.1: Sizes of internal fields of nodes in the index: m is the capacity of the table, *Node*, *Distance*, and *Tag* refer to their respective sizes, expressed in number of bits.

4.1.1 Reservation Buffers

The usage of pages for the data table, implicitly creates zones of contention. When threads want to add a new key into the hash table, the most-recently added page is the one in which the insertion is attempted. Effectively, all inserting threads may be trying to contend some entry in one page. To reduce this contention, first we pick a strategy in which contention is naturally reduced, then we additionally make threads reserve more than one entry at a time: the necessity for atomic writes on the data table is greatly diminished.

The degree to which contention is exhibited depends on the strategy with which threads decide which free entry in the page they wish to reserve for the new item. Consider a strategy in which the lowest-numbered entry is always chosen: every thread always tries to reserve the same entry, with only one thread succeeding at any given time. A simple, yet much better, alternative strategy is to treat the page itself as a hash table. That is, instead of choosing an entry based on the current availability, an entry is chosen based on the hash of the key. Thus, with sufficiently uniformly distributed hash values, the contention is greatly reduced.

The inserting thread will look for the first available entry starting from the hash of the key being inserted. If it isn't available, it linearly looks for the next one. If no entry is available, a new page is allocated, possibly triggering a grow migration. When a reservation is successful, the thread does not reserve a single entry, and instead reserves a configurable number of consecutive entries. By default it reserves 4 entries. The reserved entries are put into a reservation buffer, which is stored inside the thread's accessor storage, described later in Section 4.3.

The reservation itself needs to be carried out with an atomic write, so that the cost of inserting one key is at most two atomic operations: one write to the data table, and one to the index. The atomic write to the data table, though, is amortized depending on the size of the reservation buffer. In summary, the expected number of atomic writes per inserted key is $1 + 1/4 = 1.25$.

During a shrink migration, the data table is shrunk in addition to the index. That also entails that the entries in the reservation buffers need to be changed, because their location relative to the start of the data table has changed.

4.2 Migrations

`AtomicDict` is a general hash table that can be resized to store an arbitrary number of keys. (Up to 2^{56}.) The process of increasing or decreasing the table capacity is called a *migration*. It is not a lock-free process, because it was found that lock-free migrations are not feasible. (See Section 3.5.) When a migration is initiated, a leader thread is elected and it allocates the new hash table. It additionally performs some bookkeeping operations on the data table. After this initial phase completes, other threads can join and help the migration process. This second phase is essentially Maier's migration routine.

In `AtomicDict` a migration can be initiated from an insertion that found the approximate size of the table to be exceeding the maximum fill ratio of $2/3$, from a deletion that found the approx. size to be less than the minimum fill ratio of $1/3$, or it can be requested from the program that wishes to recover the Robin Hood invariants. The current size of the hash table is approximated based on the number of pages in the data table. (See Section 5.5 for a discussion on size approximation.)

Picking the most-significant bits of the hash to determine the position of a node in the index is crucial, in order to implement Maier's migration. In fact, the usage of the most-significant bits scheme is a core assumption for proving [17, Lemma 5.1]. CPython-generated hashes are designed for a hash table that employs the least-significant bits instead. (Recall Section 3.1.3.) Given this fundamental difference, it was necessary to re-hash Python-generated hashes, or else the d_0 position of many objects would be 0. This is done with a cheap CRC32 hardware instruction, that Maier also used in his implementation. Thus, the required property of hashes being "*large* pseudo-random numbers," is respected.

Maier et al. present different strategies for a migration process to be carried out. With regards to the strategy for "[hiding] the migration from the underlying application," two variants are presented: either "[recruit] user-threads (u)," or "[use] a dedicated thread pool (p)". We find that it would be surprising for the programmer to find out that a thread pool gets created at each instantiation of a hash table, and that choice would also not be much more performant according to [17, §8.4, Using Dedicated Growing Threads]. With regards to the consistency semantics of migrations, two variants are presented again: either "[mark] moved elements for consistency (a–asynchronous)," or "[prevent] concurrent updates to ensure consistency (s–semi-synchronized)." We find that there is one clear choice again: the semi-synchronized strategy seems easier to implement and to maintain, while also being more performant on average, according to Maier's own measurements. Thus, in the remainder of this Thesis, we will only consider what Maier refers to as the *usGrow* migration variant.

4.3 Accessor Storage

The reservation buffer, along with other things, is stored inside an accessor-local storage. It contains:

1. a mutex;
2. a local view of the size of the dictionary;
3. a flag indicating whether this accessor (thread) is participating in the current migration; and
4. the reservation buffer.

The mutex itself protects the contents of the accessor storage. This may seem counter-intuitive given that the storage is local to each thread, but it is actually very useful. The mutex itself is *generally* held only by its accessor, which releases it at the end of any insert, or delete operation.

When an instance of `AtomicDict` needs to be freed, all the allocated accessor storages need to be freed as well. In order to do this, the accessor storages are kept in a list. A thread freeing an `AtomicDict` traverses the list to free the accessor storages. Similarly, a thread accessing an `AtomicDict` instance for the first time, appends its newly created accessor storage to the list.

4.4 Synchronous Operations

When a thread manages to acquire all of the accessors' locks, a quiescent state is reached: no other thread may be found mutating the hash table. In abstract terms, the presence of the list of thread-local mutexes, described in the above Section, enables the creation of cuts in the execution of dictionary operations, inasmuch as the acquisition of all the thread-local mutexes by one thread creates a distinction between the operations that happened before this circumstance and those that happened afterwards.

This crucial characteristic, which is required e.g. for the re-ordering of data table entries, enables also many more usages. A few are described in the following sections. In particular, it enables a very simple mechanism for ensuring that all accessors come to know the presence of a hash table migration, for establishing the correct size of the hash table, and for performing a sequentially consistent iteration of the items in the hash table.

All of these operations are called synchronous because they all share the common necessity to be carried out sequentially by one thread, at least in part. For instance, the hash table migration enjoys the help of more than one thread, but requires a step in which the leader performs the necessary alterations to the data table before other threads can join in the migration.

The addition of this mechanism ensures that the hash table presented here is capable both of lock-free operations, and of complex operations that require the exclusive access to the entire dictionary in order to be performed without a prohibitive number of expensive atomic writes. The properties that a synchronous operation is sequential and ensured to be mutually exclusive make it also very simple to be explained and understood, an important characteristic for collaborative development.

Do note that the presence of a synchronous operation does not obstruct concurrent lookup operations, as also described in the relevant later sections. The meaning is that for concurrent migrations, the lookup linearizes on the state of the hash table prior to the migration (the thread performing the lookup will not be participating in the migration process). This exemption of lookups from the participation in migrations follows [17, §5.3.2, Preventing Concurrent Updates to Ensure Consistency].

4.5 Lazy Robin Hood

To reduce the cost of failing lookups we chose to employ the Robin Hood hashing scheme. We take a lazy approach to this scheme in order to maintain lock-freedom for the insertion routine. The Robin Hood scheme itself is described in Section 3.3, in this Section we describe the approach taken for `AtomicDict`.

The Robin Hood state of the hash table is essentially the distance each key has from its distance-0 slot. In other words, the number of collisions. In order to keep track of this distance, we steal some bits from the nodes in the index. According to [3, Corollary to Theorem 3], the expected number of collisions per probe for a full table is $\Theta(\log n)$. Such is a relatively low number, thus we can steal $\log \log n$ bits from the index nodes. We do so, by following the values in Table 4.1.

The laziness in our approach comes from the fact that if the *aligned* size of the write required to atomically insert the node into the index exceeds 128-bits, then we refrain from maintaining the Robin Hood invariants at all, for we would need to employ more than one atomic write in order to store the newly inserted key. This is because we cannot use the hardware [12, CMPXCHG–Compare and Exchange] instruction over more than 16-bytes. The instruction further requires that the write must be 16-byte aligned. Thus, it is not always the case that the atomic write required can be actually carried out, because the re-ordering of nodes for the Robin Hood invariants may cross a 16-byte boundary.

When the distance cannot be stored in $O(\log \log n)$ space or when a node cannot be inserted in keeping with the Robin Hood invariants, we say that its distance is ∞ and store it as $\log \log n - 1$ (the maximum distance). When a node has distance equals to the maximum distance, we say it is a non-compact node.

Before actually inserting a key that does not maintain the Robin Hood invariants, due to our lazy approach, we mark the hash table as not *compact*. Upon initialization, and after certain migrations, the hash table is marked as compact. A compact operation can be requested by the user to recover the Robin Hood invariants.

When a new node could be inserted, respecting the requirements for our lazy robin hood, but it should be inserted at a distance greater then the maximum, we initiate a grow migration, as described in Section 5.4. Effectively, we have a local trigger for our migrations that doesn't require global knowledge of the table. We can observe the local effect of collisions to determine that the table size is too small, and take action accordingly. One may argue that a degenerate set of hashes can continuously trigger migrations. While this is true in principle, do note an implication of Table 4.1: starting from a table capacity of $2^9 = 512$, the maximum distance is 16, while the number of nodes in a 16-byte-aligned region of memory (a quad word), is 8. Therefore, the described mechanism cannot be triggered indefinitely.

4.6 Amortizing Memory Accesses

It is common for hash tables to be used so that multiple keys are mutated or looked up. Not all hash table implementations provide methods for operating on multiple keys within a single operation. This is not an issue in principle, because one of the basic operations can be called multiple times. We find that providing methods that operate on multiple keys can significantly increase the efficiency of those operations. The amortization scheme described in this Section is intended to be an alternative to the one proposed in [20]. With this proposal, there is no need to radically change the interface of the hash table: some additional methods are provided instead.

Essentially, the costs that we desire to amortize are the stalls that a processor naturally encounters while waiting for the memory subsystem to respond. In a hash table, necessarily, most reads are random, and thus unlikely to enjoy caching or hardware pre-fetching. When a key is looked up, inserted, updated, or deleted, the hash is computed and the corresponding slot or bucket is retrieved from memory. If the program knows, by its own logic, that it will necessitate to read or mutate multiple keys, then the slots of the corresponding keys can be prefetched, before the processor is stalled waiting to read those locations. Even when the requested number of keys is 2 some gains can be observed from this scheme. In fact, it becomes very likely that the processor is stalled waiting for the first read to complete, and that the second one is completed together with the first, so that the processor isn't stalled again.

`AtomicDict` provides two methods for this kind of use-cases: `batch_lookup` and `reduce`, described in Sections 5.7 and 5.8. As the names suggest, the first one amortizes the cost of lookups and the second of mutations.

4.7 Garbage Collection

M. Michael wrote in [18, §2.3]:

> […] The failure or delay of the garbage collector may prevent threads operating on lock-free objects from making progress indefinitely, thus violating lock-freedom.

Such is undeniably correct, and therefore, the proposed design of `AtomicDict` cannot be considered lock-free, simply by virtue of the fact that its memory is managed through CPython's GC. Notwithstanding, this was a deliberate implementation detail and not a foundational piece in the design of `AtomicDict`. The fact that its internal data structures are traced with Python's GC is something that can be easily changed without compromising the design. The choice was chiefly made to simplify the implementation of `AtomicDict`. Furthermore, given the fact that the hash table access was subject to the interpretation of Python code by the CPython interpreter, there was no possibility of not having to deal with a stop-the-world GC.

Python code is not the only means through which `AtomicDict`'s APIs can be accessed. The library also exposes C header files (though, their standalone usage has not yet been explored). For those use-cases that don't employ Python code, but rather directly call `AtomicDict`'s C functions, true lock-freedom can be of interest.

Changing the implementation to not be based on the provided GC, but instead using a dedicated memory management system, is entirely possible; albeit requiring a lot of additional work. One way this can be achieved is by implementing what Michael describes in [19], which employs so-called *hazard pointers*, or any other suitable algorithm.

Serious effort was not put into looking at this possible implementation change, as of the time of writing.

Chapter 5

Implementation

> [Cereus greggii] usually bloom one night a year in June or July. In any given area, they all bloom at the same time.
>
> Wikipedia contributors,
Peniocereus greggii [28].

The implementation presented in this Chapter is available online at [21, src/cereggii/atomic_dict], and comprises 12 C source files, with CPython bindings, and 29 test cases (written in Python code). The bindings to the CPython runtime are not discussed here.

5.1 Lookup

The proposed lookup implementation follows from the design of the hash table described previously. The operation begins by computing the d_0 (distance-0) index slot by calling the `Distance0Of` function, which internally re-hashes the CPython-generated hash, as described in the introduction to Chapter 4, then it linearly probes the index for the searched key, and visits the data table accordingly, possibly pruning the search if the hash table is in a compact state. A fairly complete C-like pseudocode of its implementation is reported in Listing 5.1. The hash computation is omitted for brevity from Listing 5.1.

The algorithm looks up the searched element in the index, starting from position d_0, until either:

1. the item is found;
2. an empty slot is found (the only probe in which the item could be found was looked at in its entirety);
3. the hash table is in a compact state and the search can be pruned due to the Robin Hood invariant; or
4. the probe distance is equal to the table capacity.

The lookup is successful if:

1. there is a node in the index whose tag corresponds to the key's hash;
2. the node is not a tombstone (introduced later in Section 5.3.1);
3. the entry's value is not `NULL`;
4. the stored (Python) hash is not different from the key's hash; and either
    (a) the pointers of the searched key and the stored key are the same (identity); or
    (b) the keys are semantically equal, based on some arbitrary logic.

Comparisons in CPython are notoriously costly. As is well known, Timsort was implemented specifically for CPython, with the explicit goal of minimizing the amount of comparisons required. It can be seen that in lines 38–55, care is taken in order to delay the call to the comparison function as much as possible. Since the types of the involved objects may be implemented in Python code, the comparison can require interpreting an arbitrary Python function. Such function does not have any restrictions on its computational complexity, and even if it was enforced to be an $O(1)$ operation (as it would normally be expected), its constant factor would still be very high, due to the overhead of interpretation. The comparison itself may also raise an exception (a circumstance handled in line 57), either if the two objects' types are not comparable, or if the arbitrary Python code that was executed to compare them raised an exception for any reason.

In the `ReadEntry` sub-routine the key is always read before the value. This avoids the problem of torn-reads, exactly as described in [17, §4, Lookup].

On line 69 the eventuality that the compact state of the hash table has changed is taken into account. Do note that for a generation of the hash table the only transition that can occur to the compact state is from $\top$ to $\bot$. In order to restore the compact state, a migration is required. (The concept of *generation* is introduced later in Section 5.4.) Suppose the check on line 69 didn't exist, and consider two threads t_l and t_i executing respectively a lookup for key k and an insert for the same key. There can exist an execution s.t.:

1. t_l runs first and reaches line 8, reading that `is_compact` $= \top$;
2. t_i runs from start to end, turning `meta->is_compact` $= \bot$, and inserting k with a non-compact node; and
3. t_l runs again, not finding the node pertaining to k, thus returning a lookup failure.

Instead of this evidently faulty behavior, the check on line 69 makes the lookup restart from the beginning, so that item (3) above would be substituted with

3. t_l at first not finding k, sees that a non-compact insertion could have linearized, undermining its assumption that the Robin Hood invariant holds, and thus restarts from line 7, eventually finding k.

Another similar check can be found on lines 72 and 79, where concurrent migrations are considered. It is important to note that lookups are permitted to run concurrently with migrations, as they are read-only operations. When a concurrent migration from the current generation of `meta` to a newer generation has been linearized (setting this metadata's `migration_done` flag to ⊤) it is possible that a subsequent insert for the same key linearizes before the lookup completes, in a manner similar to the previously discussed compactness of the hash table. In such scenarios, the lookup operation is restarted. In this case, the d_0 position of the key must also be recomputed.

Towards the end of Listing 5.1, at lines 68–94, is where the response is written into an externally allocated C `struct`. Given the fact that this routine is also employed in other routines (e.g. see Listing 5.3, line 8), it would be insufficient to return a simple boolean value. Instead, the following is returned:

1. an `error` flag, if ⊤, the rest of the values returned are undefined;
2. a `found` flag, if ⊥, the returned values below are undefined;
3. an integer `position`, indicating at which slot of the index the relevant node was found;
4. an `is_compact` flag, indicating whether the node was found to be compact in the index;
5. a `node` structure, with the node that was found;
6. an `entry_p` pointer, to the relevant data table entry; and
7. an `entry` structure, with the values read from the data table entry.

5.1.1 Linearization

A successful lookup for a key k linearizes on line 41, where the key and value pointers are read. Failed lookups, i.e. lookups that don't find the searched key, linearize on line 15, where a node is read from the index, be it an empty node, or a node n s.t. $d_0(n) > d_0(k)$.

5.1.2 Lock-freedom

It is trivial to see that no thread's arbitrary delay can induce this lookup to also be subjected to an arbitrary delay, insofar as the code presented is concerned. Thus, the lookup operation is lock-free.

It is true that there is one aspect that evades the control of the author, namely the equality comparison of line 55. It can indeed be that the arbitrary code that is run also comprises a lock acquisition. Lock-freedom of that relevant code can in no way be evinced, nor enforced. Therefore, we can only assume (or rather, hope) that the arbitrary code is sane and does not exhibit such surprising behavior.

The `Lookup` above described is not wait-free. In fact, the check at line 69 makes it so that when a concurrent insert sets the metadata's `is_compact` flag to ⊥, the lookup operation needs to be retried. Ibid. for the check on line 72.

Listing 5.1: Lookup Operation

```
1 void Lookup(meta, key, hash, result)
  {
    // assumptions: - meta, key, result != NULL
  d_0 = Distance0Of(hash, meta);

beginning:
  is_compact = meta->is_compact;
  non_compacts = 0;

  position;

  for (d = 0; d < meta->size; d++) {
    position = (d_0 + d + non_compacts) & (meta->size - 1);
    node = meta->index[position];

    if (IsNonCompact(node, meta)) {
      // note: tombstones are non-compacts to entry #0

      d--;
      non_compacts++;
      result->is_non_compact = true;
      goto check_entry;
    }
    result->is_non_compact = false;

    if (node == EMPTY) {
      goto not_found;
    }

    if (is_compact && (
      (d_0 + d + non_compacts - node.d > d_0)
      || (d >= meta->max_d)
    )) {
      goto not_found;
    }

check_entry:
    if (node.location != 0 && node.tag == hash) {
      result->entry_p = GetEntryAt(node.location, meta);
      ReadEntry(result->entry_p. &result->entry);

      if (result->entry.value == NULL) {
        continue;
      }

      if (result->entry.hash != hash) {
        continue;
      }

      if (result->entry.key == key) {  // identity
        goto found;
      }

      cmp = PyObject_Equals(result->entry.key, key);

      if (cmp < 0) {  // exception thrown during compare
        goto error;
      }

      if (cmp == 0) {  // == false
        continue;
      }
      goto found;  // => true
    }
  } // probes exhausted

not_found:
  if (is_compact != meta->is_compact) {
    goto beginning;
  }
  if (meta->migration_done) {
    goto repeat;
  }
  result->error = false;
  result->found = false;
  return;
found:
  if (meta->migration_done) {
    goto repeat;
  }
  result->error = false;
  result->found = true;
  result->position = position;
  result->node = node;
  return;
repeat:
  this_gen = meta;
  meta = meta->new_gen_metadata;
  Decref(this_gen);
  d_0 = Distance0Of(meta, hash);
  goto beginning;
error:
  result->error = true;
}
```

5.2 Insert or Update

The `ExpectedInsertOrUpdate` routine shown in Listing 5.2 departs from Maier's design of [17, Algorithm 1] in that it doesn't expect the user to provide an update function, but rather mimics more the compare and set routines of other atomic data structures: it takes as input the expected and desired values, and it returns the value that was stored before the update was performed (or a special `NOT_FOUND` object). If the expected value was not the one stored, then a special `EXPECTATION_FAILED` object is returned. A routine that more closely resembles Maier's Algorithm 1, is described later in Section 5.8. It is based on the one discussed here.

There's also another special value called `ANY` that can be used as the expected value to signify that whatever the current value is, even an absent value, it should be replaced with the desired value. Returning the previous value, in conjunction with `ANY` as the expected value, also serves to reuse the same routine to behave like an atomic swap, rather than a CAS.

The three special objects `ANY`, `NOT_FOUND`, and `EXPECTATION_FAILED` cannot be used as keys or values in `AtomicDict`, so as to guarantee their semantics are consistent. This is not otherwise shown here.

If the expected value is either `ANY` or `NOT_FOUND`, there is an insertion fast-path used in which the d_0 element in the index is CASed with the entry reserved without going through the currently stored data in the index. If this CAS succeeds, then the operation is completed. If it doesn't, the general code path follows. Notice that this doesn't compromise correctness: if the d_0 node was in fact empty, it means that the key wasn't in the hash table.

Here, it is worth mentioning that this routine is not directly exposed, and is instead considered internal. Instead of calling this function directly, a user of `AtomicDict` is expected to call different Python idioms, which all internally call `ExpectedInsertOrUpdate`; namely:

1. `d[k] = v`, eq. `ExpectedInsertOrUpdate(k, ANY, v)`;
2. `d.compare_and_set(k, exp, des)`, eq. `ExpectedInsertOrUpdate(k, exp, des)`, which raises an exception when the expected value was not the stored value;
3. `d.swap(k, v)`, eq. `ExpectedInsertOrUpdate(k, ANY, v)`, which is similar to (1) except it also returns the previous value.

W.r.t. (2), returning a boolean value is generally considered the standard for CAS operations. Such is what the hardware primitive essentially does, and also what other languages such as Java do. Nevertheless, in order to favor the usage of the `reduce` routine of Section 5.8, which correctly implements a generic mutation to a value, it was chosen to expose this behavior instead. That is, simpler usages, such as incrementing a counter, or any other "aggregations," should instead employ the more appropriate `reduce` method. This `compare_and_set` method is thus intended to be used in scenarios where the expectation is "taken for granted," and would otherwise require debugging if unmet; that is, where the failure of the expectation has more severe impacts than mere contentious access. Do note that this design does not pertain to any correctness or performance goal; instead, it has to do with the programmer's ergonomics in the usage of this lower-level routine. Not every programmer possesses thorough knowledge of concurrency issues, and an interface that alleviates some of those complexities is what the author strives to achieve. Whether or not this is actually simpler to use remains to be seen.

The `ExpectedInsertOrUpdateCloseToDistance0` sub-routine, called at line 35, is omitted for brevity. It essentially implements our Lazy Robin Hood, inserting a compact node in the following steps:

1. consider the nodes in the 16-byte-aligned region of memory the d_0 slot is part of, and, starting from d_0:
   (a) if an empty slot is found and `expected` is neither `ANY` nor `NOT_FOUND`, set `expectation` $= \bot$; otherwise, if `expected` is either `ANY` or `NOT_FOUND`, attempt to insert the node in keeping with the Robin Hood invariant;
   (b) if a non-empty slot is found, check the tag, and possibly call the `ExpectedUpdateEntry` routine;
2. set the `must_grow` flag if the node would be inserted at a distance greater than the current maximum distance (and avoid modifying the index);
3. set the `done` flag accordingly.

The `ExpectedUpdateEntry` sub-routine is also omitted for brevity; its implementation resembles the `check_entry` part of the `Lookup` routine (Listing 5.1, lines 38–65), but also it performs an update, or it possibly signals that the expectation failed.

5.2.1 Linearization

The presented `ExpectedInsertOrUpdate` operation is linearizable. A successful insertion linearizes upon the successful write of the new node into the index. Before that happens, concurrent lookups and deletes have no visibility on the data table entry which hosts the yet-to-be inserted key, since their search is based on reading the index before looking into the data table. After it happens, either with a successful CAS on line 56 (non-compact) or equivalently with a successful CAS inside the compact sub-routine, concurrent lookups will see the new node, and visit the pointed-to data table entry, successfully finding the key. (Consider a call to `ExpectedInsertOrUpdateCloseToDistance0` to essentially boil down to a laborious computation, before picking the right value with which to call CAS. Thus, externally it can be observed to behave in much the same way as a non-compact insertion.)

Pertaining concurrent iterations (described later in Sections 5.9 and 5.10), which mainly focus on reading the data table, the linearization point thus described still holds. When an iteration visits an entry, it additionally checks for the relevant node in the index to be present and if it is not, then it avoids yielding that item.

5.2.2 Lock-freedom

The described `ExpectedInsertOrUpdate` is lock-free, inasmuch as it does not trigger a grow migration. As mentioned earlier, we maintain the lock-freedom property only for those operations that don't necessitate migrations to complete. Where migrations are not concerned, this routine is indeed lock-free: no thread's arbitrary delay can arbitrarily delay this routine as well.

`ExpectedInsertOrUpdate` is not wait-free. It may in fact be delayed in several points. They are:

1. when making a reservation in the data table (which is not shown, for brevity, in Listing 5.2);
2. when failing to make use of the insert quick-path of line 25;
3. when the CAS inside the `ExpectedInsertOrUpdateCloseToDistance0` sub-routine fails, due to the index nodes involved being contended;
4. when a non-compact node fails to be inserted on line 56, again due to contention on the index position;
5. when a concurrent update or deletion changes the already-present key; and
6. when the check on line 103–104 fails, due to another thread changing the index state to non-compact.

Note that these delays do not cause obstructions.

Listing 5.2: ExpectedInsertOrUpdate Operation

```
1  PyObject *ExpectedInsertOrUpdate(meta, key, hash, expected,
                                    desired, entry_loc, *must_grow)
   {
     // assumptions:
```

```
  //   - key \not\in {NOT_FOUND, ANY, EXPECTATION_FAILED}
  //   - expected \not\in {EXPECTATION_FAILED}
  //   - desired \not\in {NOT_FOUND, ANY, EXPECTATION_FAILED}
  //   - (expected \in {NOT_FOUND, ANY}) => entry_loc != NULL
  //   - entry_loc != NULL => (
  //         entry_loc->key == key
  //         && entry_loc->value == desired
  //         && entry_loc->hash == hash
  //     )
  //   - meta, key, expected, desired != NULL

  d_0 = Distance0Of(hash, meta)
  node = EMPTY

  if (expected == NOT_FOUND || expected == ANY) {
    // insert quick path
    node.location = entry_loc->location;
    node.distance = 0;
    node.tag = hash;

    if (CompareAndSet(meta->index[d_0], EMPTY, node)) {
      return NOT_FOUND;  // success
    }
  }

beginning:
  done = false; expectation = true; d = 0; current = NULL;
  is_compact = meta->is_compact;
  to_insert = EMPTY;

  if (ExpectedInsertOrUpdateCloseToDistance0(/* ... */) < 0) {
    goto fail;
  }

  // the above call sets done = true on success

  while (!done) {
    pos = (d_0 + d) & (meta->size - 1);
    node = meta->index[pos]

    if (node == EMPTY) {
      if (expected != NOT_FOUND && expected != ANY) {
        done = true; expectation = false;
        break;
      }
      else {
        if (is_compact) {
          meta->is_compact = false;
          is_compact = false;
        }

        done = CompareAndSet(meta->index[pos], EMPTY, NewNode(
          entry_loc->location, meta->max_d, hash
        ));

        if (!done) {
          continue;  // retry at the same position
        }
      }
    }
    else if (node == TOMBSTONE) {
      // do nothing (i.e. d++)
    }
    else if (is_compact && (
      !IsNonCompact(node, meta) && (pos - node.d > d_0)
    )) {
      if (expected != NOT_FOUND && expected != ANY) {
        done = true; expectation = false;
        break;
      }
    }
    else if (node.tag != hash) {
      // skip
    }
    else {
      updated = ExpectedUpdateEntry(/* ... */);

      if (updated < 0) {
        goto fail;
      }

      if (updated) {
        break;
      }
    }

    d++;

    if (d >= meta->size) {
      // traversed the entire dictionary
      // without finding an empty slot
      *must_grow = true;
      goto fail;
    }
  }

  // note: expected == ANY => expectation == false

  if (expected != NOT_FOUND && expectation == false
   && meta->is_compact != is_compact
  ) {
    goto beginning;
  }

  if (expectation == true && expected == NOT_FOUND) {
    return NOT_FOUND;
  }
  else if (expectation == true && expected == ANY) {
    if (current == NULL) {
      return NOT_FOUND;
    }
    return current;
  }
  else if (expectation == true) {
    return current;
  }
  else {
    return EXPECTATION_FAILED;
  }

fail:
  return NULL;
}
```

5.3 Delete

Albeit an infrequently used feature of hash tables, according to the data shown in Section 6.2, supporting deletions has taken a substantial portion of the development time for the implementation of `AtomicDict`. Several design iterations have been evaluated, and are not explored here in detail. The current proposal, which is not fully implemented as of the time of writing, is what follows.

Deleting a key k from the hash table is considered linearized when the value stored in the data table entry associated with k is set to `NULL`. Being such a conceptually simple semantics for deletes, this is the part that has remained constant throughout the iterations on the design.

After the delete is linearized, the following housekeeping actions are taken:

1. the relevant index node is exchanged for a tombstone, so as to avoid visiting the data table entry when doing a lookup;

2. a synchronous operation may be started, so as to defragment the data table; and

3. additionally, a shrink migration may be triggered, if the estimated size is less than the minimum fill ratio of 1/3, as computed by the upper-bound formula in Section 5.5.

The three additional steps are carried out in the delete routine, because:

- deletes are considered infrequent, so adding cost to them and relieving other routines of this work is considered to be a decrease in cost overall; and

- if deletes happened to be frequent instead, it would likely be that the program expects the memory consumption of the hash table to decrease.

5.3.1 Exchanging the node for a tombstone

The swapping for a tombstone node is fairly simple and is shown in lines 33–36 of Listing 5.3. Recall that this swap is necessary, as opposed to removing the node. (See Section 4.5, and [14, §6.4].) In our circumstances, it is especially useful in order to help concurrent lookups: the thread looking up a key k may visit the data table entries pertaining to the visited nodes in the index, possibly finding that k had been deleted by reading that its associated value = `NULL`. But reading a data table entry is much more costly than reading a node in the index, given that the next node to visit is very likely to already be in the processor's cache, while the entry is likely not.

The `LookupEntry` routine, omitted for brevity, is essentially equivalent to the `Lookup` routine shown in Listing 5.1, but instead of looking for a node that points to an entry that contains the searched key, it looks for a node that points to a certain fixed location of the data table; thus it is an operation that only accesses the index.

The tombstone node is possibly moved further down its probe, if such is possible within a single atomic write; that is, as long as the atomic write is of at most 16-bytes, and is 16-bytes aligned.

5.3.2 Defragmenting the data table

To help defragment the data table, we keep a per-page deletions counter noted as $\delta(p)$, $\forall p \in \Pi$, the set of pages. After a delete, $\delta(p)$ is atomically incremented. If $\delta(p) \geq \frac{|P|}{2}$, a merge is triggered.

The following is what occurs after a merge operation has been triggered.

1. Look for another page p', distinct from the page from which the merge was triggered, s.t. $\delta(p') \geq \frac{|P|}{2}$.
   (a) Since page number 0 cannot be cleared because it hosts entry number 0, which has to remain for the correctness of tombstones, start the search from page number 1 (these checks are omitted for brevity from Listing 5.3); and
   (b) if no such page can be found, abort the merge operation.
2. W.l.o.g. call p_1 the lower-numbered page, and p_2 the other.
3. Let p_0 be the lowest non-empty page in the data table. It may be that $p_0 \equiv p_1$, and that is fine.
4. Move the elements found in p_0 into p_2. When p_2 becomes full, move the remaining elements into p_1 instead. (Note that if $p_0 \equiv p_1$, then $|p_0| = |p_1| \leq \frac{|P|}{2}$, and also $|p_2| \leq \frac{|P|}{2}$, therefore $|p_0| + |p_2| \leq |P|$. Thus, all elements from $p_0 \equiv p_1$ will be moved into p_2.) That is, $\forall e \in p_0$:
   (a) clear $e \in p_0$;
   (b) write e into a free slot of p_2 (possibly overwriting deleted slots);
   (c) if e is a reservation with unoccupied slots left:
      i. write the unoccupied slots too;
      ii. look into every accessor's storage (this is already a Synchronous Operation) and edit the relevant reservation to point to the new slot; and
   (d) update the relevant node in the index to point to the new location.
5. Update $\delta(p_0)$, $\delta(p_1)$, and $\delta(p_2)$ accordingly.

The relevant pseudocode can be found in Listing 5.3, lines 40–87. Note that step (4) above is summarized on line 70.

Triggering a shrink migration. After a merge operation completes, as described in the above Section, a shrink migration may be triggered, following the increment of the greatest deleted page field in the metadata. It would cause the size of the index and of the pages array to be reduced, if usage is below a certain threshold, which by default is of 1/3.

5.3.3 The current implementation

The current implementation, which is not shown here, is essentially similar to the one described so far, but instead of the page-merge algorithm, defragmentation is carried out by swapping entries. When a key is deleted, its entry is swapped with another entry stored at a lower-numbered location, so as to maintain the possibility of eventually incrementing the deleted pages counter (P_D in Section 5.5).

Notice how the necessity for defragmentation remains, in order to provide a fairly cheap and wait-free approximation of the size, which is required by the `Delete` and `ExpectedInsertOrUpdate` routines in order to know when to trigger a migration. Other mechanisms for triggering migrations that don't require defragmentation could have been adopted, but this one is particularly useful because it also makes iterations faster, by reducing the necessity of an iterating thread to visit empty entries of a fragmented data table.

The above is considered possible to implement with the lock-freedom property. Nevertheless, it exhibits a problem concerning torn reads: suppose a thread reads the key part of the entry before the entry is marked as deleted, then it gets deleted and swapped, the thread that read the previous key now proceeds and reads the swapped value, thinking that it pertains to the prior key. It would therefore not be linearizable.

Swapping entries to defragment unoccupied slots of data table pages also doesn't provide any immediate benefit: memory reclamation is deferred to the next shrink anyway, whereas with merges, memory is reclaimed sooner and more frequently.

5.3.4 Linearization

The presented `Delete` operation is linearizable. Its semantics (i.e. removing a key k from the key-set K) are observable instantaneously at line 18 when the value associated with k is atomically set to `NULL`. Such is for a successful invocation, i.e. one that actually removes k from K. Whereas a failed invocation, i.e. one that finds $k \notin K$, is linearized inside the call to the `Lookup` routine of line 8. With the `Lookup` routine being already shown linearizable in Section 5.1.1, it follows that also a failed invocation of a delete operation is linearizable.

`Delete` is linearizable also in spite of concurrent deletions of the same key k. Under such circumstances, the successful CAS of line 18 by the one successful invocation linearizes that call to `Delete`, as above; while the failed CAS of the same line, will result in visiting line 22, forcing the delete operation to be retried, and then failed.

5.3.5 Lock-freedom

The presented `Delete` operation is lock-free, when defragmentations or migrations are not required. (And when they are, the linearization point happens before the non-lock-free part of deletions begins.) `Delete` is not wait-free, because a delete for a certain key k by thread t_1 may be delayed by a concurrent delete, or eq. an update, for the same key k by another thread t_2. When that happens, t_1 is delayed in performing a retry, as specified at line 22. It follows that the presented deletion is not wait-free.

Listing 5.3: Delete Operation

```
int Delete(meta, key, hash, *should_shrink)
{
  // assumptions:
  //   - meta, key != NULL
  //   - key \not\in {NOT_FOUND, ANY, EXPECTATION_FAILED}

beginning:
  result = Lookup(meta, key, hash)

  if (result.error) {
    goto fail;
  }

  if (!result.found) {
    goto not_found;
  }

  while (!CompareAndSet(&result.entry_p->value, result.entry.value, NULL)) {
    ReadEntry(result.entry_p, &result.entry);

    if (result.entry.value == NULL) {
      goto beginning;
    }
  }

  result.entry_p->key = NULL;
  Decref(result.entry.key);
  Decref(result.entry.value);
  result.entry.value = NULL;

  location = result.node.location;

  while (!CompareAndSet(&meta->index[result.position],
      result.node, meta->tombstone)) {
    LookupEntry(meta, location, hash, &result);
  }

  *should_shrink = false;

  page = PageOf(location);
  if (AddThenFetch(page->del_counter, 1) >= PAGE_SIZE / 2) {
    BeginSyncOp();

    p_0, p_1, p_2 = NULL;
    p = 0;

    for (; p <= meta->greatest_allocated_page; p++) {
      if (Size(p) > 0) {
        p_0 = p;
        break;
      }
    }

    for (; p <= meta->greatest_allocated_page; p++) {
      if (p->del_counter >= PAGE_SIZE / 2 && p != page) {
        p_1 = p;
        break;
      }
    }

    if (p_0 != NULL && p_1 != NULL) {
      if (page > p_1) {
        p_2 = page;
      }
      else {
        p_2 = p_1;
        p_1 = page;
      }

      Merge(p_0, p_1, p_2);  // cannot fail

      assert(Size(p_0) == 0);

      Decref(meta->pages[p_0]);
      meta->pages[p_0] = NULL;

      meta->greatest_deleted_page++;
      used_pages = meta->greatest_allocated_page
        - meta->greatest_deleted_page
        + meta->greatest_refilled_page;
      if ((used_pages) * PAGE_SIZE <= meta->size * MIN_USED_RATIO) {
        *should_shrink = true;
      }
    }

    EndSyncOp();
  }

  return 1;

not_found:
  return 0;

fail:
  return -1;
}
```

5.4 Migrations

When the hash table becomes too full ($\geq 2/3$ used slots), or too empty ($< 1/3$ used slots), a migration is triggered by the insert or delete routines. At its core, such is necessary because it is not possible to simply add more slots into an array after it has been allocated; and the index and data table pages array are in fact regular arrays. Thus, in order to allow for more keys to be inserted, or for memory to be reclaimed, it is necessary to substitute those arrays.

`AtomicDict` allows for three kinds of migrations: *grow*, *shrink* and *compact*. Respectively, they allow:

1. to double the capacity of the hash table,
2. to halve it, and
3. to recover the Robin Hood invariants, which may have been lost due to our Lazy Robin Hood approach, described in Section 4.5.

The three kinds are slightly different from each other in their implementation, but are externally signalled in the same manner: by setting a field in the current metadata to show that there is a migration in progress.

Migrations are executed entirely or in part within a Synchronous Operation. The application of a Synchronous Operation relieves the implementation from being concerned with the following problem, and its variations.

1. A thread t_i is performing an insertion of a key k.

2. Another thread t_m is performing a migration to a newly allocated index.

3. t_i reaches its linearization point before the migration completes.

4. Has k been migrated?

Instead, it becomes much easier to implement, and maintain, a logic in which thread t_m initiates a Synchronous Operation, so that all other threads either complete their mutation before the migration starts, or know that a migration is in progress and help carry it out before resuming their work. This is effectively the same approach expressed in [17, §5.3.2, Preventing Concurrent Updates to Ensure Consistency].

In principle, one may argue that updates (as opposed to inserts) don't need to wait for a migration to complete, since all they need to do is to change the value pointer in the relevant data table entry. Nevertheless, the update might need to be retried in case it returns that the searched key was not found and a concurrent migration completed in the meantime. Additionally, a migration might in fact leave the data table untouched only if it happened to be a grow migration. A compact, or shrink migration will instead try to reduce the size of the pages array of the data table, thus actually performing some operations on both the data table and the index. Depending on the migration's current state, it may be that the update is referred by an index node to a wrong entry in the data table. Therefore, updates are obstructed by migrations, in the same manner as inserts.

5.4.1 Metadata

Most of the `AtomicDict` data is stored inside its metadata field, which holds pointers to the index and to the data table, among other useful things. An external observer may perceive a migration as a change from a prior metadata object to a new one. We will also refer to them as the current generation metadata and its successor, or the new generation metadata.

In addition to the aforementioned pointers, which are the core parts of metadata objects, are also stored therein:

1. the capacity of the hash table (a power of 2);

2. the pages counters (described in Section 5.5);

3. the `is_compact` flag;

4. various information about the sizes of index nodes and their internal parts (see Table 4.1);

5. a pointer to the next metadata generation (possibly set to `NULL`);

6. the migration leader thread id;

7. the next node to migrate counter; (In Maier's terminology, this is the blocks counter. The next node to migrate is effectively the first node in what Maier denotes as a block)

8. three atomic events used to signal the progress of the three phases of a migration; and

9. other miscellaneous information.

The metadata object is stored inside an `AtomicRef`, described in detail in Appendix A. It essentially provides an interface for an atomically updatable reference to a `PyObject`, that also handles correct `PyObject` concurrent reference counting. Specifically, the objects stored therein are subject to QSBR, and thus are not immediately freed upon reaching a reference count of 0. Refer to Section 2.3 for a discussion on QSBR. Not having QSBR for metadata objects would make the implementation of `AtomicDict` subject to use-after-free bugs.

5.4.2 Migration Leader

The leader election process is very simple. We entirely rely on the processor's cache coherency protocol to solve the consensus problem.

An atomic CAS instruction is executed by possibly several threads. They try to write their own identifier into the same field in the current generation metadata. Exactly one thread will succeed in its attempt: it becomes the migration leader.

We use CPython's `_Py_ThreadId` routine to generate the identifier, which is guaranteed to associate a unique number with each thread. Therefore, it is not possible to incur in the ABA problem. (See paragraph "Thread Identifiers" in Section 2.1, and [10, §10.6] for a discussion on the ABA problem.)

5.4.3 Migration Phases

Migrations are carried out in three phases: the preparation of the new generation metadata, carried out by the leader alone, the node migration phase, which is where the bulk of the migration work is performed by both leader and follower threads, and the metadata swapping phase, performed by the leader. So as to block concurrent mutation operations while a migration is in progress, the migration leader starts a Synchronous Operation before the first phase begins.

The first phase of a migration is where the new generation metadata is prepared. The leader allocates the new index and pages array, and copies the non-deleted pages into the new pages array. Additionally, if the migration is a compact or a shrink, the leader clears the new index.

The Synchronous Operation ends here for shrink and compact migrations, in order to allow new accessors to join the migration process. Whereas for grow migrations the Synchronous Operation ends only after the entire migration process is completed. The reason for this limitation is that the threads that participate in a grow migration essentially need to hit a barrier and wait for all participants to have hit it. In order for this to work, the number of threads that will need to hit the barrier must be known in advance. Thus, we cannot allow the possibility of new accessors joining the migration process after it has started. This is not necessary for shrink and compact migrations because they use a different semantic in order to determine whether the migration has completed: namely, that the generation tag of each page in the data table is set to the new generation.

After the new generation metadata is ready, the follower threads may start migrating the nodes from the current generation to the new one. How they do so varies depending on the kind of migration. Let us refer to the shrink and compact migrations as *slow*, and to the grow migrations as *quick*. The quick migration is essentially the same as Maier's migration scheme.

Finally, after the node migration phase completes, the leader thread swaps out the current metadata for the new one and signals to the other threads that the migration has finished. This applies to both the quick and slow migrations.

A call to the `compact` method triggers a compact migration. If, after the migration, the new generation is still not compact, further grow migrations are triggered, until a compact generation is reached.

5.4.4 Slow Migrations

The slow migration is conceptually simpler than the quick counterpart: migrating threads compete to lock data pages for themselves and migrate the valid entries into the new generation index. This is done using the common insertion routine of Section 5.2. After the migration of a page is completed, the thread that previously locked the page sets the page's generation to the new generation.

To improve contention on the acquisition of pages, each thread, identified by i, starts migrating pages from the $i \mod P_A$ page, with P_A being the greatest allocated page's number. After having visited all pages, it runs a second visit to check if all the pages' generations have been set to the new generation. If it does observe that, then it signals to the other accessors that the node migration phase is complete.

5.4.5 Quick Migrations

During a quick migration, a participating thread sets a local "participation" flag to 1 when joining the migration process (initially set to 0 for all accessors). The node migration phase is then very similar to the one described in [17, §5.3.1].

A global counter is kept in the current generation metadata that specifies the next node to migrate. A thread runs an atomic fetch-and-add operation to reserve a certain block, i.e. a portion of index to migrate. By default the block size is 4096 index slots, consistent with the implementation of [17]. So that it is possible for a single thread to perform the entire migration process, if the current index size is sufficiently small. In fact, the additional contention generated for smaller block sizes is not worth the blocking of the other threads. After a block is reserved, the thread that made such reservation is free to not use atomic operations to migrate the nodes involved, because it effectively reserved both a portion in the current generation index, and in the new generation index.

Consider a block of size 4096 in the current generation index that starts at position s, i.e. $[s, s+4096]$. By [17, Lemma 5.1], we know that the involved nodes must be migrated within the $[2s, 2(s + 4096 + 1)]$ block of the new generation index. That is, grow migrations are only allowed to use a growth factor $\gamma = 2$. By the semantics of the atomic fetch-and-add operation employed, it is known that s is distinct among all participating threads, therefore the blocks in the current generation are distinct, as well as their counter parts in the new generation. Thus, there is no need for further synchronization between migrating threads than the fetch-and-add operation.

A migrating thread, having reserved a block, clears the relevant portion of the new generation index and proceeds with inserting the same nodes into the new generation index, skipping over tombstone nodes.

Additionally, each thread tries to maintain the Robin Hood invariants, falling back to linear probing if that is not possible. Note that there not being a necessity to use 16-byte aligned CAS operations, the Robin Hood invariants are maintained to a larger extent, compared to the normal insertion routine. After being done, the participating thread hits a barrier to synchronize the migration phase with the other threads.

5.5 Approximate size

An approximation of the number of elements stored in the hash table is inferred by the number of pages present. Three separate, atomic counters are kept:

- the greatest allocated page P_A;
- the greatest deleted page P_D; and
- the greatest refilled page P_R.

The P_A counter is also used to keep track of the used portion of the pages array, and is the one used when a thread makes a new reservation into the data table; possibly, triggering the allocation of a new page and increasing P_A.

The three counters serve to provide a rough estimate on the number of elements currently stored: as new elements are inserted new data table pages need to be allocated, and when numerous elements are deleted they get defragmented into their own pages, before the pages are eventually freed, or refilled. Initially, their respective values will be 0, -1, and -1.

So an upper bound for the number of elements contained in the dictionary is:

$$(P_A - P_D + P_R) \times |P|$$

The bound is then also refined by visiting the P_A and P_R pages, decrementing it of the amount of empty entries in those pages. There is also a lower bound, which is computed by decrementing the upper bound by the unused reservations, accounted by traversing the accessors' storages. Therefore, the running time complexity of this operation is $\Theta(t)$.

The mean (floor division by 2) of the lower and upper bounds is then reported to the user upon request.

This routine is also employed by the insert and delete routines to decide when to trigger migrations.

5.6 Consistent size

To provide a sequentially consistent computation of the hash table size, each accessor has a thread-local size counter: each thread keeps track of how many items it has inserted, incrementing its local counter, and how many items it has deleted, decrementing its local counter. (Thus, it may be that one thread's local counter is negative.) The thread-local counters are mutated using regular reads and writes: they are protected by the accessor's lock.

Upon request, a Synchronous Operation to safely sum the thread-local counters, begins. The local counters are reset to 0 before the operation ends. The result is cached into the `AtomicDict` and a dirty flag is reset in order to avoid restarting a Synchronous Operation at every call, if no insertions or deletions marked the cached value as dirty. The cached value is stored with a CPython integer. Upon a new request, if the dirty flag is unset, the cached value is returned.

Note that it is much faster to keep thread-local counters and then acquire locks to read them, rather than to keep a shared counter. Using a Synchronous Operation for this also implies that its semantics are trivially sequentially consistent. It can be argued that to retrieve this consistent size, there is a missed opportunity cost related to the exclusion of all other threads from using the hash table. While true in principle, it should be noted that this is a $\Theta(t)$ operation, thus independent of the table size, and generally quite fast. For these reasons, this is the routine that is exposed by default, i.e. when calling `len(AtomicDict())`. If a program cannot withstand the added cost of mutual exclusion, perhaps because it frequently asks for the size of the hash table while this is being concurrently mutated, it is still possible to retrieve the approximate size described in the above Section, which is quite accurate and lock-free. Under high-frequency concurrent mutations, it may make more sense to retrieve an approximation of the size rather than a sequentially consistent size: it may be invalidated immediately upon returning the output, so it would never actually be precise.

Notwithstanding the limitation on the number of nodes that can be present into the described hash table at any given time (2^{56}, as prescribed by Table 4.1), a counter of 64-bits is not necessarily sufficient. Consider the situation where two threads concurrently insert and delete 2^{64} items, so that one thread only performs insertions, and the other only performs deletions (or any other equivalent situation). Care then needs to be taken in order to avoid running into integer overflows or underflows. Before such an event occurs, the thread that detects it (its local counter is `MAX_INT64` during an insert, or `MIN_INT64` during a delete), the thread repeatedly tries to add (CAS) its local counter to the `AtomicDict`'s cached value, while holding its local accessor's storage mutex. This way, no Synchronous Operation can begin, and it is safe to add the protected value, before resetting it to 0. If another thread finds itself in the same condition, the competing threads will serialize based on the CAS to the cached counter.

5.7 Batch lookup

Inspired by the design of DRAM-HiT, this method was added so as to amortize the cost of multiple memory accesses when a program wants to read a number of keys from the hash table. It deviates from [20] in that it does not require the user to continuously poll the table for results. Instead, a batch of keys can be submitted for lookup, in one invocation.

An example call:

```
foo = AtomicDict({'a': 1, 'b': 2, 'c': 3})
result = foo.batch_getitem({
  'a': None,
  'b': None,
  'f': None,
})
assert result == {
  'a': 1,
  'b': 2,
  'f': cereggii.NOT_FOUND,
}
```

The `cereggii.NOT_FOUND` object is a special, global object that cannot be inserted into `AtomicDict` (as previously described in Section 5.2). The fact that it is in the output means that the key was not in the dictionary, as the name implies.

The operation involves prefetching each d_0 location of the keys in the batch. This results in an amortization of the cost of individual memory accesses: if the number of prefetches issued is sufficiently large (how large depends on the hardware), then the processor will be stalled waiting for the first read, while the subsequent ones will be read from the cache without incurring in the cost of main memory access. This behavior is cheaper-by-comparison: it is cheaper to prefetch several keys, possibly blocking at the first one, than to block at every read.

The submitted batch is subdivided into chunks of a configurable size. This is intended to prevent memory over-fetching. That is, the cache of an individual processor may become overly filled with the prefetched portions of the index, and the hardware will proceed to evict lines of cache before they are actually read by the lookups that requested them.

NUMA architectures will enjoy this mechanism particularly: when prefetches are issued at remote memory addresses, the comparative cost of the amortized reads is much less than the sum of the costs of each individual read.

Concerning NUMA architectures, it would generally also be possible to argue that the cost of a remote access may be hidden by the possibly many local memory accesses that precede it. Such is not our case: the index, which is allocated as a contiguous array, will reside entirely in a single NUMA node. Thus, a thread accessing the index will either only issue operations to remote memory, or to local memory. NUMA support is thus limited: the presence of a single shared index makes it impossible to efficiently make use of memory partitioning.

Sharding. A further endeavor may be carried out so as to provide an `AtomicShardedDict` class which creates its own pool of threads, bound to individual NUMA nodes, and affords access to the hash table shards to those threads only. The partitioning of a hash table into several shards requires that any given key must be uniquely assigned to one shard (e.g. by using the least-significant bits of the hash), each of which is managed by one or multiple threads associated with a single NUMA node. The user thread would submit requests to a message queue which is consumed by the threads associated with the relevant shard. This is resembling of the design of [20], though this design need not be asynchronous in order to gain the speed-up of stronger memory associativity. It can be implemented by making use of the current implementation of `AtomicDict`, such that one shard of `AtomicShardedDict` is an instance of `AtomicDict`.

5.7.1 Linearization

The linearization of this method is intended as the individual linearizations of the distinct lookup operations. Each distinct lookup is linearizable, although the overall invocation is not. Therefore, it is possible for a result of this method to not be sequentially consistent.

5.8 Reduce

A common pattern when using CAS operations is to repeatedly make the call to CAS in order to cope with contention. Additionally, it is common to apply a single mutation to many keys in a hash table, or eq. to insert many keys. We are especially interested into efficiently and ergonomically handle the mutation of the values of a given set of keys, for such is an important use-case for hash tables in general. We propose here a method already known in computing as *reduce*. Such is a foundational concept of many computational models, most notably of those based around the concept of map-reduce, like Hadoop. We consider that this method eases the correct mutation of a concurrent hash table: it correctly handles contention internally. Finally, this method also provides significant performance improvements. (See Section 6.1.2 for a commentary on the exhibited performance.)

The interface, which is believed to satisfy both the efficiency and the ergonomics goals, is very similar to [17, Algorithm 1]. Instead of providing an expected and a desired value, the programmer specifies the desired mutation, i.e. a function over the current values, like so:

```
d = AtomicDict()

data = [
    ("red", 1),
    ("green", 42),
    ("blue", 3),
    ("red", 5),
]

def count(key, current, new):
    if current is NOT_FOUND:
        return new
    return current + new

d.reduce(data, count)
```

A key may be already present or not; thus, the function must handles both cases. Furthermore, there should be no assumptions as to the number of times `mutation` is called before it is actually applied. In fact, it may be called multiple times due to contention.

For each key, the `reduce` method behaves like the following:

Listing 5.4: A common pattern when using CAS instructions.

```
expected = d.get(key, NOT_FOUND)
```

```
while True:
  try:
    d.compare_and_set(key, expected, mutation(expected))
  except ExpectationFailed:
    expected = d.get(key, NOT_FOUND)
  else:
    break
```

The specific method of calling CAS in the above Listing is very commonly known in the literature. (Albeit certainly more common to be written in a form where the CAS operation returns a boolean, rather than raising an exception. See also the relevant discussion in Section 5.2.) The fact that this is well known in the literature of concurrent data structures, though, does not directly imply that it is also well known in the literacy of Python programmers. For the many programmers who are not accustomed to the domain of concurrency, the above Listing should probably be explained in detail, before its semantics are appreciated. Providing a transparent and standardized way to access such common knowledge is therefore considered fruitful.

Do note that the atomicity of this `reduce` method is violated if within the provided mutation function, a lookup into `d` is performed. In other words, this is not a suitable design for a mutation that requires a sequentially consistent view of more than one key in the hash table.

The parameter `key` of `mutation` is provided so as to satisfy the possible cases in which the computation is dependent on the key itself.

Contention is greatly reduced by first accumulating an intermediate result into a thread-local table, and then applying the mutation function to the shared table. The repetition of keys in the input does not generate contention: this operation requires $\Omega(|K|)$, instead of $\Omega(n)$ atomic writes to memory, where n is the number of items in the data and $|K|$ the number of distinct keys. As keys in the data are repeated more frequently, the number of atomic writes required decreases in proportion. In fact, a piece of real-world data is often found to be skewed towards a small subset of its own keys; that is, $\Omega(|K|) \ll \Omega(n)$.

Finally, the memory accesses cost is amortized as in the `batch_lookup` method, when atomically writing the local result into the shared hash table.

Linearization and lock-freedom. The linearization and lock-freedom of `reduce` chiefly depends on the respective properties of the `ExpectedInsertOrUpdate` routine that is being called in order to implement it. Thus, refer to Sections 5.2.1 and 5.2.2.

5.9 Iterations

Iterations are designed to be particularly efficient in `AtomicDict`, mostly because of the design of the data table, which helps with cache efficiency. Speculative hardware memory pre-fetching can also help by observing the linear access pattern. (In [17], iterations are referred to as *Forall* operations.) Additionally, the partitioning scheme described in the following Section enables truly multi-threaded iterations, by creating per-partition iterators. This is something that is not available in any other reviewed hash table implementation.

The presented iteration scheme, accessible through the `fast_iter` method, has been showed to perform twice as fast as CPython's `dict` iterator, even for single threaded executions. (See Section 6.1.2 for measures.)

Iterating is done primarily over the data table. A thread starts by fetching the first page, then reads the first entry of the page, yielding the item back to the caller, after storing the next location to visit, skipping over empty or deleted cells. The Python code then loops calling this same function again. This requires the creation of a small iterator `PyObject` to keep track of the next location to visit, a necessity for adhering to the CPython loop protocol regardless of our designs.

5.9.1 Iteration Partitioning

The above iteration scheme is further enhanced with partitioning. When multiple threads want to iterate over the same hash table, it may very well make sense that they visit disjoint subsets of the stored key-set.

This is achieved by letting the thread declare the number of partitions π, and the partition number i assigned to it (a thread identifier of some sort), s.t. $i \in [0, \pi)$. For the partition to actually yield disjoint subsets, π has to be known to all participating threads, and is intended to be equal to their number, and i needs to be assigned uniquely to each thread. A call to `fast_iter(partitions=`π`, this_partition=`i`)` will then visit those pages for which their page number $p \bmod \pi = i$.

The number of keys per subset is not guaranteed to be to be uniformly distributed. Nevertheless, if the key-set is large enough (that is, $|K| \gg \pi \times |P|$), and the number of deleted keys per-page is nearly constant on average, then the distribution of keys per partition is also nearly constant on average. (That is, let $\mu = E[d(p)]$ and σ its standard deviation, $\sigma \ll \mu$.)

5.9.2 Non-linearization and lock-freedom

While the lock-freedom property this operation enjoys is trivial, its linearization is trickier. This iteration is in fact not guaranteed to be consistent with any sequentially consistent access, and is therefore not considered linearizable. In fact, repeated calls to ask for the next element in the iterator are required in order to comply with CPython's iteration protocol. As such, it cannot be that "[the] method call [appears] to take effect instantaneously at some moment between its invocation and response" [10, Principle 3.5.1].

It is therefore advised to use this faster iterator when it is somehow known that there will be no concurrent mutations, by the logic of the program. Otherwise, the following sequentially inconsistent behaviors might be observed:

1. the same key is emitted more than once; or

2. an update u_1 that happened strictly before another u_2 is not seen, while the latter is. (Such is the case when two updates are executed in succession by the same thread. Thus, it is definitely known that $u_1 \to u_2$.)

These may be very surprising behaviors for a programmer trying to debug an incorrect execution of an iterating thread.

Those behaviors may be exhibited in the following circumstances, respectively:

1. the iterating thread t_l visits the data table entry e in which some key k is stored → another thread t_d deletes k → another thread t_i re-inserts k at a location e' in the data table s.t. $e < e' \rightarrow t_l$ visits e' in which k is stored. (It may also be that $t_d \equiv t_i$.)

2. the iterating thread t_l visits the data table at location e where the key k_1 is stored → update u_1 to key k_1 is performed by another thread t_{u_1} → update u_2 to key k_2, stored at location e' s.t. $e < e'$ is performed by another thread $t_{u_2} \rightarrow t_l$ visits the data table location e', seeing the effect of update u_2. (It may also be that $t_d \equiv t_i$.)

5.10 Proposal for a consistent iteration

The following proposal for a consistent iteration has not been implemented as of the time of writing, but it is fully described here. Once implemented, it will be the default iteration scheme adopted, given that its semantics are much more immediate to understand. To implement a cost-efficient and consistent iteration seems to be possible, based on the following design:

1. the iterating thread t begins a Synchronous Operation;

2. it generates an iteration ID i (the pointer to the C structure holding the iteration data);

3. it allocates a new iteration index and copies the shared index into it;

4. it allocates a new iteration pages array;

5. it traverses the `AtomicDict`'s pages array, and for each page:

   (a) it sets the `iteration` field on the page to i;

   (b) it copies the page pointer into the local pages array;

   (c) it increments the page reference count;

6. t ends the Synchronous Operation; and

7. proceeds iterating over the new pages, in a manner similar to the iteration described in Section 5.9.

8. When the iteration completes, it decrements the reference count of the pages in the local pages array, and frees the array.

When the iteration field of a page is set, the page is considered to be immutable. A thread that wishes to mutate it, should thus first copy the page's contents into a newly allocated page, and then atomically swap the two pages in the pages array. If the swap fails, then it means that another thread has swapped the page into a mutable page. It cannot be that the newly swapped-in page is immutable because that requires holding all of the accessors' locks.

The creation of new mutable pages is therefore entirely offloaded from the iterating thread to the other `AtomicDict`'s accessors. This is a feature, rather than a compromise, of this design. Consider the opposite case in which the iterating thread additionally has to allocate more pages and swap them in the pages array, or equivalently has to create its own immutable pages. In this scenario the Synchronous Operation itself would take much longer to complete. Thus, it would hinder the performance of the other accessors which are trying to apply mutations to the hash table (recall that lookups are permitted in spite of Synchronous Operations). That is, there is a missed opportunity cost associated with such design.

Instead, the parallelization of threads which independently swap the immutable pages out for their mutable copies, entails that multiple threads can be used to create the new pages. And, furthermore, that if no mutations were actually going to happen, either at all or in a subset of the pages, then strictly less work is required overall.

It may also be entirely possible to permit the re-usage of the iteration ID. So that if a thread sees that all pages refer to the same iteration ID, then it avoids marking the pages as immutable, and instead copies the associated data into its own iterator.

Linearization and non-lock-freedom. With this being a Synchronous Operation, it is trivially not lock-free. Its linearization also follows the semantics of Synchronous Operations: the iterator linearizes when it manages to acquire all the required locks.

Chapter 6

Measurements

> How often was he appalled by
> some shrub covered with snow,
> which, like a sheeted spectre,
> beset his very path!
>
> Washington Irving,
> The Legend of Sleepy Hollow

In this Chapter we run several experiments in order to back our previous claims and to shed some light on the consequences of the design and implementation choices made. Measuring the performance of the presented hash table has proven to be particularly difficult. The implementation's performance is measurably hindered by the constraints of the environment for which it was designed.

The results presented in this Chapter are entirely reproducible and available online.

6.1 Comparison with other reviewed hash tables

In this Section we compare our hash table implementation with CPython's and Maier's hash tables. These are indirect comparisons with our hash table, which has very different constraints. Evaluating the effectiveness of our design and implementation is unfeasible to do directly. To do so would necessitate the independent development of two other concurrent hash tables for CPython, with different designs and implementations. There currently exist no such hash tables; meaning, we would need to develop them for the sole purpose of compiling this Chapter. The resources required to do so are well beyond those allocated for this entire Thesis.

Given that a direct comparison is impractical, we resort to comparing our hash table with a sequential hash table (CPython's) and a non-object-oriented hash table (Maier's). These hash tables resp. don't need to consider naturally parallel accesses, or be generalized for any possible data type; ours has to entertain both features. The lack of these two constraints will be evident in the experiments that follow.

Most of the 11 experiments presented here are drawn from the literature.

Two experiments have been introduced by us, in order to show the performance of the iteration and batch lookup operations.

6.1.1 Experimental Methodology

Before showing the results, we describe the way in which the experiments have been executed and how we measured their performance. We run each experiment in isolation and execute it five times, without running any other experiment concurrently. We pick the median-wall-clock time experiment to draw our observations from. At each run, we measure both the wall-clock time, and we sample the running time of the C functions which are called when executing the experiment's Python code.

It would not be an effective choice to compute the average of the wall-clock times because we will also observe more detailed measures, which cannot be trivially averaged.

We employ the Linux kernel's `perf` tool to sample the execution time of individual C functions. We do not trace a function's entire execution, since that would significantly alter the overall running time. Instead, the experiment's process is interrupted at a frequency of 10kHz, the current program counter is recorded for each thread, and the process is resumed. The recorded program counters are later associated with C functions.

For the experiments related to Maier's hash table we don't either measure the wall-clock time, nor sample the underlying functions. This is because (1) we rely on the output of Maier's experiments to retrieve the wall-clock times, and (2) we assume that the share of time spent in the relevant hash table routines is close to 100% of the output time (which is not the case when the program being measured is executed by a software interpreter).

To run our experiments we employed the original nogil fork, since it was the only option available at the time of implementing `AtomicDict`. This modified experimental interpreter evidently comes with some significant performance limitations.

In virtually all experiments we can see that the running time is unexpectedly high, and by refining our measurements instrumentation with `perf` we can observe that the time actually spent executing `AtomicDict`'s routines is a small share, even if it is the system being tested. Notwithstanding the small amount of Python code pertaining to each experiment, it hides several implicitly executed C code. One should argue that the experiments should thus be written in C, but the C routines of `AtomicDict` are not easily exposed to other C extensions, at this time. Apart from this limitation, we also believe that the usage from Python code will be the predominant one, and therefore the limitations of this environment should be considered.

6.1.2 Results

Throughout the experiments we have observed a comparatively poor performance of `AtomicDict`, as measured in the number of operations completed per second; for instance, consider the experiments in Figure 6.1. The performance of Maier's hash table, named Growt, is usually at least twice as high as `dict` and `AtomicDict`, often one or two orders of magnitude higher. We consider this to be caused by keys and values being of a fixed integer type. CPython's `dict`'s

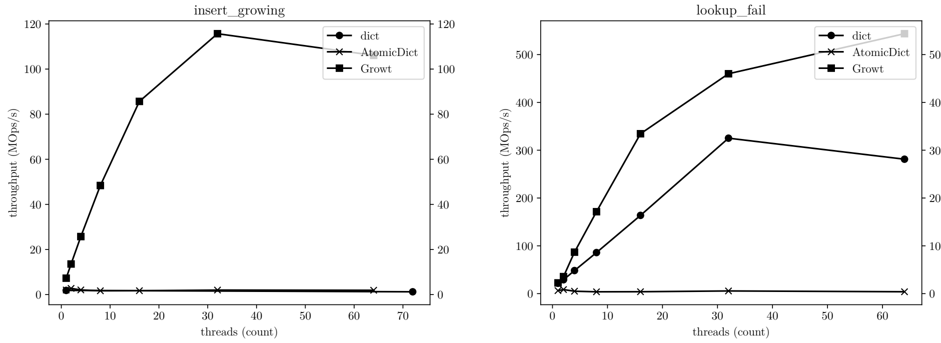

Figure 6.1: On the left, the insertion experiment, where each hash table had an initial size of 0. On the right, the failed lookup experiment, where searched keys were likely not in the table.

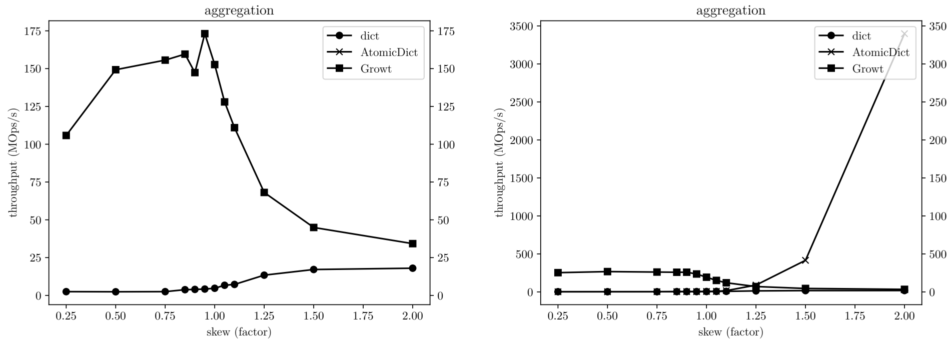

Figure 6.2: The aggregation experiment. On the left, the `AtomicDict.reduce` routine is not shown to better view the other two hash tables. The concurrent hash tables were used by 72 threads, while `dict` was used by 1 thread.

performance almost never scales with the number of threads, but is generally high. Compared to `dict` it is worth noting that while the performance may be worse, `AtomicDict` does guarantee the correctness of the output even in the face of concurrent mutations, while `dict`, crucially, does not. We find, though, that the specialized use-cases of `AtomicDict`, the `reduce`, `batch_lookup`, and `fast_iter` methods, show a very high throughput.

Aggregation, a very common use-case. As expected, the throughput of the sequential hash table `dict` almost never scales with the number of threads, across the experiments. The lookup experiments are notable outliers of this pattern, because of the optimistic lock avoidance described in Section 2.6.

Nevertheless, we find that `dict` exhibits some other less trivial patterns for performance gain, as shown in the aggregation experiment in Figure 6.2. For `dict`, the increasing skew of input data does not deteriorate performance, as is generally the case for concurrent hash tables, and instead improves it, due to the relevant data being more frequently found in the cache and the lack

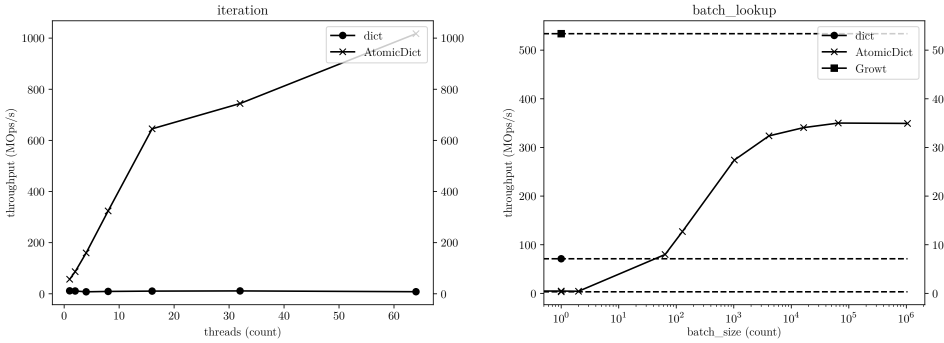

Figure 6.3: Two experiments showing some effective design choices. On the right, the dashed lines represent the tables throughput without batching.

of contention caused by atomic operations. For `AtomicDict`, we can see that its throughput is roughly on par with `dict` when the skew is small. When instead the skew is sufficiently high (≥ 1.0), the diminishing number of atomic operations required due to the accumulation of an intermediate result into a thread-local table, evidently produce very significant performance gains.

Due to limitations to our measurement instrumentation, it should be noted that the throughput of `AtomicDict` shown above should be 5–10% less. In fact, the instrumentation only associates with `AtomicDict` the code that pertains to the `cereggii` shared object. For the case of reduce, some code is executed that pertains to the CPython shared object. By deeply examining the results of our measurements, we can see that the additional code executed by the `reduce` routine outside `cereggii` is fairly small ($< 10\%$). Additional effort was not put into refining the instrumentation so as to automatically account for this additional code.

It should also be noted that while `reduce` can reach such high throughput, the overall wall-clock time is still affected by the Python code executed when calling the mutation function. In fact, most of the running time of the reduce experiment with skew $= 2.0$, is spent interpreting that routine. The cost of that routine is not associated with `AtomicDict` in the above Figure.

It is also possible to observe that the increase in skew from 0.25 to 0.95, increases the throughput for the Growt hash table as well. As the skew increases further, contention becomes a major scaling bottleneck. In fact, using `dict` with a single thread provides almost the same throughput, when the skew is 2.0. This is the reason why `reduce` was implemented for `AtomicDict` in the first place: rendering a concurrent hash table useful even with highly skewed data, a common pattern of real-world data.

Some effective design choices. There are two experiments which we could not find in the literature, but which are frequent use-cases of hash tables in general: iterating over the currently-stored set of keys, and looking up multiple keys instead of a single one. While Maier et al. mention the former use-case as “Forall” in their paper, they avoid producing an experiment. We additionally improve iterations by optionally providing partitioning between threads; refer

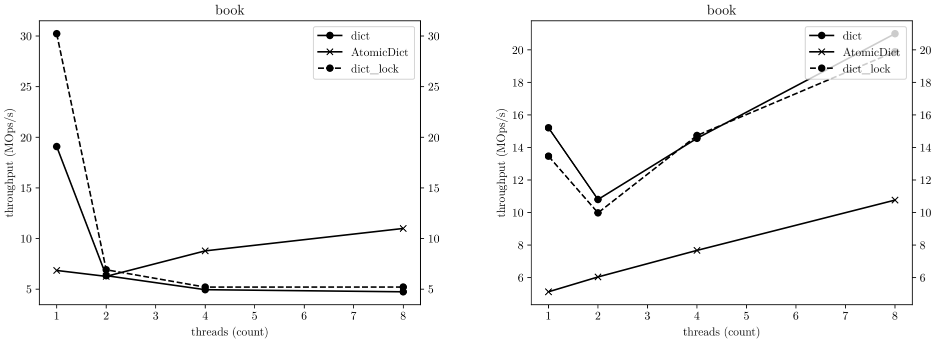

Figure 6.4: The `book` experiment. On the left as run with the nogil fork, on the right with the first release candidate of CPython 3.13.

to Section 5.9.1 for further details. The latter is the design we proposed in Section 5.7 for amortizing memory access latencies. In Figure 6.3 we can observe the effectiveness of our designs.

On the left of the Figure, we can see that the iteration can scale almost linearly with the number of threads, and that it also exhibits a much higher throughput than `dict`'s iteration, even with a single thread-partition.

On the right, we can observe that lookups in a batch can greatly improve their overall throughput with memory prefetching. We can also see that there exists a point of diminishing returns, in this case at a batch size of 1024. This is due to cache poisoning, where the system's cache is too small and prefetched elements are evicted from the cache before they are actually read. The point itself varies widely between different systems, since it depends on the capacity of the cache.

CPython 3.13. We have attempted to use a release candidate of CPython 3.13. This required some modifications to our code, which may have altered the results. The `book` experiment was run with the updated interpreter and code, and it showed a decrease in performance. This was the only experiment we have been able to run, since most others rely on the `numpy` package which does not support this new CPython version, as of the time of writing. Given that we modified the code in a way that can impact performance, we should consider the results shown in Figure 6.4 as inconclusive. However, we can notice a decrease in the single-threaded performance of `dict`, which we can confirm.

6.1.3 Reference Counting and Other Overheads

There are several bottlenecks to multithreaded-performance scaling in free-threading Python. The major one is concurrent reference counting, in spite of the many improvements detailed in Chapter 2. In fact, the objects of interest for this Thesis are actively shared between threads, therefore our objects are always forced into the reference counting slow-paths.

For instance, in the `lookup_success` experiment, we can see that the time spent in our lookup routine is only approx. 25% of the wall-clock time, while the rest is mostly spent in reference counting (≈ 28%), in the interpreter loop (≈ 13%), and in other miscellaneous CPython routines (≈ 26%). The reference counting for our internal data structures accounts for approx. 6% of the wall-clock time.

There are several other measurable bottlenecks. The main one is clearly the overhead of interpretation: a Python program is simply input data for the CPython interpreter, and it needs to be processed accordingly.

Where the performance of CPython's built-in hash table is measured, we also observe some considerable time spent waiting to acquire a mutex; approx. 21% of the wall-clock time in the `insert` experiment. That hash table is not lock-free, thus it needs to spend some time waiting for lock acquisition, temporarily idling the affected CPU cores. Our hash table is not subject to this hindrance.

6.2 Usage of CPython's built-in hash table

In this Section we back with data our previous claims that:

1. the distribution of the standard operations of hash tables is the one accepted in the literature:
   (a) 90% lookups;
   (b) 9% insertions;
   (c) 1% deletions; and
2. most lookups fail.

6.2.1 Methodology

Before showing the data, let us explain our methodology. In an online repository [22], we have slightly modified the CPython interpreter to collect metrics on its built-in hash table. Each routine and sub-routine has been modified to record the number of calls, and its running time. Additionally, some specific routines have also been altered to record their results; for instance, in the lookup routine, we recorded the number of times it returned a success or a failure. This modified interpreter was then employed to run the `pyperformance` benchmark suite. This is a suite of programs generally regarded as the standard for describing the performance of the CPython interpreter, as it evolves over time. It is, thus, the way in which the CPython core developers expect the interpreter to be used. We extrapolated the usages of the built-in hash table therein.

For a lack of a better alternative, we will consider these metrics to be the same ones as for our concurrent hash table. In fact, it is possible to argue that a concurrent hash table would incur a different usage. While that may be so, such different usages can only be observed with a functioning concurrent hash table being widely adopted. That of course, is the objective of this Thesis, but sadly we cannot foresee those programs. We therefore can only resort to consider the sequential usages at the time of writing.

Note that the modified CPython interpreter was the 3.11 release, because at the time of running those experiments the 3.12 version hadn't yet been released. The `pyperformance` suite though has not significantly changed since, so we can still make effective use of the collected metrics.

6.2.2 Results

CPython's dict is implemented over a total of 165 C functions, all of which are accounted for in [22]. By inspecting the source code we can find that most functions are in fact specialized calls over generic sub-routines. We are interested in those that implement the lookup, insert, and delete operations, which are respectively `_Py_dict_lookup`, `insertdict`, and `delitem_common`. In fact, apart from an inspection of the relevant source code, we can find that the number of calls for these routines is greater than all the other semantically-similar routines.

By inspecting data pertaining to these routines we can find that they have been called respectively 16 730 124 560, 804 910 551, and 208 639 089 times, totalling 17 743 674 200 calls. Their respective shares are thus approximately 94.3%, 4.5%, and 1.2%, showing an even higher degree of skew towards lookups than expected in claim (1).

We can see in [22] that the generic lookup routine returned 3 247 042 454 successes and 13 483 082 106 failures out of the 16 730 124 560 calls. Therefore, we can say that overall the expected ratios of lookup successes and failures are approximately 19.4%–80.6%, backing claim (2).

Chapter 7

Conclusions

We have shown that it is possible to implement an efficient concurrent hash table for CPython. This is a very important building block for its evolving ecosystem, which is increasingly interested in parallelization. To the authors' knowledge, this is the first naturally parallel hash table implemented for CPython. In the previous Chapter we have experimentally shown the limitations of a hash table designed to be concurrent and compliant to the CPython object protocol. We have also shown that our designs to circumvent those limitations were effective.

These results were presented at the 2024 Python Language Summit. The handout of the presentation is included in Appendix C. An article about the presentation and the discussion that followed is available online [15]. The article concludes with this note:

> Overall, the group seemed interested in Daniele's work on atomics
> but didn't seem willing to commit to exact answers for Python yet.
> It's clear that more experimentation will be needed in this area.

What follows in this Chapter are interesting topics for further research and some closing remarks.

7.1 Concurrent Garbage Collection

In Chapter 2 we have seen numerous clever optimizations for making concurrent reference counting very efficient. It is likely that for certain applications those optimizations are going to be fruitful. In Section 6.1.3 we have shown that for some trivial applications those optimizations are not enough, and performance is clearly subject to concurrent reference counting.

It is a belief of the author that adding a concurrent GC into CPython can help the performance of some concurrent Python programs. It could be beneficial that some objects live in a different space, one that is not subject to reference counting at all. In that "concurrent" object-space a completely different mechanism could be employed for memory reclamation. Objects that live in that space could be shared more effectively between threads, without hindering the performance of the application. The GC itself could also be designed not to be a stop-the-world GC, as is currently the case for CPython's. This would permit fully lock-free data structure implementations. The two different collectors would co-exist in the same CPython process.

7.2 Thread-Local Handles

A significant portion of the measured time involving the experiments of Chapter 6 pertains to concurrent reference counting. Such is because a lot of increments and decrements to the reference count are made when the CPython interpreter executes instructions that involve an instance of `AtomicDict`. E.g. every time a lookup is performed, the Python interpreter increments and later decrements the reference count of the involved `AtomicDict` object.

Considering the necessity to use atomic operations to perform the counting (two in the above example), the amount of time spent in those routines is great (≈15%–25% in the experiments of Chapter 6). Instead of accessing the `AtomicDict` instance directly, though, it is possible to create a handle for it, like the one described in Section B.2.1, an `AtomicDictHandle`. With the handle being a thread-local object, not intended to be shared with other threads, it can enjoy the fast-path for reference counting, thus reducing the time spent doing reference counting to a negligible amount. A thread-local handle *represents* an `AtomicDict` object, and proxies calls to it. This addition would not require a concurrent GC, and would be beneficial regardless.

We have successfully applied this design in the case of `AtomicInt`, a very simple concurrent integer implementation, detailed in Appendix B. The strategy of using a thread-local proxy object was proved successful in almost eliminating the slow-paths taken for concurrent reference counting. During an exchange with Gross, he deemed this approach to be effective both for the nogil interpreter employed for the experiments, and for free-threading Python 3.13.

Since this would entail a reduction over the total amount of work that the CPU needs to perform, it is expected that an improvement of ≈20% over the measurements of Chapter 6 would be observed.

7.3 Compatibility with Sub-Interpreters

A CPython process can have multiple sub-interpreters. Historically, since the GIL was deemed an intrinsic presence in the CPython interpreter, there have been various attempts at circumventing this limitation. Simplifying a lot of the relevant history, sub-interpreters were also designed with this problem in mind, each being endowed with a per-interpreter GIL. (Sub-interpreters have been part of CPython for over 25 years, but most of the current interest around them sparked from their recent enhancements around having a per-interpreter GIL.) When a CPython process starts several threads and associates with each of them a distinct sub-interpreter, they will all have their own GIL, enabling them to run in parallel notwithstanding the continued presence of multiple global interpreter locks. Sub-interpreters can be considered to be a fairly new technology, described in PEPs 554, 684 and 734 [24, 25, 26]. To the author, cross-compatibility of a data structure between free-threading *and* sub-interpreters constitutes an interesting research area.

Sub-interpreter parallelization comes at the expense of not being able to easily share objects between interpreters, as also noted in [9, §Per-Interpreter GIL]. Inheriting some of their design from Communicating Sequential Processes (CSP), the general idea is that the threads in sub-interpreters exchange information only through message-passing, rather than by directly sharing memory.

In fact, sharing *objects* between interpreters is generally not allowed. Given that the internal data structures of `AtomicDict` are CPython objects, it cannot be used with sub-interpreters. Nonetheless, there are mechanisms to share *immutable* data between sub-interpreters.

In [26, §Queue Objects] is described a sub-interpreters-compatible implementation of a queue, that can be used to pass *shareable* objects from one interpreter to another. Integers, strings, floats, and a queue instance in itself, are all shareable, among others. Conceptually, an object qualifies as shareable as long as it can be entirely represented by an immutable array of bytes.

Crucially, a sub-interpreters queue is not a CPython object, differently from `AtomicDict`. It is instead allocated and freed without interacting with the GC. Its internal state cannot be stored into objects, because an interpreter accessing the queue would crash when trying to access an object not pertaining to its own object-space. This also entails that there must exist an object *representation* of a queue, which has to be created from the interpreter that desires to access the queue, and no other interpreter.

A sub-interpreters queue can be entirely represented by its ID number. When a queue's memory is allocated outside CPython's object-space, the queue is inserted into a shared, global list of queues. The ID of a queue is its position in the list of queues. For a Python thread to access a queue, it is sufficient to know its ID, to then retrieve the queue from the global list. The object created to interact with a queue from Python code, i.e. the *representation* of the queue, only knows the ID of the queue: it just holds an integer, and thus it can be entirely represented by an array of bytes. Therefore, a queue can be considered *shareable*, and it can also be enqueued into another queue, or even into itself.

This *representational* behavior should hold some resemblance with the `AtomicDictHandle` described in the previous Section, whose primary purpose has nothing to do with sub-interpreters, but that can nonetheless be used to hold an immutable, and thus potentially *shareable*, representation of a hash table. In fact, it can be entirely represented by an immutable pointer to the `AtomicDict` instance. In order for the handle to work properly with sub-interpreters, an additional requirement is that the `AtomicDict`'s internal data structures should not be Python objects, putting further interest into implementing a different memory management scheme. This would also be possible by extending the concurrent GC sketched out in Section 7.1.

Supposing such is performed, more work needs to be carried out in order to ensure that the keys and values in `AtomicDict` are *shareable*, by also making use of cross-interpreter object representations, in possibly a standardized way. During an exchange with Eric Snow, author of the PEPs on sub-interpreters, the necessity for standardized, customizable ways to create cross-interpreter object representations emerged as one of Snow's planned works. Snow intends to write a PEP proposing to create a new `__xi__` standard method for Python classes to return such immutable representations.

With these enhancements, albeit quite a substantial amount of work to implement them, it becomes possible to make use of the same `AtomicDict` implementation in both concurrency models: common multithreading with free-threading, and isolated object-spaces multithreading with sub-interpreters.

7.4 Closing Remarks

Implementing a concurrent hash table was no trivial task. Implementing one that adheres to CPython's protocols while maintaining good performance, has proven to be a terrifically difficult task. I am glad I was finally able to reach this goal.

While I consider this to be quite an achievement, I don't consider this to be the final word on concurrent hash tables for the Python ecosystem. I would in fact not want it to be that. I rather hope this proves to be a solid starting point for concurrent data structures in this rich and evolving ecosystem.

Bibliography

[1] D. Anderson, G. E. Blelloch, and Y. Wei. “Concurrent deferred reference counting with constant-time overhead”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 526–541. ISBN: 9781450383912. DOI: `10.1145/3453483.3454060`. URL: `https://doi.org/10.1145/3453483.3454060`.

[2] D. Beazley. *An Inside Look at the GIL Removal Patch of Lore.* 2011. URL: `https://dabeaz.blogspot.com/2011/08/inside-look-at-gil-removal-patch-of.html` (visited on April 23, 2024).

[3] P. Celis, P.-Å. Larson, and J. I. Munro. “Robin hood hashing”. In: *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)* (1985), pp. 281–288. URL: `https://api.semanticscholar.org/CorpusID:17447073`.

[4] J. Choi, T. Shull, and J. Torrellas. “Biased reference counting: minimizing atomic operations in garbage collection”. In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques.* PACT ’18. Limassol, Cyprus: Association for Computing Machinery, 2018. ISBN: 9781450359863. DOI: `10.1145/3243176.3243195`. URL: `https://doi.org/10.1145/3243176.3243195`.

[5] CPython contributors. *dictnotes.txt.* URL: `https://github.com/python/cpython/blob/23950beff84c39d50f48011e930f4c6ebf32fc73/Objects/dictnotes.txt` (visited on April 23, 2024).

[6] CPython contributors. *dictobject.c comment on hashing function.* URL: `https://github.com/python/cpython/blob/23950beff84c39d50f48011e930f4c6ebf32fc73/Objects/dictobject.c#L262` (visited on April 23, 2024).

[7] H. Gao, J. F. Groote, and W. H. Hesselink. “Lock-free dynamic hash tables with open addressing”. In: *Distributed Computing* 18 (2003), pp. 21–42. URL: `https://api.semanticscholar.org/CorpusID:17755299`.

[8] S. Gross. *Add delayed reclamation mechanism for free-threaded build (QSBR).* February 6, 2024. URL: `https://github.com/python/cpython/issues/115103` (visited on April 23, 2024).

[9] S. Gross. *PEP 703 – Making the Global Interpreter Lock Optional in CPython.* May 4, 2023. URL: `https://peps.python.org/pep-0703/` (visited on April 23, 2024).

[10] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* 1st edition. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123973375.

[11] R. Hettinger. *More compact dictionaries with faster iteration.* December 9, 2021. URL: `https://mail.python.org/archives/list/python-dev@python.org/thread/V342SDL2AUMVCTAODAALJWW2PLDYS5GX/` (visited on April 23, 2024).

[12] Intel Corp. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 2A, 2B, 2C, and 2D: Instruction Set Reference, A-Z.* March 2024. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html` (visited on April 23, 2024).

[13] E. Kahssay. "A fast concurrent and resizable Robin Hood hash table". Graduate Thesis. Massachusetts Institute of Technology. Department of Electrical Engineering and Computer Science, 2021.

[14] D. E. Knuth. *The art of computer programming, volume 3: sorting and searching.* 2nd edition. USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0201896850.

[15] S. M. Larson. *The Python Language Summit 2024: Free-threading ecosystems.* June 14, 2024. URL: `https://pyfound.blogspot.com/2024/06/python-language-summit-2024-free-threading-ecosystems.html` (visited on April 23, 2024).

[16] D. Leijen, B. Zorn, and L. de Moura. "Mimalloc: Free List Sharding in Action". In: *Programming Languages and Systems.* Ed. by A. W. Lin. Cham: Springer International Publishing, 2019, pp. 244–265. ISBN: 978-3-030-34175-6. URL: `https://api.semanticscholar.org/CorpusID:198363081`.

[17] T. Maier, P. Sanders, and R. Dementiev. "Concurrent Hash Tables: Fast and General(?)!" In: *ACM Trans. Parallel Comput.* 5.4 (February 2019). ISSN: 2329-4949. DOI: `10.1145/3309206`. URL: `https://doi.org/10.1145/3309206`.

[18] M. M. Michael. "High performance dynamic lock-free hash tables and list-based sets". In: *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures.* SPAA '02. Winnipeg, Manitoba, Canada: Association for Computing Machinery, 2002, pp. 73–82. ISBN: 1581135297. DOI: `10.1145/564870.564881`. URL: `https://doi.org/10.1145/564870.564881`.

[19] M. M. Michael. "Safe memory reclamation for dynamic lock-free objects using atomic reads and writes". In: *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing.* PODC '02. Monterey, California: Association for Computing Machinery, 2002, pp. 21–30. ISBN: 1581134851. DOI: `10.1145/571825.571829`. URL: `https://doi.org/10.1145/571825.571829`.

[20] V. Narayanan et al. “DRAMHiT: A Hash Table Architected for the Speed of DRAM”. In: *Proceedings of the Eighteenth European Conference on Computer Systems*. EuroSys ’23. Rome, Italy: Association for Computing Machinery, 2023, pp. 817–834. ISBN: 9781450394871. DOI: `10.1145/3552326.3587457`. URL: `https://doi.org/10.1145/3552326.3587457`.

[21] D. Parmeggiani. *cereggii. Thread synchronization utilities for Python.* September 19, 2023. URL: `https://github.com/dpdani/cereggii`.

[22] D. Parmeggiani. *Collecting metrics on built-in dictionary usage.* August 29, 2023. URL: `https://github.com/dpdani/cpython/tree/3.11` (visited on April 23, 2024).

[23] M. Shannon. *PEP 412 – Key-Sharing Dictionary.* February 8, 2012. URL: `https://peps.python.org/pep-0412/` (visited on April 23, 2024).

[24] E. Snow. *PEP 554 – Multiple Interpreters in the Stdlib.* September 7, 2017. URL: `https://peps.python.org/pep-0554/` (visited on April 23, 2024).

[25] E. Snow. *PEP 684 – A Per-Interpreter GIL.* March 8, 2022. URL: `https://peps.python.org/pep-0684/` (visited on April 23, 2024).

[26] E. Snow. *PEP 734 – Multiple Interpreters in the Stdlib.* November 6, 2023. URL: `https://peps.python.org/pep-0734/` (visited on April 23, 2024).

[27] A. Waygood. *The Python Language Summit 2023: Making the Global Interpreter Lock Optional.* May 29, 2023. URL: `https://pyfound.blogspot.com/2023/05/the-python-language-summit-2023-making.html` (visited on April 23, 2024).

[28] Wikipedia contributors. *Peniocereus greggii.* URL: `https://en.wikipedia.org/wiki/Peniocereus_greggii` (visited on April 23, 2024).

Appendix A

Atomic Reference

We present in this Appendix an atomically updatable reference to a Python object. That is, a reference that can be safely updated in spite of concurrent changes and deallocations.

This was initially implemented in order to correctly handle the swapping of an `AtomicDict`'s internal metadata object, in a way that made use of CPython's new QSBR scheme. (See Section 5.4.3, when the leader thread swaps the metadata object.) Its usefulness extends beyond the internal structures of `AtomicDict`: since the problem solved is very simple, its applications can be numerous.

The implementation described in this Appendix is publicly available at [21, src/cereggii/atomic_ref.c].

A.1 Design

The design of `AtomicRef` closely resembles the interface of Java's `AtomicReference`: an `AtomicRef` object stores a single pointer to another CPython object. The choice of following the Java interface comes down to two considerations: programmers who are already accustomed to Java's interface need not learn a new one, and this being such a relatively simple problem, it would seem hard to design something completely different from the rest of the world. (Regarding the slight name difference: the shortened *ref* is more commonly used in Python, compared to *reference*, in several parts of the language.)

The "hard" part of this design is to correctly use free-threading CPython's QSBR scheme. (See Section 2.3 for an introduction to QSBR.) Namely, in lines 23–25 of Listing A.1 the read on line 24 is retried until the reference count increment succeeds on line 25. This is where serious bugs may occur in a faulty implementation. In fact, if incorrectly using QSBR, instead of failing a call to increment the reference count, it is possible that the memory pertaining to the fore-referenced object is freed, as that would be the effect of a concurrent `Decref` which sees the reference count reach 0. The exhibited behavior would thus be a segmentation fault, instead of an increase in the running time (due to contention). A significantly worse possibility.

Listing A.1: Pseudocode of `AtomicRef`'s core routines.

```
1       typedef struct {
```

```
2      PyObject_HEAD

       PyObject *reference;
     } AtomicRef;

7    int AtomicRef_CompareAndSet(self, expected, new)
     {
       Incref(new);
       int success = CompareAndSet(&self->reference, expected, new);
       if (success) {
12       Decref(expected);
         return 1;
       } else {
         Decref(new);
         return 0;
17     }
     }

     PyObject *AtomicRef_Get(self)
     {
22     PyObject *reference;
       do {
         reference = self->reference;
       } while (!TryIncref(reference));
       return reference;
27   }
```

A.1.1 Single-Field Struct

An alternative design approach could have been not making `AtomicRef` a CPython object. Instead, it would be turned into a single-field C struct, essentially removing line 2 of Listing A.1. This design change was inspired by the new `PyMutex` of the original nogil fork, and of Python 3.13t.

This choice would slightly help the performance of `AtomicDict` as well, given that there are currently two indirections required to read the metadata object: one indirection to read `dict->metadata`, which is an `AtomicRef *`, and one indirection to read `dict->metadata->reference`, which is the actual `AtomicDict` metadata object. Ipso facto, as an improvement for `AtomicDict` it would be very marginal. Instead, if a user of this library would necessitate to manage several instances of `AtomicRef`, say, stored into a C array, access to each referenced object would be hindered by the double indirection. Furthermore, `AtomicRef` not being a Python object would entail that it would not be possible to expose this presented functionality into Python code.

It is possible to both gain further performance, and retain usability in Python code: using a single-field C struct for `AtomicRef`, as previously described, intended to be accessed from C code, and also implementing a differently named Python type, say `AtomicRefObj`. The new Python type would itself be the container for a single-field C struct `AtomicRef`, thus essentially retaining equal functionality as the one presented before.

A.2 Issues with free-threading CPython 3.13

During an exchange with Gross, he pointed out that the method employed for correct concurrent handling of reference counts differed in Python 3.13t, compared to his original `nogil` fork. Namely, that the `TryIncref` routine would not behave in the same way. Instead, expecting the object to be in the *weakrefs* state, described in Section 2.1. In order to correctly implement `AtomicRef` for Python 3.13, it will therefore become necessary to enforce that the objects referenced by an `AtomicRef` to be at least in the *weakrefs* state. The state transition needs to be carried out before a new reference is stored inside `AtomicRef`, i.e. before another thread may decrement the reference count, and then deallocate the object without using the QSBR slow path for deallocation. Upon applying this modification, it is expected that the current implementation will restore correctness, without compromising the overall design.

Appendix B

Atomic Integer

This Appendix discusses the rationale and design of an atomic integer for CPython. It can be useful both in the context of the free-threading changes, and in the default build of CPython. The implementation discussed here is available in [21, src/cereggii/atomic_int].

B.1 The necessity before and after the free-threading changes

With the presence of the GIL, many concurrency problems that existed in Python code may never be observed. An atomically updatable integer is a very simple abstraction to ensure that certain concurrency problems are handled correctly.

> [...] There's currently no atomic int in Python and people go through all sorts of weird contortions to atomically count things (which often leads to broken code even with the GIL).[1]

Regardless of the CPython build, a class of the likes of `AtomicInt` is still useful in Python's ecosystem. It can additionally gain usefulness in the free-threading build because of the performance improvements that can be gained with specific design choices described later in Section B.2.1.

B.2 Design

The design of `AtomicInt` revolves around exposing common C atomic operations on 64-bit signed integers to Python code. Therefore, most of the implemented routines essentially create this mapping between C operations and Python APIs.

The APIs of `AtomicInt` closely resemble some of Java's `AtomicInteger` methods. The `AtomicInt` has some easily expected methods, like `get()` and `set()`, and also some borrowed from Java, and very similar to other C-level atomic operations, such as `getAndSet()` and `incrementAndGet()`.

[1] This quote is from a comment by Gross in a public discussion, and is accessible at `https://discuss.python.org/t/towards-a-python-util-concurrent/41325/8`, last accessed October 4, 2024.

Especially useful are the so-called “in-place” Python methods, such as my_int += 1, my_int *= 2, etc. These methods are very effective for atomic operations because, instead of requiring a separation of reads and writes in Python code, they describe the entire mutation that the program wants to see applied. The operation can thus be efficiently retried within the same call to the relevant C routine, which performs the retry when the integer's value is contended, following the semantics of CAS atomic operations.

The program can thus avoid worrying about the correct usage of atomic CAS primitives, when using in-place methods. Notice the similarity with AtomicDict's proposed reduce method, which permits to do essentially the same thing: fully describe a mutation, to be applied correctly by the internals of the routine. Instead, a usage similar to the following would yield inconsistent results, assuming that the thread function is effectively the code run by multiple threads:

```
i = AtomicInt()

def thread():
  for x in range(100):
    i = i + 1
```

What happens in this case is that the value that i is holding, is first retrieved, the sum is performed, and then the result is stored in i, similarly to calling i.set(i.get() + 1). These three actions comprising a mutation that employs two distinct methods of AtomicInt, cannot be correctly executed without the protection of an external lock. That is, the two methods being linearizable, does not entail that two successive calls to them are linearizable; in fact, the opposite is true.

B.2.1 Handles

It can easily be observed by profiling [21, examples/atomic_int/counter.py] that the running time is vastly dominated (≈99%) by concurrent reference counting and other CPython thread-safety features, with the atomic increment being such a cheap operation in comparison. Such is just a vast amount of overhead.

The reference count of AtomicInt is required to be modified only when the threads start using the object and then eventually releasing it, upon finishing their work. The many other reference counting operations are just overhead.

Therefore, a simple but very effective change was implemented. The AtomicInt object does not necessitate being referenced directly by the thread that wants to mutate it, instead it is also possible to access it indirectly through a handle. An AtomicIntHandle is an object that internally keeps a pointer to an AtomicInt object. Externally, it provides exactly the same APIs as AtomicInt, so that using or not the handle does not require substantial changes to a program's code. If the handle is created by the only thread that uses it, as it is intended, then it enjoys the fast-path for concurrent reference counting implemented in free-threading Python. (See Section 2.1.)

The same reference counting operations, thus always performed by the proxy object's owner thread, would be substantially cheaper. So much so, that the observed running time of [21, examples/atomic_int/counter.py] drops by a staggering ≈95%.

During an exchange with Sam Gross, the main author of the free-threading changes, he had indeed confirmed that the added burden of concurrent reference counting here exhibited is likely to remain in the free-threading build of CPython, for the foreseeable future. Furthermore, he was convinced that using this design of handle indirection can be effective in contrasting these performance bottlenecks.

B.3 Sharded Counter

There can be a useful specialization of `AtomicInt`. To further reduce contention, it is possible to restrict the usage of `AtomicInt` to that of a counter and then optimize it.

In fact, if the integer itself is used as a counter, a very common application, then it is possible for the T threads accessing it to have T distinct counters upon which to count in isolation with each other. Since additions are commutative, the T counters can be summed together lazily only when the program wants to read the value of the counter, say by a `get()` operation. The result might then be cached to avoid the repetition of work when the value hasn't changed.

In principle, this can be achieved in a manner similar to `AtomicDict`'s scheme for consistent size retrieval, described in Section 5.6. Each counter would be stored in a thread-local structure which also contains a lock. Such lock would behave in the same manner as `AtomicDict`'s accessors locks.

The thread-local counters would need to be stored in distinct regions of memory, so as to ensure that they are not stored into the same cache line, exhibiting false sharing. That is, the CPU cache coherency protocol would be obliged to keep the single cache line consistent among processors, resulting in no improvement, quite the opposite, over the current implementation of `AtomicInt`.

Appendix C

Language Summit

What follows in this Appendix is the handout of a presentation that the author of this Thesis gave at the 2024 Python Language Summit, on April 18th, 2024, in Pittsburgh, PA.

Free-Threading Ecosystems

Python without the GIL is coming out this fall. We can expect more Python programmers to choose multi-threading concurrency models in the future. We can also expect thread-safety guarantees to gather more interest, perhaps in the form of atomic data structures.

Given the acceptance notes to PEP 703, one should argue for caution in discussing the prospects of multi-threading ecosystems and of their growth following the release of Python 3.13.[1] With a hopeful spirit, I will disregard this caution here.

I had opened a GitHub issue[2] arguing to make a certain new C-level API called TRY_INCREF public. Sam responded as thoroughly and thoughtfully as he usually does, arguing that function shouldn't be public. And I agree with him.

The problem with making TRY_INCREF public is that its semantics are far too related to the current implementation of the free-threading changes to make it suitable for future-proof support. I now argue further that it should never be made public. Will core devs ever be able to confidently predict TRY_INCREF (and possibly other functions like it) will no more be subject to substantial changes or optimizations?

A very concrete problem still stands: how does an extension increment an object's reference count, without knowing whether it had reached zero, in the time between the pointer was read and the count updated? I would say that the level of abstraction for these functions is too low. Nevertheless, at a higher level it is possible to provide further guarantees, without constraining what's under the hood.

[1] https://discuss.python.org/t/a-steering-council-notice-about-pep-703-making-the-global-interpreter-lock-optional-in-cpython/30474

[2] https://github.com/python/cpython/issues/113920

Following with the same example, it is quite trivial to implement an atomically updatable reference to a PyObject with the new QSBR scheme that's already landed in main. But what if Python 3.14 decides to change it radically? And what if 3.15 decides to do away with it entirely? Well, making guarantees about any low-level API should seem an over-commitment. On the other hand, an API for atomically updating a reference to a PyObject seems a high-level use-case worth guaranteeing, regardless of any implementation of reference counting.

So, we could then turn to the pursuit of finding a set of use-cases we wish to support directly over the years. This being precisely what I would very much like to contribute towards. Of course, it can be the subject of much debate.

One prospect could be going through whatever is currently in the standard library, looking for spots where further atomicity guarantees could be afforded. For instance, considering collections.Counter, we can find that it doesn't guarantee no increment will ever be lost, unless a lock is manually wrapped around it. So one might improve the implementation of Counter, or maybe create a new AtomicCounter class for these new use-cases.

Another view could be looking at what other languages are doing. For instance, Java has a concurrent package, which is a large collection of atomic data structures; AtomicReference is the example I was talking about earlier. Should we offer the same or similar classes that are inside that package? Here I think that we should avoid copying too closely. Some behaviors are expected of Python data types that would be very different from Java's. For instance, AtomicInteger only stores integer numbers, the usual fashion in strongly-typed Java; but Python integers are expected to turn into floats when appropriate. So maybe an AtomicNumber rather than an AtomicInteger might be better suited, perhaps at some added cost in speed.

In general, regardless of whichever color of concurrent data structures one might prefer, the running theme will be the crucial step of reading and writing to the same memory location within the invocation of one method. That is, consider for instance Java's AtomicInteger again. The way it can be atomic to begin with, is by providing a compare-and-set method, which receives the expected value and the new desired value to update it with, if it really is the case that the currently stored value is the expected one. This compare-and-set behavior is the core of most concurrent data structures because it establishes a happens-before ordering between concurrent writes to the same memory location.

Those to whom "happens-before" rings a bell, may think I'm going to start talking about memory models, but I will refrain from doing so. Mostly because there is no optimization in Python (that I'm aware of) that actually tries to re-order memory accesses, which is the reason why memory models may be useful to begin with. And also because memory models are among the most confusing pieces of writing I've ever come across. So, if this topic that I'm presenting today is intended to lead to easily understandable abstractions of concurrent operations, then, in my view, it should avoid eliciting the necessity for a Python memory model. If the new JIT will ever decide to re-order loads and stores, then we may need to start thinking about memory models (and even then, possibly try to avoid them).

Instead, I'd rather favor the designing of interfaces that alleviate the complexity of the happens-before ordering I was mentioning. That is, suppose you want to create an atomic dictionary (as I'm currently trying to do), and you wanted to have simple APIs. Well, out of uninterpreted keys and values it is hard to find simpler interfaces than the bare compare-and-set. But suppose we have more information about the keys and the values. Suppose for instance that the semantics of this dictionary are those of collections.Counter. Then it becomes very easy to see how to provide a simpler interface: an imaginary AtomicCounter, could have an increment method which correctly calls the compare-and-set method of its underlying dictionary. The programmer using this AtomicCounter need not worry about any happens-before relationships, or any compare-and-set atomic semantics when calling the increment method. This seems to me a better direction to take.

There's also the perhaps secondary problem of performance. Secondary in the sense that I think thread-safety is usually considered more important to the average programmer.

The free-threading changes may actually decrease the performance of some multithreaded workloads. Notably, of those that frequently modify some object that is shared among threads. Now, I am not up to speed with the most recent optimizations already in main or coming to it, so my understanding may be incomplete. But according to observations based on the original fork, these write-heavy flows will be heavily impacted by the free-threading changes. Furthermore, when threads attempt to mutate one object, it will be as if the GIL was still there: they would all still be contending one lock. New concurrent data structures may alleviate this performance issue.

Also, perhaps, having atomic data structures in the standard library could give Python devs a freer hand in optimizing the single-threaded use-cases for the standard ones.

As a writer of a C extension wishing to implement concurrent lock-free data structures for Python, what I would really like to learn after this session is: does CPython wish to eventually incorporate these features I've been mentioning so far? If it doesn't, new low-level APIs like TRY_INCREF will be necessary to support external efforts towards new free-threading ecosystems.

In summary, to have a free-threaded interpreter without atomic data structures sounds like speaking Italian without saying mamma mia.

So, here are some questions to get the discussion started:

1. what primitives (high or low-level) should Python offer to programmers who target the free-threading builds, that it doesn't already offer?

2. would they substantially differ from Java's concurrent package, or to the ones found in our standard library?

3. to what extent should these be available to extension authors?

4. can we escape the need to specify a memory model? and

5. is it too early to make any call?