A Reimplementation of Pythons dict using Simple Tabulation Hashing and Linear Probing

Russell Cohen Thomas Georgiou Pedram Razavi

May 18, 2012

Abstract

Here we implement simple tabulation hashing with linear probing in Python dictionary based on the recent theoretical result of Patrascu et al. Although we achieved small performance wins on the order of 4%, we were unable to achieve significant performance wins. The time and memory required to compute the ST hash are too large to justify the speedup garnered by switching to linear probing.

1 Introduction

In this project we aim to analyze and improve the performance of Pythons dictionary or dict. Python is an extremely popular general purpose computing language for several reasons: 1. Clean, elegant near-pseudo code syntax leads to clear code 2. Widely distributed and available 3. Large quantity of external libraries for specific features

Recently, we the advent of packages like numPy, sciPy and pyPy python has began to also be used (with questionable merit) as a performance computing language due to its ease of use.

However, because Python is an interpreted language it is difficult to operate on a similar performance level to compiled languages due to the sheer overhead of the interpreter.

In order to attempt to improve the performance of the python language as a whole we plan to attempt to improve the performance of python dictionaries.

1.1 Motivation

Python heavily utilizes dictionary data structure for several of its internal language features. This extensive internal use of dictionaries means that a running Python program has many dictionaries active simultaneously, even in cases where the the users program code does not explicitly use a dictionary. Take for example, the following code which we ran in the interpreter:

```
>> i = 2
>> b = i * 2
    >> b
    10
```

One of the features that we added to the python source was a conditional flag that allowed us view dictionary statistics (collisions, probes, accesses, etc.) for a given invocation of the Python binary. With that modification we were able to see that this code used 92 lookups in string dictionaries and 24 lookups in general purpose dictionaries for a total of 116 dictionary hits. 116 hits in a 3 line piece of code which never uses dictionaries explicitly is quite significant. We explain this extensive dictionary usage below.

For instance, a dictionary can be instantiated to pass keyword arguments to a function. This means a dictionary could potentially be created and destroyed on every function call. Some other examples of internal usage of dictionaries in Python are as follows: 1. Class method lookup 2. Instance attribute lookup and global variables lookup 3. Python built-ins module 4. Membership testing

Later in this paper, we compare the performance of our new dictionary implementation for some of these special cases. The aforementioned applications of the dictionaries further highlights the need for the fast instantiation and deletion of dictionaries so that less memory is utilized once running a program. Aside from the issue of memory usage, to further enhance the runtime of a program we need fast key/value storage and retrievals which is the main focus of this paper. Therefore it becomes clear that a better dictionary structure will have significant effects on total Python performance.

2 Current Python Implementation

In this section we give a brief overview of the key implementation details of dictionaries in CPython 2.7.3. We should note that Pythons dictionary

supports several different data types as keys. However as we noted earlier, the dictionaries underlying class instances and modules have only strings as keys. Therefore optimizing a dictionary which have string-only keys is crucial to the runtime of any Python program. CPython accommodates for this optimization by changing its dictionary lookup method as necessary. First when a dictionary is instantiated, CPython uses a string-only lookup function until a search for non-string is requested, in this case CPython falls back to a more general lookup method and uses this general function onward. There are two main optimizations in the string-only lookup function. First, since the string comparisons do not raise any exceptions by design, some of the error checking logic can been removed. Second, the string-only lookup function can skip the rich set of comparisons (¡=, ¿=, ¡, ¿, ==, !=) which arbitrary data types can provide since string type does not have these special cases. As it can be seen, string-only keys can dominate the runtime of a dictionary and CPython uses some measures to optimize for this common case. Because of this importance we mostly limit ourselves to the study of new implementations of hashing the string type for the rest of the paper.

The current CPython dictionary implementation uses an iterative XORed polynomial terms for calculating the hash of the strings. In this implementation the hash value for sequential strings differ only in the least significant bits, for instance: hash("6.851")=7824665118634166871 and hash("6.852")=7824665118634166868. This behavior gives good results for sequential strings which is a common case. When the table size is 2^i taking the least significant i bits as the index can populate the table without a lot of collisions. This approach gives a better than random results in this case and is also simple and fast to compute.

On the other hand this hashing strategy has the tendency to fill contiguous slots of the hash table. CPython tries to solve this behavior by utilizing a good collision resolution strategy. The collision resolution strategy currently used is open addressing with a custom non-linear probing scheme. In this scheme CPython uses quadratic probing based on this function: $j = ((5j) + 1) \bmod 2^i$ where $0 < j < 2^i - 1$. This recurrence alone is not enough for a better behavior since again it behaves the same as linear-probing for some keys since it scans the table entries in a fixed order. To overcome this issue CPython also uses the most significant bits of the hash value with the following code snippet:

```
slot = hash;
perturb = hash;
```

```
while (<slot is full> && <item in slot does not equal the key>) {
       slot = (5*slot) + 1 + perturb;
       perturb >>= 5;
}
```

Based on this new method the probe sequence will depend on nearly all the bits of a hash value. We should note that the value of perturb shift (currently 5) is crucial for a good probing behavior. Since it should be small enough so that all the significant bits have an effect on the probe sequence but at the same time it should be large enough so that in really bad cases the high-order hash bits can affect the earlier iterations. CPython initializes its table with 8 slots. The resizing strategy is based on a load factor invariant. Whenever there are n keys in the dictionary, there must be at least $3n/2$ slots in the table. This load factor is to keep the table sparse and put a bound on the number of collisions that can happen. CPython resizes the size of table whenever the load factor invariant gets violated. It quadruples the size when there are less than 50,000 keys in the table and doubles the size otherwise. Overall, the design of CPython dictionary is simple and is optimized for more common cases such as sequential strings. Since it utilizes quadratic probing with perturbation, even if a data type does not provide a good hash function, the table still gets populated with with short chain lengths. On the other hand there are some behaviors which can be improved. For instance there is a bad cache locality because for probing a slot the table might need to jump to the other slots which are not immediate neighbors and not necessarily in cache.

3 Improvements

3.1 Theoretical Results

Recently, Patrascu et al. [2] showed that simple tabulation hashing provides unexpectedly strong guarantees and is constant competitive with a truly random hash function for hashing with linear probing. They also showed some experimental evaluations which proved that simple tabulation on fixed input length can be fast and robust when used with linear probing and cuckoo hashing.

Linear probing is appealing because of its excellent cache behavior in comparison to the probing scheme currently utilized in Python. Consider

the Intel Core 2 Duo processor with a 64 byte cache line. Each Python dictionary entry is 12 bytes so when Pythons current probing scheme probes another slot, it is not in the current cache line so it has to pull another cache line in from memory, but with linear probing, that next slot is most likely in the current cache line.

However, Patrascu et al. [2] describe simple tabulation hashing for fixed key widths. We will need to figure out a variant of simple tabulation hashing for variable length strings that does not require a table for every possible index (we could potentially cycle through a fixed set of tables). Performing table lookups is potentially an expensive operation with lots of cache hits mitigating the better cache behavior of linear probing so we will have to be careful about how we implement it.

3.2 Implementation Details

We prepopulated random tables in memory with random bits from random.org, which sources randomness form atmospheric noise.

```
while (--len >= 0) {
    register long index = (*p++) | ((len&TABLE_MASK)<<8);
    x = x ^ RAND_TABLE_NAME[index];
}
```

3.2.1 Optimizations

Cache

4 Results

4.1 Table Number Tuning

We attempted to measure the

Eight tables provided the best performance.

4.2 Benchmarks

In order to analyze the result of this new dictionary we measured three main metrics. First, we compared the performance of the simple tabulation

hashing with Pythons current hashing scheme on random strings. Second, we measured the number of probes and collisions and overall the quality of our dictionary on structured and unstructured inputs. Finally, in this section we show some of the total time spent in dictionary operations by using a set of general Python benchmarks and some real-world programs. Because of the heavy utilization of dictionaries in Python internals one of the valuable benchmarks is to see how the new dictionary performs on them. We used PyBench benchmark suit for this task. We saw significant improvements of around %20 in performance of some of the internals but at the same time some of the internals were running slower compared to current CPython implementation. On average there was %4.5 improvement in runtime for running the whole benchmark suit.

Put some of the number:

PyBench showed the performance difference in each of the individual Python internals however for a more realistic picture of the performance in real world program benchmarking the new implementation was needed. Tornado

Conclusion and future work Hardware specific performance issues dominate at small scales Simple tabulation yields slightly faster and more consistent performance Memory latency presents serious but workable issues Use different dictionaries for different purposes Improve performance bottlenecks illustrated by benchmarks