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

Russell Cohen Thomas Georgiou Pedram Razavi

May 20, 2012

Abstract

Here we implement simple tabulation hashing with linear probing in Python dictionary based on the recent theoretical result of Pătraşcu et al. Although we achieved significant performance wins on synthetic benchmarks, on real benchmarks our wins were only on the order of 4% due to interpreter overhead. Linear probing with sample tabulation hashing does improve Python performance and should be looked at as a replacement scheme. As a whole, the extra memory bandwidth and latency required to compute the simple tabulation hash negates some of the performance improvement, but if they were able to be further mitigated, the new scheme should provide more than a 4% improvement, making it even more worthwhile.

1 Introduction

In this project we aim to analyze and improve the performance of Python's dictionary or dict by changing its underlying implementation. 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, with 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 user's 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 to 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:

- Class method lookup
- Instance attribute lookup and global variables lookup
- Python built-ins module
- 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 Python's 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 predominantly focus on the study of new implementations of hashes for 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 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. 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, Pătraşcu et al. [1] showed that simple tabulation hashing provides unexpectedly strong guarantees and is constant competitive with a truly random hash function for hashing with linear probing. In their paper, they showed that for simple tabulation hashing with linear probing if the table size is $m = (1 + \epsilon)n$, then the expected time per operation is $O(1/\epsilon^2)$. This new guarantee asymptotically matches the bound for a truly random function. 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 Python probes a dictionary slot, it pulls in at least 4 more dictionary entries with it. However, when the 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. With linear probing, that next slot is most likely in the current cache line.

Pătraşcu et al. [1] describe simple tabulation hashing for fixed key widths. For implementing this new hashing scheme, first we needed 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 to implement this. In the following section we explain the implementation details of our approach to these problems.

3.2 Implementation Details

We modified the source code of CPython 2.7.3 to use simple tabulation hashing with linear probing. We use a fixed number of tabulation tables and cycle between them for variable length strings.

We prepopulated random tables in memory with random bits from random.org, which sources randomness from atmospheric noise in order to remove the reliance on pseudo-random number generators.

Following is an excerpt from our hashing code. We use bit tricks in order to prevent costly modulo instructions. For each byte in the string, we construct an index by adding the byte value and a table offset together. Then we lookup the value in a large table of random 64 bit longs and then XOR it with the previous result in the chain of XORs.

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

We also experimented with indexing using every pair of bytes in the string, treating them as 16bit integers and using 2^{16} tabulation table entries.

To implement linear probing, we modified Python's dictionary code to start at the table entry defined by the hash value and then search sequentially until it finds the item it is looking for or an empty cell.

3.2.1 Optimizations

After implementation and profiling, we found that dictionary table lookups had over twice as many level 1 cache misses with tabulation hashing than without. Hypothesizing that the tabulation tables were evicting other, more relevant data from the level 1 cache. To fix this, we inserted x86 memory prefetch instructions into our hash code to tell the CPU's caching system that it should deprioritize storing the tabulation tables in cache. We used GCC's `__builtin_prefetch(void * addr, char rw, char locality)` to generate these instructions. However, this only yielded about a 1% performance improvement, negligible compared to the performance improvement of different hash schemes.

4 Results

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 and sequential 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.

4.1 Table Number Tuning

For simple tabulation hashing with variable length strings, we decided to use a fixed number of tables N and round robin between them for every byte. We found the optimal value of N to be 8 experimentally. We limited our choices to powers of 2 so that costly modulo operations could be avoided by using bit masks Figure 1 shows the performance of different numbers of tabulation tables with both 1 and 2 byte indices. Eight hash tables indexing every byte separately resulted in the best performance. We believe this is due to the low entropy caused by fewer tables and too much memory bandwidth used by a higher number of tables. The optimal result may also be an artifact of the fact that we were using on the order of magnitude of thousands to millions of dictionary keys. Our optimal result resulted using 256 64 bit table entries in each of 8 tables using 16KB of memory.

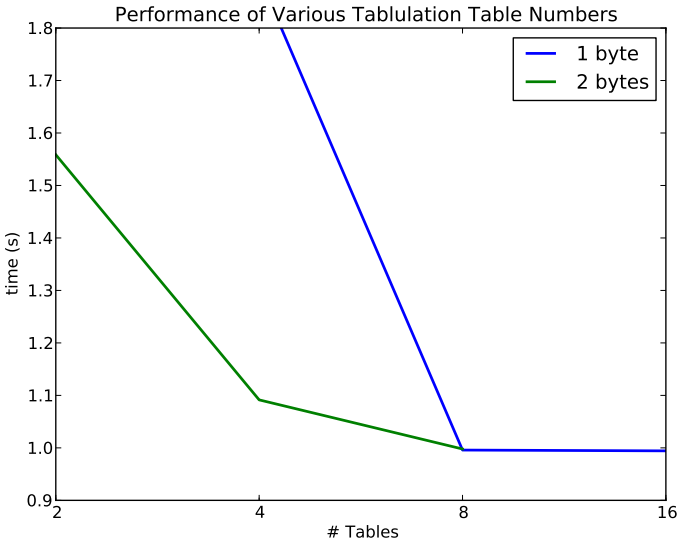

Figure 1: Eight tabulation tables with 1 byte indices provides the best performance for a dictionary with 6 million keys

4.2 Synthetic Benchmarks

Our primary benchmark, hash-table-shootout, is a synthetic one that called the CPython dictionary code directly from C so that all the interpreter overhead was bypassed. We measured performance for both sequential strings, generated by converting sequential integers into their string representation, and for random strings, generated by the string representation of random integers. For each of the benchmarks, we inserted a given number of keys into the dictionary and measured the runtime.

Figure 2 shows the performance for random strings. Both linear probing and simple tabulation result in improved performance over the current CPython dictionary implementation but combining both techniques results in over a 20% improvement. Figure 3 shows very similar results for sequential strings.

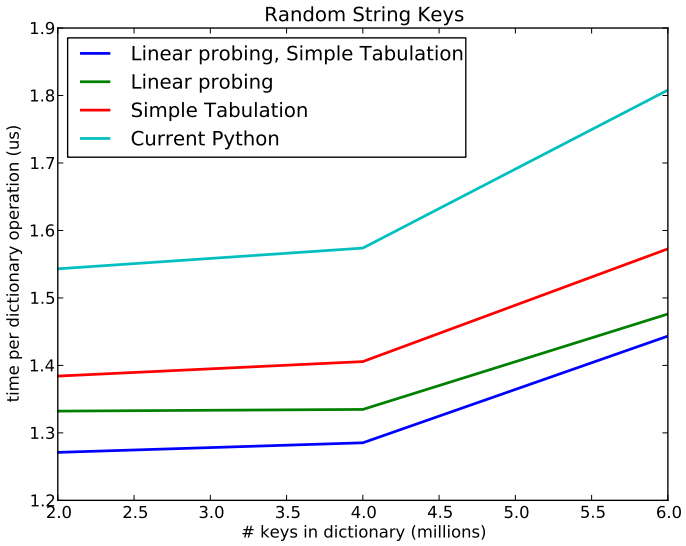

Figure 2: Random string performance

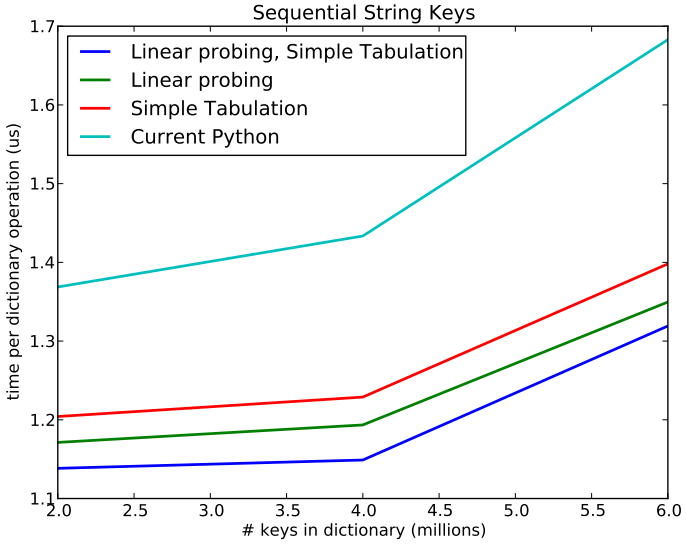

Figure 3: Sequential string performance

Figure 4 shows the average chain length, or dictionary cell probes per lookup for a run of the benchmark on sequential strings. Simple tabulation hashing with CPython's current collision resolution strategy results in the fewest number of probes. However, despite having the fewest number of probes, its real relative performance is much worse as seen in Figure 3. Linear probing performs faster than its average chain length suggests, showing the impact of looking at entries in the same cache line first rather than jumping around.

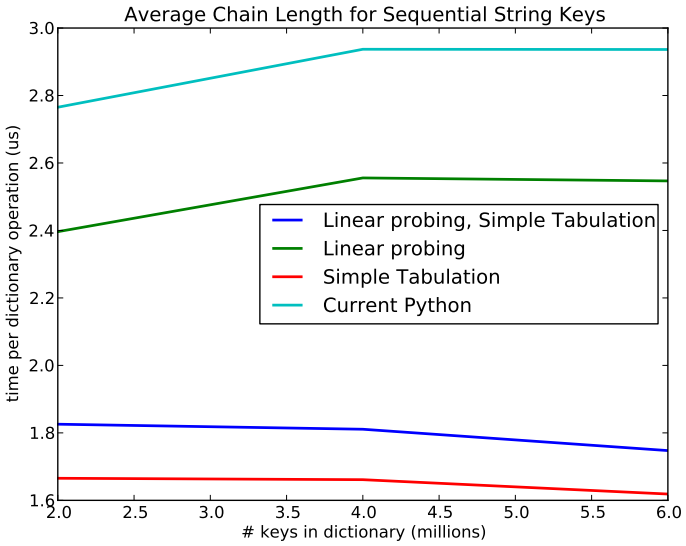

Figure 4: Simple tabulation hashing decreases the average chain length

4.3 Benchmarks

Due to the heavy utilization of dictionaries in Python internals, one of the valuable benchmarks is to see how the new dictionary performs with them them. We used PyBench benchmark suite for this task. The results can be seen in Table 1. We saw significant improvements of around 20% in performance of some of the internals such as the special and normal class attribute lookups and string predicates. However, at the same time some of the other internals such as CPython exception handler were running around 12% slower compared to the current implementation. On average a 4.5% improvement in runtime was measured for the whole benchmark suite.

PyBench showed the performance difference in each of the individual CPython internals however for a more realistic picture of the performance in real world programs more extensive benchmarking of the new implementation was needed. Especially since we wanted to know the frequency of the different CPython internals in real-world applications and therefore the corresponding runtime changes. The new implementation was tested on several standard real world programs such as Tornado web server, Django web framework, html5lib and ten other real-world programs.

We were interested to calculate if the new implementation of dictionary can handle more requests in a Python web server compared to the current CPython implementation. We chose Tornado web server for this task since most of the code is written in pure Python. An instance of Tornado server was run in a virtual machine with one assigned processor. The code of the application used is provided in the appendix.

Different request rates (up to 2000 requests per second) were tested on this sample server and their corresponding response rates were calculated using httperf, a HTTP performance measurement tool. Considering the variance of request rates on different iterations of the test, we could not find any significant improvement of the response rate of the new implementation. We found that on the average performance was similar however the new implementation showed a smaller standard deviation in the response rates.

Table 1: Performance results on selected components of the PyBench suite. We have omitted the least significant performance results in terms of percent change.

Test Name	**Performance Difference**
NormalClassAttribute	19.20%
SpecialClassAttribute	18.88%
NormalInstanceAttribute	17.77%
StringPredicates	15.25%
UnicodePredicates	13.82%
SimpleListManipulation	13.03%
CompareUnicode	11.27%
SimpleComplexArithmetic	10.83%
SpecialInstanceAttribute	10.52%
TryFinally	10.35%
CreateUnicodeWithConcat	8.52%
StringSlicing	6.52%
Small Performance Changes Omitted	...
UnicodeMappings	-1.27%
ListSlicing	-2.40%
ComplexPythonFunctionCalls	-3.55%
TupleSlicing	-3.72%
WithRaiseExcept	-5.89%
TryRaiseExcept	-11.99%
Totals	4.54%

This finding as well as those by other benchmarks showed that the new implementation gives more consistent runtime on different iterations of a same code compared to the current CPython implementation. This consistency is a desired behavior. We suspect that this behavior is a result of the "better" randomness of hash values in simple tabulation hashing and also the smaller average chain lengths.

As mentioned before, another set of the benchmarks that we utilized was based on different standard real-world Python applications. We ran and timed the performance of the new implementation and CPython 2.7.3 side by side using the Python-Mirrors suite. The performance of more than ten applications were recoded as shown in the figures.

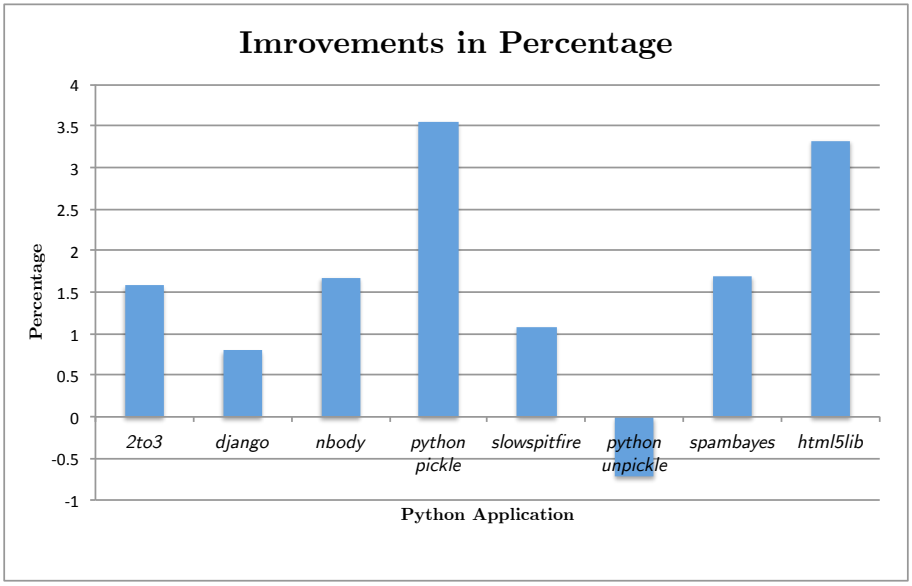

Figure 5: Percentage of overall improvements in some real-world Python applications. Nearly none of the applications performed worse. Python Pickle and html5lib faster around 3% to 4%.

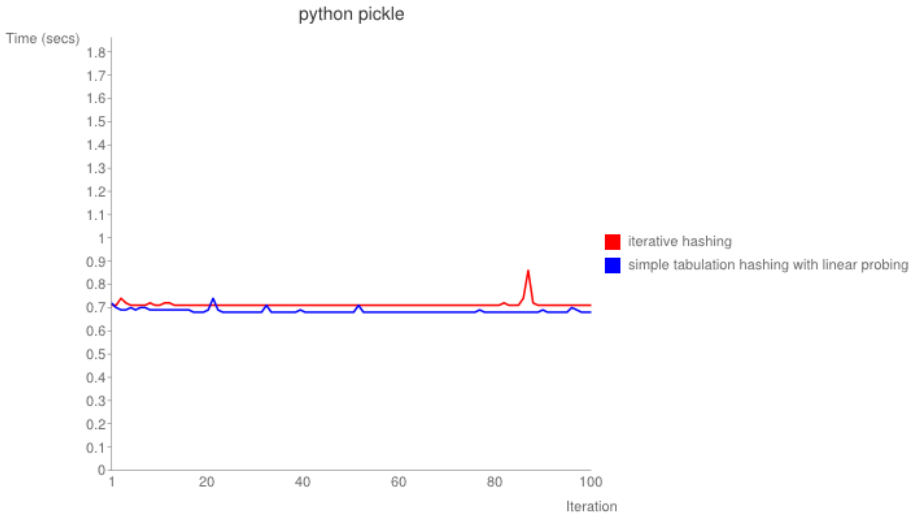

Figure 6: Python Pickle module benchmark. Minimum Iteration: changed from 0.705s to 0.681s: 1.04 times faster. Average iteration: changed from 0.710s to 0.686s: 1.04 times faster. Standard deviation of new implementation 2.0383 times smaller

In a couple of the programs such as pure Python Pickle (an object serialization library) and html5lib (specification parser library) we calculated an improvement of 3% to 4%. In other cases we did not calculate an speedup but at the same time we did not find any real-world case which the new implementation performs significantly slower. These results were consistent with the previous results of PyBench that showed about 4.5% improvement.

5 Conclusion and future work

Throughout the course of our work we came to the following conclusions:

1. Hardware specific performance issues dominate at small scales: When you are attempting to work on a process that runs as fast as a hash table lookup or insertion already does, there is little room for error. Improving hash performance with simple tabulation is not an easy task simply because any change you make will probably end up making the hash function run slower. That said:

2. Simple tabulation yields slightly faster and more consistent performance: With a careful implementation simple tabulation can yield increased hash performance on linear probing work loads. Simple tabulation definitely leads to shorter chain lengths, however, those shorter chain lengths dont always justify the performance cost in all cases.

3. Memory latency presents serious but workable issues. Doing a look up in the simple-tabulation table *requires* a random memory access, exactly what we were trying to avoid. With careful memory management we can maintain good performance, but a solution requires careful thought about cache sizes and special purpose compiler directives.

In terms of future work, we have 2 avenues of attack. First, we could use different dictionaries for different purposes. Many of the Python dictionaries are small, and their keysets never change.

New built in functions are not added during the runtime of the program and thus they could use perfect hashing.

Second, we could improve the performance bottlenecks illustrated by benchmarks. These mainly consist of the extra time and memory bandwith and latency cost for computing simple tabulation hashing. If we optimize the hashing further through more profiling, memory layout, and instruction order optimizing, we may be able to achieve more real world gains. We could also look into applying a modifying of the perturbation trick currently used in Python for quadratic probing.

Given the strong performance of simple tabulation hashing and the only downside being the cost of memory lookups, processor manufacturers should look into adding random tables into hardware as a ROM. We found that for what we found optimal for Python, we only needed 16KB of randomness. This would be a trivial cost in ROM given that processors have MBs of higher level caches nowadays.

References

[1] Mihai Pătraşcu and Mikkel Thorup. The power of simple tabulation hashing. In *Proc. of 43rd STOC*, 2011.

[2] Python Software Foundation , `http://python.org` 2012.

[3] Nick Welch. Hash-table-shootout. `https://github.com/mackstann/hash-table-shootout` 2012.

[4] Nicholas Pil. Benchmark of Python WSGI Servers.

`http://nichol.as/benchmark-of-python-web-servers` 2010.

Appendix

A Tornado Application

Sample Tornado 2.2.1 application used for benchmark:

```
import tornado.httpserver
import tornado.ioloop
import tornado.wsgi

def application(environ, start_response):
    status = '200 OK'
    output = 'Hello World!'
    response_headers = [('Content-type', 'text/plain'),
        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)
    return [output]
def main():
    container = tornado.wsgi.WSGIContainer(application)
    http_server = tornado.httpserver.HTTPServer(container)
    http_server.listen(8000)
    tornado.ioloop.IOLoop.instance().start()
if __name__ == "__main__":
    main()
```