February 25, 2015

TWO-STEP BUCKETING TO ACHIEVE HIGH PERFORMANCE DATA STRUCTURES

Geoffrey Pike

Justin Lebar

TWO-STEP BUCKETING TO ACHIEVE HIGH PERFORMANCE DATA STRUCTURES

Abstract:

A variety of techniques and strategies are presented for protecting a data structure, e.g., a hash table, from an attacker who wishes to force worst-case performance. Many data structures are designed with fast and efficient insertion and lookup functionality under normal use-case scenarios. However, when forced into a worst-case context, the performance of the data structure is undermined and services relying on the data structure are impacted. The techniques presented maintain the high performance under normal use-case while protecting the data structure from malicious use.

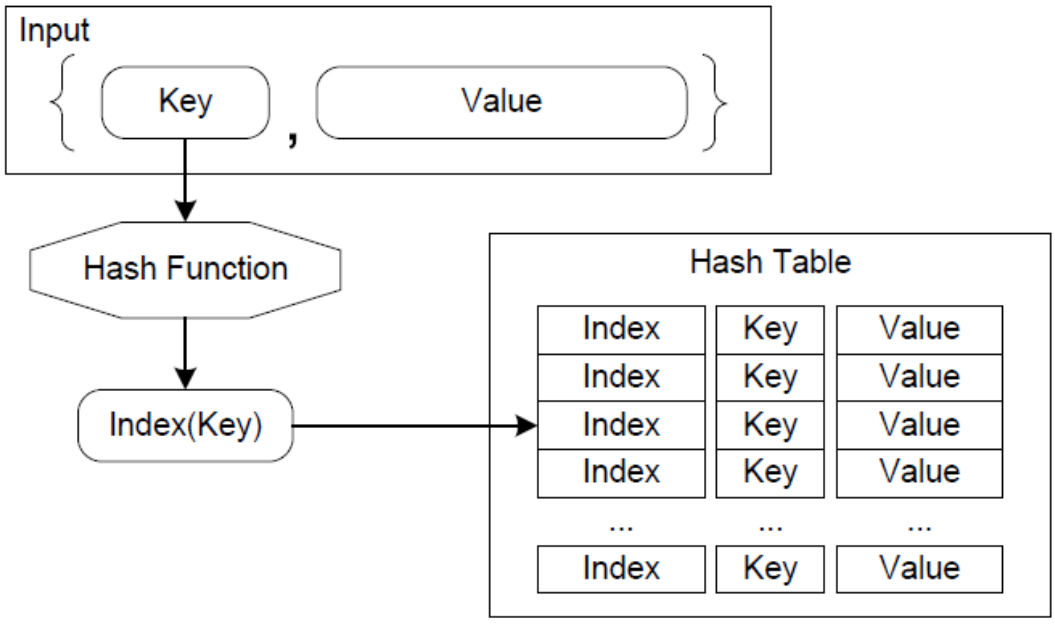
Introduction:

In certain data structures such as a hash table or map, an insert function uses a key to identify an index or address for a memory "bucket" in which to store an entry associated with the key. A lookup function then uses the key to identify the index and directly retrieve the corresponding entry from the bucket. Typically, the entry is a {key, value} pair, although the value portion of the pair may be inconsequential, i.e., the entry can just be the key itself.

A collision occurs when multiple keys correspond to the same bucket index. If insertion of an entry collides with data already present in the data structure, the insert function may use a second data structure to represent multiple entries in association with the same bucket (e.g., as a linked list), or the insert function may attempt to locate an unused bucket in the structure in which to store the new entry (known as "open addressing"). However, resolving the collision problem requires the insert function to take these additional steps (or to rebuild the data

structure), and requires the lookup function to search through the data structure for an entry matching the lookup key. The additional steps, e.g., the search and verification, can undermine the efficiency of the data structure, particularly if an excessive number of entries collide.

A malicious third-party (an "attacker") can attempt to insert entries in a manner that causes excessive collisions. Because the collisions require additional computational steps to resolve, the attacker can slow down or cripple a computational system by selecting keys (e.g., as system input) that correspond to the same bucket index. Therefore, it is desirable to have insert and lookup functions that minimize the ability of an attacker to do so, without sacrificing speed or efficiency of the data structure.



Description:

A simple insert function attempts to place an entry (e.g., a {key, value} pair) into a data structure using an index calculated as a function, e.g., a hash function, H, of the key, k, modulo

the number of buckets, B, in the data structure. That is: Index = H(k) % B. However, an attacker may be able to construct a set of keys such that the same index is computed for each, thereby causing collisions. If it is trivial to construct such a set, then an attacker who performs N computational steps can impose αN^2 computational steps, where α is a constant, on typical hash-based data structures.

However, even if H and B are unknown, by trial and error it may be possible to construct a set of keys that mostly hash to the same bucket. For example, the attacker can observe or estimate how long the data structure takes to process the insertions or deletions of various keys. In this case, an attacker who performs N computational steps can impose $\beta N^{1.5}$ computational steps, where β is a constant, on typical hash-based data structures.

A number of strategies have been developed to make index resolution less predictable, making it more difficult for an attacker to undermine the data structure. However, these strategies generally suffer from one or more flaws. For example, a large number of buckets, B, may make it more difficult to randomly find keys that hash to the same index. However, if the context of the data structure doesn't require the large number of buckets, the extra space allocation wastes resources. Another strategy is to pick an unusual or complex hash function, H. However, as mentioned above, this may still fail to prevent an observant attacker from doing damage.

A variety of strategies modify an open addressing probing sequence. In some data structure designs, when an insert function encounters a pre-existing entry, the insert function identifies a sequence of alternative location indices to probe until an unused bucket is found. A simple probing sequence is linear probing, in which the probes are at locations H(k) mod B,

H(k) + m mod B, H(k) + 2m mod B, and so forth, where *m* is a constant value that is either 1 or is selected to be relatively prime to B. Linear probing is subject to clustering. Another strategy is quadratic probing, in which the probes are at locations H(k) mod B, H(k) + p(0) mod B, H(k) + p(1) mod B, H(k) + p(2) mod B, and so on, where p is a linear function. However, these incremental probing sequences each suffer the same flaw, which is that any number of keys that hash to the same value mod B will probe the same sequence of buckets, resulting in collisions deeper and deeper into the probe sequence.

A slightly different strategy for modifying the open addressing probing sequence is known as double hashing. In double hashing, the bucket number for the i^{th} probe is H(k) + *i* H′(k), mod B, where H′ is a second has function. This, too, is subject to an attack if an attacker finds keys that map to the initial bucket (because they have the value for H(k) mod B) and the same probe sequence (because they also have the value for H′(k) mod B).

On the other hand, hash tables that use linked lists in buckets can suffer from the worst case performance of a linked list if an attacker manages to find many keys that hash to the same bucket. Similarly, hash tables that use balanced trees in buckets can suffer from the worst case performance of balanced trees.

One solution is to combine open addressing strategies in multiple stages. For example, an insert function can use quadratic probing up to some specific number of probes. Then, if no available bucket has been identified, switch to the double hashing approach. This burdens the attacker with finding keys that undermine double hashing, but frees the actual data structure from the computational expense of double hashing in the best or average-case scenario. In this

strategy, the hash function, H′, used for the increment can be a more computationally intense hash function, e.g., a secure hash function such as SHA-3.

Other complex open addressing strategies also can be used, including, for example, using a set of hash functions H_1, H_2, … H_n, for the open addressing probe sequence. An insert function computes a first index as $H_1(k)$ % B. If the first bucket for the first index is already in use, the insert function replaces the hash function H_1 with the next function in the set, i.e., H_2, and computes another index, e.g., $H_2(k)$ % B. If this index is also already in use, the insert function continues to proceed through the set of hash functions until the set is exhausted. It then uses the same process to relocate the existing entries within the structure until a place is opened up for the new entry. That is, the existing entry at $H_1(k)$ % B, which has key k′, is moved to, for example, $H_2(k')$ % B, or the next open bucket by sequencing through the set of hash functions on key k′. A corresponding lookup function computes the first address, second address, and so forth until it finds the entry corresponding to the lookup key. However, an attacker with knowledge of each of the hash functions, H_1, H_2, … H_n, can overwhelm this strategy in much the same way as before. There is also the added computational complexity of calculating the various hash values. Worse, the relocation steps in the insert function can be computationally cost prohibitive.

In a random-probe approach, the probe sequence is dependent on a hash function (which could be a secure hash function) and a pseudo-random number generator (which could be a secure pseudo-random number generator). As before, an initial bucket, b_0, is at H(k) mod B. If b0 is occupied, the insert function generates a pseudo-random number generator, r, using one of k, H(k), or H′(k) as the seed, such that the output of r.next() is deterministic for a given key. Each probe *i*, is based on the next output from the pseudo-random number generator, r.next(). For example, each probe *i* can be directed to an index location equal to r.next() mod B, or to a

location equal to x + r.next(), modulo B, where x is the previous bucket index probed. As an optimization, all (or a portion) of the possible bucket indices for a given key can be generated using the pseudo-random number generator, and cached as a set-permutation of some or all of the buckets to search for the given key.

The random-probe approach can be further protected from an adversary by introducing entropy into the probe sequence. A secondary data structure stores information used to track the introduced entropy, so that probe sequences can be reproducible. An adversary who knows everything about the data structure cannot know the entropy information, because it is determined as new keys are encountered.

For example, an array (z) of entropic values is maintained with each value corresponding to a bucket. Initially, all of the values in z are set to 0, the number of probes (i) is 0, and an entropy parameter q is set to a positive integer. For each probe (or each probe after the initial check of b_0), the insert function sets x = r.next(), calculates bucket index $b_i = x$ mod B, and increments the probe count i. If the entry can be inserted at b_i, then the insert function does so. Otherwise, the insert function determines whether to introduce entropy: if i modulo q is non-zero, the probe sequence simply continues, otherwise (when i modulo q is zero) the pseudo-random number generator, r, is re-seeded before continuing with the probe sequence. If $z[b_i]$ is still set to 0, a random value is selected and stored there, that is, $z[b_i]$ is set to a random value using a different source of randomness, e.g., another pseudo-random number generator with a non-deterministic seed. When re-seeding r, the entropic value stored at $z[b_i]$ is used as the seed, or to generate the seed (e.g., using a combination of $z[b_i]$ and x as the seed). The resulting probe sequence is thus unpredictable, while also computationally efficient and reproducible by the lookup function.

In a sufficiently large data structure, using some of the strategies discussed herein, it is unlikely that an insert function would have to step through too many probes in a probe sequence. Therefore, it is reasonable to suspect that, after some number of probes, an insert function has been called maliciously. As a defense against a denial of service attack, the insert function can de-prioritize the insert after some number of probes. For example, a guarding function can have an initial probe number limit (e.g., limit = 12), where the guard function attempts to insert an entry using a key, but only up to the limit number of probes. If the insert fails, the limit is increased by some percentage, all relevant locks are release, and the guard function thread sleeps for some length of time (e.g., a number of milliseconds equal to the limit). The guard function then re-acquires any necessary locks and re-attempts the insert with the higher limit.

Another alternative defense against an attacker is to modify the data structure itself. For example, if an insert function is unable to find an available bucket after some number of probes, the insert function stores the entry in a second fallback structure, e.g., a trie. To improve performance, a third structure, e.g., a Bloom filter, may be used to estimate whether a given key cannot be present in the trie. An alternative third structure is a set of values (w), where $w_{\text{key}} = H''(\text{key}) \bmod y$, for some constant value of y. If the entry of a given key is relegated to the fallback structure, a value w_{key} is entered into the set. In practice, the fallback structure will contain few keys unless an adversary is attempting to flood the data structure, so searching the set of (w) values should be a minimal additional cost.

References/Background material:

Donald E. Knuth. The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley, Reading, MA, 1973.

Dietzfelbinger et al. Dynamic Perfect Hashing. SICOMP 23 (1994) 738-761.

Per-Åke Larson, Dynamic Hash Tables, CACM 31(4):446-457, April 1988.

http://xlinux.nist.gov/dads//HTML/extendibleHashing.html

Scott Crosby and Dan S. Wallach, Denial of Service via Algorithmic Complexity Attacks, 12th Usenix Security Symposium (Washington, D.C.), August 2003.