Introduction to the HAMT and Tcl

Don Porter
NIST
donald.porter@nist.gov

October 18, 2017

Abstract

The structure and function of the Hash Array-Mapped Trie (HAMT) is presented from the perspective of those familiar with analogous hash table data structures in the implementation of Tcl. The immutable nature of values represented by the HAMT structure is a perfect match to the value semantics of the Tcl language. A prototype implementation of the HAMT within a branch of the Tcl codebase permits comparisons with the incumbent dictionary values. Catastrophic performance collapse in scripts caused by the ”copy on write” of shared values disappears, as the HAMT design does not rely on that scheme. Even in a prototype with little effort toward explicitly tackling performance, the speed of basic HAMT operations are within a factor of 2 of the corresponding dict operations. Opportunities for further improvement abound. Related data structures such as Relaxed Radix Balance (RRB) Trees may have similar application to Tcl lists and strings following the example of their use as the basis of immutable vectors in Clojure. Success promises to deliver greatly improved multithreading results, as concurrent sharing of explicitly immutable values is an inherently simpler problem. Memory usage and extension interfaces may also see substantial improvements.

1 Value Semantics in Tcl

Tcl programmers recognize the mantra that “Everything is a String”. It brings to mind the concrete nature of the Tcl 7 implementation where every value is indeed implemented as a dynamically allocated, NULL-terminated char array.

The days of Tcl 7 are long in the past. We now regularly enjoy strained, prolonged discussions as people try to invent a new modified mantra that says what is important about the nature of Tcl values, but doesn’t give a false impression suggesting the continued use of a non-performing implementation. The interchangability of any Tcl value with a string value remains true. The ability to make string-like queries of any value is still presumed. How long is this value? What is it’s nth character? The utility of the string as a medium of value exchange remains important.

That said, much of the utility of the nature of Tcl values lies in the simpler truth that Tcl follows value semantics. A value is a value. Two values that are the same in every way behave the same in every way. There is no place for the identity or the container of a value to revise its interactions. This is in contrast to languages where reference semantics play an essential role. In such languages, programming becomes not so much an exercise of data processing applied to fully known values, but a matter of directing entities to take actions or transform themselves. The distinction is between dead values and living objects. You can count on a dead value to remain unchanged. This inherently limits the kind of troubles they can cause. This is a considerable part of what makes programming in Tcl a more manageable task than programming in some other languages. There's simply less trouble available to get into. To be fair, there is some loss of conventional programming techniques that are built on reference semantics.

The value semantics in Tcl is one of the key things it shares with Lisp. It gives much of Tcl a near-functional character. Recent years have shown pockets of renewed enthusiasm for the value semantics approach in many developer circles. Several "new" languages (newer than Tcl) have explicitly embraced value semantics, implicitly endorsing some of Tcl's model of operation. Examples include Scala and Clojure. For the most part their development heritage is rooted in Lisp instead of Tcl, but that just indicates we have shared taste in what foundations are more attractive.

There is a key difference in the value semantics delivered by Tcl and those delivered by Clojure for example. In Clojure, immutable values are provided by implementations where their immutable nature is directly structured in the implementation. In contrast, Tcl's implementation is written in C, and most of the underlying C code is the use and management of C primitives that do not themselves guarantee value semantics. Consider the Tcl string implemented as a NULL-terminated char array. The elements of the char array can be written as well as they can be read. Overwriting a byte in a string changes the value of the string. The Tcl implementation is not inherently protected against such failure to deliver value semantics to the script level. Instead, the value semantics are properly implemented only via careful attention to the programming discipline of Copy on Write. Any scheme that relies on perfect application of discipline is at risk of failure. The Tcl world does fail from time to time as the very existence of the phrase "EIAS violation" makes clear.

Note that sustaining the "Copy on Write" discipline is the key reason Tcl values must be handled with precisely correct management of reference counts. An implementation that can inherently deliver immutable value without a visible Copy on Write burden makes escape from the reference counting a plausible future. Further note that Tcl's current inability to directly share values between threads is mostly rooted in the use of a reference counting scheme that is not thread safe. Escaping that implementation may well hold the key to letting any number of threads share read accesss to an immutable value they all share an interest in. As the multicore model of hardware systems has fully taken over the world, this has increasing importance to the ability to produce software that efficiently makes full use of available hardware capability.

Developments in computer science starting in the late 1990s have produced data structures designed to directly provide immutable values of various kinds, while achieving reasonable levels of both time and memory efficiency. Clojure is built on such structures, like the Hash Array-Mapped Trie and the Relaxed Radix Balance Tree. Here we begin to explore the prospects for re-implementing portions of Tcl on data structures that directly support Tcl's value semantics. This may increase the complexity of the inner core of Tcl's implementation, but it may have the effect of making the interfaces available to extensions easier and safer to use.

2 From Hash Table to HAMT

In this manuscript, we consider the Hash Array-Mapped Trie, an immutable data structure that provides the same function as Tcl's long established hash tables. The core concept of the hash table is to store and retrieve key-value pairs, and allow a particular key to be found quickly. A hash value computed from the key determines where it is to be found. Given a Hash function,

```
size_t Hash(Key key)
```

that produces a pointer-sized value for each key in the `Key` domain, the simple strategy is to use that hash to select one "bucket" from among all buckets to look in for the desired key.

```
Search bucket[Hash(key)] for key.
```

A quality `Hash` function is one that distributes the domain of `Key`s evenly over the range of hash values. The consequence is that each bucket search should be through only a small number of entries. All the entries that land in the same bucket are stored in a linked list that must be scanned one entry at a time. A `Hash` function should also not be time-consuming beyond necessity.

A literal interpretation of this scheme runs into immediate practical difficulty. On the usual 64-bit systems of today, the domain of hash values has size 2^{64}. This implies a bucket array of that size, imposing a memory requirement of 128 exibytes, almost all of which exists just to hold NULL pointers. Some revision to the concept is needed to make a practical implementation. The clear appropriate choice for hash value size today is size_t, as that is the size the hardware is capable of computing without additional cost. Tcl hash tables are instead based on a hash value of type unsigned int. That implies a much smaller requirement of *only* 32 gibibytes, but that's still impractical (and beyond the capability of Tcl's incumbent memory allocators).

For Tcl's hash tables, the practical implementation takes the form of

```
Search bucket[Hash(key) & mask] for key.
```

where the mask limits the number of bits in the hash that determine the bucket chosen for searching. Tcl hash tables are born using a mask of only two bits.

This implies four buckets, an easily manageable size. As more entries are added to the table, the buckets fill and grow entry lists extending from each. The Tcl hash tables are tuned to continue in this way until the total number of entries exceeds three times the number of buckets. At that point a typical bucket search starts to exceed three entries in length, and the judgment recorded in the design is that search size is too large. A new hash table is created with a mask with two more bits, or four times as many buckets. All entries in the original table have to be re-inserted into the new table, requiring an infrequent but substantial penalty. Either all keys must be hashed again, or the hash values need to have been stored within the entries. The price is paid in either time or memory. Note also that at a typical moment of hash table operations, each bucket can be expected to hold one or two entries, not the zero or one expected in the original hash concept. Tcl hash tables are implemented to never shrink, so once any table is large in size, it will continue to make use of that enlarged memory space, even if the number of keys stored in it drops to zero. Finally, if memory limitations have not yet already arisen, when the $805,306,368$th key is added to a Tcl dict, the hash table within that dict will crash out of an inability to scale up the bucket array again. (See Tcl Bug 3298012.) Even with some of these shortcomings, though, the Tcl hash table has served well for a lifetime; its function powers a large number of tasks underlying the Tcl implementation.

The HAMT is the result of a different series of steps of re-design away from the impractical giant bucket array of the original hash map conception. It achieves a different set of features along the way. Its fundamental purpose of being an efficient store of key-value pairs making up a dictionary remains effective.

The first step away from the giant bucket array is in the direction of even greater impracticality. Instead of imagining each bucket as an element in an array of 2^{64} elements, imagine each bucket as the leaf of a binary tree. To reach any particular bucket, we use the bits of a hash value, one by one, to navigate the tree from root to the bucket leaf determined by the entire hash value. This requires 64 steps from a root node all the way to the appropriate bucket. Taken as a literal desgin, this is an even more inefficient scheme, but it gives us an important conception of the structure that we can further modify. This is an example of a Trie, a tree structure with all content stored at the leaves, with a leaf selected by following a path directed by an index value. Here that index value is the hash value of the key, so this structure is known as a Hash Trie.

Like the giant bucket array, the Hash Trie uses vast amounts of space for buckets to do nothing more than store NULL pointers indicating the absence of any key matching the hash of that bucket. Recall that there are 2^{64} buckets in this conception. For any imaginable amount of data to store, almost every bucket will be empty. The number of buckets is exponentially larger than any practical storage use. Even storing a billion keys in a Hash Trie makes use of about one in a billion of the conceptual buckets.

This observation leads to the first step toward making a practical implementation. We can simply determine not to store any leaf that holds a NULL bucket. Furthermore, we can determine not to store any internal node that has no leaves. Every stored key still rests in a bucket, and that bucket is reached following a path of 64 steps from the root, but the massive waste of allocating space to store nothing is eliminated. Since each bucket is still reached by a 64 step path, it is implicitly identified with a particular 64-bit hash value that determines that path.

At this point, in most examples of valid states of the Hash Trie, there will be long chains of internal nodes representing the paths from root to a bucket leaf containing one or more keys. Some nodes will have two children, and represent an actual decision point in the path taking, but many will not. There is no need to store multiple steps of a path that has only one destination, so long as we can identify that destination. The next step, then, is to store in each bucket the hash value associated with it. At that point we no longer need the complete trail of breadcrumbs to tell us where we arrive, and to check whether that bucket of arrival is a match to the hash value we are trying to look up. This removes another large amount of unnecessary storage from the structure, by pulling all leaves up the tree to the point where they least differ from another leaf in the tree. Nodes exist to distinguish buckets actually stored in the tree, not buckets that could potentially be in the tree.

The next revision is to make a similar reform to paths reaching internal nodes. So long as each internal node has stored within it its identity in the complete binary trie, we no longer need to follow every step in a path to reach it. When following a path through the trie, a check at each internal node that its value is still consistent with the path we intend to take is sufficient to tell us whether our hash is leading to a bucket or to nothing.

At this point, we are storing more information in each bucket and each internal tree node, but we are storing many many fewer of them. In fact we are storing only internal nodes with two children. We have a binary tree. Each child can be either another internal node with two children of its own, or a bucket terminating any successful path. This covers all numbers of buckets except for zero and one. These get handled as special trivial cases. An empty Hash Trie doesn't contain the hash, and a singleton Hash Trie contains it only if the single bucket has a matching hash value. Note that the tree's structure is determined by its nature as a trie. In general it will not be balanced, though a good hash function will tend to produce nearly balanced trees. Even without guarantees of balance, the finite hash value size and the trie structure determine a maximum depth of 64, with depths closer to the base 2 log of the number of stored keys far more likely, and no possibility of a depth larger than the number of stored keys.

Lets pause a moment to reflect on the structure of an internal node at this point. We often think of an internal node of a binary tree as a simple matter of two pointers, one to the left child and one to the right child. As the design has evolved however, we require more information than that. We have two kinds of children, leaves and nodes, and we must distinguish them. Furthermore, we know that in the case of hash collisions a bucket might hold multiple key-value pairs, but there is no need for it to store the hash value multiple times.

Because of this, it makes more sense to store the hash value with the pointer to the bucket and not within the bucket. Given these needs, we determine to encode the various cases in a “map” field of the internal node. In our current conception this encoding can be achieved in two bits. The first bit is set if the left child is a bucket leaf. The second bit is set if the right child is a bucket leaf. We can then arrange to store all the data we need to describe the children of the node. When both map bits are set we need storage for two hash values and two pointers to buckets, a total of four pointers of space. An array of four void pointers will do the job. With other map values, we can use less space, as a pointer to a child node is all we need. We choose to organize the storage with the hash values first, in order, followed by the bucket pointers, in order, followed by the node pointers in order. Stepping through a pointer array is something modern hardware is built to do very well. The map is used to interpret the meaning of each pointer value in the array as we step through it. For some maps, a smaller pointer array is sufficient and we can allow other considerations to determine whether it is better to be frugal with memory and allocate only what is necessary, or whether there are benefits to a small waste that allows less complexity. This mechanism of internal node traversal is where the ”Array-Mapped” portion of the HAMT name comes from.

There is one more step in design evolution before we reach the implementation that has been coded. We have a map of two bits at each node, because that is all that is necessary to encode the cases of a node in our binary tree. The tree is binary because it is the remnant after reduction of a trie that was binary, based on a trie path guided one bit at a time. A trie need not be limited to binary form. Any number of bits can be used at each internal node to select among a set of children. We can easily make use of a tree branching factor that is any power of two. Storage for two bit maps are not convenient, but storage of pointer sized maps are. It happens that the programming is most convenient when we make use of two pointer-sized maps, one for the children that are buckets, and one for the children that are nodes. The simple encoding is that a set bit in the bucket map indicates that child is a bucket. A set bit in the node map indicates that child is a node. A child can only be one or the other, so the same bit cannot be set in both maps. The encoding in the maps is not of minimal size, but it is of very useful simplicity. A pointer-sized map contains 64 bits, so the implication is that we use a tree with a branching factor of 64. The same scheme of storing hash values first in order, then bucket pointers in order, then node pointers in order, with no NULLs anywhere in between is continued. In a 64-tree, we no longer have a guarantee that all child slots are filled. In most nodes, many slots will be empty. Our storage needs for the array pointer can vary from two (only two child nodes) to 128 (all 64 children are buckets with associated hash values) elements. With this variability, it tends to make more sense to allocate only the space that will actually be used. A HAMT with a branching factor of 64 can reach any bucket in any state in a maximum depth of 11, with depths closer to the base 64 log of the number of stored keys far more likely, and no possibility of a depth larger than the number of stored keys. Recall that the Tcl hash table was designed to expect to follow pointer chains of up to 3 links on average. Although that is considered a ”constant” access design, while this one might most properly be terms a logarithmic access design, the limits in practice of how large that logarithm can be makes the two schemes competitive, especially in practical storage volumes.

This section introduced the structure and functioning of an HAMT, with a step-by-step evoution of design that estalblished its effectiveness as a properly functioning implementation of the key-value pair storage and retrieval function. Furthermore, analysis suggests this approach at least has a chance at being competitive with incumbent hash tables in terms of time and memory efficiency.

3 Efficient Immutability with the HAMT

Now we come to the new capabilities the HAMT structure makes possible. The hash table is implemented as a mutable container. When we add a key-value pair to a hash table, we overwrite parts of it to reflect the new contents. In contrast, the HAMT is implemented as an immutable value. Any set of key-value pairs present in the HAMT determines the structure of that HAMT. When we seek to add a key-value pair to an existing HAMT, the idea is not to overwrite the HAMT with a new value, but to produce a completely new HAMT with the right structure to represent the new hash map value. We implement in the system language the same value semantics that serve us so well in Tcl scripts.

The key to success is that the new HAMT we create to represent the new value after key-value insertion shares large portions of the same structure as the original HAMT. Adding or revising or deleting a single bucket from our HAMT structure involves tracing a single path from the root to that bucket. The new HAMT will have new values for the internal nodes along that path to achieve a structure suitable for the new HAMT value. However, all the other branches off that path in the original HAMT tree remain unchanged. Because our design is premised on immutability, there is no need to make copies of those portions of the HAMT that will be the same in both trees. We just point to them from both HAMTs. This is the technique that allows us to keep a series of persistent values representing an arbitrarily long history of modification without paying the large price of making complete copies of entire structures at every step. History can be kept as long as it is useful. Sharing is not an enemy to the efficiency of an in-place operation. No operations are in-place, but all operations are (acceptably) efficient, by design. There’s no advice to teach scripters about how to carefully apply artful code techniques to avoid sharing that imposes expensive operations. We just don’t have expensive operations in the first place that need avoiding.

With an understanding of how hash map values are encoded into the HAMT structure, there are no great mysteries in how to code the creation of new values from old in the insert and remove operations. We do not belabor the details here.

There is one issue that the prolific sharing of this approach raises. How can we know when some node that is a part of some large number of HAMT trees is no longer in use by any of them and can have its resources returned to the system? The current implementation uses reference counting to track this, and that approach brings with it the issues of thread safety. The constant building up of new structures, even with only a fraction of nodes involved, also appears to imply a heavier burden on memory allocation facilities. Both of these issues indicate that HAMT performance may benefit greatly from support by custom memory systems tailored to frequent allocations, and designed to reap memory no longer used without the need for (as much) programmer caretaking. Garbage collecting systems may be the right choice. The languages that have made most use of these data structures to date all have such memory systems in their foundations.

4 Implementation and Results

The fossil development branch `dgp-refactor` contains several prototype experiments on re-designs of Tcl internals. As of this writing, one of those experiments is the implementation of the HAMT data structure in the files `tclHAMT.h` and `tclHAMT.c`. There is also a file `tclHAMTObj.c` that defines a `Tcl_ObjType` for holding a HAMT structure as the internal representation of a dictionary value. It also includes an implementation of an ensemble command `hamt` that has a handful of subcommands in common with the `dict` command and is meant to be a near replacement of that command in terms of function. With this implementation we can explore that at least the key promises of the HAMT implementation are kept.

```
# Create 10,000 key-value pairs
set data [lmap _ [lrepeat 20000 {}] tcl::mathfunc::rand]

# Make a dict of them...
set d [dict create {*}$data]

# ... and tear it down without care about sharing.
time {foreach {k v} $data {set d [dict remove $d $k]}}
==> 23839420 microseconds per iteration

# Make a hamt of them
set h [hamt create {*}$data]

# ... and tear it down, No need to care about sharing!
time {foreach {k v} $data {set h [hamt remove $h $k]}}
==> 77113 microseconds per iteration

# Repeat dict teardown, but code artfully
set d [dict create {*}$data]
```

```
time {foreach {k v} $data {dict unset d $k}}
==> 28610 microseconds per iteration
```

As designed, the HAMT avoids the performance collapse when sharing drives the Copy On Write mechanism of incumbent dicts into spasms of copying. Note this example makes use of a hash map of only 10,000 key-value pairs. If we try to demonstrate with 100,000 pairs, the failing use of `dict remove` takes dozens of minutes! The HAMT teardown time increases in the predictable way, from about 80 ms to about 800 ms.

When the dict command is used properly according to sharing-aware training, `dict unset` can tear down the dict in under 30 ms, less than half the time of the `hamt remove` approach. For a first pass implementation to be about a factor of two of the performance of a released production level command seems like a promising beginning.

A second point of comparison where the HAMT comes out on top is the merge operation.

```
# Create 1,000,000 key-value pairs
set data [lmap _ [lrepeat 2000000 {}] tcl::mathfunc::rand]

# Make two dicts of 500,000 each...
set d1 [dict create {*}[lrange $data 0 999999]]
set d2 [dict create {*}[lrange $data 1000000 1999999]]

# ...and merge them together...
time {set d [dict merge $d1 $d2]}
==> 681783 microseconds per iteration

# ...and then with itself...
time {dict merge $d $d}
==> 1032838 microseconds per iteration

# ...and with a piece of itself.
time {dict merge $d $d1}
==> 927085 microseconds per iteration

# Repeat with the hamt command
set h1 [hamt create {*}[lrange $data 0 999999]]
set h2 [hamt create {*}[lrange $data 1000000 1999999]]
time {set h [hamt merge $h1 $h2]}
==> 294936 microseconds per iteration

time {hamt merge $h $h}
==> 65 microseconds per iteration

time {hamt merge $h $h1}
==> 218641 microseconds per iteration
```

5 Future Prospects

As of this writing, the HAMT implementation includes only the most necessary primitives. It supports hamt creation, insertion and removal of key-value pairs, fetching of a value corresponding to a key, merging hashmaps, and reporting hashmap size. It also supports a mechanism to iterate over all key-value pairs in the map, as was necessary to enable string generation from an arbitrary HAMT value. Finally a `hamt info` command produces a statistical summary of the structures making up the HAMT. These outputs comfirm that both the total memory required for a HAMT and a hash table are competitive, and the balance tips to HAMTs as we seek to keep a larger number of closely related hashmaps. The path length through a HAMT to a key-value pair is also competitive with the search path of Tcl hash tables.

In contrast with the `dict`, the `hamt` does not preserve storage order. The order of iterations through a HAMT is imposed by the sorting of hash values. For this reason, it is not on the path to be an exact replacement for `dict`.

As already mentioned, the performance of the HAMT implementation may be significantly improved by supporting it with the foundation of a more suitable memory allocation scheme. This work would also be key to enabling thread-sharing of HAMT values to bring the benefits of this work to improved Tcl concurrency capabilities.

The research and development into the HAMT and related structures has been reasonably active in recent years, and further variations and innovations are out there to adapt and adopt. One particular scheme is the use of transients. Although most HAMT operations continue to act on and act to produce strictly immutable values, transients are means for programmers to mark some HAMT values as ones that are known temporary in nature and known unshared. An example of this is the series of HAMT values we pass through in a `hamt create` command acting over several key-value pairs. In such a situation, the HAMT algorithms can be modified to perform a controlled set of overwrites that honors all commitments, but promises to incrementally improve performance.

6 Conclusions

Early experiments with a HAMT implementation successfully demonstrate its ability to provide basic dictionary functionality. By design as an immutable value, it avoids the severe pitfalls that arise when value sharing forces the incumbent dict implementation into its full Copy On Write operation. Even when artful scripting in Tcl avoids the pitfalls, the HAMT appears to achieve performance within a small factor, even in this first sketch implementation where known improvements have not yet been applied. Further pursuit of this approach appears to be worthwhile.