LC-trie implementation notes

Node types

leaf
: An end node with data. This has a copy of the relevant key, along with 'hlist' with routing table entries sorted by prefix length. See struct leaf and struct leaf_info.

trie node or tnode
: An internal node, holding an array of child (leaf or tnode) pointers, indexed through a subset of the key. See Level Compression.

A few concepts explained

Bits (tnode)

The number of bits in the key segment used for indexing into the child array - the "child index". See Level Compression.

Pos (tnode)

The position (in the key) of the key segment used for indexing into the child array. See Path Compression.

Path Compression / skipped bits

Any given tnode is linked to from the child array of its parent, using a segment of the key specified by the parent's "pos" and "bits" In certain cases, this tnode's own "pos" will not be immediately adjacent to the parent (pos+bits), but there will be some bits in the key skipped over because they represent a single path with no deviations. These "skipped bits" constitute Path Compression. Note that the search algorithm will simply skip over these bits when searching, making it necessary to save the keys in the leaves to verify that they actually do match the key we are searching for.

Level Compression / child arrays

the trie is kept level balanced moving, under certain conditions, the children of a full child (see "full_children") up one level, so that instead of a pure binary tree, each internal node ("tnode") may contain an arbitrarily large array of links to several children. Conversely, a tnode with a mostly empty child array (see empty_children) may be "halved", having some of its children moved downwards one level, in order to avoid ever-increasing child arrays.

empty_children

the number of positions in the child array of a given tnode that are NULL.

full_children

the number of children of a given tnode that aren't path compressed. (in other words, they aren't NULL or leaves and their "pos" is equal to this tnode's "pos"+"bits").

(The word "full" here is used more in the sense of "complete" than as the opposite of "empty", which might be a tad confusing.)

Comments

We have tried to keep the structure of the code as close to fib_hash as possible to allow verification and help up reviewing.

fib_find_node()
: A good start for understanding this code. This function implements a straightforward trie lookup.

fib_insert_node()
: Inserts a new leaf node in the trie. This is bit more complicated than fib_find_node(). Inserting a new node means we might have to run the level compression algorithm on part of the trie.

trie_leaf_remove()
: Looks up a key, deletes it and runs the level compression algorithm.

trie_rebalance()
: The key function for the dynamic trie after any change in the trie it is run to optimize and reorganize. It will walk the trie upwards towards the root from a given tnode, doing a resize() at each step to implement level compression.

resize()
: Analyzes a tnode and optimizes the child array size by either inflating or shrinking it repeatedly until it fulfills the criteria for optimal level compression. This part follows the original paper pretty closely and there may be some room for experimentation here.

inflate()
: Doubles the size of the child array within a tnode. Used by resize().

halve()
: Halves the size of the child array within a tnode - the inverse of inflate(). Used by resize();

fn_trie_insert(), fn_trie_delete(), fn_trie_select_default()
: The route manipulation functions. Should conform pretty closely to the corresponding functions in fib_hash.

fn_trie_flush()
: This walks the full trie (using nextleaf()) and searches for empty leaves which have to be removed.

fn_trie_dump()
: Dumps the routing table ordered by prefix length. This is somewhat slower than the corresponding fib_hash function, as we have to walk the entire trie for each prefix length. In comparison, fib_hash is organized as one "zone"/hash per prefix length.

Locking

fib_lock is used for an RW-lock in the same way that this is done in fib_hash. However, the functions are somewhat separated for other possible locking scenarios. It might conceivably be possible to run trie_rebalance via RCU to avoid read_lock in the fn_trie_lookup() function.

Main lookup mechanism

fn_trie_lookup() is the main lookup function.

The lookup is in its simplest form just like fib_find_node(). We descend the trie, key segment by key segment, until we find a leaf. check_leaf() does the fib_semantic_match in the leaf's sorted prefix hlist.

If we find a match, we are done.

If we don't find a match, we enter prefix matching mode. The prefix length, starting out at the same as the key length, is reduced one step at a time, and we backtrack upwards through the trie trying to find a longest matching prefix. The goal is always to reach a leaf and get a positive result from the fib_semantic_match mechanism.

Inside each tnode, the search for longest matching prefix consists of searching through the child array, chopping off (zeroing) the least significant "1" of the child index until we find a match or the child index consists of nothing but zeros.

At this point we backtrack (t->stats.backtrack++) up the trie, continuing to chop off part of the key in order to find the longest matching prefix.

At this point we will repeatedly descend subtries to look for a match, and there are some optimizations available that can provide us with "shortcuts" to avoid descending into dead ends. Look for "HL_OPTIMIZE" sections in the code.

To alleviate any doubts about the correctness of the route selection process, a new netlink operation has been added. Look for NETLINK_FIB_LOOKUP, which gives userland access to fib_lookup().