March 23, 2015

PARALLEL, SPACE-EFFICIENT HASH TABLE RESIZE

Geoffrey Pike

Justin Lebar

PARALLEL, SPACE-EFFICIENT HASH TABLE RESIZE

TECHNICAL FIELD

[0001] This disclosure generally relates to hash tables.

BACKGROUND

[0002] A hash table may be a data structure used to map keys to objects. A hash table may use one or more hash functions on keys to compute an index into an array of buckets. Hash tables may be re-sized as the number of keys mapped by the hash table increase or decrease. However, the process of re-sizing a hash table may use significantly more memory than used by the hash table itself.

SUMMARY

[0003] In general, an aspect of the subject matter described may involve a process for resizing a hash table. In some implementations, a hash table may be resized by incrementally de-allocating buckets of an old hash table and incrementally allocating buckets of a new hash table. In some implementations, a hash table may be resized by re-allocating buckets from the old hash table to the new hash table and then re-arranging the buckets of the new hash table. In some implementations, a hash table with chaining may be resized by copying the elements of the old hash table to corresponding buckets of the new hash table and indicating which elements are not necessarily in a final position. After copying, final positions may be determined for the buckets that are indicated as not necessarily in a final position.

DESCRIPTION OF DRAWINGS

[0004] FIG. 1 is an example process for resizing a hash table.

[0005] FIG. 2 is another example process for resizing a hash table.

[0006] FIG. 3 is an example process for resizing a hash table with chaining.

DETAILED DESCRIPTION

[0007] FIG. 1 is an example process 100 for resizing a hash table by incrementally de-allocating buckets of an old hash table and incrementally allocating buckets of a new hash table. The process may 100 include starting at the beginning bucket of an old hash table (110). For example, the process 100 may include starting at a bucket with an index zero, e.g., bucket [0], of a hash table with four buckets.

[0008] The process 100 may include determining if an element is stored in the bucket, and if so, trying to insert the element into a new hash table (120). For example, the process 100 may include determining that bucket [0] includes an element, and inserting the element into the new hash table. The process 100 may include, if the insertion fails, storing the element at the beginning of the old hash table in a bucket after any other elements that have failed to be inserted (130). For example, if inserting an element of bucket [2] fails, the element may be stored in bucket [0] if no other element has failed to be inserted.

[0009] The process 100 may include, de-allocating buckets of the old hash list including the current bucket to the bucket after the last bucket that stores an element that failed to be inserted (140). If no elements have failed to be inserted, then the buckets of the old hash list including the current bucket to the beginning bucket may be de-allocated. For example, if bucket [3] is the current bucket and bucket [0] stores the last unsuccessfully inserted element, buckets [1]-[3] may be de-allocated.

[0010] The process 100 may include iterating until all buckets are iterated through (150). The process 100 may include inserting elements stored at the front of the old hash table that were previously failed to be inserted in the new hash table (160).

[0011] FIG. 2 is an example process 200 for resizing a hash table by re-allocating buckets from the old hash table to the new hash table and then re-arranging the buckets of the new hash table. The process 200 may include re-allocating buckets of an old hash table to a new hash table and including a paired duplicate bucket following each bucket in the new hash table from the old hash table (210). For example, if an old hash table includes buckets [0]-[3], those buckets may be re-allocated to buckets [0], [2], [4], and [6], respectively, of a new hash table. Bucket [1] may be duplicate from bucket [0], bucket [3] may be duplicated from bucket [2], etc.

[0012] The process 200 may include starting at a beginning bucket of the new hash table (220). The process 200 may include determining if elements of the current bucket and a following bucket are the same, and if so, inserting the element into the new hash table and emptying the bucket and the following bucket (230). For example, the process 200 may include determining if bucket [0] and bucket [1] include the same element, and if so, inserting the element into the new hash list. In the example, bucket [0] and [1] may be set to empty after the insertion of the element into bucket [4].

[0013] The process 200 may include determining if the element was inserted into a bucket that had the same element stored in that bucket's paired bucket, and if so, emptying the paired bucket (240). For example, the process 200 may include determining if bucket [4] previously stored the same element as bucket [5], and if so, emptying bucket [5]. In another example, if an element is inserted into bucket [3], and bucket [2] stored a different element than bucket [3] had stored, then bucket [2] may not be emptied. The process 200 may include iterating (230) and (240) sequentially through the buckets of the new hash table until (230) and (240) are performed for all the remaining even numbered buckets (250).

[0014] FIG. 3 is an example process 300 for resizing a hash table with chaining may be resized by copying the elements of the old hash table to corresponding buckets of the new hash table and indicating which elements are not necessarily in a final position. The process 300 may include starting at a beginning bucket of an old hash table (310). For example, the process 300 may start with bucket [0] of the old hash table with four buckets. The process 300 may include determining if the bucket includes an element, and if the bucket includes an element, storing the element in a corresponding bucket of a new hash table (320). For example, the process 300 may determine that bucket [1] of the old hash table includes an element, and store the element in bucket [1] of a new hash table that includes eight buckets. The process 300 may include indicating that the element stored in the new hash table is not necessarily in a final position (330). For example, the process 300 may include setting a next value of the element to point to itself. The process 300 may include de-allocating buckets of the old hash table (340). For example, the current bucket of the old hash table may be de-allocated. The process 300 may include iterating through buckets of the old hash table (350). For example, (320)-(340) may be sequentially repeated for each bucket of the old hash table. The process 300 may include for each element of the new hash table that is indicated as not necessarily in a final position, determining a final position for the element (360).

[0015] More details of implementation are described in the following, where src may refer to a source hash table and dst may refer to a destination hash table with a different number of buckets. A process similar to process 100 shown in FIG. 1 may be implemented with the following, Algorithm 1:

```
        for (i = 0, j = 0; i < B; i++)
                if (src[i] contains a key) {
                        if (!TryInsert(src, i, dst)) {
                                If touching src[j] might fail because told the
                        OS didn't need that memory, tell the OS need it.
                                src[j++] = src[i];}}
                (Optional) Tell the OS that memory between src[j] and src[i] is not
        needed }
        while (j>0) {
                Insert(dst, src[j]);
                (Optional) Occasionally tell the OS that memory between src[j]
        and the end of src is not needed}
Subroutine Insert:
        This can be a hashtable insertion algorithm.
Subroutine TryInsert:
        Try to insert src[i] into dst using the same algorithm as Insert, but do access
        elements of dst outside of some set, S. If successful, return true. Otherwise,
        return false.
```

[0016] Variation V1 - Permute src before starting one of the above algorithms. For example, resize from B to 1.99B buckets, with a probing strategy of open addressing with linear or quadratic probing. For example, first permute src so that items likely to land close together in dst are close together in src. For example, compute e(k) for each key and sort by that, with e(k) = floor((16/1.99B) * (likeliest destination index in [0, 1.99B] for k)). If the sort causes long runs of empty buckets, may tell the OS that won't access those empty buckets again.

[0017] Variation V2 - Intermix moving elements around in src and moving elements from src to dst. For example, during the sort step of V1, if some key's e(k) is less than 5, then immediately move it to dst, or move it to dst if TryInsert() can succeed without touching any destination page that hasn't yet been touched. Or, rather than doing a sort, opportunistically swap elements of src with empty buckets in src that are found by probing near their estimated to be optimal spot.

[0018] A process similar to process 200 shown in FIG. 2 may be implemented with the following, Algorithm 2: Rearrange data in the destination array. In some cases, the data structure may be able to increase the size of the array of buckets without copying any data, e.g., via reallocation. Even if that isn't possible, elements may be copied to the destination array without regard to placing them in their correct positions, and then rearranged so that all the data structure's invariants are met. This may be performed with an additional bookkeeping data structure, e.g., a set implemented as a bit vector that keeps track of which items in the destination array needn't be moved because they're known to be in a reasonable position. Alternatively, memory of the destination array may be used only. For example:

```
for (j = 0, i = 0; i < B; i++)
        if (src[i] contains a key) {
                dst[j] = src[i]; dst[j+1] = src[i]; j += 2;}
        if ((i+1) mod some constant is 0) tell the OS that memory from
src[0] through src[i] is no longer needed}
src is no longer needed
for (i = 0; i < j; i += 2) Rearrange(i);
```

```
Subroutine Rearrange:
        if dst[i] != dst[i+1] return;
        tmp = dst[i]; dst[i] = empty; dst[i+1] = empty
        while (true) {
                        Insert tmp using the hashtable's normal insertion routine, except
        that buckets, b, with dst[b] == dst[b xor 1], are treated as empty
                        If that insertion was to a bucket b that didn't have dst[b] == dst[b
        xor 1] then return
                        tmp = dst[b xor 1]; dst[b xor 1] = empty;}
```

[0019] A process similar to process 300 shown in FIG. 3 may be implemented with the following:

```
for (i = 0; i < B; i++)
        Node *n = src[i];
        while (n != NULL) {
                dst[j] = n; n = n->next; j++;
                dst[j]->next = dst[j];} // oneNode circular list, to indicate
        "not necessarily in final position"
        if ((i+1) mod some constant is 0) tell the OS that memory from
src[0] through src[i] is no longer needed}
src is no longer needed
for (i = 0; i < j; i++) if (dst[i] != NULL && dst[i]>next ==dst[i]) ...etc.
```

[0020] Additionally or alternatively, parallelizing algorithms may be applied. For example, transactional memory or mutexes may be used. In the latter case, a constant Q may be chosen and have a mutex for chunks of Q elements in src or dst, with the code arranged to take locks in lowest to highest order in cases where multiple locks may be held simultaneously. A parallel implementation of Algorithm 1 with Variation V1 or V2 may provide advantages. For example, with a parallel permutation step, and worker threads each responsible for inserting in dst all elements in some part of the permuted src array, contention may be reduced because the elements handled by a given worker may likely be moving to destinations near each other.

ABSTRACT OF THE DISCLOSURE

Methods, systems, and apparatus, including computer programs encoded on a computer storage medium, for parallel, space-efficient hash table resize. An aspect may include a hash table that may be resized by incrementally de-allocating buckets of an old hash table and incrementally allocating buckets of a new hash table. Additionally or alternatively, an aspect may include a hash table that may be resized by re-allocating buckets from the old hash table to the new hash table and then re-arranging the buckets of the new hash table. Additionally or alternatively, an aspect may include a hash table with chaining that may be resized by copying the elements of the old hash table to corresponding buckets of the new hash table and indicating which elements are not necessarily in a final position. After copying, final positions may be determined for the buckets that are indicated as not necessarily in a final position. Additionally or alternatively, an aspect may include parallezing algorithms for resizing hash tables.

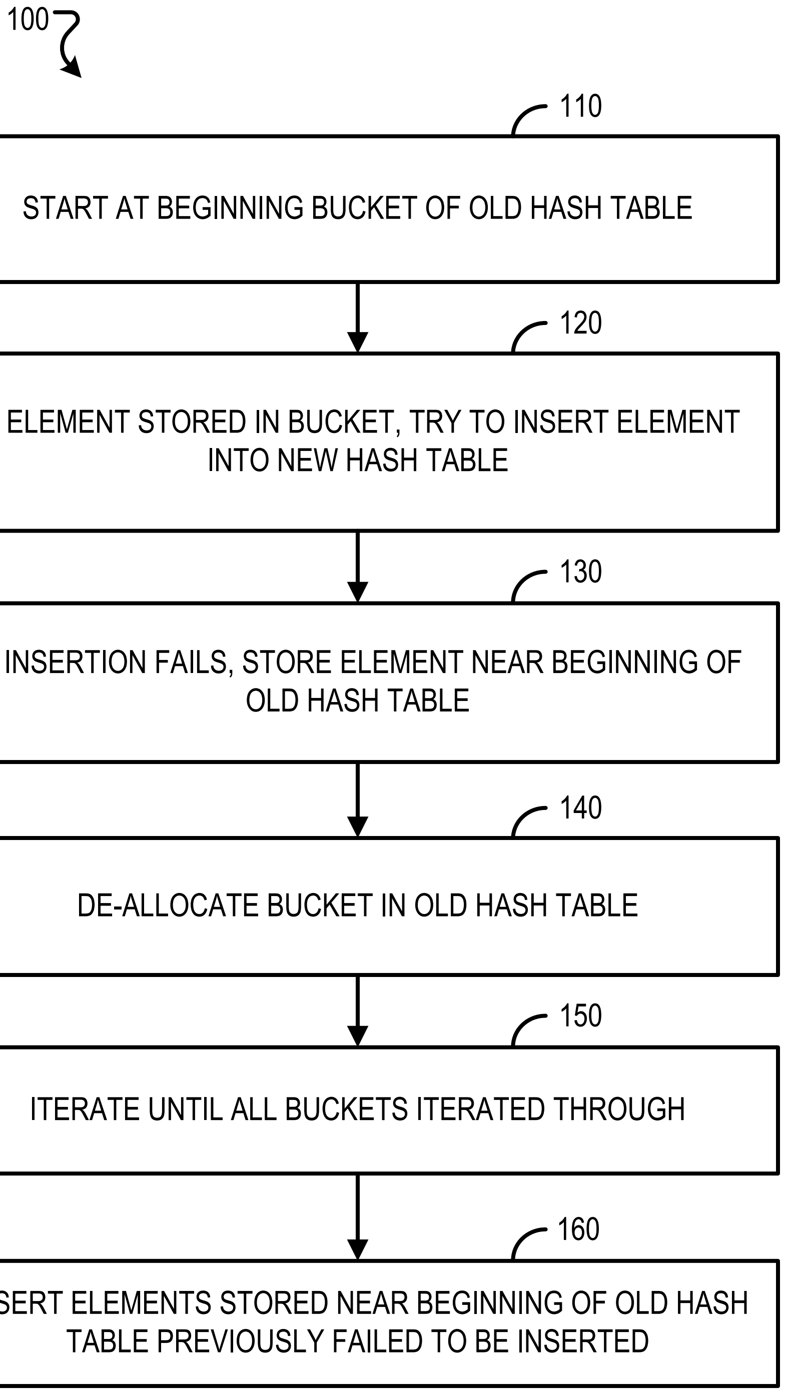
100
110
START AT BEGINNING BUCKET OF OLD HASH TABLE
120
IF ELEMENT STORED IN BUCKET, TRY TO INSERT ELEMENT INTO NEW HASH TABLE
130
IF INSERTION FAILS, STORE ELEMENT NEAR BEGINNING OF OLD HASH TABLE
140
DE-ALLOCATE BUCKET IN OLD HASH TABLE
150
ITERATE UNTIL ALL BUCKETS ITERATED THROUGH
160
INSERT ELEMENTS STORED NEAR BEGINNING OF OLD HASH TABLE PREVIOUSLY FAILED TO BE INSERTED

FIG. 1

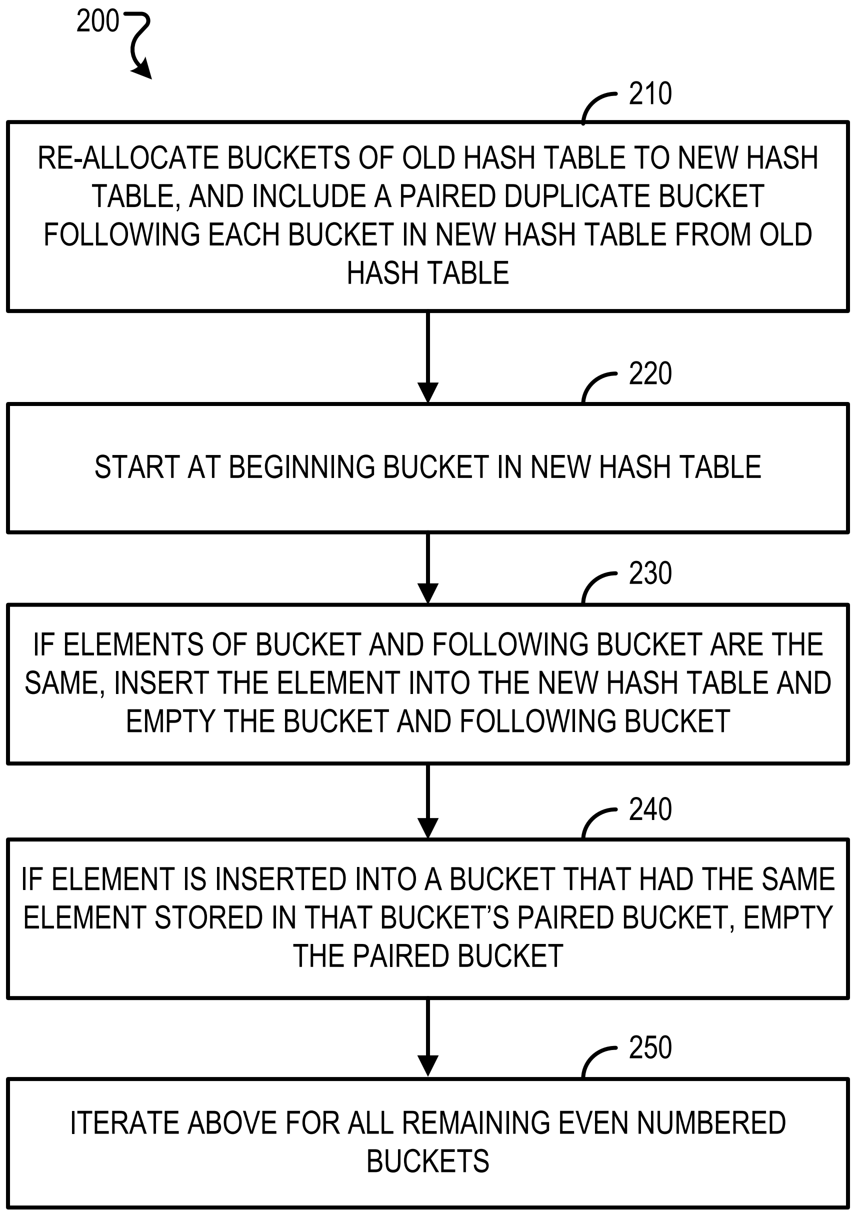
200
210
RE-ALLOCATE BUCKETS OF OLD HASH TABLE TO NEW HASH TABLE, AND INCLUDE A PAIRED DUPLICATE BUCKET FOLLOWING EACH BUCKET IN NEW HASH TABLE FROM OLD HASH TABLE
220
START AT BEGINNING BUCKET IN NEW HASH TABLE
230
IF ELEMENTS OF BUCKET AND FOLLOWING BUCKET ARE THE SAME, INSERT THE ELEMENT INTO THE NEW HASH TABLE AND EMPTY THE BUCKET AND FOLLOWING BUCKET
240
IF ELEMENT IS INSERTED INTO A BUCKET THAT HAD THE SAME ELEMENT STORED IN THAT BUCKET'S PAIRED BUCKET, EMPTY THE PAIRED BUCKET
250
ITERATE ABOVE FOR ALL REMAINING EVEN NUMBERED BUCKETS

FIG. 2

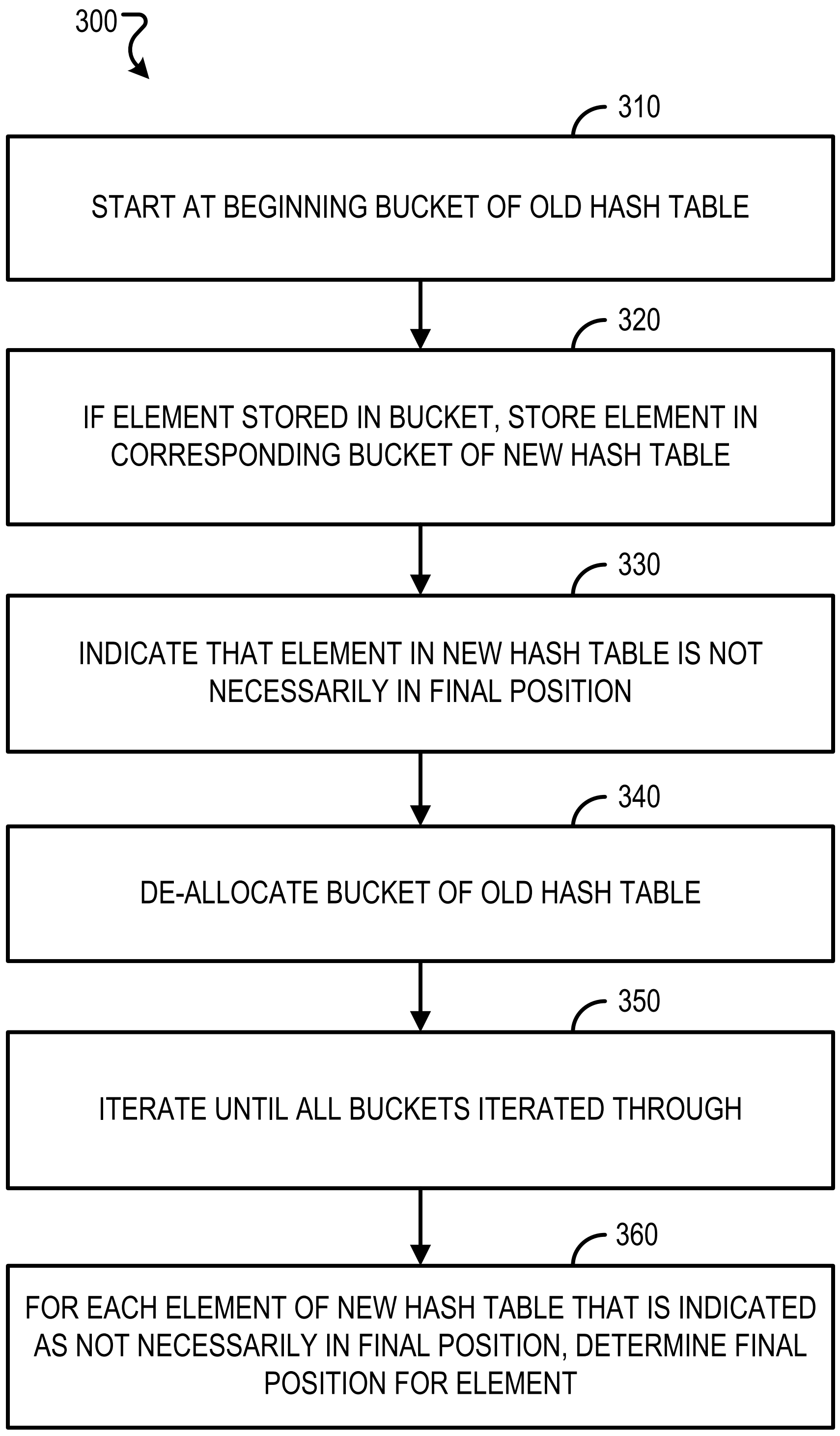
300
310
START AT BEGINNING BUCKET OF OLD HASH TABLE
320
IF ELEMENT STORED IN BUCKET, STORE ELEMENT IN CORRESPONDING BUCKET OF NEW HASH TABLE
330
INDICATE THAT ELEMENT IN NEW HASH TABLE IS NOT NECESSARILY IN FINAL POSITION
340
DE-ALLOCATE BUCKET OF OLD HASH TABLE
350
ITERATE UNTIL ALL BUCKETS ITERATED THROUGH
360
FOR EACH ELEMENT OF NEW HASH TABLE THAT IS INDICATED AS NOT NECESSARILY IN FINAL POSITION, DETERMINE FINAL POSITION FOR ELEMENT

FIG. 3