

Incremental Reorganization of Open Hash Tables

Dale Parson (dparson@agere.com)

Agere Systems

Abstract

Hybrid open hashing is a composite algorithm that combines the key-based retrieval properties of open addressing hash tables with the temporal responsiveness of incremental copy garbage collection.¹ Hybrid open hashing's table reorganization algorithm alternates between incremental construction of a new table via selective copying of entries, and incremental cleaning of an aged table via emptying of entries. Hybrid open hashing limits worst-case reaction time to an event requiring key-based retrieval, such as state-driven address matching in network packet processing. Emphasis is on predictable retrieval time. This report investigates the drawbacks of conventional hash table algorithms, the properties of open address hashing and incremental garbage collection that the hybrid algorithm combines, performance simulation, and refinements of the basic incremental reorganization algorithm.

Keywords: database, embedded system, garbage collection, hash table, network processor, real-time system

1. Open Hashing and Garbage Collection

Key-based access to information is essential for many information processing systems including real-time, embedded systems. Application data supply identifying fields that serve as keys for storing, retrieving and updating associated data. In many embedded systems key manipulation and access to associated data must occur within a hard real-time limit.

The width of an application key can be reduced by a hashing algorithm that reorders and combines bits of the key into an index into a hash table that stores key-associated data [1]. Hash tables are among the most common data structures used for key-based retrieval. Unfortunately, conventional hash tables suffer from performance degradation over time that limits their usefulness in real-time systems.

1. The hybrid open addressing hash table mechanism reported in this paper, including performance enhancements based on limiting the number of garbage collection calls, has been submitted to the U.S. Patent Office in a patent application by Agere Systems Inc. and the author.

An open hash table maintains a fixed number of entries at each index (a.k.a. bucket), in the limit 0 or 1 entry, with size depending on the amount of memory per bucket. A single hashing step maps a key to an index, but if the key is not found there, one or more rehash steps are taken in order to inspect additional buckets. Searching terminates at a bucket with finding the key, or with an empty entry, or when the table has been searched exhaustively. With an effective, randomizing hash function and a relatively sparse table (e.g., < 50% density), open hashing can map a typical key to its dependent values in as little as one memory reference. In contrast, chained hashing, which keeps a linked list of colliding entries in each bucket, entails multiple memory references to access and manipulate the linked list at a bucket as well as the free list of unused list entries. Embedded systems that need to minimize memory latency benefit from using an open rather than a chained hash table.

Unfortunately, performance of open hash tables suffers after many insertions and deletions. Each bucket has a status of either *empty* (initial state), *used* (containing a key mapping), or *deleted*. It is necessary to distinguish the empty and deleted states because search for a key terminates upon arrival at an empty bucket. If a collision causes a search to continue past bucket B_N for insertion of new key K , and B_N is later marked as empty upon deletion of its key, subsequent searches for K would terminate prematurely at the empty B_N . In contrast, when deletion marks B_N as deleted, subsequent searches for K proceed past B_N . While sound, the problem with this approach is that after many insertions and deletions, a large number of buckets may be marked as deleted, incurring search steps through buckets that do not contain key mappings. What is needed is a way to periodically reorganize a table to minimize the number of buckets tagged as deleted.

A brute force approach for non-real-time applications is to read all used entries from an aging table periodically and hash their keys into a new table consisting, initially, of empty entries. The algorithms of Knuth [1] and Szymanski [2] improve on this brute force approach by allowing table reorganization within the space of one existing table, but all of them suffer from the same defect with respect to real time applications. They reconstruct the remaining valid keys into a new table

organization monolithically — the entire table must be reconstructed within one operation before it can be used — incurring a large execution time penalty during table reorganization. This penalty causes real-time, reactive applications that rely on a hash table to miss their deadlines.

The hybrid algorithm presented here avoids this monolithic reconstruction of a hash table by continually building a new hash table as it copies used entries from an aging hash table into the new hash table, skipping over deleted and empty entries. This incremental approach to reorganization is inspired by incremental copy garbage collection [3]. The hybrid algorithm maintains two hash tables, with the *current table* being the table receiving new insertions from Put, and the *alternate table* being the aging table from which garbage collection filters out deleted entries. Table reorganization proceeds in two phases. During the *copy phase*, both tables may contain valid keys. In copy phase, Get searches the current table for a key and, if it does not find the key, Get searches the alternate table; likewise Put searches both tables before inserting new entries in the current table; Remove deletes its key from both tables. The garbage collector is invoked at the end of Get, Put and Remove to perform one table reorganization step; the garbage collector advances an index variable *cleanix* through the alternate table, one entry per invocation of the garbage collector. In the

copy phase, when the garbage collector finds a used entry (i.e., a valid key) in the alternate table, it puts that key into the current table as a hashed insertion. Eventually, *cleanix* advances to the end of the alternate table, and the garbage collector moves into the *clean phase*, resetting *cleanix* to the start of the alternate table. At this point the garbage collector has copied all keys from the alternate table to the current table, and all new Put insertions are going into the current table.

During the clean phase, each call to the garbage collector sets all entries in one bucket within the alternate table to empty. The alternate table is not consulted by Get, Put or Remove during the clean phase. At the conclusion of the clean phase, when *cleanix* advances to the end of the alternate table, the garbage collector returns into the copy phase, resetting *cleanix* to the start of the alternate table. When returning to copy phase, the garbage collector reverses the roles of the current and alternate tables, so that the previous alternate table (which is now completely empty) becomes the current table (for new insertions), and the previous current table now becomes the alternate table to be filtered for deleted entries by having its used entries copied. Table 1 summarizes the actions of Get, Put and Remove during the two phases of the algorithm. Incremental table reorganization could also be invoked from other functions in a system, e.g., by a background thread that runs during lulls in real-time activity.

Table 1: Summary of hash table operations in the hybrid algorithm

	Get	Put	Remove
copy phase	Retrieve from either table.	Retrieve from either table, insert into current table if not found	Delete from both tables.
	Garbage collector walks through alternate table, a step at a time, copying used entries into the current table via hashing. When it reaches alternate’s end, it changes to the clean phase.		
clean phase	Retrieve from current table.	Retrieve from current table, insert into current table if not found.	Delete from current table.
	Garbage collector walks through alternate table, a step at a time, converting all entries to empty. When it reaches alternate’s end, it changes to the copy phase, and reverses the roles of the tables (a “flip”). The new current table is empty; copying begins from the populated alternate table.		

Given the complexity that garbage collection adds to open hashing, there is a chance that the performance costs outweigh the benefits. Rather than go through complete implementations of assorted algorithm

variations for performance evaluation, this project’s means for comparing performance of open hash tables, chained hash tables, hybrid open tables, variations of hybrid tables, and assorted hashing and rehashing

strategies is by running representative application data through a series of related Java classes that implement an abstract Java interface *hashtablei*. This interface specifies table operations Get, Put and Remove, along with some adjunct operations and a set of measurement operations. Performance is characterized in terms of several concrete Java classes that implement hash table management strategies. Class *hashtableplain* implements standard open hashing without any table reorganization, and class *hashtablemono* adds reorganization by building a new hash table in one monolithic step when the number of deleted entries in the current table exceeds a threshold specified on the command line. *Hashtablehybrid* uses the hybrid open hashing - incremental garbage collection algorithm discussed above. Other hash table variants are discussed shortly.

Table 2 shows results from a series of tests, hashing realistic Internet addresses and port numbers to hash table entries, using *hashtableplain*, *hashtablemono*, and *hashtablehybrid*. *Hashtablemono* builds a new table when a threshold of 5632 deleted entries is reached for this sample data set; the effectiveness of this threshold for the data was determined empirically. In these tests, the table can hold up to 16384 entries, with 8 entries per bucket, giving 2048 buckets. A probe count in these measurements corresponds to the number of buckets that hashing must inspect or modify to achieve one sample Put or Get or Remove operation for network traffic. Sample data was constructed so that after 8000 operations, the number of Remove operations balances the number of Put operations, eliminating the possibility of filling the table with used entries. It is trivial to see that if the Put rate consistently exceeds the Remove rate, any table must eventually run out of room.

Table 2: 2,000,000 operations, limit of 8000 used entries before Remove balances Put

algorithm	Max probes	Min probes	Average probes	Std. deviation
plain	2048	1	89.1208805	284.3068
monolithic	18162	1	1.746935	60.18176
hybrid	15	2	3.4318165	1.187051
timeout	8	2	3.2463965	1.032094
dynamic 3-4	6	2	3.245097	1.030479
dynamic 1-2	6	2	2.496241	0.5020706
adaptive	6	2	2.496241	0.5020706

The *plain* approach with no table reorganization degrades when the number of deleted entries exceeds empty entries after many deletions, because inspection of deleted entries comes to dominate search time. The *monolithic* approach reduces the average expense of a hash table operation by periodically building a new table without deleted entries, but the worst case time of a table operation, represented here by “max probes,” skyrockets because reorganization-triggering table operations must await table reorganization. Each table-accessing step in table reorganization is a “probe.”

The *hybrid* approach of incremental table reorganization shows a 2X average probe count increase over monolithic because each hash operation incurs additional table-reorganization probes, but the worst-case operation probe count drops to 15 by avoiding monolithic reorganization.

The subsequent rows of Table 2 are refinements of the

hybrid reorganization algorithm. Timeout replaces an explicit Remove operation with removal of an entry during the copy phase if the entry has not been accessed within some decay period. Timeout is appropriate for applications such as network processing that delete mappings that have not been used within some interval. It improves performance simply by eliminating the steps of explicit deletion; its work is integrated into the copy phase by copy avoidance of outdated entries.

A minor flaw in the basic hybrid algorithm is that it taxes every hash table operation with reorganization work, regardless of whether the operation itself entailed a high probe count. The *dynamic* approach refines the hybrid approach by applying a fixed threshold such that only operations that have not exceeded the threshold are taxed with a table reorganization step. Expensive operations avoid the tax, while cheap operations perform incremental table reorganization, lowering the

maximum probe count further. Dynamic 3-4 uses a threshold of three operation probes for copy phase and four application probes for clean phase. The clean phase threshold is higher because a clean step involves less additional overhead than a copy step. Dynamic 1-2 lowers the copy-clean thresholds to 1 and 2 respectively. Note that for this sample data set, the worst case probe count drops from 8 for timeout to 6 for dynamic 3-4, while the average count drops further for dynamic 1-2.

Two problems with the fixed thresholds of the dynamic approach are that a threshold may be too high or too low for a given sequence of keys. If the threshold is too high, operations that need not be taxed with reorganization work are taxed, increasing max probe count. If the threshold is too low, reorganization occurs too infrequently, and average probe count increases because of accumulation of deleted entries. Degenerate data can shut off reorganization entirely by exceeding fixed thresholds, causing average performance to degrade to that of plain. To avoid these problems, *adaptive* samples the probe count of a sequence of table operations within a temporal window. At the end of the window it adjusts reorganization thresholds based on sampled probe counts, and then begins counting operation probes within the next temporal window. The adaptive approach avoids the problems of fixed thresholds by adapting to the key-probe count relationships that actually occur at run time. Table 2 shows that adaptive matches dynamic 1-2 for this test data. It is easy to create test data for which dynamic 1-2 never reorganizes the table because of high operation probe counts (low fixed thresholds); adaptive continues to perform reorganization in such cases.

2. Related Work

Knuth [1] gives an algorithm for in-place reorganization of an open hash table to remove deleted entries that requires linear probing for collision resolution. Szymanski [2] improves on Knuth by removing the requirement for linear probing, but both approaches are monolithic in their table reconstruction. They avoid the memory cost of maintaining two tables during reconstruction, but they do not avoid the worst-case table access time encountered during reconstruction.

The work of Friedman, et. al. on hash tables for embedded and real-time systems is the only other work to the author's knowledge that takes an incremental approach to hash table reorganization that is similar to incremental garbage collection [4]. The Friedman approach applies to chained hash tables, which do not suffer from the deleted entry crossing problem of open hash tables. Instead, Friedman's work concentrates on solving performance problems that arise when a chained hash table becomes too heavily loaded, resulting in many collisions and resulting linear searches through linked lists. The conventional solution is to build a new, bigger chained hash table monolithically, using keys scanned from the overloaded hash table for insertion into the new hash table. Friedman's approach incrementally inserts entries from existing chains in a smaller table into a bigger table whenever table growth is occurring. Like the hybrid open hashing algorithm of this report, this chained approach copies entries as an adjunct activity of Get, Put, or Remove processing. Friedman avoids creating the memory fragmentation and storage allocation time spikes that come with new table creation and old table deletion by preallocating a storage area in which to house the hash tables. Friedman, et. al state that open hashing is an alternative to chained hashing but that it is not well suited to real-time embedded system applications. Clearly the use of hybrid open hashing with garbage collection, including the measurements given in this paper, stand as a counterexample.

3. References

1. D. Knuth, *The Art of Computer Programming*, Volume 3, Searching and Sorting. Reading, MA: Addison-Wesley, 1973.
2. T. Szymanski, "Hash Table Reorganization." *Journal of Algorithms* 6(3), 1985, pp. 322-335.
3. R. Jones, *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. Chichester, England: John Wiley & Sons, 1999.
4. S. Friedman, N. Leidenfrost, B. Brodie and R. Cytron, "Hashtables for Embedded and Real-Time Systems." *Proceedings of IEEE Real-time Embedded System Workshop*, Dec. 3, 2001.