LHlf: Lock-free Linear Hashing (poster paper)

Donghui Zhang
Microsoft Jim Gray Systems Lab
634 W Main St, 4th floor, Madison, WI 53703
dozhan@microsoft.com

Per-Åke Larson
Microsoft Research
One Microsoft Way, Redmond, WA 98052
palarson@microsoft.com

Abstract

LHlf is a new hash table designed to allow very high levels of concurrency. The table is lock free and grows and shrinks automatically according to the number of items in the table. Insertions, lookups and deletions are never blocked. LHlf is based on linear hashing but adopts recursive split-ordering of the items within a bucket to be able to split and merge lists in a lock free manner. LHlf is as fast as the best previous lock-free design and in addition it offers stable performance, uses less space, and supports both expansions and contractions.

Categories and Subject Descriptors E.1 [**Data Structures**]: distributed Data Structures.

General Terms Algorithms, Performance, Design.

Keywords lock-free; hash table; linear hashing; split-order.

1. Introduction

LHlf (Linear Hashing, lock-free) is a new lock-free hash table design. LHlf automatically grows and shrinks the table according to the load; previous lock-free designs have only supported expansion. LHlf is space efficient; in a quiescent state, its representation in memory is essentially the same as that of a sequential hash table. Finally, LHlf's performance is stable because the work of resizing the table is spread over threads and over time.

LHlf is based on linear hashing (LH) which was originally designed for hash files by Litwin [5] and adapted for main-memory hash tables by Larson [4]. LHlf implements dynamic arrays using segments of exponentially increasing size as proposed by Griswold and Townsend [1].

LH gradually expands a hash table by splitting buckets. During a split, on average, half of the items in the bucket are relocated to a new bucket. We adopt an idea by Shalev and Shavit [8] and order the items in a bucket in recursive-split order. This allows a bucket to be split in a lock-free manner; all that is required is to identify the split point and cut the linked list in two.

Table 1: Feature comparison of four hash table designs.

	Fixed size, lock free	LH with locks	RSO [8]	LHlf
Efficient in time	Not always	May block	X	X
Expandable		X	X	X
Contractible		X		X
Scales to a large number of threads	X		X	X
Efficient in space	Not always	X		X
Stable performance	X	X		X

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
ACM 978-1-4503-1160-1/12/02.

Table 1 summarizes six desired features of a hash table and compares four hash table designs. A fixed-size hash table is very efficient when correctly sized but if it is undersized, performance suffers; if it is oversized, space is wasted. LH with locking does not scale to a very large number of threads. Shalev's and Shavit's design (RSO) [8] does not support table contraction, requires extra space for dummy nodes, and its performance is not stable. Only LHlf has all six desired features.

2. Main Ideas

Figure 1 shows an example of an LHlf hash table. The table is essentially a variable-size array of pointers, which is implemented by a two level structure: a small directory and a set of array segments. Each segment is an array of bucket pointers where a bucket pointer is the head of a single linked list of data nodes.

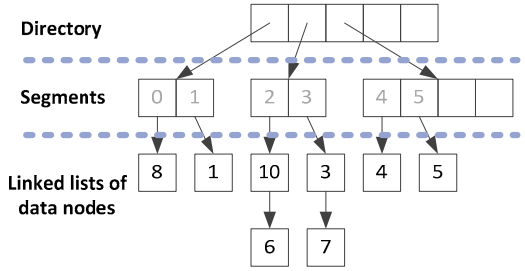

Figure 1. An example of an LHlf with six buckets.

2.1 Dynamic Array Implementation

The original version of linear hashing used array segments of fixed size. This has the drawback that for a very large table the directory also becomes very large. We want to use a small, fixed size directory for LHlf that fits entirely in an L1 cache. We adopted the solution proposed by Griswold and Townsend [1] of using segments of exponentially increasing size. In Figure 1, the second segment is 1X larger than the first segment; the third segment is 2X larger; the fourth segment will be 4X larger; and so on. We modified this idea slightly by allowing the table to have an upper size limit on a single segment.

2.2 Parallel Splitting and Merging

The original version of linear hashing expands or contracts the table by splitting or merging one bucket at a time, which means that the table cannot expand fast enough under very high insertion rates.

To avoid this limitation LHlf performs expansions and contractions in larger steps (multiple buckets at a time) and allows multiple threads to split (or merge) different buckets concurrently. When a thread detects that the table needs to be expanded, all it does is increment the table size by a fixed amount, set the table state to Splitting and the new buckets to Uninitialized. Any thread that hits an Uninitialized bucket first performs a bucket split. Similarly, any thread can contribute by splitting a bucket, allowing the splitting of a group of buckets to proceed in parallel.

2.3 Recursive-Split-Ordered Lists

The original version of linear hashing did not impose any ordering of the items on a list so splitting a bucket required a complete scan of its linked list. LHlf keeps the lists sorted in recursive-split order [8], that is, in increasing order of the reverse-bit values of the hash keys. This ordering makes it possible to split a bucket by cutting the list into two pieces and connecting the tail piece to the new bucket.

2.4 Atomic Operations Required

LHlf relies only on 64-bit and 128-bit atomic load, store and compare-and-swap (CAS) instructions to achieve consistency. Our actual implementation uses the machine-independent interlocked-compare-exchange family of functions available on Windows, such as *InterlockedCompareExchange128*, which are compiled into the appropriate assembly instruction.

2.5 Hazard Pointers

LHlf adopts Michael's hazard pointer scheme [7] to protect from memory access violations. Each thread uses up to five hazard pointers, two protecting segments and three protecting nodes. Note that other memory management schemes are possible, e.g. the Repeat Offender Problem and its solutions [3].

2.6 State Transitions and Helping

Like other lock free data structures, threads in LHlf help each other. LHlf breaks an operation into smaller atomic steps and carefully tracks what steps have been completed. Any thread can attempt the next atomic step. If it succeeds, it records that the step has been completed by atomically changing the state of the target object of the operation.

LHlf has state machines for four types of objects: hash table, segment, bucket, and node. Every state machine has a Normal state. When the data structure is quiescent, it looks just like a normal hash table in main memory. When the data structure is being updated, the threads do not immediately follow pointers that are not Normal; instead, they first help to fix up the state of the data structure to make the pointer Normal again.

3. Performance Evaluation

We experimentally evaluated four different hash table designs.

LHlf: lock-free linear hashing.

RSO: the dynamic lock-free hash table [8] that stores RSO values in every node and uses dummy nodes.

Fixed[8M]: a fixed-size lock-free hash table [2][6] with 8*1024*1024 buckets.

LH_lock: lock-based linear hashing using 1M spinlocks to protect buckets and one spinlock to protect expansions and contractions.

The experimental data consists of 480M distinct string keys with average length 12 bytes. 30M records were first loaded into the hash table and then a mixed workload (90% search, 6% insertions, and 4% deletions) was run for one second and the total number of operations counted. The experiments were performed on an HP ProLiant DL980 machine with eight Intel Xeon X7560 @ 2.27GHz Nehalem processors with a total of 64 cores and 128 hardware threads.

Figure 2 plots the throughput as the number of threads increases from 1 to 128. Average bucket size was 4 records. LH_Lock has the best performance when the number of threads is small but it does not scale to a large number of threads due to lock contention. Among the lock-free versions Fixed[8M] has the best performance, because it does not have to check for and handle concurrent table expansions and contractions, but the others are close.

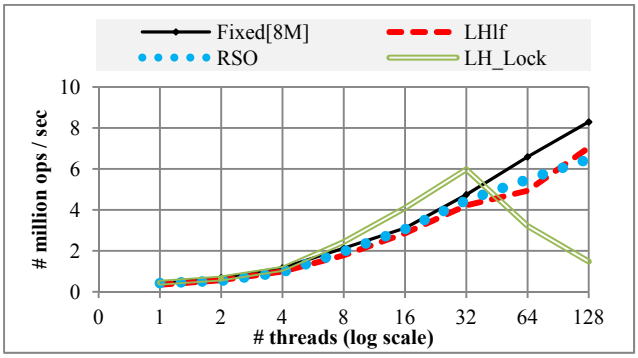

Figure 2: Varying #threads.

Figure 3 compares the performance when varying the number of pre-inserted records. Here the number of threads is 128, and the dynamic hash schemes expand the table when the average bucket size exceeds 4. The performance of Fixed[8M] drops as the number of records increases because the average chain length increases. The performance of RSO fluctuates significantly and in a cyclic manner with the length of each cycle doubling. The reason for this phenomenon is that, right after each directory doubling, half of the newly added buckets are uninitialized and accessing one of these buckets leads to an expensive bucket split. LHlf has much more stable performance because expansions are smooth and gradual.

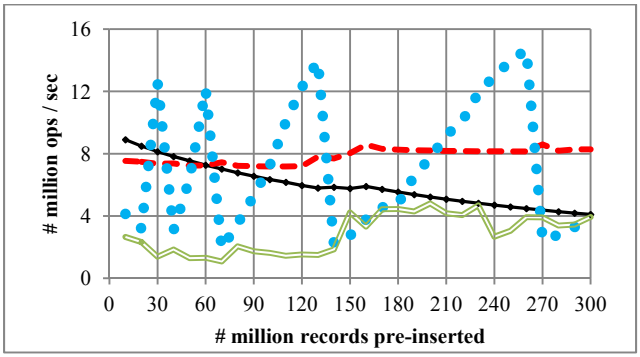

Figure 3: Varying amount of pre-insertions.

References

[1] Griswold, W. G. and Townsend, G. M. 1993. The Design and Implementation of Dynamic Hashing for Sets and Tables in Icon". *Software - Practice and Experience*, 23(4):351-67.

[2] Harris, T. L. 2001. A Pragmatic Implementation of Non-Blocking Linked-Lists. *DISC '01*: 300-314.

[3] Herlihy, M., Luchangco, V., and Moir, M. 2002. The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures. *DISC '02*: 339-353.

[4] Larson, P.-Å. 1988. Dynamic Hash Tables, *CACM* 31(4): 446–457.

[5] Litwin, W. 1980. Linear Hashing: A New Tool for File and Table Addressing, *VLDB '80*: 212–223.

[6] Michael, M. M. 2002. High Performance Dynamic Lock-free Hash Tables and List-based Sets. *SPAA '02*: 73-82.

[7] Michael, M. M. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE TPDS* 15(6): 491-504.

[8] Shalev, O. and Shavit, N. 2006. Split-ordered Lists: Lock-free Extensible Hash Tables. *JACM* 53(3): 379-405.