LKRhash: Scalable Hash Tables

Per-Åke Larson, Murali Krishnan, and George V. Reilly
Microsoft, One Microsoft Way, Redmond, WA 98052
palarson@microsoft.com, muralik@microsoft.com, georgere@microsoft.com

Abstract

LKRhash is a hash table implementation designed to allow high levels of concurrent operations and to run fast on contemporary processors that rely heavily on caching. It is based on linear hashing so the table can grow (and shrink) from a few elements to millions, always providing constant expected lookup, insertion, and deletion times. Compared with a standard implementation, the design reduces the number of cache and page misses, executes fewer instructions per operation, and scales well on multiprocessor machines. Experimental and anecdotal evidence indicate outstanding performance.

1. Introduction

Hashing is an efficient and popular technique for fast lookup of items based on a key value. Most research on hash tables has been focused on programs with a single thread of control. However, many modern applications are multithreaded and run on multiprocessor systems. Server applications such as web servers, database servers, and directory servers are typical examples. Server applications often make use of one or more (software) caches to speed up access to frequently used items. The number of items in a cache may vary greatly, both over time and among installations. Hashing is often used to provide fast lookup of items in a cache. The hash table then becomes a shared, global data structure that should grow and shrink automatically with the number of items and that must be able to handle a high rate of concurrent operations (insert, lookup, and delete), all without wasting memory. This paper describes LKRhash, a hash table implementation designed to meet these requirements.

Some programmers still believe that hash tables have to be of fixed size, that is, the table size has to be determined in advance and stay fixed thereafter. In the late 70s, several researchers proposed schemes that allow hash files to grow and shrink gradually in concert with the number of records in the file [1,4,6,7]. Two methods, namely, *linear hashing* and *spiral storage*, were subsequently adapted for main-memory hash tables[5]. LKRhash is based on linear hashing.

It is easy to make a hash table thread-safe: just add a single, global lock protecting all access to the table. The single lock serializes all operations on the table so that they cannot possibly interfere with each other. However, for multithreaded applications with many concurrent threads, the single lock easily becomes a bottleneck resulting in poor scalability. We have seen applications with poor or even negative scaling because of lock contention; that is, total throughput actually decreased after adding processors. LKRhash allows many operations on a hash table to proceed concurrently and offers excellent scalability.

All modern CPUs rely on multilevel processor caches to bridge the latency gap between memory and processors. The cost of a (complete) cache miss is substantial in the cycles wasted while the processor is stalled waiting for data to arrive from memory. Today, it is already is high as 100 cycles on some processors and we can only expect it to get worse. Significant performance improvements may result from cache-friendly data structures and algorithms that reduce the number of cache misses. Reducing cache misses and page faults were important design objectives for LKRhash.

The rest of the paper is organized as follows. Section 2 briefly describes linear hashing and shows how the hash table grows and shrinks as records are inserted and deleted. Section 3 describes the design of LKRhash. Section 4 presents experimental results on the performance and scalability of LKRhash on multiprocessor SMP machines. Finally, section 5 summaries the findings and offers some conclusions.

2. Linear Hashing

A higher load factor on a hash table increases the cost of all basic operations: insertion, retrieval, and deletion. If performance is to remain acceptable when the number of records increases, the table size must somehow be increased. The traditional solution is to create a new, larger hash table and rehash all the records into the new table. Typically, the new hash table is twice the size of the old one. Linear hashing allows a smooth growth: the table grows gradually, one bucket at a time. When a new bucket is added to the table, a limited local reorganization is performed. Linear hashing was developed by W. Litwin [6] for external files and adapted to in-memory hash tables by P.-Å. Larson [5]. Griswold and Townsend subsequently suggested improvements to handle very small tables more efficiently [2].

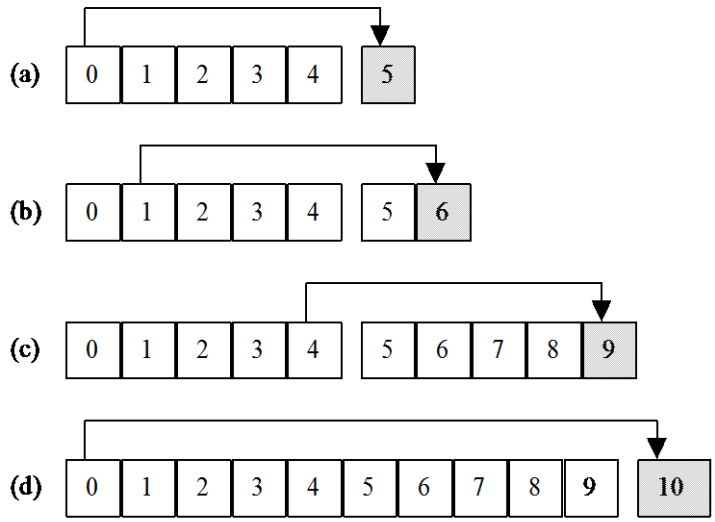


Figure 1: Expansion process of linear hashing for a five-bucket table

Consider a hash table consisting of N buckets with addresses 0, 1,…, N–1. A bucket is simply a linked list containing all the records hashing to the same address. Linear hashing increases the address space gradually by splitting the buckets in a fixed order: first bucket 0, then bucket 1, and so on, up to and including bucket N–1. When a bucket is split, about half of its records are moved to a new bucket at the end of the table. The splitting process is illustrated in Figure 1 for an example table with five buckets (N=5). A pointer p keeps track of the next bucket to be split. When all N buckets have been split and the table size has doubled to 2N, the pointer is reset to zero and the splitting process starts over again. This time, the pointer travels from 0 to 2N–1, doubling the table size to 4N. This expansion process can continue as long as is required.

Figure 2 illustrates the splitting of bucket 0 for an example table with 5 buckets. An entry in the hash table contains a single pointer, which is the head of a linked list connecting all records that hashed to that address. When the table is of size 5, all records are hashed by $h_0(K) = K \bmod 5$. Once the table size has doubled to 10, all records will be addressed by $h_1(K) = K \bmod 10$. However, instead of doubling the table size immediately, we expand the table one bucket at a time.

Now consider the keys hashing to bucket 0: To hash to 0 under $h_0(K) = K \bmod 5$, the last digit of the key must be either 0 or 5. Under the hashing function $h_1(K) = K \bmod 10$, keys with the last digit equal to 0 still hash to bucket 0, while those with the last digit equal to 5 hash to bucket 5. None of the keys hashing to buckets 1, 2, 3, or 4 under h_0 can possibly hash to bucket 5 under h_1. To expand the table, all we need to do is allocate a new bucket (with address 5) at the end of the table, increase the pointer p by one, scan through the records of bucket 0, and relocate to the new bucket all those hashing to 5 under $h_1(K) = K \bmod 10$.

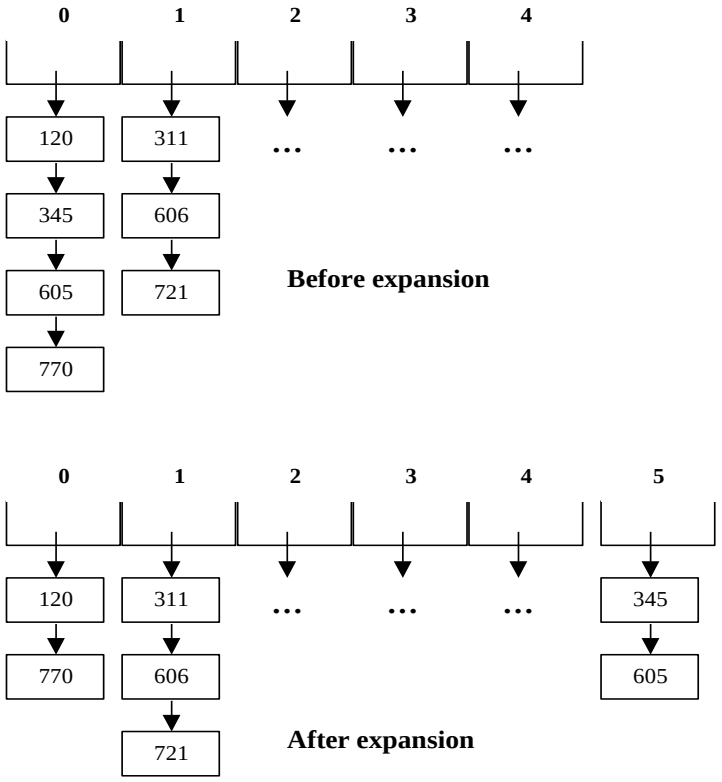


Figure 2: Splitting bucket 0 in a five-bucket table.

The address of a record changes as the table size changes but the current address of a record can be computed quickly. Given a key K, we first compute $h_0(K)$. If $h_0(K)$ is less than the current value of p, the corresponding bucket has already been split, otherwise not. If the bucket has been split, the correct address of the record is given by $h_1(K)$. When all the original buckets (buckets 0–4) have been split and the table size has increased to 10, all records are addressed by $h_1(K)$, which then becomes the new $h_0(K)$ for N=10.

The address computation can be implemented in several ways but the following solution appears to be the simplest. Let g be a normal hash function producing addresses in some interval [0, M]. M should be sufficiently large, say $M > 2^{20}$. To compute the address of a record, we use the following sequence of hashing functions:

$$h_i(K) = g(K) \bmod (N \times 2^i), \quad i = 0, 1, ...$$

where N is the minimum size of the hash table. (If N is a power of two, the modulo operation reduces to extracting low-order bits of $g(K)$.) The hashing function $g(K)$ can also be implemented in several ways. Functions of the type $g(K) = (cK) \bmod M$, where c is a constant and M is a large prime have experimentally been found to perform well. Different hashing functions are easily obtained by choosing different values for c and M.

We must keep track of the current state of the hash table. This can be done by two variables:

L = number of times the table size has doubled (from its minimum size, N).

p = pointer to the next bucket to be split, $p < N \times 2^L$.

When the table is expanded by one bucket, these variables are updated as follows:

```
p++ ;
if ( p == N*(1 << L ))  { L++ ; p = 0 ; }
```

Given a key *K*, the current address of the corresponding record can be computed simply as

```
addr = h_L(K);
if ( addr < p )  addr = h_{L+1}(K);
```

Contracting the table by one bucket is exactly the inverse of expanding it by one bucket. First the state variables are updated as follows:

```
p-- ;
if ( p < 0 )  { L-- ; p = N*(1 << L) - 1 ; }
```

Then all the records of the last bucket are moved to the bucket pointed to by *p*, and the last bucket can be freed.

The above process describes how to expand or contract, but not when to do these operations. The key idea is to keep the overall load factor bounded. The overall load factor is defined as the number of records in the table divided by the (current) number of buckets; i.e., the average chain length. We fix a lower and an upper bound on the overall load factor and expand (contract) the table whenever the overall load factor goes above (below) the upper (lower) bound. This requires that we keep track of the current number of records in the table, in addition to the state variables *L* and *p*.

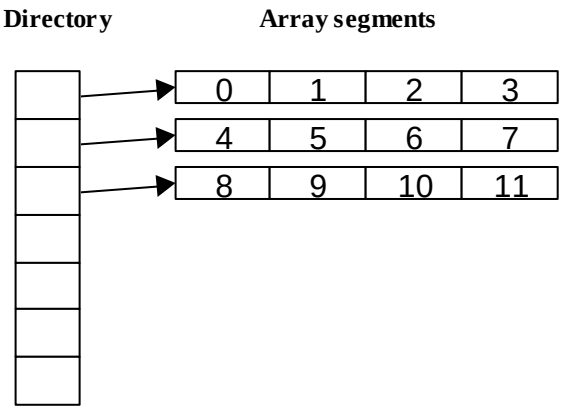


Figure 3: Implementing a variable-size array

Implementation

The basic data structure required is an expanding and contracting array. The simplest way to implement such an array is to use a two-level data structure, as illustrated in Figure 3. The array is divided into segments of fixed size. When the array grows, new segments are allocated as needed. When the array shrinks and a segment becomes superfluous, it can be freed. A segment directory keeps track of the start address of each segment in use. When the directory array becomes full, we simply double its size and copy the existing pointers into the first half. Similarly, when the directory becomes less than half full, we reduce its size by half.

It is easiest to let the minimum table size correspond to one segment. If the segment size is a power of two, the offset within the directory and the offset within a segment can be computed from the bucket address by masking and shifting. A directory size and segment size of 256 gives a maximum address space of $256 \times 256 = 64K$ buckets.

The hash function used in our implementation is shown below. The variables *L* and *p* were explained earlier. *SegBits* is the number of bits needed for addressing within a segment, i.e., a segment size of 256 entries corresponds to 8 bits, 512 to 9 bits and so on. The function takes as input the (hash) signature of a key. If keys are integers or any 4-byte data type, the key value itself (treated as an integer) is taken as the

signature. If keys are longer, the key value must first be converted into an integer, which is then taken as the hash signature. The function `convertkey()` performs this conversion.

```
const unsigned int RNDM_CONSTANT= 314159269; // default scrambling constant
const unsigned int RNDM_PRIME   =1000000007; // prime, also used for scrambling

unsigned int hash(unsigned int key_signature, hashtable *pT)
{
  unsigned int h = (RNDM_CONSTANT * key_signature) % RNDM_PRIME;
  unsigned int mask = (1 << (pT->L + pT->SegBits)) - 1;
  unsigned int address = h & mask;
  if (address < pT->p)  address = h & ((mask << 1) | 1);
  return address;
}

const unsigned int MAGIC_NO=101;

unsigned int convertkey( void  *key, int key_len)
{
  unsigned char *cp = (unsigned char *) key;
  unsigned char *sentinel = cp + key_len;
  unsigned int   signature = 0;
  for ( ; cp < sentinel; cp++)
    signature = MAGIC_NO * signature + (unsigned int) *cp ;
  return signature;
}
```

The "magic" constants used in the two functions are somewhat arbitrary but the ones shown have been tested on many key sets and found to work well. The primary requirement on the function `convertkey()` is to produce as few duplicate signatures as possible. The function shown was tested on a variety of data sets with different values for the constant `MAGIC_NO`. Small primes in the range 101 to 251 consistently produced signatures with few duplicates. Values containing small factors (less than 23), especially factors of 2, produced many more duplicates than theoretically expected. Signatures need not be uniformly distributed; spreading them uniformly over a range is taken care of by the multiplication and modulo operation on the first line of function `hash()`. The "scrambling" operation applied is the same as that used in multiplicative random number generators. Ramakrishna and Zobel [11] tested different hash functions on several sets of string-valued keys. They also found that functions of the above type produced good results but that they were slower than functions using shifting and XOR operations.

3. LKRhash

We want a hash table implementation that can support very high rates of concurrent operations (insert, delete, and lookup) on contemporary shared-memory multiprocessor (SMP) systems. To this end, we modified the basic linear hashing design to reduce the number of L1 and L2 cache misses, added a light-weight locking scheme, and modified the algorithms to reduce the amount of time that locks are held. These changes are described in more detail in this section.

3.1 Reducing cache misses

The original paper [5] suggested a very simple data structure for a linear hash table: an entry in the hash table consists of nothing more than a pointer to a hash chain, that is, a linked list containing all items hashing to that address. The hash chain can be either intrusive or non-intrusive. In the first case, the pointer field needed for the hash chain is embedded in the items themselves. In the second case, the hash chain consists of separate link nodes containing two fields: a pointer to the next link node and a pointer to the target item. The second alternative requires more space but has the advantage that the target items are completely unchanged.

Unfortunately, this organization has poor processor cache behavior. To see why, consider the retrieval of an item that happens to be the fifth item on its hash chain and count the potential cache misses. First take the case of embedded hash chains.

1. Get hash table parameters: one miss.
2. Get segment pointer from directory: one miss.

3. Get pointer to first item: one miss.
4. Repeat five times
    4.1. Get key of item: one miss (assuming the key is contained in one cache line).
    4.2. Get pointer to next item: one miss (assumed to be in a different cache line than the key).

The total is 13 potential cache misses. This assumes that locating the key and comparing it with the search key results in only one cache miss. If the key is long or not located at a fixed position within the item, the number of cache misses may be higher.

Note also that the various items read during retrieval may be scattered over multiple pages of memory. In the worst case, we may touch as many as 13 different pages.

Thirteen potential cache misses is far too high. To reduce the number, we applied three different ideas: comparing hash signatures, packing link nodes into cache lines, and pulling the first node of the hash chain into the table itself.

Comparing hash signatures. Accessing and comparing keys is expensive. To avoid key comparisons, each link node contains a 32-bit hash signature computed from the key of the associated item. During lookup, we first compute a signature for the search key. This allows fast comparison of the stored signatures with the signature of the search key. If signatures don't match, the keys cannot match either. A full key comparison is necessary only when the signatures match. R. Morris originally proposed the use of hash signatures in 1968 [9]. In our implementation, the signature is simply the value returned by `convertkey()`.

A well-designed signature function produces very few signature clashes; i.e., different keys with the same signature value. If there are no clashes, a successful lookup will require only one key comparison, namely, the final comparison confirming that we have indeed found the right item. An unsuccessful lookup will require no key comparisons at all. In practice, a small number of clashes will occur but not many. In our example, hash signatures will eliminate four potential cache misses and the same number of key comparisons.

One other benefit of storing hash signatures is worth mentioning. Because the hash address is computed based on the signature of a key, we need not recompute the signature during expansion; we can simply use the stored signature.

Packing link nodes into cache lines. Looking at our example again, we see that there is one potential cache miss for each link node traversed during a search. To improve cache locality, LKRhash uses larger link nodes and stores multiple pointers and corresponding hash signatures in each node. These larger nodes are laid out carefully to fit exactly into one or a few cache lines and to minimize the number of cache misses during search.

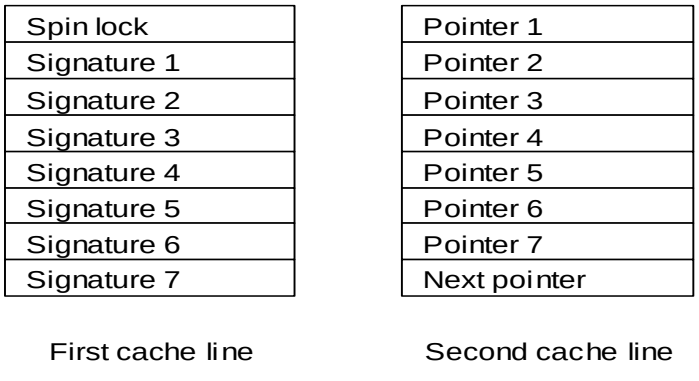


Figure 4: Bucket node layout

In our implementation, the default node size is 64 bytes so it fits into two 32-byte cache lines. A node contains seven pointer-signature pairs plus a spin lock and a next pointer, stored as shown in Figure 4. For our example, this change reduces the number of potential cache misses to six misses from the original thirteen.

Pulling in the first node. As an additional optimization, LKRhash stores the first link node in the hash table itself. This reduces the potential cache

misses by one, to a total of five in our example. However, the main benefits are elsewhere: less time spent on memory management and a much smaller working set (pages touched).

Assume that each node contains seven slots and that we limit the average chain length to five. In that case, the great majority of hash table entries will receive seven items or less. These will all fit into the first node stored in the hash table and no additional link nodes need be allocated. Furthermore, space for the first node is allocated automatically when the corresponding array segment is allocated. Very little time is spent on memory allocation but some space is wasted because space is preallocated in larger chunks.

Now consider working set size. Assume that we have a hash table consisting of 500 entries, storing 2500 items, and that the page size is 4KB. In the original design, the hash table data structure may be spread over as many as 2502 pages; one for the "root" of the hash table, one for the 500 pointers, and one each for the 2500 link nodes. By using larger link nodes capable of holding seven pointer–signature pairs, this is reduced to 502 pages or slightly more, that is, one for the "root" of the hash table, one for the 500 pointers, and one link node for each of the 500 addresses (a small fraction will have more than one link node). By pulling the first node into the hash table array, this is reduced to eight pages (500×64/4096) plus one for each overflow link node. This is a worst-case analysis, so in practice the reduction will not be as dramatic but still very significant. We have not explicitly measured this effect in our experiments but one user reported a 75% reduction in the working set compared with another hash table implementation.

3.2 Reducing lock contention

LKRhash is targeted for multithreaded applications and must support a high rate of concurrent operations. Most data structures require some changes to be applied serially to preserve the integrity of the data structure. This is true for linear hash tables as well. For example, changes to the split pointer must be serialized. To achieve high scalability, we must keep the serially executed sections of code short and minimize the synchronization overhead. We applied the following ideas to improve the scalability of LKRhash tables: use spinlocks for synchronization, lock buckets rather than tables, modify algorithms to reduce lock time, and partition items among multiple subtables.

Bucket Locks. LKRhash uses two levels of locks: a table lock and a lock for each bucket. The table lock protects access to table-wide data like the split pointer, counters, and the directory of segments. A per-bucket lock serializes all operations on items in that bucket. Bucket locks allow the table lock to be released much sooner. Bucket locks provide fine-granularity locking and greatly reduce lock contention.

Spinlocks. LKRhash uses carefully crafted spinlocks for synchronization. A simple mutual exclusion spinlock is used for the per-bucket locks. It occupies four bytes and stores the thread ID,[1] if locked, or 0, if unlocked. When a thread tries to acquire a lock that is already locked, the thread spins in a tight loop testing the lock repeatedly.[2] We use bounded spinlocks; i.e., there is a bound on how many times a thread tests a lock before it sleeps, relinquishing the processor. Our default bound was 4000 times. If the lock is still unavailable when control returns to the thread, the spin time is halved repeatedly in an exponential backoff strategy.

Spinlocks use less memory than most other synchronization mechanisms; in particular, much less than Windows™ critical sections. This means that we can afford to use more locks, protecting smaller parts of a data structure. Spinlocks greatly reduce the frequency of context switches but only if locks are held for a very short time. Spinlocks are best thought of as cheap, short-duration locks.

Algorithm changes. We made several changes to the insert, delete, and lookup algorithms to reduce lock hold time. Here are the most important ones.

- The hash signature is computed before the table lock is acquired. The signature of the key is deterministic and does not depend on the state of the hash table.

[1] Storing the thread ID aids debugging.

[2] No spinning occurs on uniprocessor systems, as it is fruitless. The thread that holds the lock cannot possibly release the lock if the thread that wants the lock is busy spinning. Instead, the thread that wants the lock yields the CPU immediately. Since uniprocessor systems provide an illusion of concurrency by coarse-grained time-slicing, lock contention tends to be much lower than on multiprocessor systems.

- The table lock is held just long enough to find and lock the target bucket. All operations on the table must acquire the table lock before finding and locking a bucket (to avoid race conditions such as a concurrent expansion changing the values used in the h_0 and h_1 functions). Minimizing the hold time of the table lock reduces contention and improves concurrency considerably. Since there are typically several hundred buckets in a hash table, contention on the per-bucket locks is negligible.
- Test whether to expand or contract without locking. Linear hashing promises to keep the overall load factor bounded. Locking the table to test whether to expand would provide no useful extra accuracy and would increase contention.
- Release the table lock early when expanding (contracting) the table. The table lock is held while finding and locking the old and new buckets and possibly expanding (shrinking) the directory of segments, but it is not necessary to hold it while splitting (merging) the two bucket chains.

Table Partitioning. An LKRhash table may consist of multiple separate subtables, where each subtable is a linear hash table as described above. Items are assigned to subtables by hashing on the key value. Each subtable receives a much lower rate of operations, resulting in fewer conflicts.

The number of subtables is determined when an LKRhash table is created and cannot be changed at run time. (Note: There is no limitation in growing or shrinking sub-table count, but the benefits are not high – shouldn't we have this point as well.) The subtables, including their directories, are also allocated at creation time. By disallowing changes in the number of subtables at run time, we avoid having an additional level of locks. The key's hash signature is also used to multiplex into the correct subtable. The top-level lookup routine becomes something like this:

```
Record* FindKey(Table* pTable, Key* pKey, int KeyLen)
{
  unsigned int hash = convertkey(pKey, KeyLen);              // calculated only once
  unsigned int i = (hash % 104853) % pTable->m_NumSubTables; // scramble with a prime
  return FindKey2(pTable->m_SubTables[i], pKey, hash);
}
```

3.3 Further refinements

This section briefly outlines three possible further refinements. All three are included in our implementation.

Recursive spinlocks. If a thread is allowed to apply an operation when it already holds some lock on the table, the thread may block itself. We noticed this phenomenon when implementing a table scan operation. The scan retained a bucket lock to safeguard its position in the table. However, if the thread performing a scan tries to perform another operation, such as an insert, while the scan is active, it may hit the locked bucket and block itself. Therefore, if this type of lock-retaining operation is needed, the simple spinlock explained above is not sufficient; we need a spinlock that can be acquired safely multiple times by the same thread. We implemented a spinlock with this property. Each spinlock still requires four bytes and consists of two fields: a field storing the ID of the thread holding the lock, in the lower 28 bits, and a counter field, in the top 4 bits. The counter simply counts the number of times the lock has been acquired by the thread, which allows the lock to be safely reacquired.

The insert, delete, and find operations cannot be composed safely. For example, it is a common requirement to insert an item in a table only if it is not already present. This is trivial to implement in a single-threaded environment, but requires locking the table in a multithreaded environment. To accommodate such scenarios, we give users explicit access to the table lock. For the simple, atomic operations like delete, they do not need to acquire the lock—indeed, to maximize concurrency, they must not—but more complex operations like insert-if-not-present require explicit control of the table lock.

The table lock is a multi-reader, single-writer spinlock, occupying eight bytes. The table lock keeps track of the number of readers (16 bits), the number of waiting writers (16 bits), a thread ID field (28 bits), and a recursion count (4 bits). The thread ID and the recursion count are used only if a writer holds the lock, so that the owning thread can reacquire the lock without deadlocking itself.

Safe expansion/contraction. In a server environment, the hash table should always be left in a usable state. In particular, a memory shortage detected during insertions should be handled gracefully. This is easy if the

insertion does not trigger an expansion: if a new node is required and it cannot be allocated, we just reject the insertion. However, during an expansion multiple new nodes may be required (for the new bucket) and possibly a new segment and space for an enlarged directory. If the required space cannot all be allocated, the table should not be expanded. The simplest way to handle this is to first compute what space will be needed and preallocate the space. The actual expansion will only proceed if this phase succeeds.

The same problem arises for contraction: a contraction may require additional nodes (and possibly a smaller directory). The same solution also applies, namely, preallocate the space needed in an initial phase.

Typesafe template wrapper. Internally, the hash table manipulates untyped `void*` pointers. For full generality, it makes no assumption about the layout of records and requires callback functions that extract a key from a record, calculate a hash signature for a key, and compare two keys for equality. For convenience and type safety, we provide a C++ template wrapper to simplify use of the table.

4. Experimental results

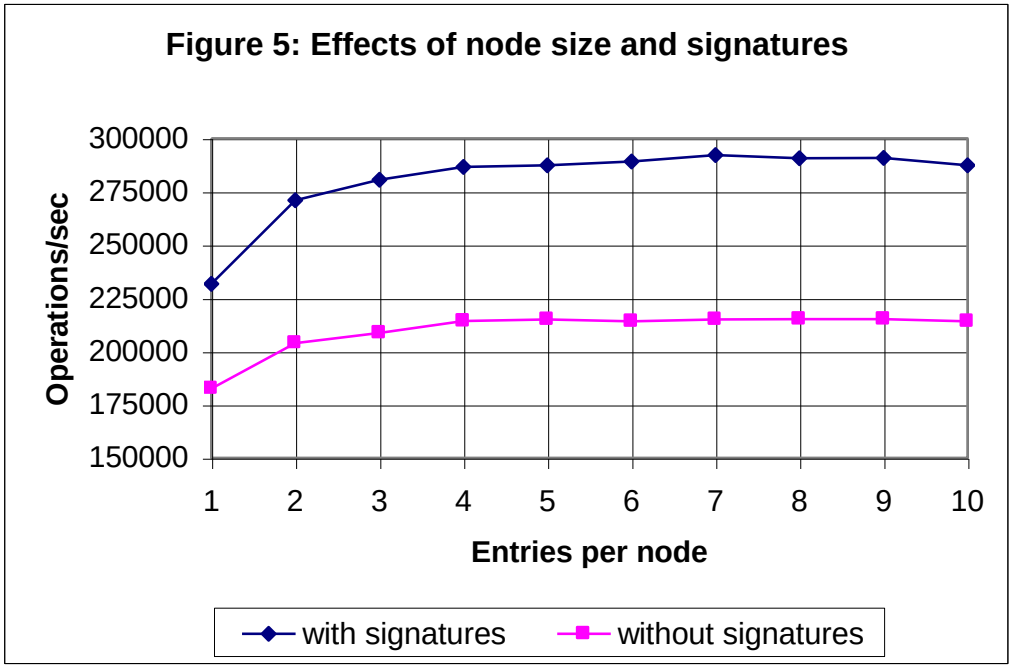
This section reports experimental results. The first experiments attempt to determine the performance benefits of using signatures and of increasing the node size. Later experiments focus on scalability. We measure performance by throughput, that is, the number of insertion, deletion, and search operations per second.

The basic input consists of a 25,084-entry dictionary of English words. To create any number of distinct keys, the original 25,084 entries are replicated prefixed with a digit; e.g., "abba", "0abba", "1abba", "2abba". In our experiments we expanded the key set to 100,000 or 200,000 keys.

Unless indicated otherwise, the load factor was 4.0. For a 100,000-element table, this means that there will be approximately 100,000/4.0 = 25,000 bucket chains.

4.1 Varying the Node Size

Figure 5 shows the effect of changing the node size (how many pointer–signature pairs are packed into a link node) and of using hash signatures. These experiments were run on a 300 MHz Pentium II machine with one processor. The input was the same as described above. The hashtable used a single subtable and the load factor limit was set to 5.0. The input consisted of 200,000 keys, yielding a maximum table size of 40,000 buckets. Five random lookups were performed for each insert or delete operation. The experiment was repeated 5 times, that is, each point in the diagram is based on 1,000,000 inserts, 1,000,000 deletes, and 10,000,000 lookups.



The 1 record/node case corresponds to a traditional (non-embedded) linked list. Clearly, increasing the node size improves performance up to 4 records/node, but makes little difference thereafter. Using signatures also increases throughput significantly. The combined effect is an increase of 60%, from 182,423 operations/sec for node size one and no signatures to 292,127 operations per second for node size seven with signatures. Hash signatures save cache misses and key comparisons. In our case the keys are just case-sensitive strings so key comparisons are relatively cheap. One can expect even greater benefits for cases where key comparisons are more expensive, for example, when keys have multiple parts.

The increased throughput is caused by a combination of fewer cache misses and executing fewer instructions. We used the hardware counters available on Pentium processors to measure the number of clock cycles, the number of instructions executed, and the number of L1 and L2 cache misses per operation. Each experiment consisted of three phases: (1) build a hash table by inserting 200,000 keys, (2) perform 1,000,000 random (successful) lookups and (3) empty the hash table by deleting the 200,000 keys. Table 1 below lists the observed results for load factors 5 and 10, node sizes 1 and 7, and with and without signatures.

	Load factor 5.0				**Load factor 10.0**			
Entries/node	1	1	7	7	1	1	7	7
Signatures	No	Yes	No	Yes	No	Yes	No	Yes
Inserts								
Instructions	958	900	602	545	1085	967	736	624
Cycles	2316	1629	1800	1067	3325	1961	2547	1152
Cycles/instruction	2.42	1.81	2.99	1.96	3.06	2.03	3.46	1.85
L1 cache misses	22.56	13.63	16.07	7.73	36.17	18.81	24.83	9.42
L2 cache misses	13.07	7.14	9.84	3.95	23.01	10.99	15.89	5.09
Lookups								
Instructions	362	348	342	328	420	389	382	351
Cycles	1561	1236	1389	1044	2140	1490	1762	1095
Cycles/instruction	4.1	3.55	4.06	3.18	5.10	3.83	4.61	3.12
L1 cache misses	14.93	10.36	12.06	7.92	22.72	13.54	16.46	8.41

L2 cache misses	9.20	6.46	7.85	5.15	14.28	8.76	11.05	5.66
Deletes								
Instructions	740	718	513	493	718	678	589	542
Cycles	2099	1821	1459	1199	2339	1867	1883	1292
Cycles/instruction	2.84	2.54	2.84	2.43	3.26	2.75	3.20	2.38
L1 cache misses	20.51	17.14	10.59	7.39	25.19	19.41	16.46	9.37
L2 cache misses	11.84	9.67	6.52	4.46	14.66	11.04	10.08	5.25

Table 1: Instructions, cycles, and cache misses per operation.

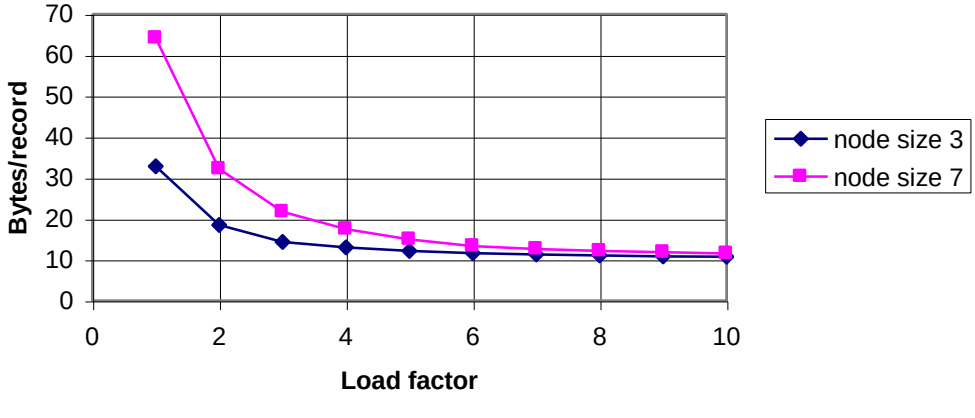
The two optimizations speed up insertions and deletions more than lookups. For load factor 5, insert time is reduced by 54%, delete time by 43% but lookup time only by 33%. Even though the reductions in L1 and L2 cache misses are impressive, it appears that the savings in instructions executed has more effect on run time. Fewer cache misses do not affect the number of instructions executed but they reduce processor stalls, thereby reducing the average number of cycles per instruction. For lookups with load factor 5, the total reduction is 1561 – 1044 = 517 cycles. Of this, about 180 cycles can be attributed to fewer cache misses (L1: 35 cycles; L2: 145) and the remaining 337 to fewer instructions. For inserts, the corresponding number is 2316 – 1067 = 1249 cycles, of which about 400 can be attributed to fewer cache misses (L1: 75; L2: 325) and the remaining 849 to fewer instructions. This analysis is only approximate. Pentium processors can execute instructions out of order which makes an exact analysis difficult.

Hash table throughput and space per record used by the hash table depend on the node size and the load factor. The node sizes that make sense in practice are 3 (32 bytes, one cache line) and 7 (64 bytes, two cache lines). Figure 6 plots space per record as a function of the load factor for these two node sizes. Figure 7 plots the throughput in operations per second. The data is for a table of 200,000 records. Signatures were used.

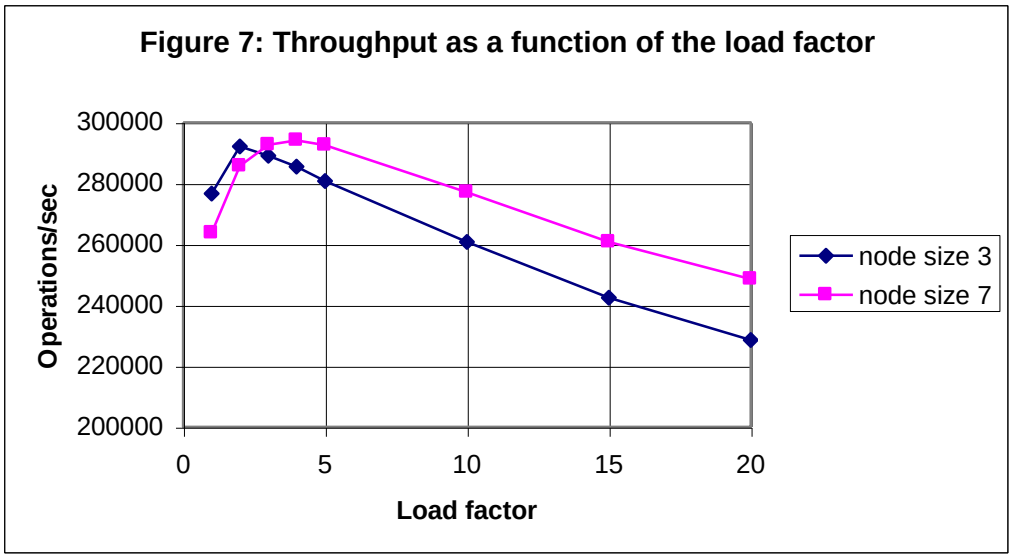
4.2 Space usage and throughput versus load factor

For a load factor of 1.0, slightly over one node (32 bytes or 64 bytes) is needed per record. Space used per record tapers off quickly as the load factor increases. For a load factor of 4.0 it is already down to 13.0 and 17.4 bytes and for a load factor of 6.0 it is 11.6 and 13.3 bytes.

Figure 6: Space usage as a function of the load factor



As shown in Figure 7, throughput is fairly stable over a wide load factor range, varying between 260,000 and 300,000 operations per second for a load between 1.0 and about 10. The best performance is achieved with a moderate load factor, somewhere between 2 and 6. For node size 3 the maximum is 292,127 at a load factor of 3 and for node size 7 the maximum is slightly higher, namely, 294,139 at a load factor of 4. For both node sizes, a load factor around 5 is a good compromise, combining high throughput with good storage utilization.



4.3 Scalability Results

The following experiments examined the scalability of LKRhash on multiprocessor systems. In an ideal situation, throughput should increase linearly with the number of threads concurrently accessing the hash table. This cannot be achieved in practice because of resource contention, both in software (locks held) and in hardware (system bus and memory).

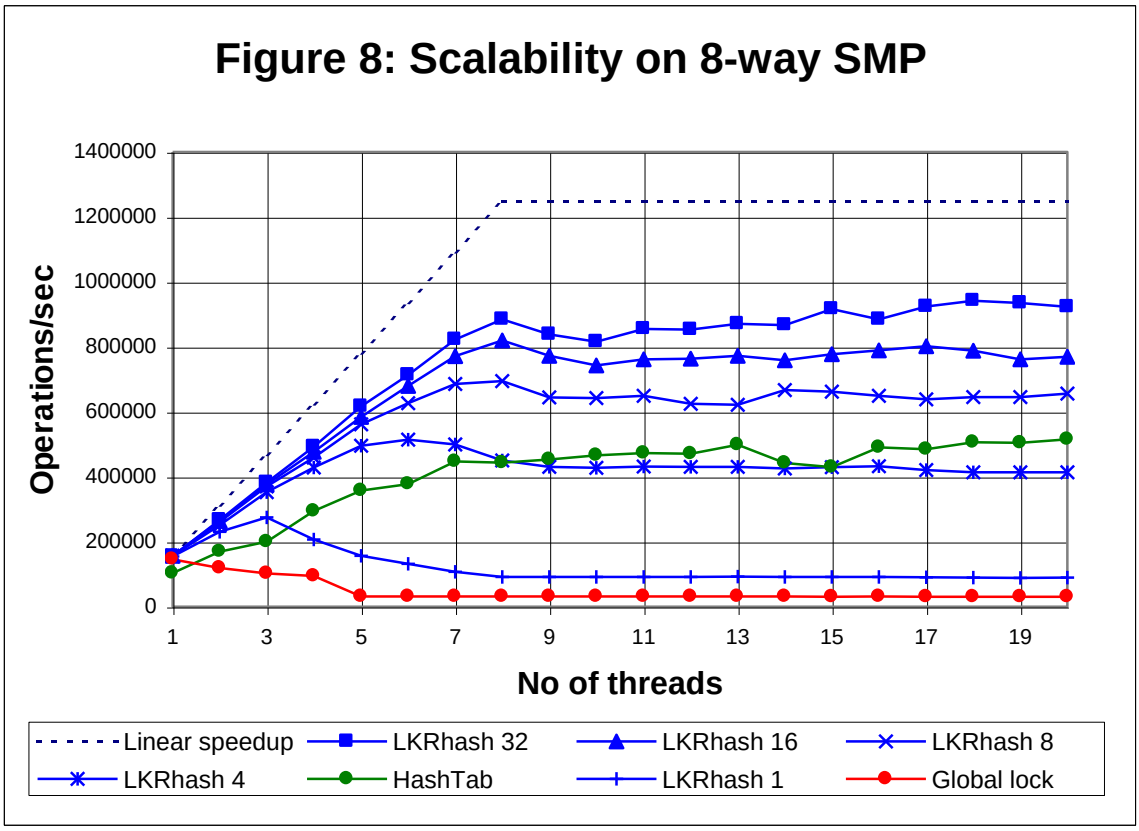
The following test is repeated for *T* threads, where *T* runs from 1 to 20. The input consists of a 100,000-element array of keys, partitioned into 100,000/*T* subarrays, one for each of the *T* threads. Each thread adds one element to the shared hash table at a time. As each entry is added, each thread makes *S* searches for words that it has already added to the hash table, each search being chosen at random. Once all 100,000 words have been added to the hash table, the table is torn down, one element at a time. As each element is deleted, another *S* random searches are made for words known to still be in the hash table. Each of the *T* threads does *R* rounds of insertions, deletions, and 2*S* searches. When the *R* rounds have completed, the entire set of operations begins again for *T*+1 threads. In the charts below, each insertion, deletion, and search is counted as a separate operation.

The primary metric of interest is throughput: the number of insertion, deletion, and search operations per second. We are also interested in scalability: the number of operations/second for *T* threads compared to the number of operations/second for one thread. Ideally, we would achieve linear scalability. The tests were run on an 8-way SMP with 200MHz Pentium Pro processors, each with 512KB L2 cache

We compared the performance of LKRhash tables with varying numbers of subtables with the other hash tables.

1. **Global Lock**. This is linear hashing using a single table with a single global lock. Every operation acquires this mutual exclusion–type lock at the beginning of the operation and releases it at the end of the operation.
2. **HashTab:** This implements a fixed-size hash table with a Windows critical section per bucket.

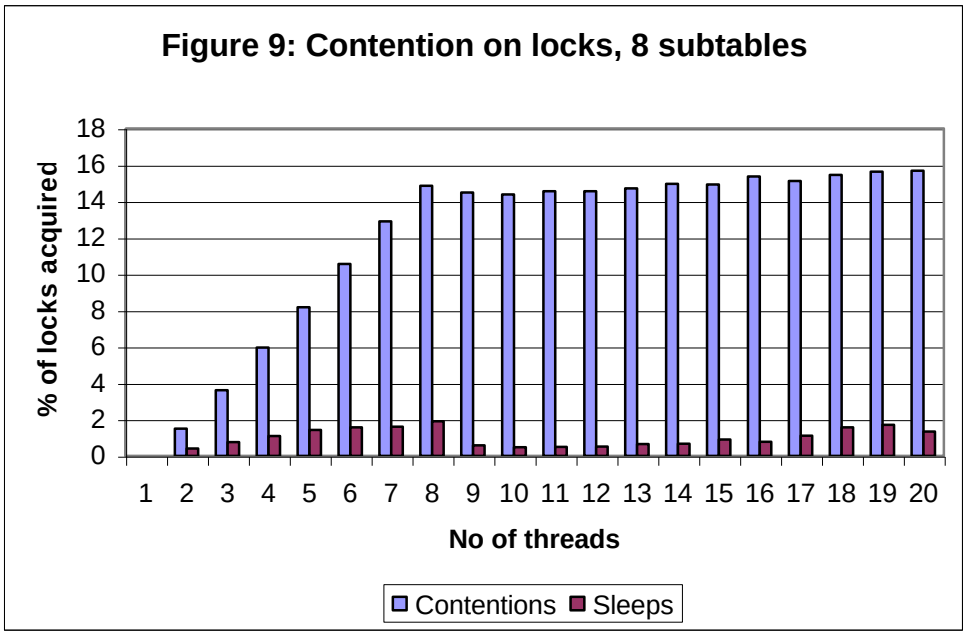
3. **LKRhash *X*:** This is an LKRhash table with *X* subtables, with *X* ranging over 1, 4, 8, 16, and 32.



We can draw several interesting conclusions from this chart.

- LKRhash is fast, even when there is only one thread, and hence, no contention. It beats HashTab, which is itself a well-crafted hash table.
- Gating all operations through a global lock kills scalability. The more threads, the worse the throughput. Partitioning locks helps scalability.
- The higher the number of subtables, the better the throughput as threads are added. Having only one subtable becomes quite a bottleneck, as all operations contend for the single tablelock. Multiplexing operations across even four tables improves throughput. In practice, one subtable per CPU gives good results. Adding more than that yields little benefit, since most applications do work other than updating hashtables.
- LKRhash scales extremely well, achieving throughput rates of over 75% of the ideal rate. This is highly unusual on an 8-way system.

Figure 6 below illustrates lock contention for a hash table with 8 subtables. The contentions are almost all due to the locks on the eight subtables. Only 0.1% of acquisitions of a per-bucket locks experience contention. Recall that there are approximately 25,000 buckets in the table when it is full, so the probability of contention on any bucket lock is negligible. The subtable locks experience contention for about 15% of all operations or, phrased differently, about 85% of all lock acquisition succeed immediately. The spincount is adequate for almost all cases, as less than 2% of lock acquisitions yield the processor. If the load were lighter, this number would approach zero.



5. Summary and conclusion

Server applications use hash tables routinely when fast lookup is required, for example, for managing a cache of objects of some type. Server applications need fast hash tables capable of growing and shrinking automatically as the number of objects in the table varies. Servers are normally multithreaded and often run on multiprocessors so the hash table must scale well as the level of concurrent access increases.

Linear hashing efficiently solves the problem of growing and shrinking a hash table—all operations take constant average time regardless of table size. However, the original version was not intended for concurrent access. LKRhash is a version of linear hashing designed to handle high levels of concurrency. In addition, it attempts to reduce cache misses—an increasingly important consideration for modern processors.

Four ideas are exploited to reduce cache misses: hash signatures, separate hash chains (i.e. not embedded in the objects), using multi-object nodes on the hash chains, and pulling the first node into the hash table. Our experimental results show that these changes reduce cache misses and instructions executed very significantly. The reduction in instructions executed is more important and caused mainly by the use of signatures.

Three ideas are exploited to improve scalability under concurrent access: lightweight spin locks, a lock per bucket and dividing the table into multiple subtables. Our experimental results show close-to-linear scaling: as high as 75% of the ideal on an 8-processor system.

LKRhash is used in several Microsoft products and users report excellent performance. One user reported a 10X speedup, 75% reduction in the working set, and 90% reduction in page faults compared with the hash table implementation in a third-party class library used previously. Another user reported a 30% run time reduction for an application that relies heavily on hash table lookup.

6. References

1. R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong, Extendible hashing — a fast access method for dynamic files, *ACM Transactions on Database Systems*, 4, 3 (1979), 315-344.
2. W.G. Griswold, G.M. Townsend, The design and implementation of dynamic hashing for sets and tables in Icon", *Software - Practice and Experience*, 23(4):351–67, 1993.
3. D.E. Knuth, *The Art of Computer Programming, Volume 3, Sorting and Searching*, (2nd Ed), Addison–Wesley, 1998.
4. P.-Å. Larson, Dynamic hashing, *BIT*, 18, 2 (1978), 184–201.
5. P.-Å. Larson, Dynamic hash tables, *Communications of the ACM*, Vol. 31, No 4, 1988, 446–457.
6. W. Litwin, Linear Hashing: A new tool for file and table addressing, *Proceedings of the 6th Conference on Very Large Databases (VLDB '81)*, 1981, 212–223.
7. G.N.N. Martin, Spiral Storage: Incrementally augmentable hash addressed storage, Theory of Computation Rep. 27, University of Warwick, England, 1979.
8. B.J. McKenzie, R. Harries, and T. Bell, Selecting a hashing algorithm, *Software — Practice and Experience*, 20(2), 1990, 209–224.
9. R. Morris, Scatter storage techniques, *Communications of the ACM*, Vol. 11, No.(1), 1968, 38–44.
10. J.K. Mullin, Spiral Storage: Efficient dynamic hashing with constant performance, *The Computer Journal*, 28(3), 1985, 330–334.
11. M.V. Ramakrishna and J. Zobel, Performance in practice of string hashing functions, *Proc. International Conference on Database Systems for Advanced Applications (DASFAA)*, 1997, 215–223.