Fast and Compact Hash Tables for Integer Keys

Nikolas Askitis

School of Computer Science and Information Technology,
RMIT University, Melbourne 3001, Australia.
Email: `naskitis@cs.rmit.edu.au`

Abstract

A hash table is a fundamental data structure in computer science that can offer rapid storage and retrieval of data. A leading implementation for string keys is the cache-conscious array hash table. Although fast with strings, there is currently no information in the research literature on its performance with integer keys. More importantly, we do not know how efficient an integer-based array hash table is compared to other hash tables that are designed for integers, such as bucketized cuckoo hashing. In this paper, we explain how to efficiently implement an array hash table for integers. We then demonstrate, through careful experimental evaluations, which hash table, whether it be a bucketized cuckoo hash table, an array hash table, or alternative hash table schemes such as linear probing, offers the best performance—with respect to time and space—for maintaining a large dictionary of integers in-memory, on a current cache-oriented processor.

Keywords: Cuckoo hashing, integers, cache, array hash, dictionary, in-memory.

1 Introduction

In-memory data structures are fundamental tools used in virtually any computing application that requires efficient management of data. A well-known example is a hash table (Knuth 1998), which distributes keys amongst a set of slots by using a hash function (Ramakrishna & Zobel 1997, Zobel, Heinz & Williams 2001). A hash table can offer rapid insertion, deletion, and search of both strings and integers but requires a form of collision resolution to resolve cases where two or more keys are hashed to the same slot. The simplest and most effective collision resolution scheme for when the number of keys is not known in advance is the use of linked lists. This forms a chaining hash table (Zobel et al. 2001) also known as a *standard-chain* hash table (Askitis & Zobel 2005).

Linked lists (or chains) are simple and flexible structures to implement and maintain but are not particularly cache-friendly. Nodes and their associated keys are typically scattered in main memory and as a consequence, traversing a chain can result in poor use of CPU cache. In addition, nodes are accessed via pointers which incurs pointer-chasing, a phenomenon that hinders the effectiveness of hardware data prefetchers (Askitis 2007).

Askitis and Zobel (Askitis & Zobel 2005) addressed these issues by replacing the chains used to resolve collisions with dynamic arrays, yielding a cache-conscious collision resolution scheme called the array hash. The key advantage offered by the array hash is the elimination of nodes and string pointers that were used to access strings. This configuration permits good use of CPU cache and hardware data prefetch while simultaneously saving space.

We can avoid the use of chains and arrays altogether by storing homogeneous keys such as integers directly within hash slots. This forms an open-address hash table (Peterson 1957) which can be simpler to implement and can result in better use of both CPU cache and space (Heileman & Luo 2005). When a collision occurs in an open-address hash table, several well-known techniques can be applied such as linear probing or double hashing (Askitis 2007). Our focus is with a relatively newer technique called cuckoo hashing (Pagh & Rodler 2004).

Cuckoo hashing uses two open-address hash tables and two independent hash functions. A key is stored in either one of the hash tables but not both. When a collision occurs in the first hash table (using the first hash function), the key occupying the slot is evicted and replaced by the new key. The evicted key is then hashed using the second hash function and stored in the second hash table. If this causes another collision, then the eviction process repeats until all keys are hashed without collision. Cuckoo hashing offers a constant worst-case probe cost since at most only two hash slots are accessed during search (Pagh & Rodler 2004, Panigrahy 2005).

Cuckoo hashing, however, requires the average load factor (i.e., keys/slots) to be kept less than 0.49 (Pagh & Rodler 2004). If we were to exceed this threshold, then the probability of an insertion failure (an irresolvable collision) will greatly increase. Recent innovations of cuckoo hashing have addressed this issue by employing several hash functions to reduce the cost of collisions (Fotakis, Pagh, Sanders & Spirakis 2003), and with hash slots that can store more than one key—otherwise known as *bucketized* cuckoo hashing. These variants, among others, can efficiently support a load factor of almost 1 (Zukowski, Héman & Boncz 2006, Ross 2007).

Despite its popularity (Kirsch, Mitzenmacher, & Wieder 2008), cuckoo hashing has yet to be compared against an integer-based array hash. The array hash is structurally dissimilar to cuckoo hashing, since it employs dynamic arrays to resolve collisions. The main disadvantage with this approach in comparison to cuckoo hashing, is that more space can be required as a result of slot pointers and memory allocation overheads. In addition, more keys can be compared in the worst-case since arrays are dynamic (each hash slot is able to store an unbounded number of keys). However, the array hash has a useful property that cuckoo hashing lacks—it can make good use of cache while supporting a load factor greater than 1. The array hash is therefore a *scalable* hash table; it can remain efficient to use as the number of keys increase (Askitis 2007). The number of keys that can be stored in an open-address hash table, in contrast, is bounded by the number of slots and their capacity. Hence, to accommodate an unexpected increase in the number of keys processed, an open-address hash table would need to be resized and rehashed, which can be an expensive process particularly with a large number of keys (Cieslewicz & Ross 2007, Kirsch et al. 2008, Askitis & Zobel 2008*b*).

Drawing together the themes we have sketched, this paper provides three important contributions. First, we will show how to develop an array hash table for integers which we experimentally compare against a standard-chain hash table (Zobel et al. 2001), a more cache-efficient variant known as a clustered-chain hash table (Chilimbi 1999, Askitis 2007), and a linear probing open-address hash table (Peterson 1957, Heileman & Luo 2005). Our experiments measure the time, space, and actual cache performance of these hash tables using large volumes of 32-bit integers. Second, we experimentally compare the performance of the integer-based array hash against leading implementations of bucketized cuckoo hashing, which are designed to operate efficiently with a load factor of almost 1 (Ross 2007, Kirsch et al. 2008). Third, based on our results, we show which hash table is ideally suited—with respect to overall performance and space usage—for the task of storing and retrieving a large set of integers with payload data (that is, to maintain a dictionary) in-memory.

2 Background

A hash table is a data structure that distributes keys amongst a set of slots by using a hash function. A hash function should be both fast and from a universal class (Ramakrishna & Zobel 1997, Askitis 2007, Ross 2007), so that keys are distributed as well as possible. For integers, a multiplicative-based hash table is considered to be universal and computationally efficient (Ross 2007). Alternatively, assuming that the input sequence of integers is random, a simple modulo calculation can also serve as a fast and effective hash function (Zukowski et al. 2006).

A hash function, however, can not guarantee that each slot will have at most one key when the total number of keys is not known in advance (Knuth 1998). Therefore, a form of collision resolution is needed. For strings, a simple and effective technique is the use of linked lists with move-to-front on access (Knuth 1998), forming a *standard-chain* hash table (Askitis & Zobel 2005).

Linked lists, however, are not cache-efficient data structures (VanderWiel & Lilja 2000, Kowarschik & Weiß 2003). As a consequence, the standard-chain hash table is likely to attract high performance penalties (cache misses) on current cache-oriented processors. Node clustering can, to some extent, improve the cache performance of a linked list by packing its homogeneous nodes into blocks of memory that are sized to match the CPU cache-line (typically between 64 and 256 bytes) (Chilimbi 1999, Badawy, Aggarwal, Yeung & Tseng 2004). In this manner, access to the first node in a chain will automatically cache the next few nodes, which can improve cache usage. The application of node clustering to the standard-chain hash table yields a *clustered-chain* hash table (Askitis 2007).

Similar techniques that relocate or group data to improve cache utilization include virtual cache lines (Rubin, Bernstein & Rodeh 1999), customized memory allocators (Truong, Bodin & Seznec 1998, Kistler & Franz 2000, Berger, Zorn & McKinley 2002), and memory relocation techniques (Calder, Krintz, John & Austin 1998, Lattner & Adve 2005, Chilimbi & Shaham 2006). A similar node clustering technique known as hash buckets was applied to the two chained hash tables used to perform the hash-join algorithm in SQL (Graefe, Bunker & Cooper 1998). Another variant combines hash buckets with software prefetching and multi-threaded access to further speed up hash-join (Garcia & Korth 2006). Ghoting et al. also proposed a node clustering technique to improve the cache performance of the FP-tree, which is used in frequent-pattern mining (Ghoting, Buehrer, Parthasarathy, Kim, Nguyen, Chen & Dubey 2006). In this approach, the original pointer-based FP-tree is copied into a static contiguous block of memory which represents a cache-conscious FP-tree.

Askitis and Zobel advanced the idea of clustering by replacing the linked lists used by the standard-chain hash table with dynamic arrays, forming an *array* hash table (Askitis & Zobel 2005). In this manner, strings are stored contiguously in memory eliminating nodes and pointers which allows for high reductions in both cache misses and space usage—speed gains of up to 97% with around a 70% simultaneous reduction in space was reported over standard chaining. The dynamic array techniques were also applied to the string burst trie (Heinz, Zobel & Williams 2002) and binary search tree (Knuth 1998), yielding superior cache-conscious variants known as the *array* burst trie and *array* BST (Askitis 2007, Askitis & Zobel 2008*b*).

We can eliminate the use of chains and arrays altogether by storing homogeneous keys such as 32-bit or 64-bit integers directly within hash slots, forming an open-address hash table (Peterson 1957). A collision resolution scheme is still required, however, with the simplest being *linear probing*; the hash table is scanned from left to right (beginning from a hash slot selected by the hash function) until a vacancy is found (Peterson 1957). Instead of scanning, heuristics can be used to guess the location of a vacant slot. This is known as *double hashing*; another hash function independent of the first is used to skip to a location that may be vacant. This approach requires, on average, fewer probes than linear probing (Knuth 1998). A similar approach called *quadratic probing* guesses the next available hash slot (from left-to-right) by using a quadratic function. Another is a *reordering scheme* (Brent 1973) which moves items around to reduce expected probe costs. Alternative techniques include coalesced chaining, which allows lists to coalesce to reduce memory wasted by unused slots (Vitter 1983), and to combine chaining and open-addressing, known as pseudo-chaining (Halatsis & Philokyprou 1978).

Linear probing and double hashing were surveyed and analyzed by Munro and Celis (Munro & Celis 1986), and were found to offer poor performance compared to a chained hash table when the load factor (i.e., keys/slots) approached 1. In addition, since keys are stored directly within slots, the total number of keys supported by an open-address hash table is bounded by the number of slots and their capacity. Resizing an open-address hash table to accommodate more keys can be an expensive and space-consuming process (Askitis 2007, Kirsch et al. 2008). As a consequence, open-address schemes are typically best suited for applications where the total number of keys is known in advance, or when keys can be evicted (Cieslewicz & Ross 2007). Linear probing and double hashing have recently been investigated in the context of CPU cache (Heileman & Luo 2005), with linear probing found to be the overall efficient option as a result of its better use of cache.

Cuckoo hashing (Pagh & Rodler 2004) is a relatively new open-address hash table that uses two hash tables and two hash functions to resolve collisions—often a single open-address hash table is divided equally to form the two independent hash tables required. A key is stored in either one of the hash tables, but not both. When a collision occurs in the first hash table (using the first hash function), the key occupying the slot is evicted and replaced by the new key. The evicted key is then hashed using the second hash function and stored in the second hash table. If this causes another collision, then the eviction process repeats until all keys are hashed without collision.

Cuckoo hashing offers a constant worst-case cost during search but requires the load factor be kept below 0.49 (Pagh & Rodler 2004, Ross 2007). This implies that over half of the slots are to be kept empty. If the load factor is increased beyond this threshold, then an insertion will likely fail due to an irresolvable collision (Pagh & Rodler 2004). Fotakis et al. proposed a *d*-ary cuckoo hashing scheme which employs a set of independent hash functions to probe the hash table (Fotakis et al. 2003). The authors showed that by employing four independent hash functions, each assigned to a different region of the hash table, a load factor of approximately 0.97 can be achieved before the first irresolvable collision occurs.

A simple variant of cuckoo hashing known as bucketized cuckoo hashing allows hash slots to contain up to a fixed number of keys. With a slot capacity of two keys, for example, a load factor of 0.89 can be achieved before the first irresolvable collision occurs (Panigrahy 2005, Dietzfelbinger & Weidling 2007). Erlingsson et al. studied the combination of *d*-ary cuckoo hashing and bucketized cuckoo hashing (Erlingsson, Manasse & Mcsherry 2006), and showed that two independent hash functions and a slot capacity of up to four keys can yield good performance and space utilization.

Kirsch et al. proposed the use of a *stash*, a simple data structure independent of the cuckoo hash table that is used to store keys that cause irresolvable collisions. The use of a stash can greatly improve the failure probability bounds of insertion, and given knowledge of the total number of keys inserted, only a constant amount of additional space is required (Kirsch et al. 2008). Additional variants of cuckoo hashing include one that is engineered for use in hardware (Kirsch & Mitzenmacher 2008) and history-independent hashing (Naor, Segev & Wieder 2008). Kutzelnigg (Kutzelnigg 2008) analyzed the performance of an asymmetric cuckoo hash table (Pagh & Rodler 2004) (where the two hash tables used contain a different number of slots). However, this variant was found to increase the probability of insertion failure.

Zukowski et al. studied the application of a non-bucketized cuckoo hash table in database operations such as aggregation and join. The authors proposed a simple variant that eliminates the *if-then-else* branches used during search, which can improve lookup performance relative to a chained hash table (Zukowski et al. 2006).

Similarly, Ross explored the application of a bucketized cuckoo hash table for database operations such as aggregation (Ross 2007), and proposed combining a bucketized cuckoo hash table with *d*-ary cuckoo hashing, branch elimination and the use of SIMD operations (single instructions, multiple data), to further speed-up processing of hash slots in database operations. This variant of cuckoo hashing, also known as a splash table (Ross 2007), was shown to offer superior space utilization and performance with both small and large hash tables. Furthermore, unlike the variants of cuckoo hashing previously discussed, the splash table can efficiently support payload data (a counter or a pointer that is associated with every homogeneous key inserted).

The cache-sensitive search tree is another data structure that can offer efficient sorted access to integers (Rao & Ross 1999). It is a binary search tree (with nodes sized to match the cache-line) that is built on top of an existing static array of sorted integers. As a consequence, however, it can not be updated efficiently. The adaptive trie is another cache-efficient data structure for strings and integers (Acharya, Zhu & Shen 1999), but was shown to be slower and more space-intensive against other data structures such as the array burst trie and variants of binary search tree, particularly under skew access—due to the overhead of maintaining adaptive nodes (Askitis 2007, Crescenzi, Grossi & Italiano 2003). Similarly, the Judy trie (Silverstein 2002) can also maintain sorted access to integers, but was shown to offer poor performance against alternative trie-based data structures, particularly under skew access (Askitis 2007, Askitis & Zobel 2008*b*).

Nash et al. studied the performance of several integer-based data structures that can efficiently retrieve integers in sort order (Nash & Gregg 2008). The authors showed that a variant of burst trie (Heinz et al. 2002, Askitis 2007) that is designed for integers offers the best performance. However, their focus was solely on sorted data management. As a result, the authors did not explore the performance of integer-based hash tables.

Cache-oblivious data structures attempt to perform well on all levels of the memory hierarchy, including disk, without knowledge of the characteristics of each level (Kumar 2003). Some examples include the cache-oblivious string B-tree (Bender, Farach-Colton & Kuszmaul 2006) and a cache-oblivious index for approximate string matching (Hon, Lam, Shah, Tam & Vitter 2007), but of which often assume uniform distribution in data and operations (Bender, Demaine & Farach-Colton 2002, Askitis & Zobel 2008*a*). An recent implementation of a cache-oblivious algorithm involving matrix calculations and homogeneous keys was shown to offer inferior performance, compared to alternative algorithms that were tuned to exploit CPU cache (Yotov, Roeder, Pingali, Gunnels & Gustavson 2007). The data structures that we consider in this paper reside solely within volatile main memory and are not cache-oblivious.

3 Implementing Hash Tables for Integers

Array hash table. The array hash table can be easily adapted to store, delete, and retrieve fixed-length keys, namely 32-bit or 64-bit integers. In this paper, we only consider 32-bit integers but the algorithms we describe below are readily adaptable to 64-bit integers.

To search for an integer (a key) in an array hash table, we first hash the key to acquire a slot. If the slot is empty, then the key does not exist in the hash table and the search fails. Otherwise, we scan the array acquired, a key at a time, until a match is found or until we exhaust the array—in which case, the search fails. On successful search, the key and its payload data are not moved to the front of the array due to the high computational costs involved (Askitis 2007); as we show in later experiments, cache-efficiency more than compensates.

When a search fails, we can then insert the key into the hash table. Insertion proceeds as follows: if the slot was empty, a new 12 byte array is allocated and assigned. The first 4 bytes in the array are reserved to store the number of entries (which is initialized to 1). The next 8 bytes store the key (4 bytes) followed by its payload data, a 4 byte integer. Otherwise, we resize the existing array by 8 bytes (4 bytes for the key and 4 bytes for its payload data) using the *realloc* system call. The key and its payload are then appended to the array and the number of entries in the array is incremented by 1, completing the insertion process.

Deletion proceeds in a similar manner: the key is hashed, the acquired array (if any) is searched and assuming the key is found, we overwrite the key and its payload data by sliding the remaining keys (if any) with their payload data one entry towards the start of the array. The number of entries in the array is then decremented by 1. The array can then be resized using the *realloc* system call to shrink the array by 8 bytes. Figure 1 shows an example of an array hash.

As shown in Figure 1, we interleave keys with their payload data. Alternatively, we can separate keys from their payload by storing the keys first followed by their payload data (in order of occurrence). However, preliminary experiments using this array configuration yielded no significant gains in performance.

Standard-chain hash table. Inserting, searching, or deleting a key in a standard-chain hash table is a relatively straight-forward task. The key to search or insert is first hashed to acquire a slot. If the slot anchors a linked list, the list is traversed until a match is found. If found, the

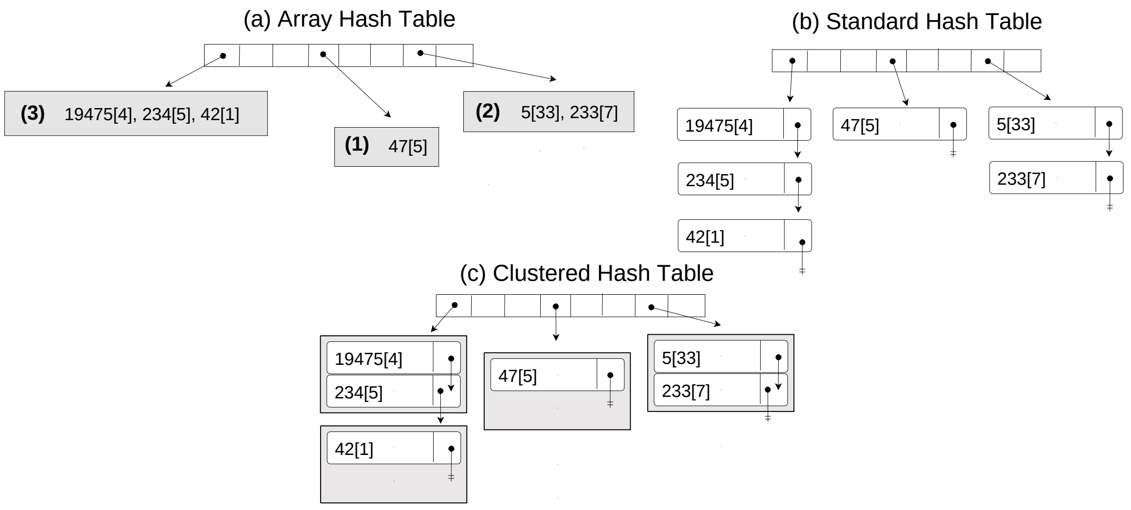

Figure 1: *Some integers (keys) along with their payload data (square brackets) are inserted into an array hash, a standard-chain hash, and a clustered-chain hash table. Each dynamic array starts with the no. of keys (in parenthesis).*

node containing the required key and payload is moved to the front of the list via pointer manipulation.

The key can only be inserted on search failure, in which case, the key (4 bytes) and its payload data (4 bytes) are encapsulated within a 12 byte node which contains a null 4-byte next-node pointer. The node is then appended to the end of the list completing the insertion process. Deletion proceeds in a similar manner: assuming that the key is found, its node is deleted and the list is re-arranged via pointer manipulation. Figure 1 shows an example of a standard-chain hash table.

Clustered-chain hash table. The standard-chain hash table is a simple and flexible data structure but it is not particularly cache-efficient, as nodes are likely to be scattered in memory. We can address this issue by clustering nodes (Chilimbi 1999). That is, each slot anchors a block of memory that is sized to match the CPU cache-line (64 bytes in our case). Searching for, or inserting a key (along with its payload data) proceeds as described for the standard-chain, except that during insertion, a node is appended to the acquired block. If the block is full, we store the node in a new empty block that is linked to the old. This effectively forms a linked list of blocks which should improve spatial access locality and should also reduce the impact of pointer-chasing (Askitis 2007). Figure 1 shows an example of a clustered-chain hash table.

Details regarding deletion in a clustered-chain are generally not available in the literature (Chilimbi 1999). To delete a key in a clustered-chain, we must overwrite it and its payload by sliding all remaining keys (and payloads) up one entry towards the start of the chain. However, this can involve accessing and manipulating keys from several blocks which can be expensive (Askitis 2007).

We can pre-allocate space prior to building the standard or clustered-chain hash table which will reduce the computational cost of allocating nodes (or blocks), but not their cache-efficiency (Askitis 2007). We can also use a cache-efficient memory allocator such as *Ialloc* (Truong et al. 1998), which attempts to interleave nodes to improve cache-line utilization. However, Askitis demonstrated that both pre-allocation and *Ialloc* are ineffective for the chained hash table (Askitis 2007). Berger et al. also reported similar results after comparing the performance of pointer-intensive programs using several custom memory allocators (Berger et al. 2002). The general-purpose *malloc* allocator—sometimes known as the Doug Lea allocator which is provided by most *gcc* compilers—was found to offer superior performance in the majority of cases. Compiler based techniques (Calder et al. 1998)—which attempt to reallocate stack and heap data to better utilize cache—can also be applied, but often require program profiles and are therefore not particularly effective for dynamic data structures (Askitis 2007).

The clustered-chain hash table can also be optimized to exploit SIMD instructions (single instruction, multiple-data). On initial access to a block of nodes, we can copy its keys and their payloads into SIMD registers. This will allow the set of keys in the block to be compared against a key using a single instruction. However, exploiting SIMD instructions in a clustered-chain hash table (and also in an array hash) resulted in poor overall performance. Ross also observed inferior performance after attempting to exploit SIMD instructions in a chained hash table (Ross 2007).

Bucketized cuckoo hashing. We implement bucketized cuckoo hashing where each hash slot is a *bucket* that can store up to four keys and their payload data. A bucket is structured as an array of eight 32-bit integers. The first four store keys while the remaining four store their payload data, in order of occurrence. This bucket structure and its capacity were shown to offer the best performance (with respect to both time and space) for bucketized cuckoo hashing (Ross 2007, Erlingsson et al. 2006). We assume that zero is an invalid payload value. This will allow us to efficiently count the number of keys stored in a bucket. Alteratively, we could reserve the first 32-bit entry (to maintain word-alignment) in each bucket as a counter, but at a cost in space (Ross 2007). An example of a bucketized cuckoo hash table is shown in Figure 2.

To search for a key (i.e., the query key), we hash it using the first hash function. We then access the required bucket from the first hash table and compare its keys against the query key. If the query key is not found, we hash it again using the second hash function to acquire a bucket from the second hash table. We search the second bucket and if the query key is found, we return its payload data. Otherwise, the search is unsuccessful. Zukowski et al. suggests deriving the second hash value from the first by shifting it by *n* bits to the right (Zukowski et al. 2006). However, we found this approach to be unsuitable for a large number of keys, due to poorer distributions of keys to slots which as a consequence, led to poorer overall performance.

Deletion proceeds in a similar manner. We search for the query key as described, and if found, we remove it and its payload from the acquired bucket by sliding any remaining keys (along with their payload data) one entry towards the start of the array. If the last key is deleted, we

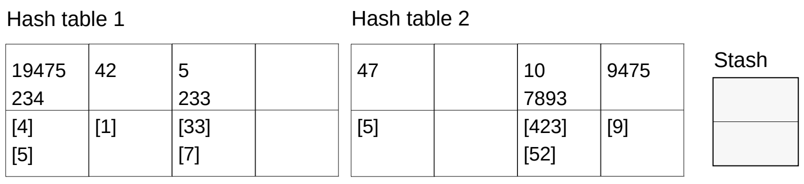

Figure 2: *Some integers (keys) along with their payload data (in square brackets) are inserted into a bucketized cuckoo hash table with two hash functions and a bucket capacity of two. The stash stores keys that cause irresolvable collisions.*

simply set its payload value to zero.

It is possible to parallelize the search phase by processing both buckets at once by using threads (Pagh & Rodler 2004). However, since buckets are small in size (they can fit entirely with a cache-line), they can be processed rapidly. As a consequence, the overhead of creating and synchronizing threads is likely to impact overall performance (Askitis 2008). Leading implementations of bucketized cuckoo hashing have instead relied on careful coding to exploit compiler optimizations, hardware-level parallelism and out-of-order execution of instructions (Zukowski et al. 2006, Ross 2007, Kirsch et al. 2008).

We implement search by manually unrolling the loop used to compare the keys in a bucket. We attempted to eliminate branches (i.e., the *if* statements) used to compare keys (Zukowski et al. 2006). However, because buckets can store more than one key with payload data that is returned if a key matches the query key (two properties not considered by Zukowski et al.), we found that manual loop unrolling combined with compiler optimizations yielded the best results. Similarly, the use of SIMD instructions during search yielded no improvements compared to our original code. One issue regarding the use of SIMD instructions is that we must first load the keys of a bucket into SIMD register(s). This would be beneficial if the bucket is accessed multiple times during search, which can occur in database operations (Ross 2007). However, in our case, with an unpredictable input sequence of keys, buckets can be accessed at random and are not re-used during the search phase, hindering the effectiveness of SIMD.

When the search ends in failure, we can insert the query key by appending it (and its payload) to the bucket acquired from the first hash table. If the bucket is full, we then check if the bucket from the second hash table can accommodate. If so, the key and its payload data are appended, completing the insertion. Otherwise, we evict the oldest key from either the first or second bucket. Since keys and their payload data are added in order of occurrence, the first key in a bucket is the oldest. The remaining keys (and their payload data) are moved one entry towards the start of the bucket to allow the new key and its payload to be appended. We then take the evicted key (storing its payload in a temporary space), and hash it using the two independent hash functions to acquire a bucket from both hash tables. We then attempt to re-insert the evicted key (with its payload) as described above.

It is possible for the eviction process to loop indefinitely causing an irresolvable collision. We allow the eviction process to repeat 1000 times before we declare the insertion as a failure. If an insertion fails, we can attempt to re-hash all of the keys using two different hash functions (Pagh & Rodler 2004), or we can double the number of slots available, say, and use the same hash functions to re-hash the keys. Both options are, however, expensive particularly with a large number of keys (Kirsch et al. 2008).

We implement two variants of bucketized cuckoo hashing. Our first does not handle insertion failures—we simply halt the construction of the hash table. Our second includes a *stash* (Kirsch et al. 2008), an auxiliary data structure that stores the keys (and their payload) that cause irresolvable collisions. The stash, in our case, is an array hash table with 8192 slots that uses a simple modulo (a mask) as a hash function.

4 Experimental Design

We compare the performance of the hash tables described in Section 3 by measuring the elapsed time (in seconds), the space required (in megabytes) and the actual cache performance incurred for the task of inserting, searching, and deleting a large set of integers. We also compare against a simple linear probing hash table as discussed in Section 2, which is known to be both compact and cache-efficient (Heileman & Luo 2005). Our measure of time was averaged over a sequence of ten runs (we flooded main memory after each run with random data to flush system caches). Our measure of space includes an estimate of the operating system overhead (16 bytes) imposed per memory allocation request, which we found to be consistent with the total memory usage reported in the `/proc/stat/` table.

We used PAPI (Dongarra, London, Moore, Mucci & Terpstra 2001) to measure the actual number of L2 cache misses and CPU cycles incurred during search. The cache performance of insertion and deletion was found to be similar and was thus omitted due to space constraints. Unfortunately, we could not measure TLB misses as our primary machine lacked a working TLB hardware counter; simulating TLB misses using *valgrind* (available online) is currently not an option. Our measure of CPU cycles, however, compensates to some extent, as it includes the CPU cycles lost from both L2 and TLB misses.

Our primary machine was an Intel E6850 Core 2 Duo running a 64-bit Linux Operating System (kernel 2.6.24) kept under light load (single user, no X server); the machine had 4GB of generic DDR2 1066 RAM with 4MB of L2 cache and 64-byte cache-lines. We also tested the data structures on alternative architectures: a 1.3Ghz UltraSPARC III, a 700Mhz Pentium III, and a 2.8Ghz Pentium IV and observed similar results which we report in our full paper.

Our first dataset was derived from the complete set of words found in the five TREC CDs (Harman 1995). The dataset is therefore highly skew, containing many hundreds of millions of words but of which only around a million are distinct. We converted (up to) the first 4 bytes of each word, in order of occurrence, into a 32-bit unsigned integer. This generated our `skew` dataset which contains 752 495 240 integers (approx. 3GB), but which only 48 283 are distinct. Our second dataset labeled `distinct` was created by randomly generating (from a memory-less source) 60 million distinct unsigned integers between the range of 0 to 2^{32}. Satellite data associated with each key was set to a random (non-zero) number.

We varied the number of slots used by the hash tables from 2^{15}, doubling up to 2^{26}. We employed the same multiplicative-based (64-bit) hash function with a mask as a modulo[1] in all of our hash tables, which we found to be both fast and effective. We also used the same hash function for our implementations of bucketized cuckoo hashing, but derived two independent hash functions by using two different hash seeds (set to large prime values). Our bucketized cuckoo hash tables implement two independent hash tables by evenly dividing the number of slots allocated to a single open-address hash table.

We are confident after extensive profiling that our data structures (implemented in C) are of high quality and are available for study upon request[2]. We compiled the data structures using *gcc* version 4.3.0 with all optimizations enabled. We discuss the results of our experiments below.

5 Results

Skewed data. Figure 3 shows the time and space required to build and then search the hash tables respectively, using our skew dataset. Despite its effort to exploit cache, the clustered-chain hash table was only slightly faster to build and search relative to the standard-chain hash table, when under heavy load. With only 2^{15} slots for example, the clustered-chain hash table required about 18.13s to build and search, whereas the equivalent standard-chain required about 18.19s to build and search. These results may be surprising at first, since we would normally expect better performance from a hash table that uses cache-efficient linked lists to resolve collisions. However, our results do coincide with those involving a clustered-chain hash table and string keys (Askitis 2007).

The reason why the clustered-chain was generally slower in these experiments was due to a lack of efficient support for move-to-front on access. Move-to-front is beneficial under skew access as it can reduce the cost of accessing frequent keys. In a standard-chain, moving a node to the start of the list requires only pointer manipulation which is computationally efficient. The impact on performance becomes more apparent as we decrease the load factor (i.e., by increasing the number of slots), in which case, the standard-chain hash table becomes noticeably faster to build and search than the clustered-chain. Pointer manipulation wont work in a clustered-chain as it can violate the principle of node clustering. In order to implement move-to-front in a clustered-chain, we must explicitly move nodes in and amongst blocks which can be expensive (Askitis 2007).

Comparing the L2 and CPU cycle costs incurred during search shown in Figure 3, we observe consistent results to our timings. The clustered-chain hash table incurs slightly fewer L2 misses per search but only under heavy load. However, the clustered-chain also consumes more CPU cycles relative to the standard-chain due to the absence of move-to-front on access. As we reduce the average load factor (by adding more slots), the standard-chain begins to rival the cache-efficiency of the equivalent clustered-chain; and coupled with fewer CPU cycles per search as a result of move-to-front on access, the standard-chain offered overall superior performance.

Building an array hash table, in contrast, is clearly more expensive than a standard or clustered-chain hash table, but is more space-efficient due the elimination of nodes and pointers. The array hash is more expensive to build due to the high computational cost involved with resizing arrays. We observe this fact by comparing the time required to build against the time required to search. During search, arrays are not resized, and as a result, the array hash table is consistently faster than the standard and clustered-chained hash tables. The array hash does not employ move-to-front on access due to the high computational costs associated with moving keys (Askitis & Zobel 2005). Nonetheless, our results show that cache-efficiency more than compensates.

The high costs observed with building the array hash table is also caused by the small number of insertions made. In these experiments only 48 283 keys were inserted. The remaining 752 446 957 keys were duplicates. As we increase the number of keys inserted, the computational cost of resizing arrays will be compensated by a high reduction in cache misses (Askitis 2007). We demonstrate this fact in a later experiment involving our distinct dataset.

The bucketized cuckoo hash table was found to be the slowest hash table to build and search under skew access. At best—with 2^{18} hash slots—the bucketized cuckoo hash table (without a stash) consumed about 8.39 megabytes of space and required 24.09s to build and 21.57s to search. The equivalent array hash table required only 3.37 megabytes of space and 20.47s to build and 17.68s to search—which is up to 19% faster. Similarly, the clustered and standard-chain hash tables also offered consistently superior performance relative to bucketized cuckoo hashing. The equivalent standard-chain, for example, consumed only 3.64 megabytes of space and required around 18.88s to build and 18.78s to search. The use of a stash in bucketized cuckoo hashing was of no benefit in these experiments and was thus omitted, as no irresolvable collisions occurred (hence, the stash was never accessed).

The results of L2 cache misses and CPU cycle costs during search were consistent to our timings, with the bucketized cuckoo hash table incurring the most cache misses and consuming the most CPU cycles. Under heavy load, the bucketized cuckoo hash table rivaled the cache-efficiency of the equivalent array hash, but so did the chained hash tables; they were all small enough to remain cache resident.

Linear probing was consistently the fastest hash table to build and search in these experiments when given enough space—a minimum of 2^{16} slots. The L2 cache misses and CPU cycle costs measured were also consistent with our timings, with linear probing incurring the fewest cache misses per search and simultaneously consuming the least amount of CPU cycles. Our results also coincide with previous research that reported linear hashing to a cache-efficient open-address hash table (Heileman & Luo 2005).

Although linear probing requires at least one available slot per key inserted, no buckets, linked lists, or arrays are used to resolve collisions. Hence, each hash slot occupies just 8 bytes of memory (for a key and its payload). As such, in addition to its high performance under skew access, linear probing is relatively space-efficient. With a just 2^{16} slots for example, linear probing consumed only 0.52 megabytes of space and required 15.59s to build and 16.25s to search. The equivalent array hash table consumed 1.59 megabytes of space and required 20.08s to build and 17.41s to search. Similarly, the equivalent bucketized cuckoo hash table consumed 2.10 megabytes of space and required 25.17s and 23.11s to build and search, respectively.

Despite its impressive performance, linear probing has poor worst-case performance and as a consequence, it can become a slow hash table to use. Consider the following experiment where we build the hash tables using our distinct dataset and then search for all of the keys in our skew dataset. The difference in this experiment—despite the increased size of the hash tables—is that search failures will occur. From the 752 495 240 searches made, only 21 262 337 are successfully found; the rest incur the full cost of traversal. With 2^{26} slots, the linear probing hash table required a total of 70.93s. The equivalent array hash required only 26.20s while the bucketized cuckoo hash table (with its constant worst-cost probe cost) required

[1]http://www.concentric.net/ Ttwang/tech/inthash.htm
[2]http://www.cs.rmit.edu.au/~naskitis/

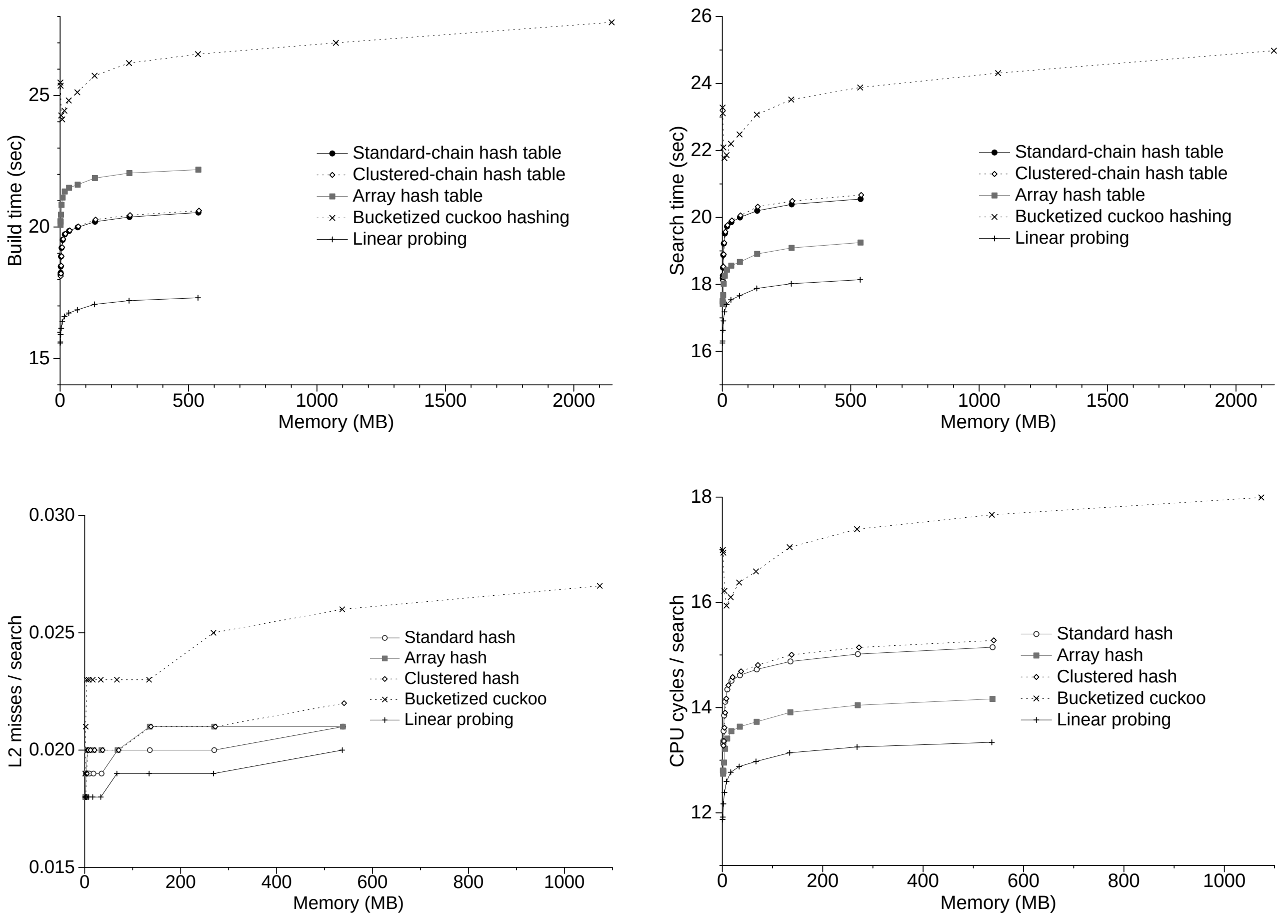

Figure 3: *The time and space (top graphs) required to build and search the hash tables using our* `skew` *dataset. The actual L2 and CPU cycles incurred during search is also shown (bottom graphs). The plots represent the number of slots starting from* 2^{15} *(left-most), doubling up to* 2^{26}.

33.15s—53% faster than linear probing.

With a load factor of about 0.9, searching for a key that does not exist in a linear probing hash table will likely involve scanning and comparing a large proportion of the hash table, which will consume a surplus of CPU cycles. We can reduce this cost by reducing the average load factor (by adding a surplus of empty slots) but at the expense of both space and overall performance, since fewer frequently accessed slots are likely to remain cache-resident. In contrast, when we traverse a slot in an array hash, we are only comparing keys that have the same hash value. As such, fewer keys are likely to be compared resulting in better use of cache and the consumption of fewer CPU cycles. Similarly, with bucketized cuckoo hashing, we process only two buckets per search. However, unlike scanning a single contiguous array of integers, these buckets are likely to be scattered in main memory and can therefore result in more cache misses (relative to the array hash) on initial access, which is reflected in our timings.

Distinct data. The time and space required to build and then search the hash tables using our `distinct` dataset is shown in Table 1. With no skew in the data distribution, move-to-front on access is rendered ineffective. With only 2^{15} slots, for example, the clustered-chain hash table required about 2194s to be built and 1.2 gigabytes of space. The equivalent standard-chain required over 6000s and consumed over 1.9 gigabytes—which is 63% slower and around 37% more space-intensive than the clustered-chain hash table. The standard-chain was slower than the clustered-chain because 60 million distinct keys were inserted. As a result, longer linked lists, on average, were traversed during search and with no skew each key (node) has equal probability of access. As a consequence, an excessive number of L2 cache-misses were incurred by the standard-chain. Hence, with no skew in the data distribution, node clustering is indeed an effective technique for improving the cache utilization of a chained hash table (Chilimbi 1999).

Despite the substantial performance gains offered by the clustered-chain it still remained, in all cases, inferior to the array hash. With only 2^{15} slots, for example, the array hash required 116s to be built and about 85s to search while simultaneously consuming approximately 480 megabytes of space. This is an improvement in speed of about 95% relative to the equivalent clustered-chain, with a simultaneous reduction in space of about 60%. These results also demonstrate that despite the high computational costs incurred from frequent array resizing, cache efficiency more than compensates allowing the array method to be more efficient in practice (Askitis 2007). These results also highlight that the array hash is a *scalable* hash table—the cost of inserting or searching a key increases gracefully as the number of keys (or load factor) increase. The standard and compact-chained hash tables can only rival the performance of the array hash once given a surplus of space.

The bucketized cuckoo hash table (without a stash) displayed competitive performance but required a minimum of 2^{24} slots to prevent irresolvable collisions. At best (with 2^{24} slots), the bucketized cuckoo hash table consumed 536.8 megabytes of space and required 13.5s and 8.3s to build and search, respectively. This is considerably more space-efficient than the equivalent standard-chain, clustered-chain, and the array hash table which could only rival in speed once given the same number of slots. With

Table 1: *Time (sec) and space (MB) required to build and then search the hash tables using our* `distinct` *dataset.*

Num	**Standard hash**			**Clustered hash**			**Array hash**		
slots	Build (s)	Search (s)	Space (MB)	Build (s)	Search (s)	Space (MB)	Build (s)	Search (s)	Space (MB)
2^{15}	6112.2	6106.8	1920.2	2194.9	2196.5	1201.2	116.6	85.6	480.9
2^{16}	3092.6	3085.0	1920.5	1109.1	1110.2	1202.4	71.6	50.2	481.8
2^{17}	1560.9	1555.5	1921.0	561.3	563.7	1204.9	46.2	32.3	483.6
2^{18}	784.4	780.7	1922.1	285.3	285.5	1209.9	33.5	23.3	487.3
2^{19}	397.3	394.9	1924.1	147.2	146.5	1219.9	26.7	18.4	494.6
2^{20}	203.0	201.0	1928.3	78.5	76.4	1239.8	22.3	15.3	509.3
2^{21}	106.2	104.3	1936.7	43.9	41.0	1279.6	19.8	13.3	538.7
2^{22}	57.6	55.6	1953.5	26.7	22.6	1359.3	18.3	11.7	597.4
2^{23}	33.4	31.1	1987.1	18.1	12.7	1518.7	16.7	9.7	714.7
2^{24}	21.3	18.3	2054.2	14.1	8.3	1841.4	15.0	8.0	940.6
2^{25}	15.4	11.2	2188.4	12.5	7.1	2602.0	13.1	7.1	1308.6
2^{26}	12.5	7.3	2456.8	12.5	8.8	3734.7	11.8	6.2	1813.6

Num	**Cuckoo hash**			**Cuckoo hash + stash**			**Linear probing**		
slots	Build (s)	Search (s)	Space (MB)	Build (s)	Search (s)	Space (MB)	Build (s)	Search (s)	Space (MB)
2^{23}	–	–	–	1821.1	72.7	480.2	–	–	–
2^{24}	13.5	8.3	536.8	13.7	8.4	536.9	–	–	–
2^{25}	14.1	9.3	1073.7	14.3	9.3	1073.8	–	–	–
2^{26}	15.1	10.3	2147.4	15.5	10.3	2147.5	5.1	4.7	536.8

2^{24} slots, for example, the array hash consumed 940.6 megabytes of space and required about 15s to build and 8s to search.

Increasing the number of slots available to the bucketized cuckoo hash table yields poorer performance relative to the equivalent chained and array hash tables, as fewer buckets are likely to remain cache-resident; this reduction in locality impacts bucketized cuckoo hashing more than the equivalent chained and array hash tables, since up to two random buckets are accessed during search. In addition, by doubling the amount of space used, more TLB misses are likely to occur in bucketized cuckoo hashing as fewer virtual to physical memory translations are likely to be stored within the TLB (Askitis 2007).

With a stash enabled, we can safely reduce the number of slots available to the bucketized cuckoo hash table as shown in Table 1. With 2^{23} slots, 26 449 132 keys out of the 60 million were stored in the stash. The stash, however, is not intended to maintain such a large volume of keys. As a result, the overall space consumed by the bucketized cuckoo hash table drops to around 480 megabytes, since almost half of the keys are stored in an auxiliary array hash that uses only 8192 slots (hence, the space overhead incurred from slot pointers and allocating dynamic arrays is tiny). However, performance is severely jeopardized as a result. We can reduce the time required to build and search by increasing the number of slots in the stash, but at the expense of space.

Linear probing required at least 2^{26} slots to operate in these experiments, since 60 million distinct keys were inserted. However, as each slot is only 8 bytes long, the total space consumed was just over double the space required by our `distinct` dataset—536 megabytes. This is still reasonably space-efficient when compared to the array hash table, which can only rival in space-efficiency when given less than 2^{21} slots. In all, even with a load factor of 0.9 and with no skew in the data distribution, linear probing offered the fastest insertion and search. Bucketized cuckoo hashing can match the space-efficiency of linear probing with 2^{24} slots, but remained around twice as slow. However, as we highlighted during our previous skew search experiments, linear probing can be expensive in the worse-case which occurs when we search for a large batch of keys that are not found. We demonstrate this in the next section that involves key deletion.

Deletion. For our next experiment we build the standard-chain hash table, the array hash, the linear probing hash table, and the bucketized cuckoo hash table (without a stash) using our `distinct` dataset. We then measure the time required to delete 30 million integers selected at random from our `distinct` dataset. Under heavy load (with 2^{16} slots), the standard-chain hash table required 478.8s to delete the random batch of keys. The equivalent array hash, in contrast, needed only 57.3s. We observed consistent results as we reduced the average load factor (by adding more slots). With 2^{23} slots for example, the standard-chain hash required 11.4s whereas the equivalent array hash needed only 8.5s. When we delete a random key from the standard-chain hash table, a linked list is likely to be traversed which will incur cache misses—the cost of which will out-weigh the overall cost of array scanning and resizing.

Bucketized cuckoo hashing required only 4.8s to delete the random batch of keys using 2^{25} slots. Its speed, however, is due to the fact that no real deletion took place. Since hash slots are represented as fixed (cache-lined) sized buckets, a key and its payload data are deleted by either overwriting or by setting the payload value to zero. Hence, unlike the standard-chain or array hash tables, no memory is actually freed—no time is lost to system calls (i.e., to free a node) or to resize an array. Similarly, deleting a key in a linear probing hash table involves flagging the key and its payload as having been deleted. As a result, with 2^{26} slots, linear probing required only 1.8s to delete the random batch of keys.

Although the array hash table was shown to be superior to the standard-chain for random key deletion, it can be slower in the worse-case which occurs when we continually delete the first key from any slot. This process is expensive because we have to scan and resize the entire array. In a linked list, we only need to delete the first node and reassign its pointers. With 2^{16} slots, the standard-chain hash required only 2.1s to delete 60 million integers (our entire `distinct` dataset), all of which were found at the start of each slot. The equivalent array hash table needed over 90s due to the high computational costs involved with array scanning and resizing. This worst-case cost can be reduced if we resize arrays periodically, but at the expense of maintaining unused space within arrays (Askitis & Zobel 2005). Similarly, by relaxing some constraint on space-efficiency, we can effectively eliminate this worst-case cost altogether by employing a type of lazy deletion. We flag the key and its payload as having been deleted, and optionally resize the array only when, say, it becomes half empty.

In our final experiment we build our set of hash tables using the `distinct` dataset. We then generate 100 million unsigned integers at random (from a memory-less source) to form a new batch of keys to delete. Unlike the previous batch, only 2 731 980 keys are found in the hash tables. The remaining 97 268 020 keys are not present in our `distinct` dataset and only 63 878 in total are duplicates. With 2^{20} slots, the array hash table required about 32.3s to delete the skew batch of keys. The equivalent standard-chain hash table required over 686.1s. We observed consistent results as we reduced the load factor by adding more slots, with the standard-chain (using 2^{25} slots) at 20.7s and the equivalent array hash at 13.4s. The standard-chain is more expensive because the majority of searches end in failure, thereby incurring the full cost of traversal (traversing a linked list will incur more cache misses than scanning its equivalent array). Move-to-front on access is also rendered ineffective since the majority of keys are distinct.

Though not as fast as the array hash, with 2^{25} slots, the bucketized cuckoo hash table required only 17.1s to delete the skew batch of keys. Bucketized cuckoo hashing offers a constant worst-case probe cost since at most only two buckets are processed. However, these buckets are unlikely to be physically located close to each other and as a consequence, can incur more cache misses (and TLB misses) on initial access compared to scanning and resizing a single array. As expected, linear probing was slower than the array hash and the bucketized cuckoo hash table, requiring 22.8s with 2^{26} slots due to the excessive number of hash slots accessed.

6 Conclusion

We have described how to efficiently implement a cache-conscious array hash table for integer keys. We then experimentally compared its performance against two variants of chained hash table and against two open-address hash tables—linear probing and bucketized cuckoo hashing—for the specific task of maintaining a dictionary of integer keys (with payload data) in-memory.

When we can afford to allocate as many hash slots as the number of distinct keys inserted (with a surplus of empty slots), linear probing is by far the fastest hash table with or without skew (duplicate keys) in the data distinction. Linear probing, however, has poor worst-case performance which can jeopardize its efficiency as a dictionary. The array hash table was shown to be a better option in this case.

Despite a constant worst-case probe cost, bucketized cuckoo hashing was consistently slower than the array hash to build, search, and delete keys with a skew distribution. The bucketized cuckoo hash table could only rival the performance of the array hash when under heavy load and with no skew in the data distribution. However, unlike linear probing or bucketized cuckoo hashing, the array hash can operate efficiently (that is, it does not need to be resized) with a load factor greater than 1, making it a more scalable option.

The array hash was also among the fastest hash tables to delete random keys, but it can become slower than both a chained hash table and bucketized cuckoo hashing in a specific case involving key deletion. Yet with some relaxation on space-efficiency, we can employ simple lazy deletion schemes with periodic array resizing to effectively eliminate this worst-case deletion cost, making the array hash an ideal option for use as a dictionary.

References

Acharya, A., Zhu, H. & Shen, K. (1999), Adaptive algorithms for cache-efficient trie search, *in* ‘ALENEX’, pp. 296–311.

Askitis, N. (2007), Efficient Data Structures for Cache Architectures, PhD thesis, RMIT University. RMIT Technical Report TR-08-5. http://www.cs.rmit.edu.au/˜naskitis.

Askitis, N. (2008), CORE-hash: A cache and core conscious string hash table, Technical report. RMIT University, TR-08-8. Manuscript in submission.

Askitis, N. & Zobel, J. (2005), Cache-conscious collision resolution in string hash tables, *in* ‘SPIRE’, pp. 91–102.

Askitis, N. & Zobel, J. (2008*a*), B-tries for disk-based string management, *in* ‘Int. Journal on Very Large Databases’. To appear.

Askitis, N. & Zobel, J. (2008*b*), ‘Redesigning the string hash table, burst trie, and BST to exploit cache’. RMIT University TR-08-04. Manuscript in submission.

Badawy, A. A., Aggarwal, A., Yeung, D. & Tseng, C. (2004), ‘The efficacy of software prefetching and locality optimizations on future memory systems’, *Journal of Instruction-Level Parallelism* **6**(7).

Bender, M. A., Demaine, E. D. & Farach-Colton, M. (2002), Efficient tree layout in a multilevel memory hierarchy, *in* ‘ESA’, pp. 165–173.

Bender, M. A., Farach-Colton, M. & Kuszmaul, B. C. (2006), Cache-oblivious string B-trees, *in* ‘PODS’, pp. 233–242.

Berger, E. D., Zorn, B. G. & McKinley, K. S. (2002), Reconsidering custom memory allocation, *in* ‘ACM OOPSLA’, pp. 1–12.

Brent, R. P. (1973), ‘Reducing the retrieval time of scatter storage techniques’, *Communications of the ACM* **16**(2), 105–109.

Calder, B., Krintz, C., John, S. & Austin, T. (1998), Cache-conscious data placement, *in* ‘Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems’, pp. 139–149.

Chilimbi, T. M. (1999), Cache-Conscious Data Structures–Design and Implementation, PhD thesis, Computer Sciences Department, University of Wisconsin-Madison.

Chilimbi, T. M. & Shaham, R. (2006), Cache-conscious coallocation of hot data streams, *in* ‘Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation’, pp. 252–262.

Cieslewicz, J. & Ross, K. A. (2007), Adaptive aggregation on chip multiprocessors, *in* ‘VLDB Conf.’, pp. 339–350.

Crescenzi, P., Grossi, R. & Italiano, G. F. (2003), Search data structures for skewed strings, *in* ‘Experimental and Efficient Algorithms: Second Int. Workshop, WEA’, pp. 81–96.

Dietzfelbinger, M. & Weidling, C. (2007), ‘Balanced allocation and dictionaries with tightly packed constant size bins’, *Theoretical Computer Science* **380**(1-2), 47–68.

Dongarra, J., London, K., Moore, S., Mucci, S. & Terpstra, D. (2001), Using PAPI for hardware performance monitoring on Linux systems, *in* ‘Proc. Conf. on Linux Clusters: The HPC Revolution’. *http://icl.cs.utk.edu/papi/

Erlingsson, U., Manasse, M. & Mcsherry, F. (2006), A cool and practical alternative to traditional hash tables, *in* 'Workshop on Distributed Data and Structures'.

Fotakis, D., Pagh, R., Sanders, P. & Spirakis, P. G. (2003), Space efficient hash tables with worst case constant access time, *in* 'Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science', pp. 271–282.

Garcia, P. & Korth, H. F. (2006), Database hash-join algorithms on multithreaded computer architectures, *in* 'Conf. CF', pp. 241–252.

Ghoting, A., Buehrer, G., Parthasarathy, S., Kim, D., Nguyen, A., Chen, Y. & Dubey, P. (2006), 'Cache-conscious frequent pattern mining on modern and emerging processors', *Int. Journal on Very Large Databases* **16**(1), 77–96.

Graefe, G., Bunker, R. & Cooper, S. (1998), Hash joins and hash teams in Microsoft SQL server, *in* 'Proc. Int. Conf. on Very Large Databases', pp. 86–97.

Halatsis, C. & Philokyprou, G. (1978), 'Pseudochaining in hash tables', *Communications of the ACM* **21**(7), 554–557.

Harman, D. (1995), 'Overview of the second text retrieval conf. (TREC-2)', *Information Processing and Management* **31**(3), 271–289.

Heileman, G. L. & Luo, W. (2005), How caching affects hashing, *in* 'Proc. ALENEX Workshop on Algorithm Engineering and Experiments', pp. 141–154.

Heinz, S., Zobel, J. & Williams, H. E. (2002), 'Burst tries: A fast, efficient data structure for string keys', *ACM Transactions on Information Systems* **20**(2), 192–223.

Hon, W.-K., Lam, T. W., Shah, R., Tam, S.-L. & Vitter, J. S. (2007), Cache-oblivious index for approximate string matching., *in* 'CPM', pp. 40–51.

Kirsch, A. & Mitzenmacher, M. (2008), The power of one move: Hashing schemes for hardware, *in* 'IEEE INFOCOM', pp. 106–110.

Kirsch, A., Mitzenmacher, M., & Wieder, U. (2008), More robust hashing: Cuckoo hashing with a stash, *in* 'To appear in Proceedings of the 16th Annual European Symposium on Algorithms (ESA)'. http://www.eecs.harvard.edu/~kirsch/pubs/.

Kistler, T. & Franz, M. (2000), 'Automated data-member layout of heap objects to improve memory-hierarchy performance', *ACM Transactions on Programming Languages and Systems* **22**(3), 490–505.

Knuth, D. E. (1998), *The Art of Computer Programming: Sorting and Searching*, Vol. 3, second edn, Addison-Wesley Longman.

Kowarschik, M. & Weiß, C. (2003), An overview of cache optimization techniques and cache-aware numerical algorithms., *in* 'Algorithms for Memory Hierarchies', pp. 213–232.

Kumar, P. (2003), Cache oblivious algorithms., *in* 'Algorithms for Memory Hierarchies', pp. 193–212.

Kutzelnigg, R. (2008), An improved version of cuckoo hashing: Average case analysis of construction cost and search operations, *in* 'To appear, Workshop on Combinatorial Algorithms (IWOCA).'.

Lattner, C. & Adve, V. (2005), Automatic pool allocation: improving performance by controlling data structure layout in the heap, *in* 'Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation', pp. 129–142.

Munro, J. I. & Celis, P. (1986), Techniques for collision resolution in hash tables with open addressing, *in* 'Proc. ACM Fall Joint Computer Conf.', pp. 601–610.

Naor, M., Segev, G. & Wieder, U. (2008), History-independent cuckoo hashing., *in* 'International Colloquium on Automata, Languages and Programming (ICALP)', Vol. 5126, pp. 631–642.

Nash, N. & Gregg, D. (2008), Comparing integer data structures for 32 and 64 bit keys, *in* 'WEA', pp. 28–42.

Pagh, R. & Rodler, F. F. (2004), 'Cuckoo hashing', *Journal of Algorithms* **51**(2), 122–144.

Panigrahy, R. (2005), Efficient hashing with lookups in two memory accesses, *in* 'Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms', Society for Industrial and Applied Mathematics, pp. 830–839.

Peterson, W. W. (1957), 'Open addressing', *IBM Journal of Research and Development* **1**(2), 130–146.

Ramakrishna, M. V. & Zobel, J. (1997), Performance in practice of string hashing functions, *in* 'DASFAA', Vol. 6, pp. 215–224.

Rao, J. & Ross, K. A. (1999), Cache conscious indexing for decision-support in main memory, *in* 'Proc. Int. Conf. on Very Large Databases', pp. 78–89.

Ross, K. A. (2007), Efficient hash probes on modern processors, *in* 'IEEE 23rd International Conference on Data Engineering', pp. 1297 – 1301.

Rubin, S., Bernstein, D. & Rodeh, M. (1999), Virtual cache line: A new technique to improve cache exploitation for recursive data structures, *in* 'Proc. Int. Conf. on Compiler Construction', pp. 259–273.

Silverstein, A. (2002), 'Judy IV shop manual'. http://judy.sourceforge.net/.

Truong, D. N., Bodin, F. & Seznec, A. (1998), Improving cache behavior of dynamically allocated data structures, *in* 'Proc. Int. Conf. on Parallel Architectures and Compilation Techniques', pp. 322–329.

VanderWiel, S. P. & Lilja, D. J. (2000), 'Data prefetch mechanisms', *ACM Computing Surveys* **32**(2), 174–199.

Vitter, J. S. (1983), 'Analysis of the search performance of coalesced hashing', *Journal of the ACM* **30**(2), 231–258.

Yotov, K., Roeder, T., Pingali, K., Gunnels, J. & Gustavson, F. (2007), An experimental comparison of cache-oblivious and cache-conscious programs, *in* 'SPAA', pp. 93–104.

Zobel, J., Heinz, S. & Williams, H. E. (2001), 'In-memory hash tables for accumulating text vocabularies', *Information Processing Letters* **80**(6), 271–277.

Zukowski, M., Héman, S. & Boncz, P. (2006), Architecture-conscious hashing, *in* 'International workshop on Data management on new hardware', pp. 6–14.