Retrieval and Perfect Hashing using Fingerprinting

Ingo Müller[1,2], Peter Sanders[1], Robert Schulze[2], and Wei Zhou[1,2]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany
{ingo.mueller,sanders}@kit.edu,wei.zhou@student.kit.edu
[2] SAP AG, Walldorf, Germany
robert.schulze@sap.com

Abstract. Recent work has shown that perfect hashing and retrieval of data values associated with a key can be done in such a way that there is no need to store the keys and that only a few bits of additional space per element are needed. We present FiRe – a new, very simple approach to such data structures. FiRe allows very fast construction and better cache efficiency. The main idea is to substitute keys by small fingerprints. Collisions between fingerprints are resolved by recursively handling those elements in an overflow data structure. FiRe is dynamizable, easily parallelizable and allows distributed implementation without communicating keys. Depending on implementation choices, queries may require close to a single access to a cache line or the data structure needs as low as 2.58 bits of additional space per element.

1 Introduction

Consider a set S of n keys from some universe U. Often we want to map S to unique integer IDs from a small range. A mapping with this property is called a *perfect hash function*. Similarly, we often want to store data values associated with the keys. This is known as a *retrieval data structure*. These two problems are closely interrelated. In particular, perfect hash functions can be used to implement a retrieval data structure by indexing an array of values. The classical way to implement these data structures uses hash tables storing the keys and/or values. However, it turns out that it is not necessary to store the key values. If the keys are big, this optimization can be important. For example, suppose that S is a set of URLs and we want to store one out of a small number of categories for each element of S. Another application example is storing flags for graph exploration in a large implicitly defined graph where the keys are quite large state descriptions of a finite automaton [1]. For further applications refer to [2]. We also encountered the retrieval problem in context of a the SAP HANA main memory column oriented data base [3]. In such a *column store* DB, each attribute of a relation is stored as a separate column. In order to keep large data sets in main memory, data compression is important in column stores. Perhaps the most important compression technique replaces elements from a large universe U (e.g., strings) by an ID that can be encoded with a few

bits in many cases (dictionary compression). A retrieval data structure can be used to provide efficient mapping from U to IDs. SAP was interested in variants with very high query performance even at the price of somewhat larger space consumption than previous work. Since many data structures are accessed at the same time in HANA, an additional aspect was that we cannot assume lookup tables to remain in cache between accesses. Hence a small "cache footprint" of the data structure was an important design consideration.

Previous retrieval data structures in one way or the other cause multiple cache faults for each query. Refer to Section 3 for details. Here we explore the possibility to achieve higher performance by using a single hash function evaluation leading to access of a single cache line which yields the desired result. Our basic approach is quite simple minded and builds on traditional hash table data structures, in particular those for external memory: Keys are mapped to *buckets* capable of storing several values. The new aspect is that we do not store the keys themselves but only small *fingerprints* based on another hash function. On the first glance, this idea does not work since several elements in the same bucket may have the same fingerprint making them indistinguishable. This problem is solved by moving *all* colliding elements to an overflow data structure. Similarly, surplus elements from overfull buckets are moved to the overflow data structure. The overflow data structure can be based on the same principle using a fresh hash function. Elements not fitting there are moved to a secondary overflow data structure and so on. We may stop the recursion once the number of elements is small enough to use a more expensive data structure. This yields constant worst case access time and the expected number of cache faults can be close to one. Section 4 describes the Fingerprint Retrieval approach (FiRe) in more detail. In particular, we explain how using compression of the set of fingerprints in a bucket, the required space can become close to the space needed just for storing the function values. It turned out that the FiRe approach has additional advantages that may be even more important for some applications. In particular we can dynamize the data structure allowing insertions and deletions in expected constant time. In Section 5 we explain how the FiRe approach can be adapted to perfect hash functions. In Section 6 we report on experiments with a performance oriented implementation of FiRe. FiRe significantly outperforms competing solutions with respect to construction time and query time. The price is sometimes but not always higher space overhead which is nonetheless much smaller than for ordinary hashing. Section 7 summarizes the results and discusses possible future work.

2 Preliminaries

We use $i..j$ as a shorthand for $\{i, \ldots, j\}$. Let $n = |S|$. An obvious lower bound for the space consumption of a retrieval data structure is rn bits. For the analysis we assume that the used hash functions $h : U \to 1..m$ behave like truly random functions, i.e., we assume that they are drawn uniformly at random from the set of mappings from U to $1..m$. This can be justified theoretically using a "splitting

trick" [4]. A *perfect hash function* $h : U \to 1..m$ is an injective mapping from S to $1..m$. h is a *minimal perfect hash function* if $m = n$. In this paper, $\log x$ denotes the base two logarithm. $f(n) \sim g(n)$ expresses that f converges to g as $n \to \infty$. A pair $(s, t) \in M \subseteq \mathbb{R}^2$ is *Pareto optimal* with respect to M if no other element $(s', t') \in M$ *dominates* x, i.e., $s' \leq s$ and $t' \leq t$.

3 Related Work

Fingerprinting is a well known technique [5] for indexing data structures but it has not been applied to perfect hashing or retrieval so far. Most applications use fairly large fingerprints in order to avoid collisions. The cuckoo filter [6] uses small fingerprints to obtain an approximate dictionary. Unique features of FiRe are its simplicity and that it is able to repair collisions.

Theoretical solutions for perfect hashing with a constant or even optimal number of bits per element have been known for a long time [7]. However, practical solutions have emerged only recently – raising significant interest in the topic. The compressed hash-and-displace algorithm [8] (CHD) uses a primary hash function to identify the index of a secondary hash function. This index is geometrically distributed with constant expectation and with clever compression [9] needs only a constant number of bits. CHD has relatively expensive queries since it performs select operations on large sparse bit vectors. CHD is also inherently sequential since it relies on a greedy construction algorithm. The BPZ algorithm by Botelho, Pagh and Ziviani [10] computes the hash function value based of three random table lookups. Hence, using BPZ for retrieval implies about four cache faults for each access for large inputs. Computing the table is based on an inherently sequential greedy algorithm for ordering the edges of a 3-regular random hypergraph. The EPH algorithm [2] uses the splitting trick to compute minimal perfect hash functions. The main difference to FiRe is that the resulting buckets in EPH have variable size and use no fingerprints but the BPZ algorithm to build bucket local perfect hash functions. Another external hash table represents buckets using *entropy coded tries* (ECT) [11] storing the longest distinguishing prefix of a hash value. These can be viewed as "perfect" fingerprints leading to a single stage lookup desirable for external memory but also introducing complication and computational overhead not appropriate for our high performance setting.

The retrieval problem can also be solved directly. The CHM algorithm [12] uses yet another greedy algorithm to compute a table of $m = \mathcal{O}(n)$ r-bit values so that the retrieved value is the xor of $k \geq 2$ values at hashed table positions.

These results can be viewed as show cases of algorithm engineering since they combine interesting ideas and highly nontrivial theoretical analysis in such a way that one gets surprisingly good results that are practically useful. FiRe is different in that it starts from a simple-minded idea and has a very simple analysis and implementation. The surprising part is that one gets competitive and in some aspects superior results this way.

4 Retrieval using Fingerprint Hashing

4.1 The FiRe Data Structure

The first level of a FiRe data structure consists of an array B of $m = n/b$ buckets. A bucket is an array of a values with r bits each. For each stored value, the bucket also stores a fingerprint in the range $1..k$. An element $s \in S$ is mapped to a bucket by a hash function $h_B : U \to 1..m$. Its fingerprint is obtained by a hash function $h_f : U \to 1..k$. We can equivalently assume that we have a single hash function $h \to 1..km$ defining both bucket position and fingerprint. The first level can only hold values for elements with a unique value of h, i.e., no two elements of S stored in the first level may be mapped to the same bucket *and* have the same fingerprint – a fingerprint collision. If more then a such eligible elements are mapped to the same bucket, any a of them can be chosen.

The first level is constructed by mapping elements to their buckets, removing elements involved in fingerprint collisions, and then removing elements from overloaded buckets. Elements not stored in the first level are moved on to the second level which is built in an analogous way. This process is repeated until a maximum number of levels L is reached. Layer $L+1$ is a *fallback* data structure which stores the remaining elements and guarantees constant worst case access time using any of the previous techniques. We can also use $L = \infty$, eliminating the fallback data structure. Figure 1 summarizes the structure of FiRe. As long as a constant fraction of the remaining elements considered can be stored in each iteration, the overall construction time is linear.

A query for key u checks whether an element with fingerprint $h_f(u)$ is stored in bucket $h_B(u)$ of level 1. In the positive case, the associated data element is

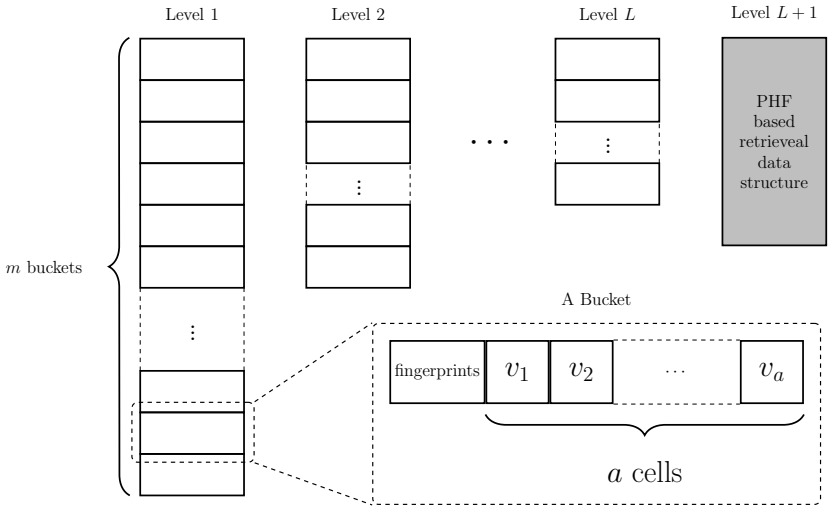

Fig. 1. Schematic diagram of a FiRe data structure.

returned. Otherwise, the next level is queried. As long as L and a are constants, this yields constant worst case query time. For $L = \infty$ the worst case query time can be made logarithmic by restarting the construction process of a layer whenever it is much smaller than expected.

Representing Fingerprints There are several ways to represent fingerprint information with different trade-offs between space consumption and query time. Refer to the full paper for details. Most of the time we will consider bit vectors using k bits for representing all fingerprints in a bucket. Also interesting are information theoretically optimal representations requiring $\left\lceil \log \sum_{i=0}^{a} \binom{k}{i} \right\rceil$ bits.

Asymptotic Analysis For details refer to the full paper. Here we only outline the basic ideas of an analysis as $m \to \infty$ assuming that $L = \infty$ and the parameters a, b, and k are the same on all levels. We first show that the probability of a particular fingerprint value to represent exactly one element is $p_1 \sim \frac{b}{k} e^{-b/k}$. We then argue that the number of non-colliding elements allocated to a bucket is approximately $B(k, p_1)$ binomially distributed which implies that the expected number of empty cells in a bucket is $a_0 \sim \sum_{i=0}^{a-1} (a-i)\binom{k}{i} p_1^i (1-p_1)^{k-i}$. Since the *overall* number of empty cells is sharply concentrated around its expectation, we can use a_0 to estimate the space overhead per element as $s \sim \frac{r a_0 + s_f}{a - a_0}$ bits per element where s_f denotes the number of bits needed to store the fingerprints of a bucket. The expected number of accessed levels is $\ell \sim \frac{b}{a - a_0}$.

4.2 Choosing Parameters

The performance of the FiRe data structure with respect to space consumption and query time depends on the parameters a, b, k, on implementation choices, in particular for representing fingerprints, on r, and on hardware parameters like the cache line size. Hence, it is a complex problem how to actually set the parameters. Moreover, in most situations there is not *one* optimal choice but a trade-off between space and time. Hence, we are interested in a set of parameter settings representing Pareto optimal solutions with respect to space and time while excluding suboptimal choices that are dominated by other choices. We propose to attack this problem by starting from hardware and implementation constraints generating a small set of reasonable choices for setting a, k, and the fingerprint representations. For each of these choices, b will be the only remaining free parameter. We then have to find out which of these choices yield Pareto optimal solutions for some values of b.

We exemplify this methodology for an example oriented at the column store application mentioned in the introduction. We consider a machine with cache line size 64 and we want to store values with $r = 32$ bits. We are interested in very fast access and thus choose the bit vector representation for fingerprints. For the same reason, we want buckets to fit perfectly into a cache line. This implies that k should be a multiple of 32 and that $a = 16 - k/32$. Figure 2 plots the resulting trade-off for $k \in \{32, 64, 96, 128\}$. These plots were obtained by

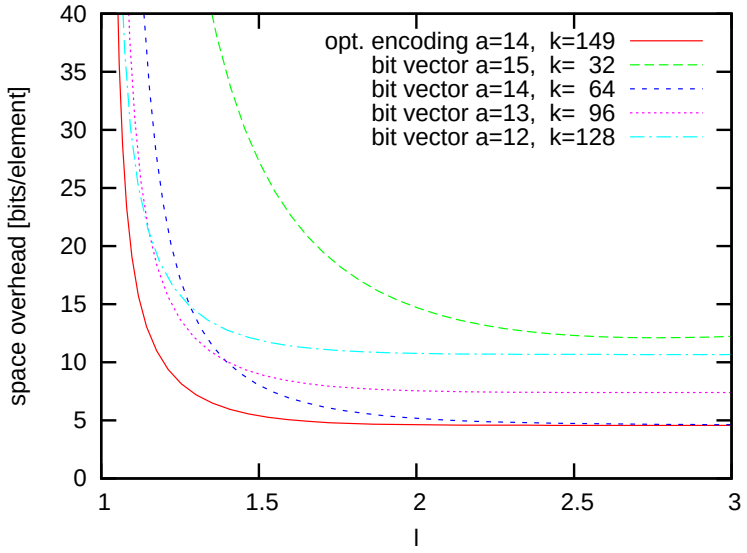

Fig. 2. Expected number of levels accessed versus space overhead for $r = 32$ and bit-vector encoding of fingerprints.

computing the pairs (ℓ, s) for $b \in 1..100$. The case $k = 32$ is not useful since it leads to too many fingerprint collisions. The case $k = 64$, $a = 14$ is perhaps the most useful one since it is good for a wide range of trade-offs. In particular, its space overhead becomes as low as 4.586 bits per element for $b = 64$. If we are willing to spend 5 bits per element, $\ell = 2.1$ expected level access suffice (at $b = 29$). To achieve $\ell < 1.3$, it is better to use $k = 96$, $a = 13$. To achieve $\ell < 1.15$, it is better to use $k = 128$, $a = 12$. The corresponding ranges of b for which the respective cases are Pareto optimal are 1..11 for $k = 128$, 12..16 for $k = 96$, and 17..64 for $k = 64$. For comparison, the solid curve shows the values for $a = 14$, $k = 129$ assuming an information theoretically optimal encoding of the fingerprints. This curve dominates all the other curves. However, note that the query time for this case is likely to be quite large anyway due to overhead for decoding the fingerprints. Hence, this curve should rather be viewed as an optimistic estimate for the performance of some very clever implementation that offers a combination of space efficient encoding and fast decoding.

External, Parallel, and Distributed Processing In the full paper we explain how to construct a FiRe data structure in external memory and on a parallel machine. Due to the decomposition into independent buckets this is very easy and efficient. Perhaps more interestingly, FiRe can be implemented in various distributed settings so that communication volume is very low. In particular, there is no need to communicate keys.

4.3 Dynamization

FiRe directly supports an update operation – changing the value associated with a key. The same holds for retrieval data structures based on perfect hashing but other data structures such as CHM [12] do not allow updates.

An advantage of the FiRe is that it can be augmented to allow *modifications* (insertions and deletions) in expected constant time. Existing superlinear lower space bounds [13,14] make this appear difficult without access to the key information. However, we will now argue that additional information only needs to be available during modifications. This setting does not save memory but has useful applications anyway. One example is when the FiRe data structure fits into fast memory (e.g., L3 cache) and the augmented information fits into the next level of the memory hierarchy (e.g., main memory). In this situation, the dynamized FiRe will be faster than alternative solutions when there are much more queries than modifications. Another scenario is distributed computing where the key information is available on one site A and another site B only stores the data needed for queries. A modification will then be done on A which sends only the information required to change the FiRe data structure at site B.

In order to support insertions, we need to store two kinds of additional information. First, we need to know the keys of the elements stored in a bucket. When a newly inserted element x suffers a fingerprint collision with an element y, both x and y need to be moved on to the next level. When x collides with a fingerprint that already suffered a collision previously, this also needs to be known. We call such a fingerprint "blocked". To insert an element x, we inspect the bucket $i = h_B(x)$. If the fingerprint $f = h_f(x)$ is unused, and block i is not full, the value for x is inserted into block i and f is marked as used. The dynamic part of the data structure remembers x. In all other cases, f becomes blocked for block i and the insertion attempt moves on to the next level. If x had to move on because of another element y stored there (i.e., f was used but not blocked previously), element y is also moved to the next level.

Note that in the worst case, a single insertion can cause a chain reaction leading to a number of element moves exponential in the number of levels L. In the full paper we analyze this effect using branching processes and show that the situation remains stable as long as $k > b$. When this condition becomes violated, the data structure should be rebuilt.

The easiest way to handle deletions is to ignore them – the specification of retrieval data structures does not specify anything about the result of a query for an element outside S. The stability condition for the branching process should then define n (and, as a consequence $b = n/m$) as the number of stored elements plus the number of inserted elements. We can also trigger rebuilding when this value of n differs too much from the number of non-deleted elements. Actual deletion of an element stored in a bucket is easy. We just remove it from the cell it previously occupied and set its finger print from used to unused.

If we insist on keeping the retrieval part of the FiRe data structure identical to what we would get in the static case, we additionally need to be able to unblock blocked fingerprints. For this we need to count the number of times

a fingerprint is used. When this count goes down to one, we have to find the element which wants to move there, delete it from the subsequent layers and move it to the current layer. A similar case applies when an element is deleted from a full bucket. Then the block has room for an element associated with a blocked fingerprint with count one.

5 Fingerprint Based Perfect Hashing (FiPHa)

Perfect hashing can be viewed as FiRe with $r = 0$, i.e., there is no need to store any associated information – we only store fingerprint information in the buckets. The fingerprint information can be used to define the injective function $h_p(x) := ah_b(x) - a + \text{rank}_B(x)$ if $h_f(x)$ is a valid fingerprint in bucket $B[h_b(x)]$ and where $\text{rank}_B(x)$ counts the number of one-bits in the fingerprint bit vector of bucket $B[h_b(x)]$ up to position $h_f(x)$. If $h_f(x)$ is not a valid fingerprint in bucket $B[h_b(x)]$, we return the value of $h_p(x)$ for the next layer and add the offset am. The analysis of FiPHa is analogous to the analysis of FiRe. We get expected space overhead of $s = ns_f/(a - a_0)$ bits and as before $\ell = b/(a - a_0)$. The expected range of the perfect hash function is $n/(1-\frac{a_0}{a})$. This can be seen as follows: In layer 1, we will consume range am. In expectation, $am-a_0m$ elements will be mapped to this range. The resulting ratio is $am/(am-a_0m) = 1/(1-\frac{a_0}{a})$. Since the same ratio will be observed on all levels, the overall expected range is $n/(1 - \frac{a_0}{a})$.

For example, assuming information theoretically optimal representation of fingerprints, cache line size 64 bytes, and optimizing for space, we set $k = b = 543$ and $a = 200$. We get expected space $s = 2.61$ bits per element, $\ell = 2.77$ expected layer accesses and the perfect hash function has expected range $1.026n$. In this case, compression does not actually help a lot. Consider uncompressed bit vector representation, $k = b = 512$ and $a = 188$. We then get space 2.79 bits, 2.78 layer accesses, and range $1.023n$.

FiPHa is dynamizable in a way analogous to FiRe.

Minimal perfect hashing. We can get a minimal perfect hash function by using only a single large bucket (i.e., $b = n$, $m = 1$, $a = \infty$) and by setting the bucket size a to the actual number of elements stored in that bucket (i.e., all those elements not moved on to the next layer). The price we pay for this conceptual simplification is that now the rank function has to work on an input of size $\mathcal{O}(n)$. Fortunately, it is well known how to do this with constant query time and information theoretically optimally up to lower order terms. There are even practical implementation, e.g., [15]. The asymptotic analysis is also greatly simplified since there is no need to account for empty cells. In the full paper we argue that the expected space consumption per stored element is $\frac{H(p_1)}{p_1}$ where H denotes the entropy function and p_1 is the probability that a fingerprint is used exactly once in a bucket. The expected number of accessed levels is $e^{n/k}$. Optimizing for space consumption we get $p_1 = 1/e$ at $k = n$. This value yields $H(1/e) \approx 2.58$ bit of space per element and expected number of accessed levels

e. If we are willing to spend 3 bits per element, we get about 1.58 expected level accesses. For 4 bits per element the same figure becomes 1.21 expected level access.

6 Experiments

We show results for 10^8 32 bit integers (data set **INT**) and and 10^8 3-grams (sequences of 3 words) randomly choosen from the $1.33 \cdot 10^8$ 3-grams from Google Books [16] starting with `n` (data set **NGRAM**). We evaluate the retrieval data structures on both data sets and with $r \in \{8, 32, 64\}$ bits for data values.

We have implemented the FiRe and FiPHa data structures using bit vector representation of fingerprints. Refer to the bachelor thesis of Wei Zhou [17] for more details. Buckets are aligned to cache lines. We use Jenkins [18] fast and simple hash function (we also tried the newer SpookyHast with similar results). The implementation uses GNU C++ 4.8.1 with compilation options `-O3 -m64 -msse4.2 -fopenmp -std=c++11 -march=native`. Elements are assigned to buckets with the fast parallel radix sort algorithm from [19]. All experiments have been performed using a single core of a machine with 48 GByte RAM and 2 Intel Xeon X5650 hexa-core processors with 2.66 GHz clock frequency and 12 MByte L3 cache. The cache line size of this processor is 64 byte. For each of value of r we have configured FiRe with three parameter settings that achieve ℓ close to 1.05, 1.25, and 1.5 respectively. In the full paper we list these configurations. Here we denote them FiRe5, FiRe25, and FiRe50 respectively. All these variants use $L = 8$ levels. The fallback data structure uses our own implementation of the BPZ algorithm [10]. FiPHa is configured as introduced in Section 5.

We compare FiRe and FiPHa with five state of the art implementations from the CMPH library [20]. All these algorithms use the Jenkins hash function [18]. CHM-x refers to the CHM algorithm [12] where x indicates how many values are xor-ed. CHD-α stands for the CHD algorithm [8] where α denotes the load factor used in [8] – 0.99 means that a nearly minimal perfect hash function is generated. We have also made measurements using the STL `unordered_map` hash table (using Jenkins hash function and preallocating as many buckets as elements). STL not only needs much more space but is also about three times slower than FiRe both with respect to construction time and query time.

Figure 3 visualizes the remaining results. The full paper gives the corresponding numeric values. With respect to construction time FiRe is four times faster than the fastest competitors (CHD-0.5) and 17 times faster than the slowest one (CHD-0.99). FiPHa is considerably slower to construct, but still faster than all competitors. With respect to space consumption, FiRe50 is competitive with the other implementations for $r \geq 32$. Only FiPHa and CHD-0.99, which compute a near minimal perfect hash function, beat all the other codes significantly, with a small advantage for CHD-0.99. However, this comes at the price of a much larger construction time. For $r = 8$ or for FiRe5 and FiRe25, FiRe needs significantly more space then the other codes. However this is still much less than an ordi-

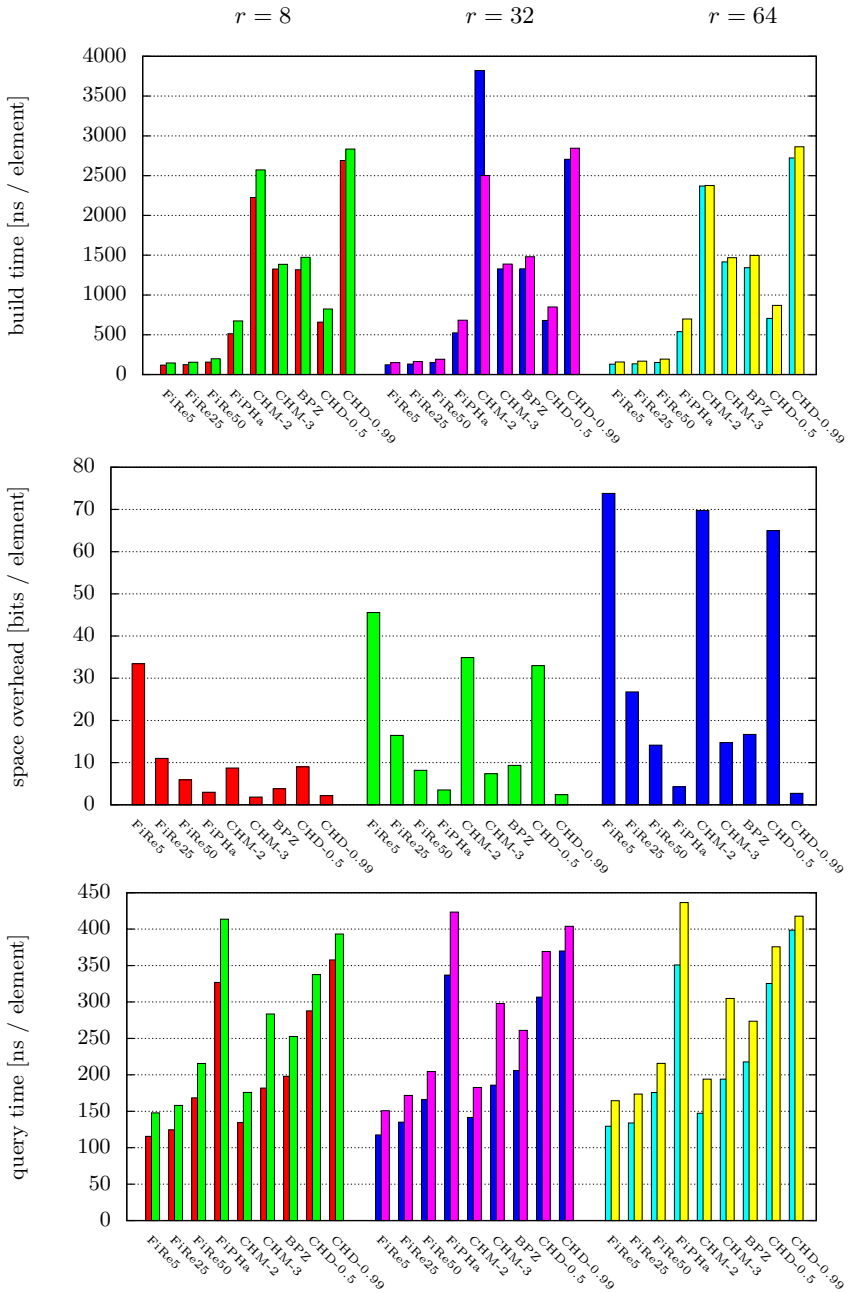

Fig. 3. Comparison of space, construction time and query time. The left and the right bar show the values of the INT and the NGRAM data set respectively.

nary hash table. However, more space efficient implementations of fingerprint sets may improve this in the future, in particular for small r.

As expected, FiRe has the best query times. The only competitor with comparable performance, CHM-2, actually needs a similar amount of space as FiRe but is clearly beaten with respect to construction time. Also recall that CHM does not support updates of the retrieved information. As expected FiPHa has worse query performance than FiRe but remains competitive with CHD-0.99, the only competitor with a similar space overhead, while having a considerably lower construction time. Overall, we also see a significant performance advantage compared to competitors of comparable functionality.

7 Conclusion

In retrospect, we find it surprising that fingerprint based hashing was not the first method tried for space efficient perfect hashing and retrieval data structures. At least conceptually it looks simpler than previous methods. We have shown that it also allows faster (and parallel) construction and faster queries. The advantages of fingerprinting for dynamization and distributed implementation seem even more fundamental. Since so many results follow from the fingerprinting approach, it looks interesting to look for further applications and refinement.

We could radically reduce the number of empty cells by mapping overflowing elements to the same level. For example, we could adapt the bucket-cuckoo hashing approach [21] to fingerprinting as in [6] for approximate dictionaries. We pay with more expensive construction and we will need somewhat larger fingerprints but for large r this should overall save space.

It might be possible to find better trade-offs between space and query time by using different parameters on different levels of the FiRe data structure. It looks promising to optimize for space efficiency on the first level and change the parameters in favor of lower query time on the following levels. Good combinations could be found systematically using dynamic programming – we maintain a set of Pareto optimal configurations using i levels of hierarchy and use this to build solutions with $i + 1$ levels.

In order to get a good practical compromise between space efficiency and speed, it would be interesting to look for a representation of the fingerprint sets that allows fast rank-queries and need space close to the lower bound.

Acknowledgements We would like to thank Martin Dietzfelbinger for valuable suggestions including the idea to combine fingerprints with cuckoo hashing. Sebastian Schlag provided a very good implementation of the radix sorter.

References

1. Edelkamp, S., Sanders, P., Simecek, P.: Semi-external LTL model checking. In: 20th International Conference on Computer Aided Verification. (2008) 530–542

2. Botelho, F.C., Ziviani, N.: External perfect hashing for very large key sets. In: 16th ACM Conference on Information and Knowledge Management. (2007) 653–662
3. Färber, F., et al.: SAP HANA Database: Data management for modern business applications. SIGMOD Rec. **40**(4) (January 2012) 45–51
4. Dietzfelbinger, M.: Design strategies for minimal perfect hash functions. In: Stochastic Algorithms: Foundations and Applications. Volume 4665 of LNCS. Springer Berlin Heidelberg (2007) 2–17
5. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development **31**(2) (1987) 249–260
6. Fan, B., Andersen, D.G., Kaminsky, M.: Cuckoo filter: Better than bloom. ;login **38**(4) (2013)
7. Hagerup, T., Tholey, T.: Efficient minimal perfect hashing in nearly minimal space. In: STACS 2001. Volume 2010 of LNCS. Springer (2001) 317–326
8. Belazzougui, D., Botelho, F.C., Dietzfelbinger, M.: Hash, displace, and compress. In: Algorithms-ESA. Volume 5757 of LNCS. Springer (2009) 682–693
9. Fredriksson, K., Nikitin, F.: Simple compression code supporting random access and fast string matching. In: Experimental Algorithms (SEA). Volume 4525 of LNCS. Springer (2007) 203–216
10. Botelho, F.C., Pagh, R., Ziviani, N.: Simple and space-efficient minimal perfect hash functions. In: 10th WADS. Volume 4619 of LNCS. Springer (2007) 139–150
11. Lim, H., Andersen, D.G., Kaminsky, M.: Practical batch-updatable external hashing with sorting. In: ALENEX. (2013) 173–182
12. Dietzfelbinger, M., Pagh, R.: Succinct data structures for retrieval and approximate membership. In: 35th ICALP. Volume 5125 of LNCS., Springer (2008) 385–396
13. Demaine, E., Meyer auf der Heide, F., Pagh, R., Pătraşcu, M.: De dictionariis dynamicis pauco spatio utentibus. In: LATIN 2006: Theoretical Informatics. Volume 3887 of LNCS. Springer (2006) 349–361
14. Eppstein, D., Goodrich, M.: Straggler identification in round-trip data streams via newton's identities and invertible Bloom filters. IEEE Trans. Knowl. Data Eng. **23**(2) (2011) 297–306
15. Navarro, G., Providel, E.: Fast, small, simple rank/select on bitmaps. In: 11th Symposium on Experimental Algorithms. Volume 7276 of LNCS. Springer (2012) 295–306
16. Google: Google books Ngram Viewer
17. Zhou, W.: A compact cache-efficient function store with constant evaluation time. Bachelor thesis at KIT and SAP (2013)
18. Jenkins, B.: Algorithm alley: Hash functions. Dr. Dobb's Journal (1997)
19. Sanders, P., Wassenberg, J.: Engineering a multi-core radix sort. In: Euro-Par. Volume 6853 of LNCS., Springer (2011) 160–169
20. de Castro Reis, D., Belazzougui, D., Botelho, F.C., Ziviani, N.: CMPH – C Minimal Perfect Hashing Library `http://cmph.sf.net`.
21. Dietzfelbinger, M., Weidling, C.: Balanced allocation and dictionaries with tightly packed constant size bins. Theoretical Computer Science **380**(1–2) (2007) 47–68
22. McDiarmid, C.: Concentration. In: Probabilistic Methods for Algorithmic Discrete Mathematics. Springer (1998) 195–247
23. Sanders, P., Schlag, S., Müller, I.: Communication efficient algorithms for fundamental big data problems. In: IEEE Int. Conf. on Big Data. (2013)