# Design of a Near-Minimal Dynamic Perfect Hash Function on Embedded Device

Derek Pao, Xing Wang and Ziyan Lu
Department of Electronic Engineering,
City University of Hong Kong, HONG KONG
E-mail: d.pao@cityu.edu.hk, {xingwang4, ziyanlu2}@student.cityu.edu.hk

***Abstract*— There has been a general opinion that it is difficult to construct perfect hash tables with high load factor for large datasets having a million records. The problem is even more challenging if new records can be added to the hash table incrementally. In this article, we shall demonstrate the design of a dynamic perfect hash function on embedded device based on simple bit-shuffle and bit-extraction operations. The achievable load factor can be up to 100%, and the amortized memory cost of the hash function is about 7 to 15 bits per key for 32-bit keys. Incremental updates to the hash table are allowed. The perfect hash function for a dataset with 1 million keys can be constructed in a few seconds of CPU time.**

***Keywords*— Searching, Dynamic Perfect Hash Table, Embedded System, Pipelined Architecture.**

## I. Introduction

Searching is ubiquitous in computing. A dataset $D$ is a collection of $N$ records $\{\langle k_i,v_i\rangle \mid 1 \le i \le N\}$, where $k_i$ and $v_i$ are the key and data of the $i$-th record, respectively. All the keys in $D$ are distinct. Let $K = \{k_i \mid 1 \le i \le N\}$ be the set of keys in $D$. Given an input key $\kappa$, we want to determine if $\kappa$ is a member in $K$. If $\kappa = k_i$, the search operation returns the data $v_i$ associated with $k_i$ in $D$. Searching is a core operation in many real-time packet processing tasks, e.g. IP lookup [11] and content inspection [3, 12, 13]. To meet the stringent processing requirements, high-end routers are equipped with special-purpose hardware accelerators built into the network processor [2] or implemented on FPGA/ASIC.

Hashing [8] is a technique that can solve the searching problem in $O(1)$ time on average. A hash function $H$ is a mapping of $K$ to integer values within the range from 0 to $m-1$, i.e. $H$: $\{k_1, k_2, \ldots, k_N\} \rightarrow \{0, 1, 2, \ldots, m-1\}$, where $m$ is the size of the hash table and $m \ge N$. The ratio $\lambda = N/m$ is called the *load factor*, which represents the memory utilization. Two distinct keys $k_i$ and $k_j$ in $K$ are said to have *collided* under $H$ if $H(k_i) = H(k_j)$. When collisions happen, a search operation may involve multiple probes to the hash table. In general, collision probability increases with $\lambda$. $H$ is said to be a *perfect hash function* (PHF) if $H(k_i) \ne H(k_j)$ for any two distinct keys $k_i$, $k_j \in K$. A PHF is said to be a *minimal perfect hash function* (MPHF) if $m = N$.

PHFs are highly desirable in hardware-based processing. First, the hardware needs not handle collision resolution and bucket overflow. Second, the search operation is completed in exactly one probe to the hash table. This property helps to simplify the control of the hardware pipeline and ensures deterministic throughput. There are three major challenges in the design of PHF in hardware, namely the memory efficiency, scalability and dynamic updates.

In this article, we shall present the design of a *near-minimal perfect hash function* for dynamic datasets. We test our method using datasets with 10K to 2M records, and the achievable load factor can be up to 100%. The memory cost of the hash function is about 7 to 15 bits per key for 32-bit keys, and the logic circuits are fairly simple. Incremental updates to the hash table can be allowed.

This work was supported by a grant from the Research Council of the Hong Kong Special Administrative Region, China (Project No. CityU 119809).

## II. Related Work

Hashing is a fundamental concept in computing, and it has been studied extensively. However, none of the previously proposed methods on the construction of PHF in hardware can adequately address the three major issues on efficiency, scalability and dynamic updates. Effective algorithms to find MPHF for *static* datasets have been presented in [6], where the memory cost of the MPHF is $O(\log_2 N)$ bits per key. Whenever a new key is added, the hash table needs to be rebuilt completely. Also the MPHFs of [6] use multiplication and division operations making the hardware realization expensive in terms of logic elements and circuit delay time.

A classical dynamic perfect hash table was proposed by Fredman and Komlos [4]. The search operation involves the evaluation of 2 multiplications and 4 divisions, and the achievable load factor for Fredman's method is between 0.2 to 0.3.

Cuckoo hashing [10] is another well-known method to construct dynamic perfect hash table. It uses 2 hash tables $T_1$ and $T_2$. A key $k_i$ may be stored in one of the two possible locations, namely $T_1[H_1(k_i)]$ or $T_2[H_2(k_i)]$. After the hash tables have been constructed, a query operation can be completed with at most 2 probes. Keys are inserted one by one into the initially empty hash tables. If the home bucket of the new key is occupied, the old key is knocked out and the insertion algorithm will be called recursive to insert the knocked-out key in the other hash table. If the insertion procedure gets into an infinite-loop, the hash tables are rebuilt by choosing larger table sizes and/or using two new hash functions. The allowable load factor is up to 0.5 in order to keep the probability of rehash within an acceptable level.

Bloom Filter (BF) [1] is an efficient method to determine if an input key κ is a member of the dataset. A BF uses *r* hash functions, and contains an *m*-bit vector where $m \geq rN$. Let $b[j]$ denotes the *j*-th bit of the *m*-bit vector. Initially all the bits in the bit-vector are set to zero. In the programming phase, for each $k_i \in K$, $b[H_j(k_i)]$ is set to 1 for $1 \leq j \leq r$. In the query phase, for a given input key κ, if all the bits in the bit-vector indexed by $H_j(\kappa)$ for $1 \leq j \leq r$ are equal to 1, then κ is considered to be a member of *K*. BF may generate *false-positive*. The probability of false-positive can be tuned by adjusting the parameters *r* and *m*. According to the design guidelines of [3], the memory cost of the basic BF implemented in hardware with a false-positive probability of 0.1% is about 14.5 bits per key.

Besides the possibility of producing false-positive, a fundamental limitation of BF is that it does not identify the matching key. There have been a number of research efforts [7, 9, 14] to extend BF such that the matching key can be located. Despite the sophistication of the extended methods, their memory efficiencies are suboptimal. First, multiple BFs or counting BFs are required and the on-chip memory cost is on the high side, e.g. about 50 bits per key. Second, the load factor of the resultant hash table is on the low side, e.g. about 0.25 to 0.3. This is because the size of the hash table depends on the bit-vector length of the underlying BF.

Ficara et al. [5] presented a method to construct PHF with a lower on-chip memory cost by introducing external *discriminators*. A fingerprint hash code for each key is evaluated, which will be used to access a *discriminator table* (DT). In a query operation, the system will first look up the DT and appends the discriminator bits retrieved from the DT to the input key before evaluating the hash function. The memory cost of the DT is about 2 to 4 bits per key for data sets with a few thousand keys. Ficara's method has two major disadvantages. It takes very long computation time to set up the DT such that collisions (for the given dataset) are eliminated. Another disadvantage of this method is that incremental updates to the data set are not supported.

## III. Perfect Hashing with Bit-Shuffling

The proposed method is a generalization of the bit-shuffled trie for IP lookup [11]. The motivation of our method is different from [5]. Since the keys are distinct, we do not see any needs to look for external discriminators to differentiate the keys. We shall instead apply a hierarchical approach to successively divide the dataset into smaller subsets by selecting appropriate discriminator bits within the key. When each non-empty subset contains one item, a PHF is obtained. We shall present a design with 4 processing steps implemented in a hardware pipeline. Let the size of the dataset be *N*, where $2^k \leq N < 2^{k+1}$. A fixed number of bits are chosen in the first 3 steps, and the objective is to divide the dataset into $2^{k-2}$ non-empty subsets such that most of the subsets contain no more than 8 members. In the last processing step, individual members in a subset are mapped to unique addresses in the hash table. The proposed method has been applied in [13], and in this article we shall present more detailed designs and performance evaluations.

The block diagram of the hardware pipeline is depicted in Fig. 1. The bits in the input key are shuffled in each pipeline stage according to the control data stored in some internal registers or index tables. The first stage bit-shuffle circuit (implemented by simple multiplexors controlled by internal registers) divides the input key into 3 chunks, *A*, *B* and *C*. Chunk *A* is used as the address to access table $T_1$. Two bit indexes are stored in each entry of $T_1$, and they are used to control the shuffling of *B* in the second stage. Chunk *B* is divided into $B_0$ with the two specified bits and $B_1$ with the remaining bits. $B_0$ is then concatenated to *A* to access table $T_2$ to retrieve another pair of bit indexes. The indexes obtained from $T_2$ will be used to control the shuffling of *C* in the next stage. Chunk *C* is divided into $C_0$ with the two specified bits and $C_1$ with the remaining bits. Again, $C_0$ is concatenated to $\{A, B_0\}$ to access table $T_3$, and $\{B_1, C_1\}$ constitutes the *residue key* (remaining bits of the input key not yet matched).

Three data fields are stored in $T_3$, a *mask-vector*, an 8-bit *block-vector*, and a *base* address. The mask-vector is used to specify the bits in the residue key that would be chosen to uniquely differentiate the members in the corresponding subset. The offset generation circuit takes the residue key, mask-vector, and the block-vector as inputs and produces an offset value. The final hash address is obtained by adding the offset value to the base address retrieved from $T_3$. The hash table can be on-chip or stored in external SRAM depending on the application requirements.

We shall explain the operation of the pipeline with an example. To simply the discussion, we assume 12-bit keys and the system only contains 2 index tables $T_1$ and $T_3$. In this example, the key is divided into two chunks, *A* and *B*, in the first bit-shuffle operation. The selected bits in *A* are highlighted in Fig. 2. Let $T_k[i]$ refer to the *i*-th entry of table $T_k$. All the keys shown in the example are mapped to $T_1[0]$. Bits are numbered from right to left starting from 0. Assume the two bit indexes stored in $T_1[0]$ are {3, 0}. The second stage bit-shuffle circuit divides *B* into $B_0$ (bits 3 and 0) and $B_1$ (the residue key). The keys are mapped to $T_3$ according to the value of $\{A, B_0\}$. Consider the group $\{a, b, c, d, e\}$ mapped to $T_3[0]$. By taking the 4 selected bits defined by the mask-vector to form a 4-bit offset value, the 5 items are mapped to addresses {0001, 1010, 1100, 0000, 1000} in the hash table. This group of 5 items occupy a block of size 16 in the hash table, and 11 out of 16 entries in the block are vacant. We shall use two strategies, namely *block compaction* and *block overlaying*, to improve the memory efficiency of the hash table.

The block compaction strategy is as follow. If the number of selected bits in the mask-vector is equal to *s*, then the group is mapped to a logical block of size $2^s$ in the hash table. The logical block is said to be *partially-filled* if the number of members in that group is less than $2^s$. A partially-filled block of size 8 or below can be compacted to eliminate the empty slots. The 8-bit *block-vector* encodes the original offset positions of the members before compaction.

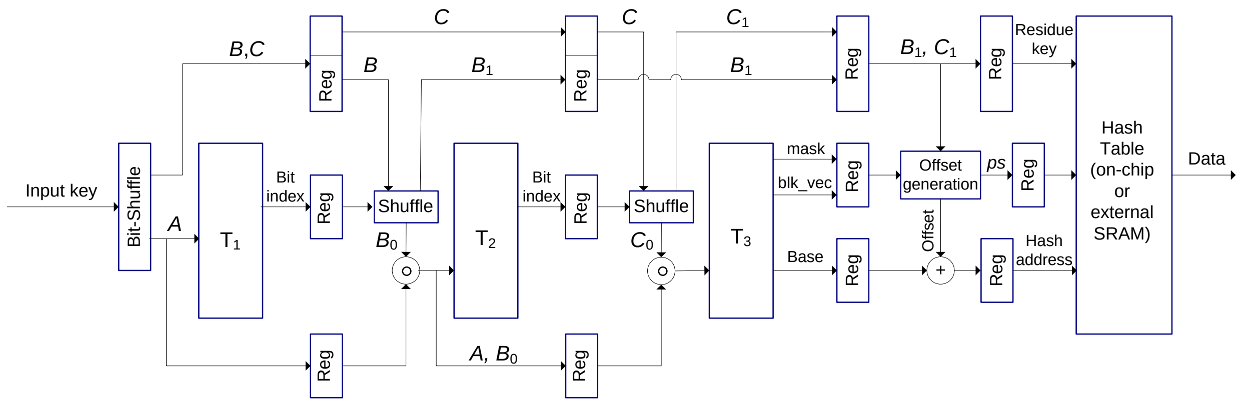


**Figure 1.** Block diagram of the hardware architecture. $T_1$, $T_2$ and $T_3$ are on-chip index tables.

Consider the group $\{f, g, h\}$. Members in this group are mapped to a logical block of size 4 with offsets {01, 00, 11}. Hence, the block-vector of this group is equal to 0000-1011. The offset generation circuit will first extract the bits defined by the mask-vector to form the initial offset value $i$. If the block-vector is non-zero (i.e. the block is compacted), then the offset value is adjusted by counting the number of 1's to the right of the $i$-th bit in the block-vector. For item $h$, its offset value is adjusted to 2 after compaction.

The block overlay strategy is as follow. Logical blocks of size 16 and above are called *primary-blocks*, and logical blocks of size 8 and below are called *secondary-blocks*. Primary-blocks (secondary-blocks) are mapped to disjoint address spaces with respect to other primary-blocks (secondary-blocks). To improve memory efficiency, we allow secondary-blocks to be overlaid onto the vacant spaces of primary-blocks. Members of the primary-block are distinguished from those of the secondary-blocks by the *ps*-bit. The hash table entries correspond to secondary-blocks will have the *ps*-bit equal to 0, whereas members of the primary-block have the *ps*-bit equal to 1. In the example of Fig. 3, the three secondary-blocks of the groups $\{f, g, h\}$, $\{i, j\}$, and $\{k\}$ are mapped to the vacant spaces of the primary-block at addresses 2, 5 and 7, respectively. The offset-generation circuit will also derive the *ps*-bit value from the mask-vector. The pipeline will output the triple (hash address, residue key, *ps*-bit). If the residue key and the *ps*-bit stored at the given address of the hash table match the outputs of the pipeline, then a matching key is found.

## IV. Hardware design and evaluation

Implementation of the first stage bit-shuffle circuit is straightforward. We can use $L$ multiplexors (MUXs), where $L$ is the key length, to divide the input key into chunks $A$, $B$ and $C$. The control bits of each MUX are pre-loaded to some internal registers during system initialization. Let the length of $A$, $B$ and $C$ be $L_A$, $L_B$ and $L_C$, respectively, and $L_B = \lfloor (L - L_A)/2 \rfloor$, and $L_C = \lceil (L - L_A)/2 \rceil$. The second and third stage bit-shuffle circuits are essentially the same. Let $B = \{ b_{L_B-1},\ldots,b_0 \}$ and the two indexes retrieved from table $T_1$ are $I_1$ and $I_0$, where $I_1 > I_0$. The bit-shuffle circuit produces two sets of outputs $B_0 = \{x_1, x_0\}$ and $B_1 = \{ y_{L_B-3},\ldots,y_0 \}$. $B_0$ can be produced by 2 MUXs. The output $y_i$ is generated based on the equation:

$$y_i = \begin{cases} b_i & \text{if } i < I_0 \\ b_{i+1} & \text{if } I_0 \le i < I_1 - 1 \\ b_{i+2} & \text{if } I_1 - 1 \le i \end{cases}$$

Next we consider the offset generation circuit. Let's consider a simple case for a 3-bit bit-extract circuit. The two inputs to the circuit are the mask-vector $\{m_2, m_1, m_0\}$ and the data bits $\{b_2, b_1, b_0\}$, and the outputs bits are $\{x_2, x_1, x_0\}$. The Boolean equations for the output bits are:

$$x_0 = m_0 b_0 + \overline{m_0}(m_1 b_1 + \overline{m_1} m_2 b_2)$$
$$x_1 = m_0(m_1 b_1 + \overline{m_1} m_2 b_2) + \overline{m_0} m_1 m_2 b_2$$
$$x_2 = m_0 m_1 m_2 b_2$$

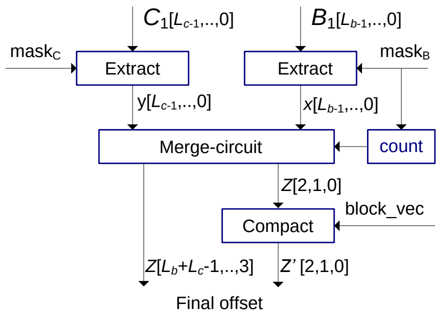


**Figure 2.** Block diagram of the offset generation circuit.

The number of terms in the Boolean equations for the basic bit-extract circuit increases rapidly with the length of the mask-vector. Hence, we adopt a hierarchical approach to implement the offset generation circuit as depicted in Fig. 2. We use two separate basic bit-extract circuits to extract the specified bits from $B_1$ and $C_1$, where the length of $B_1$ and $C_1$ are no more than 10 bits for 32-bit keys. The outputs of the two bit-extract circuits are then merged. A bit-counting circuit is used to count the number of 1's in the mask-vector $mask_B$. The output bits of the merge-circuit are produced based on the equation shown below, where $L_c$ is the length of the input $y$.

| Item | 12-bit key value | *A* | *B* | $B_0$ | $B_1$ (residue key) | Offset (underlined bits in $B_1$) | {*A*, $B_0$} |
|---|---|---|---|---|---|---|---|
| *a* | 00***0***0-0***0***00-001***0*** | 000 | 0-0000-***0***01***0*** | 00 | 000-0001 | 0001 | 000-00 |
| *b* | 10***0***0-0***0***00-010***0*** | 000 | 1-0000-***0***10***0*** | 00 | 100-0010 | 1010 | 000-00 |
| *c* | 10***0***0-0***0***10-000***0*** | 000 | 1-0001-***0***00***0*** | 00 | 100-0100 | 1100 | 000-00 |
| *d* | 00***0***0-1***0***00-000***0*** | 000 | 0-0010-***0***00***0*** | 00 | 000-1000 | 0000 | 000-00 |
| *e* | 10***0***1-0***0***00-000***0*** | 000 | 1-0100-***0***00***0*** | 00 | 101-0000 | 1000 | 000-00 |
| *f* | 00***0***0-0***0***00-101***0*** | 000 | 0-0000-***0***10***1*** | 01 | 000-0010 | 01 | 000-01 |
| *g* | 10***0***0-1***0***00-001***0*** | 000 | 1-0010-***0***00***1*** | 01 | 100-1000 | 00 | 000-01 |
| *h* | 00***0***0-1***0***10-101***0*** | 000 | 0-0011-***0***10***1*** | 01 | 000-1110 | 11 | 000-01 |
| *i* | 00***0***0-0***0***01-010***0*** | 000 | 0-0000-***1***01***0*** | 10 | 000-0001 | 0 | 000-10 |
| *j* | 10***0***1-0***0***11-010***0*** | 000 | 1-0101-***1***01***0*** | 10 | 101-0101 | 1 | 000-10 |
| *k* | 00***0***1-0***0***11-011***0*** | 000 | 0-0101-***1***01***1*** | 11 | 001-0101 | 0 | 000-11 |

Sample set of keys, and the bit-chunks produced by the bit-shuffle operations. Bit indexes stored in $T_1[0] = \{3, 0\}$.

**Table $T_3$**

| Address | mask-vector | block-vector | base |
|---|---|---|---|
| 000-00 | 100-0111 | 0000-0000 | 0000 |
| 000-01 | 000-0110 | 0000-1011 | 0010 |
| 000-10 | 000-0100 | 0000-0000 | 0101 |
| 000-11 | 000-0000 | 0000-0000 | 0111 |

**Hash Table**

| Address | *ps*-bit | Residue key | Item |
|---|---|---|---|
| 0000 | **1** | 000-1000 | ***d*** |
| 0001 | **1** | 000-0001 | ***a*** |
| 0010 | 0 | 100-1000 | *g* |
| 0011 | 0 | 000-0010 | *f* |
| 0100 | 0 | 000-1110 | *h* |
| 0101 | 0 | 000-0001 | *i* |
| 0110 | 0 | 101-0101 | *j* |
| 0111 | 0 | 001-0101 | *k* |
| 1000 | **1** | 101-0000 | ***e*** |
| 1001 | 0 | *empty* | |
| 1010 | **1** | 100-0010 | ***b*** |
| 1011 | 0 | *empty* | |
| 1100 | **1** | 100-0100 | ***c*** |
| 1101 | 0 | *empty* | |
| 1101 | 0 | *empty* | |
| 1110 | 0 | *empty* | |
| 1111 | 0 | *empty* | |

**Figure 3.** Example to illustrate the organization of the hash table.

$$z_i = \begin{cases} x_i & \text{if } count > i \\ y_{i-count} & \text{if } count \le i \text{ and } i - count < L_c \\ 0 & \text{if } count \le i \text{ and } i - count \ge L_c \end{cases}$$

The compact-circuit is used to support the block compaction strategy, which is essentially a bit-counting circuit that count the number of 1's in the block-vector to the right of the given index position. For longer key length, e.g. $48 \le L \le 64$, the residue key is divided into 4 to 6 logical partitions, and the extracted bits of individual partitions are then merged.

We evaluate the feasibility of the proposed architecture by implementing it using the Xilinx Virtex-5 XC5VSX240T device model with speed grade -2. The hardware resource utilization and system clock frequency for different system configurations are summarized in Table 1. From the design reports generated by the design tools, we observe that the system clock frequency is limited by the third pipeline stage, where $T_3$ is constructed using a relative large number of block RAMs.

## V. Construction of Lookup Tables and Memory Cost Evaluation

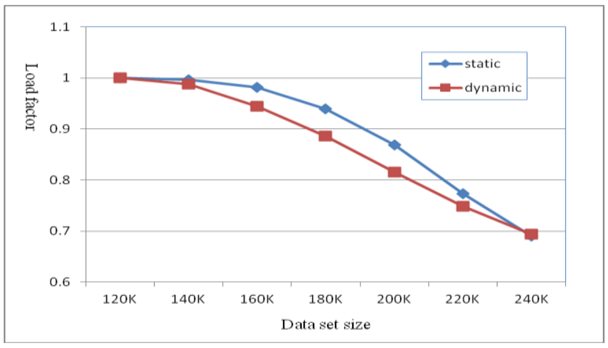
We start with a data set of *N* keys. If the average number of items mapped to each $T_1$ entry is expected to be about 64, the number of selected bits in chunk *A* is $L_A = \lceil \log_2(N/64) \rceil$. A simple algorithm is used to select the $L_A$ bits of *A*. We will first perform a bit-counting for each bit position. Let $C_i$ denote the number of keys whose *i*-th bit is equal to 1. The score of the *i*-th bit is $Score_i = min\{N - C_i, C_i\}$. Chunk *A* is obtained by selecting the $L_A$ bits with the highest scores. The remaining bits are divided into two equal-sized chunks *B* and *C*. In the second and third pipeline stage, each group of keys is further divided into 4 subgroups. We shall try to balance the size of the subgroups in selecting the 2 discriminator bits. The computer program developed in C language can construct the hash function for data sets with 1 million 32-bit keys in about 5 seconds when running on a Window XP system with Intel Core2 6400 CPU @2.13GHz.

The memory cost of the proposed method is evaluated using randomly generated key sets. In the following discussion we shall assume generic key-value pairs, where both the key and the value are 32-bit entities.

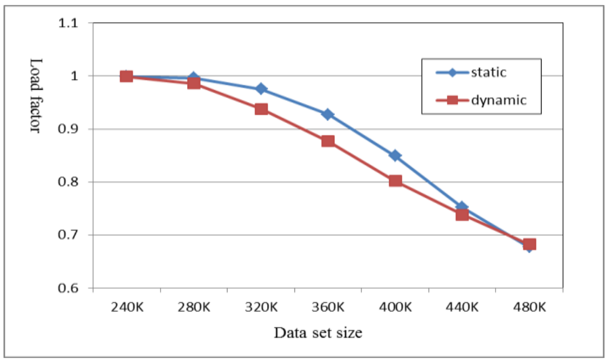
**TABLE 1.** SUMMARY OF HARDWARE DESIGN EVALUATION.

| | ***L* = 32 bits** | | | | ***L* = 48 bits** | | ***L* = 64 bits** | |
|---|---|---|---|---|---|---|---|---|
| Size of $T_1$ | 1K | 2K | 4K | 8K | 4K | 8K | 4K | 8K |
| No. of registers | 310 | 309 | 308 | 306 | 491 | 492 | 615 | 620 |
| No. of LUTs | 607 | 627 | 751 | 905 | 1390 | 1689 | 2129 | 2484 |
| No. of BRAM (36Kb) | 22.5 | 43.5 | 85 | 170 | 116 | 227.5 | 140 | 279.5 |
| Clock freq. (MHz) | 280 | 262 | 200 | 168 | 188 | 156 | 168 | 147 |

Fig.4 shows the load factor of the hash table when the size of the dataset is increased while the sizes of the index tables $T_1$ to $T_3$ are kept constant. For the static curve, tables $T_1$ to $T_3$ are computed from sketch with each dataset instance of the given size. For the dynamic curve, the initial size of the dataset is equal to 120K in Fig. 4(a) and 240K in Fig. 4(b). The plots show the performance when the dataset is increased to the given size by dynamic insertions. The contents of $T_1$ and $T_2$ remain unchanged when new keys are added. In general, the achievable load factor with dynamic insertions is slightly lower than the static data sets. However, the two become almost the same when the size of the data set grows to 2 times the initial value. One explanation for this observation is that the distribution of keys to $T_3$ is less even when dynamic insertions are allowed. As a result, there can be a higher number of smaller blocks of sizes 1 to 4, and it is easier to overlay these smaller blocks to the empty spaces of the primary-blocks.

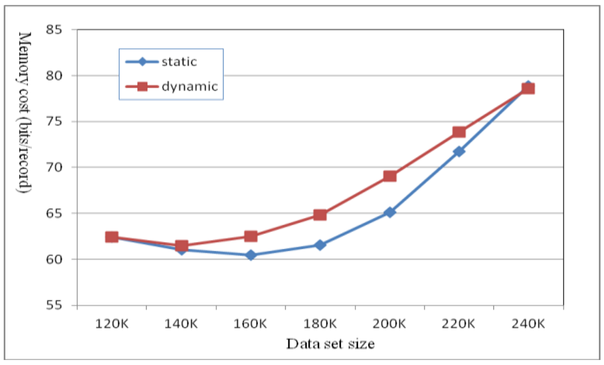


(a)

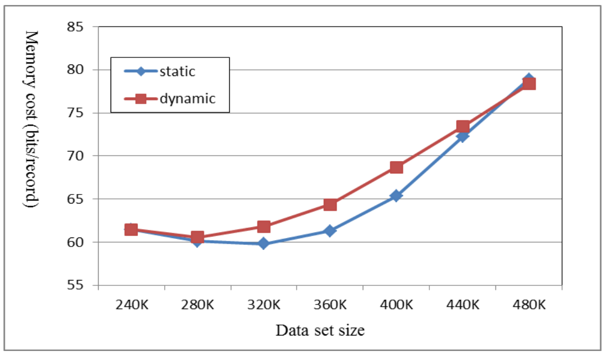


(b)

**Figure 4.** Load factor versus dataset sizes. (a) $|T_1|$ = 2K entries, (b) $|T_1|$ = 4K entries.

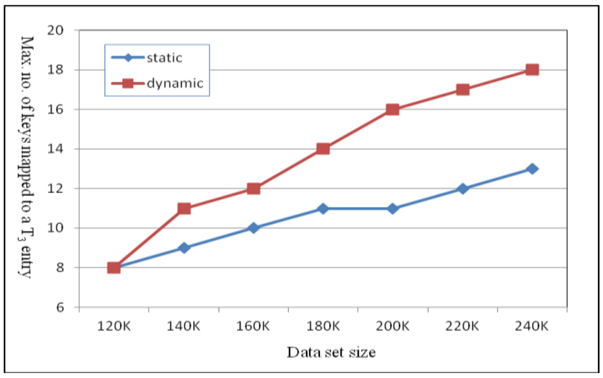


(a)

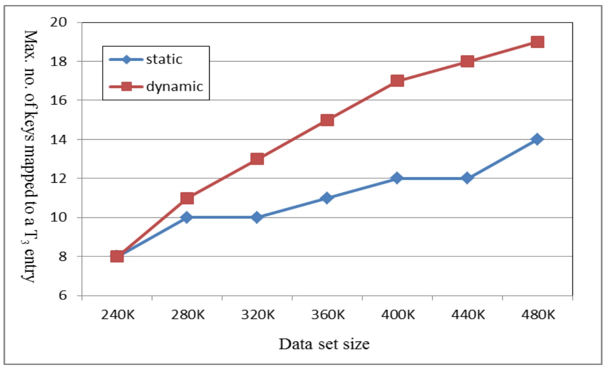


(b)

**Figure 5.** Overall memory cost (bits per record) versus dataset sizes. (a) $|T_1|$ = 2K entries, (b) $|T_1|$ = 4K entries.



(a)



(b)

**Figure 6.** Dynamic update cost (max. no. of records moved per update) versus dataset sizes. (a) $|T_1|$ = 2K entries, (b) $|T_1|$ = 4K entries.

**TABLE 2.** COMPARISON OF PERFECT HASH FUNCTIONS

| | **Fox [6]** | **Fredman [4]** | **Cuckoo hashing [10]** | **Extended BF [14]** | **Extended BF [9]** | **Ficara [5]** | **Proposed method** |
|---|---|---|---|---|---|---|---|
| **Memory cost of the hash function** | $\text{Log}_2 N$ bits/key | N/A | N/A | 40 bits/key | 50 bits/key | 2 to 4 bits/key | 7 to 15 bits/key |
| **Processing logic (implementation in hardware)** | Complex, 2 multiplications and 1 division | Complex, 2 multiplications and 4 divisions | Depends on the hash functions | Moderate | Moderate | Simple | Simple |
| **Load factor** | 100% | 20% to 30% | Up to 50% | 25 to 30% | About 25% (100% if the table is compacted) | About 50% | 80% to 100% |
| **Scalability** | Good | Good | Good | Good | Good | Limited | Good |
| **Incremental updates** | No | Yes | Conditional (may require rehash) | Yes (if there is no counter overflow) | Yes (if the hash table is not compacted) | No | Yes |

The memory cost of the hash function is fixed for a given hardware configuration, i.e. sizes of $T_1$ to $T_3$ are fixed. Fig. 5 shows the overall memory cost per record taking into account the memory space of the hash table. If the key-value pairs are stored in a linear list, 64 bits are required per record. The overall memory cost of the proposed method can be less than 64 bits per record when $N/|T_1|$ is between 60 to 90. The reduction in length for the residue keys stored in the hash table contributes to the memory savings.

Incremental updates to the hash table can be allowed. Contents of $T_1$ and $T_2$ remain unchanged in the insertion procedure. When new members are added to a $T_3$ entry, the mask-vector and the block-vector of the given $T_3$ entry may be modified. Members mapped to the given $T_3$ entry will be relocated to new locations in the hash table. From Fig. 6, we see that up to 19 records may be moved in an insertion operation.

With reference to the above evaluation results, a general design guideline based on the ratio $w = N/|T_1|$ is proposed. A suitable range for $w$ is between 50 to 100, where the load factor of the hash table would be between 0.8 to 1. When the value of $w$ grows beyond 100, it would be more efficient to reorganize the hash function with 2 times the original size of $T_1$ if on-chip hardware resources can be available. For a given value of $w$, we can estimate the memory cost of the hash function as follows. Two indexes of $\lfloor \log_2 L \rfloor$ bits are stored in each entry of $T_1$ and $T_2$. The total storage of $T_1$ and $T_2$ is equal to $(10N \log_2 L)/w$ bits. An entry in $T_3$ has 3 data fields, mask-vector, block-vector and the base address. The length of the mask-vector is equal to $L - \log_2|T_1| - 4$. The base address field has $\lceil \log_2 N \rceil$ bits. The total storage of $T_3$ is equal to $16N(L+\log_2 w+4)/w$ bits. The memory cost of the hash function is $C_M = 16(L+\log_2 L+\log_2 w+4)/w$ bits per key. With $w$ between 50 to 100, the values of $C_M$ are 7.5 to 15, 10 to 20, and 13 to 25, for $L$ = 32, 48, and 64, respectively.

## VI. CONCLUSION

We have demonstrated a practical design of a near-minimal dynamic perfect hash function on embedded device. A comparison with existing methods is shown in table 2. Our method has better overall performance in terms of memory efficiency, scalability and dynamic updates. The proposed method can be applied to real-time applications that require high-speed table lookup, e.g. IP address lookup, packet classification, pattern matching, and Named Data Networking.

## REFERENCES

[1] B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors", *Communications of ACM*, Vol. 13, No. 7, pp. 422-426, 1970.

[2] J. D. Brown, S. Woodward, B. M. Bass, C. L. Johnson, "IBM Power Edge of Network Processor: A Wire-Speed System on a Chip", *IEEE Micro*, Vol. 31, Issue 2, pp. 76-85, 2011.

[3] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, J. W. Lockwood, "Deep Packet Inspection Using Parallel Bloom Filters", *IEEE Micro*, Jan/Feb 2004.

[4] M. L. Fredman and J. Komlos, "Storing a Sparse Table with O(1) Worst Case Access Time", *Journal of the ACM*, Vol. 31, No. 3, pp. 538-544, 1984.

[5] D. Ficara, S. Giordano, S. Kumar and B. Lynch, "Divide and Discriminate: Algorithm for Deterministic and Fast Hash Lookups", *ACM/IEEE ANCS*, pp. 133-142, 2009.

[6] E. A. Fox, L. S. Heath, Q. F. Chen and A. M. Daoud, "Practical Minimal Perfect Hash Functions for Large Databases", *Comm. of ACM*, Vol. 35, No. 1, pp. 105-121, 1992.

[7] J. Hasan, S. Cadambi, V. Jakkula, S. Chakradhar, "Chisel: A Storage-Efficient, Collision-free Hash-based Network Processing Architecture", *ACM/IEEE Int. Symp. on Computer Architecture*, 2006.

[8] T. G. Lewis and C. R. Cook, "Hashing for Dynamic and Static Internal Tables", *IEEE Computer*, Vol. 21, Issue 10, pp. 45-56, 1988.

[9] Y. Lu, B. Prabhakar, and F. Bonomi, "Perfect Hashing for Network Applications", *IEEE Symp. on Information Theory*, pp. 2774-2778, 2006.

[10] R. Pagh and F. F. Rodler, "Cuckoo Hashing", *J. of Algorithm*, Vol. 51, No. 2, pp. 122-144, 2004.

[11] D. Pao, Z. Lu, Y. H. Poon, "Bit-Shuffled Trie: IP Lookup with Multi-Level Index Tables", *IEEE Int. Conf. on Communications*, 2011.

[12] D. Pao, X. Wang, X. Wang, C. Cao, Y. Zhu, "String Searching Engine for Virus Scanning", *IEEE Trans. on Computers*, Vol. 60, No. 11, pp. 1596-1609, 2011.

[13] D. Pao and X. Wang, "Multi-Stride String Searching for High-Speed Content Inspection", *The Computer Journal*, Vol. 55, No. 10, pp. 1216-1231, 2012.

[14] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing", *ACM SIGCOMM*, pp. 181-192, 2005.