适用于高速检索的完美 Hash 函数①

王 兴[1,2], 鲍志伟[2]

[1](浙江广厦建设职业技术学院 信息与工程控制学院, 东阳 322100)
[2](香港城市大学 电子工程系, 香港)

摘 要: 软件实现的 Hash 函数在当前检索领域应用非常广泛, 但是由于处理速度不高, 很难满足骨干网以及服务器海量数据的高速实时查找要求. 硬件 Hash 函数处理速度快, 但普遍存在设计电路复杂、存储空间利用率不高以及无法支持数据集动态更新等问题. 基于位提取(Bit-extraction)算法, 利用位选择(Bit-Selection)操作与位逻辑运算在 FPGA 上仿真实现一种 Hash 函数, 可生成负载因子(Load factor)接近于 1 的近似最小完美 Hash 表. 仿真结果表明, 该 Hash 函数中每个 24 bits 长度 Key 的存储空间只要 2.8-5.6 bits, 系统时钟频率可以达到 300MHz 左右(吞吐率超过 14Gbps). 可以应用于 IP 地址查找、数据包分类、字符串匹配以及入侵检测等需要实时高速表查找的场景.

关键词: 硬件 Hash 表; 完美 Hash 函数; 高速搜索; 最小完美 Hash 表

Perfect Hash Function for High-Speed Searching

WANG Xing[1,2], PAO Derek[2]

[1](School of Information & Control Engineering, Zhejiang Guangsha College of Applied Construction Technology, Dongyang 322100, China)
[2](Department of Electronic Engineering, City University of Hong Kong, Hong Kong , China)

Abstract: Software-based hash function has been popularly applied to current networking searching area, but it is difficult to meet the demand of high-speed real-time applications in backbone network and services with mass data. In the general design of hardware-based hash function, there are still some drawbacks such as complex logic circuit memory inefficiency, and incremental updates for dataset needed to be solved. This paper proposed a design of hardware perfect hash function on FPGA based on bit-extraction algorithm, using simple bit-selection and bit logic operation. The achievable load factor can be up to 1, and the amortized memory cost of the hash function is about 2.8-5.6 bits/ key for 32-bit keys, and the system clock frequency is about 300MHz (Throughput more than 14Gbps). The proposed method can be applied to real-time applications that require high-speed table lookup, e.g. IP address lookup, packet classification, string matching and intrusion detection systems.

Key words: hardware-based hash table; perfect hash function; high-speed searching; minimal perfect hash table

假设数据集$\{\langle k_i,v_i\rangle \mid 1 \le i \le N\}$, 由 N 条记录组成, 其中 k_i 是第 i 条 *Key*, v_i 是与该 *Key* 对应的 *Value* 值. 令 *Key* 的集 $K = \{k_i \mid 1 \le i \le N\}$, 构造 Hash 表时, 通过 Hash 函数 F 将 K 映射成从 0 到 m-1 的整数, 即 $F(K)\rightarrow \{0, 1, 2, \ldots, m-1\}$, 那么 0 到 m-1 就是生成的 Hash 表 T 的地址, v_i 就保存在 T[$F(k_i)$]上. m 为 Hash 表的大小, $m \ge N$, 衡量 Hash 表存储空间利用率的负载因子(Load factor)$\lambda=N/m$. 如果存在两个不同的 k_i 和 k_j, 通过 F 映射后存在 $F(k_i) = F(k_j)$, 那么说明存在 Hash 冲突, 这将影响到 Hash 表的查找性能. 往往 λ 越大, Hash 冲突的可能性就越大. 不存在冲突的 Hash 函数叫完美 Hash 函数, 负载因子 λ=1 的完美 Hash 函数叫最小完美 Hash 函数.

Hash 函数通过输入的 *Key* 直接读取得到对应的

① 基金项目:香港研究资助局项目(CityU 119809);浙江广厦建设职业技术学院重点项目(2015005)
收稿时间:2015-05-16;收到修改稿时间:2015-09-16

Value, 保证搜索算法的平均时间复杂度为 $O(1)$, 可以大大提高了表搜索效率, 所以在互联网搜索领域应用非常广泛, 比如在深度包检测(Deep packet inspection)[1]、路由表查找[2]、数据包分类[3]、高速入侵检测[4]以及模式匹配[5]等方面. 基于软件实现的 Hash 函数处理速度很难满足高速实时的查找要求, 硬件 Hash 函数的查找速度往往可以达到 1Gb/s 以上, 但是适用于几十万特别是百万级别数据集的高速硬件 Hash 函数的设计在当前计算机领域还存在一些困难, 包括设计电路复杂、运算速度慢、Hash 冲突频繁、不支持动态更新等. 特别是由于 FPGA 芯片的存储空间成本很高, 比如 Xilinx 较新的 Virtex-6 产品系列只有 4.5MB 可用的存储空间, 要在硬件上设计支持大数据集的 Hash 函数, 在保证处理速度的前提下, 如何提高存储资源利用率是当前的一个最大挑战.

本文针对以上问题, 基于位提取(Bit-extraction)算法, 利用位选择(Bit-Selection)操作与位逻辑运算, 在 FPGA 上实现一种完美 Hash 函数. 用 120K 到 480K 大小的静态和动态数据集分别进行测试表明, 该 Hash 函数每个 24 bits 长度 Key 的存储空间只要 2.8-5.6 bits, 系统时钟频率可以达到 300MHz 左右, 吞吐率大于 14Gbps, 生成的 Hash 表负载因子可以接近于 1. 具有硬件设计电路简单、资源消耗低, Hash 表查找速度快、存储空间利用率高等特点, 适用于各种需要高速实时表查找的应用.

1 相关工作

硬件实现的 Hash 函数研究工作主要集中在解决 Hash 冲突方面, 最常见的方法[6]就是利用已知的数据集构建独特专用的 Hash 函数, 实现完美 Hash 表. Cuckoo hashing[7]是一个非常经典且原理简单的 Hash 算法, 利于硬件实现, 但是允许达到的最大负载因子是 0.5. 该算法利用两个 Hash 表 T_1 和 T_2, 以及两个 Hash 函数 F_1 和 F_2 来处理 Hash 冲突. 对于输入的任意一个 *Key* k_i, 都可以两个可能的存储位置 $T_1[F_1(k_i)]$ 和 $T_2[F_2(k_i)]$. 构建 Hash 表时, 如果 *Key* k_i 对应的 2 个位置中有一个为空, k_i 就可以插入到那个位置. 否则, 任选一个位置将 k_i 插入, 并把已经在那个位置的 *Key* 踢出来. 被踢出来的 *Key* 需要重新插入, 不断递归循环, 直到没有 *Key* 被踢出为止. 当陷入无限循环时, 则需要重新选择 Hash 函数. 增加 Hash 函数的个数或者在每个 Hash 表地址空间存储多个 *Key*, 可以提高 Cuckoo hashing 表的负载因子.

FICARA[8]等人通过引入外部区分表实现完美 Hash 函数. 在查表操作过程中, 首先读取外部区分表中的当前 *Key* 对应的区分位, 然后将区分位合并到当前 *Key* 后再进行 Hash 函数求值. 1K 大小的数据集中每个 *Key* 占用外部区分表的空间大概是 2-4 bits. 这种方法主要的缺点是构建外部区分表很耗时, 而且不支持新的 *Key* 的动态插入. KUMAR[9]以及 ISTVÁN[10]等人通过采用类似于链接的数据结构, 增加 Hash 表每个相同 Hash 地址对应存储空间里 *value* 的个数, 降低 Hash 冲突. 该类方法支持的数据集大小比较有限, 而且不能保证持续稳定的读取速度. Hash 表的容量都是有限的, 插入新的 *Key* 时, 肯定有溢出发生的可能. BANDO [11]等人利用片内内容寻址存储器(Content Addressable Memory, CAM)作公共溢出缓存区, 降低插入时 Hash 冲突概率, 改善 Hash 表最坏访问时间的问题.

Bloom Filter (BF) [12]可以有效地确认输入的 *Key* 是否是某个数据集的成员, 但是 BF 算法本身存在误判率. 如果将 BF 算法在硬件上实现, 并把误判率控制在 0.1%以下, 则每个 *Key* 的存储空间大概需要 14.5 bits. 另外, BF 算法只能判断输入的 *Key* 是否属于某个数据集, 但是无法对该 *Key* 进行区分. 针对此问题, 很多研究对 BF 算法进行了扩展[13,14], 可以对匹配到的 *Key* 进行准确的区分, 但是硬件实现后每个 *Key* 需要 50 bits 左右的存储空间, 而且负载因子仅为 0.25 左右.

2 Hash函数设计

本文提出的硬件 Hash 函数的基本设计思路就是利用基于贪心法则(Greedy Approach)的 Bit-Extraction 算法, 对已知数据集中的 *Key*(二进制)提取几个位作为特殊位. 根据特殊位进行位选择操作, 将原始的 *Key* 集分成一系列子集, 通过位运算操作生成 Hash 表地址, 实现快速的 Hash 表查找.

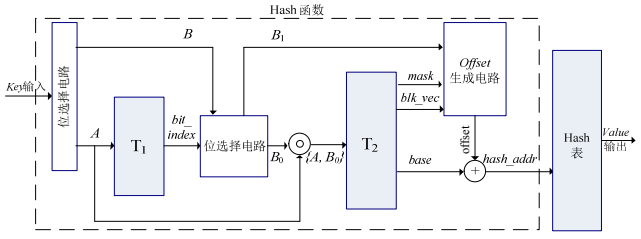


图 1 Hash 函数硬件结构设计框图

本设计的 Hash 函数硬件框图如图 1 所示, 可以分成三级的流水结构. 第一级是用第一个位选择操作, 将定长为 L 的二进制 *Key* 分成长度分别为 L_A 的 A 与长度为 $L_B(L_B=L-L_A)$的 B. 在常见的应用场景中, *Key* 的长度 L 一般在 16bits 到 64bits 之间. 假设数据集的大小为 N, 即 *Key* 的个数为 N, 则我们计算 A 的长度 $L_A\approx\lceil\log_2 N/64\rceil$. 用 A 作为查找表(Lookup Table)T_1 的地址, 读取存储在表中相应位置的 *bit_index*. 在第二级位选择操作中, 利用 2 个位索引 *bit_index* 将 B 分成长度为 2bits 的 B_0 和长度为(L_B-2)的 B_1. 然后, 将 A 和 B_0 合并后的$\{A, B_0\}$作为查找表 T_2 的地址, 读取存储在表中相应位置的位掩码 *mask*, 块占有向量 *blk_vec* 以及基础地址 *base*. 最后利用 *mask*, *blk_vec* 与 B_1 计算出地址位移 *offset*, 而{*base*+*offset*}就是我们所要得到 Hash 表地址 *hash_addr*. 以上的整个过程, 也可以看作是 Hash 算法的设计过程.

为了方便理解, 我们选择一个长度 L 为 20bits 的 *Key* 集作为例子说明. 如表 1 所示, 在第一个位选择操作中, 将标注深灰的 9 个特殊位选择作为 A, 剩余的作为 B, 则 L_A=9, L_B=11. 本例中 A 均为 *0-0000-0000*, 即所有 *Key* 都映射到了 $T_1[0]$, 假设读出存储在 $T_1[0]$的 *bit_index* 值为 3, 0. 根据 *bit_index* 的值, 在第二个 Bit-Selection 操作中, 选择第 0 位和第 3 位的 B 值(浅灰标注)作为 B_0, 剩余的作为 B_1.

表 1　举例说明设计原理

Item	20-bit input key	*A(9-bit)*	*B(11-bit)*	B_0	B_1	T_2-Address$\{A,B_0\}$	*offset*
a	*0*000-*0*010-*0*000-*0*000-11*0*0	0-0000-0000	000-1000-*0*11*0*	00	0-0010-001*1*	0	01
b	*0*010-*0*110-*0*100-*0*000-01*0*0	0-0000-0000	101-1100-*0*01*0*	00	1-0111-000*1*	0	11
c	*0*010-*0*100-*0*010-*0*000-10*0*0	0-0000-0000	101-0010-*0*10*0*	00	1-0100-101*0*	0	10
d	*0*000-*0*100-*0*010-*0*000-*0*000	0-0000-0000	001-0010-*0*000	00	0-0100-100*0*	0	00
e	*0*000-*0*010-*0*000-*0*000-10*1*0	0-0000-0000	000-1000-*0*10*1*	01	0-0010-001*0*	1	01
f	*0*010-*0*100-*0*010-*0*000-00*1*0	0-0000-0000	101-0010-*0*00*1*	01	1-0100-100*0*	1	00
g	*0*000-*0*110-*0*010-1*0*00-10*1*0	0-0000-0000	001-1011-*0*10*1*	01	0-0110-111*0*	1	11
h	*0*000-*0*110-*0*000-0*1*00-0*1*00	0-0000-0000	001-1000-*1*01*0*	10	0-0110-000*1*	2	0
i	*0*010-*0*010-*0*100-1*1*00-0*1*00	0-0000-0000	100-1101-*1*01*0*	10	1-0011-010*1*	2	1
j	*0*000-*0*110-*0*100-1*1*00-01*1*0	0-0000-0000	001-1101-*1*01*1*	11	0-0111-0101	3	/

将 A 与 B_0 合并成$\{A, B_0\}$作为地址读取 T_2, 获得存储在表中相应位置的 *mask*, *blk_vec* 以及 *base* 的值. *mask* 的长度与 B_1 的长度相同, *mask* 中所有值为 1 的位对应的 B_1 的值就是 *offset*. 我们在设计中用 8bits 的 *blk_vec* 来提高 Hash 表的负载因子. 每一个相同 T_2 地址对应的 Hash 表存储区域称为块, 如果块内记录数等于 2^n, n 为 *offset* 的位数(*mask* 中位值为 1 的个数), 那么表示该块已经被全部利用, *blk_vec* 的位值全设为 0; 否则表示块只是部分被利用, 被利用的块对应的位的值设为 1, 其它位设为 0. 比如 T_2 的地址为$\{A, B_0\}$=000-0000-0001 对应 Hash 表中的块包含 e, f, g 三个项的值, 对应的 2 位的 *offset* 分别为 01, 00, 11, 很明显, *offset* 值为 10 对应的 Hash 表存储区域没有被利用, 所以它存储在表 T_2 对应的 *blk_vec* 为 0000-1011, 如表 2 所示. 在对 Hash 表进行插入操作的时候, 可以根据 *blk_vec* 的值, 调整新插入的 *Key* 的 *offset* 值, 将其插入到 Hash 表中空余区域, 实现支持新 *Key* 动态插入, 同时提高 Hash 表负载因子.

表 2　相应的 T_2 与 Hash 表存储内容

Table T2

Address	*mask*	*blk_vec*	*base*
0	1-0000-0001	0000-0000	0
1	0-0000-0110	0000-1011	4
2	0-0000-0100	0000-0000	8
3	0-0000-0000	0000-0000	10

Hash Table

Address	*Item*
0	d
1	a
2	c
3	b
4	f
5	e
6	empty
7	g
8	h
9	i
10	j

3 设计实现与结果分析

3.1 Bit-Extraction 算法

在 Hash 函数硬件实现之前, 要利用 Bit-Extraction 算法预先处理原始 *Key* 集, 从已知数据集中的 *Key* 提取 *m* 个位作为特殊位, 将数据集分成一系列子集, 并得出位掩码 *mask*. Bit-Extraction 算法利用贪心法则, 根据选择成本 *cost* 尽可能地平衡子集的大小. 给定一个数据集大小为 N, Bit-Extraction 算法先计算出每个位中位值为“1”的 *Key* 的个数以及位值为“0”的 *Key* 的个数, 如果第 i 位值为“1”的 *Key* 的个数为 n_i, 那么通过选择第 i 位作为特殊位将把数据集分成大小分别为 n_i 和 $(N\text{-}n_i)$的两个数据集, 选择成本 *cost* 为 $\min(n_i, N\text{-}n_i)$. 从 *Key* 中选择成本 *cost* 值的和最大的 L_A 个位作为 A, 剩余的位作为 B. Bit-Extraction 算法基本思想用伪代码描述如下:

```
输入: u 个长度为 k 的 key 构成的数据集 p[0…u-1]
输出: 选择的 m 个特殊位与掩码 mask, 子 Group
1: group_size[0] = u;   //初始时所有的 key 都放在 group 0
2: for (s = 0; s < m; s++)   //每一次循环迭代选择一个特殊位
3: {   for (i = 0; i < u; i++)
4:         for (j = 0; j < k; j++)
5:             if (j-th bit of p[i] == 1)
6:                     cost[p[i].group][k]++; //求出每一个位对应的 cost 值
7:     for (i = 0; i < pow(2, s); i++):
8:         for (j = 0; j < k; j++)
9:             if (cost[i] > group_size[i] – cost[i]):
10:                 cost[i] = group_size[i] – cost[i];//选择小的值作为 cost
11:    for (i = 1; i < pow(2, s); i++)   //把子集中所有的 cost 值相加, 找出 cost 最大和, 求出相应选择的位和 mask
12:        cost[0]+=cost[i];
13:        max_cost = cost[0][k-1];
14:        selected_bit = k-1;
15:    for (j = k-2; j >= 0; j--)
16:        if (cost[0] > max_cost)
17:        {   max_cost = cost[0];
18:          selected_bit = j;
19:        }
20:    if (max_cost == 0)   直接选择最右的位, break; //无需继续迭代
21:    将选择位的 mask 值设为 1;
22:    if (s < m-1) 为下一次迭代更新变量; }
```

用 C 语言实现 Bit-Extraction 算法程序, 在配置为 Intel Core2 6400 CPU @2.13GHz, Window XP 操作系统的计算机上运行测试, 得出只需要 5s 左右就可以对大小约为 500K, 长度为 32bits 的数据集构建本 Hash 函数中所需的数据内容.

3.2 硬件设计实现

本文的完美 Hash 函数在 FPGA 上设计实现, 第一个位选择操作对长度为 L 的 *Key* 通过 L 个多路复用器(MUX)完成, T_1 , T_2 以及 Hash 表则通过配置 FPGA 自带的 Block RAM IPCore 实现. 利用 Bit-Extraction 算法预先处理原始 *Key* 集, 根据得到提取的特殊位确定 L 个 MUX 的控制位的值, 以及预计算得到 T_1, T_2 表格中存储的各项数据.

第二个位选择操作本质上也是通过 MUX 实现, 假设 $B = \{b_{L_B-1},...,b_0\}$, 从 T_1 表中读出的两个 *bit_index* 的值分别为 I_1 和 I_0, 且 $I_1>I_0$, 则 B_0 直接用两个 MUX 就可以得出, 而 $B_1 = \{y_{L_B-3},...,y_0\}$, y_i 则由以下等式算出:

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

生成 offset 的电路由简单的位运算操作生成, 我们用一个简单的只有 3bits 数据的 *mask* 和 B_1 举例说明, 假设 *mask* $=\{m_2, m_1, m_0\}$, $B_1 = \{b_2, b_1, b_0\}$, 那么 *offset* 的输出 $\{x_2, x_1, x_0\}$ 可由以下 Boolean 等式求出:

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

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

$$x_2 = m_0 m_1 m_2 b_2$$

位选择操作和位运算操作逻辑电路具有运算速度快、硬件消耗资源少的特点, 非常适合利用其进行高速硬件 Hash 表的设计. 但是在 *Key* 集的长度较长(L>24bits), 而且数据 N 较大(N≥64K)的情况下进行位选择操作时, 出现每一个相同 T_2 地址对应的 Hash 表存储块内存储 *Key* 的数目过多的情况, 容易导致 Hash 冲突, 降低 Hash 表的负载因子. 我们可以对 Hash 函数设计进行改进, 在第二级位选择操作之后, 再增加一个与 T_1 结构一样的查找表和一个与第二级位选择操作一样的位选择操作, 将 *Key* 的子集进一步分集. 改进后设计有三级位选择操作, 虽然会降低硬件电路运行速度, 但是能够提高 Hash 表的利用率和可支持的数据

集的大小.

3.3 结果分析与对比

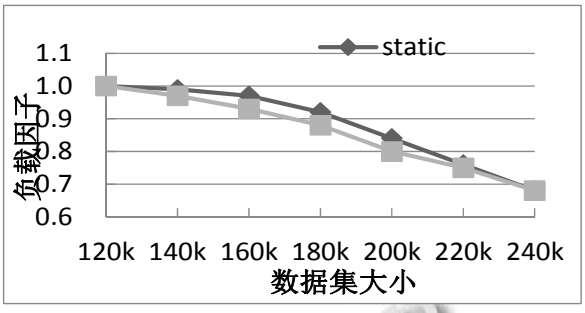
本设计在 Xilinx ISE 上用 Virtex-6 XC6VSX475T 实现, 根据 *Key* 集的长度 L 以及 L_A 的值也即 T_1 的大小(2^{LA})不同分别进行仿真, 得到硬件资源消耗的数量和系统时钟频率如表 3 所示, 其中 32 bits 和 48 bits 使用三级位选择操作实现.

表 3　FPGA 硬件资源消耗和最大系统时钟频率

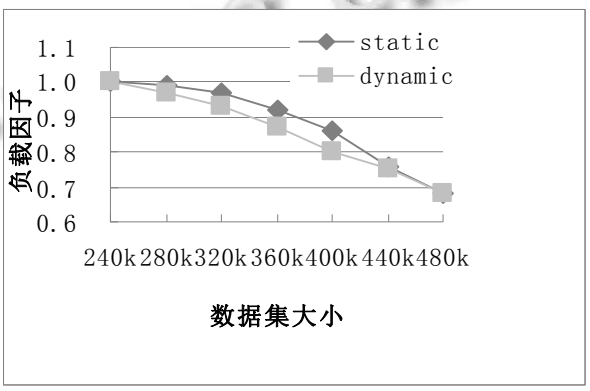
	L = 24 bits			L = 32 bits			L = 48 bits	
Size of T_1	1K	2K	4K	2K	4K	8K	4K	8K
Registers	217	214	210	347	345	342	530	531
LUTs	354	315	338	624	742	891	1364	1681
Slices	212	183	202	329	336	537	838	953
Block RAM (36Kb)	5	9.5	18	43.5	85	170	116	227.5
Max clock freq. (MHz)	279	288	322	252	220	179	189	167

另外, 把 T_1 的大小分别设置为 1K、2K, 将 Key 集的大小 N 从 120K 增加到 480K, 仿真得到负载因子结果如图 2 所示. 其中, 静态曲线是通过多个固定大小的数据集, 预计算得到 T_1、T_2 和 Hash Table 中各项值, 算出负载因子. 而动态曲线是通过在一个初始大小的数据集中动态插入新的 Key 得到的, 图 2(a)中对应的初始数据集大小为 120K, (b)中初始数据集大小为 240K. 可以看出, 当$|T_1| = 1$K, 数据集N小于140K, 负载因子将近 1, 设计的 Hash 表是近似最小完美 Hash 表. 随着数据集 N 的增大, 负载因子逐渐降低, 在 N 等于 240K 左右时, 负载因子约为 0.7. 动态插入得到的 Hash 表负载因子往往比静态生成的大. 当$|T_1| = 2$K 时, 得到类似的结果, 但是相同条件下支持的数据集大小是$|T_1| = 1$K 时的 2 倍.

本设计支持新 *Key* 的不断更新插入, 在持续插入过程中, 表 T_1 中的数据内容不发生变化. 表 T_2 中的数据 *mask* 以及 *blk_vec* 将发生变化, 而且已经存储在 T_2 表格中的一整条记录, 随着新 *Key* 的插入需要移动到新的存储位置. 对 *Key* 集的大小 *N* 从 120K 到 480K 分别进行动态插入和静态插入, 得到的 T_2 表格中需要移动的最大数据记录个数如图 3 所示. 当数据集大小为 480K 时, 需要移动的最大数据记录个数为 19.

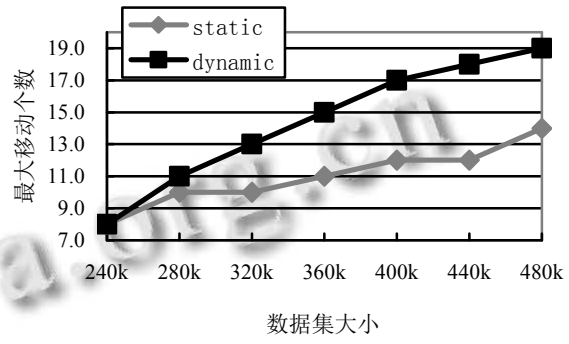


(a) $|T_1| = 1$K

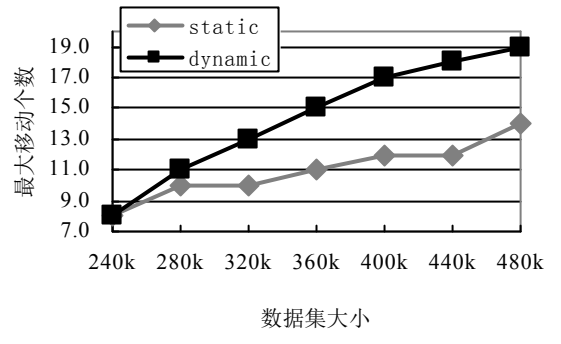


(b) $|T_1| = 2$K

图 2　负载因子随数据集大小的变化



(a) $|T_1| = 1$K



(b) $|T_1| = 2$K

图 3　插入新 key 时数据最大移动个数随数据集大小变化

最后分析本设计中每个 *Key* 的存储空间, 假设数据集大小 N 与 T_1 查找表大小之比为 $\beta= N/|T_1|$, T_1 表存两个长度为$\lceil \log_2 L_A \rceil$的 *bit_index*, 那么 T_1 表所需的存储空间为 $2N\lceil \log_2 L_A \rceil/\beta$. T_2 表存储的内容包括 *mask*, *blk_vec* 以及 *base*, 其中 *mask* 的长度为 $L-\log_2|T_1|-4$, *blk_vec* 的长度为 8 bits, *base* 的长度为$\lceil \log_2 N \rceil$, 所以 T_2 表所需的存储空间为 $4N(L+\log_2\beta+4)/\beta$. 每个 *Key* 总的存储空间可以通过$(2\lceil \log_2 L_A \rceil+4L+4\log_2\beta+16)/\beta$ 计算得出. 考虑到 Hash 冲突和负载因子, 将合理的 β 取值设为 50 到 100, 则平均每个 24 bits 的 *K*ey 所需的存储空间为2.8 bits到5.6 bits, 与表3中实际测试得到的block RAM 的消耗量一致.

4 结语

本文设计并验证了一种适用于硬件实现的近似最小完美 Hash 函数. 通过对定长 Key 进行预处理后, 设计构建完美Hash函数, 利用简单的硬件逻辑电路实现, 可以支持满足几百 K 的大数据量高速表检索要求. 在表 4 中, 我们将该设计与典型的硬件 Hash 函数进行详细对比, 得出该 Hash 表设计具有电路结构简单、存储资源消耗低、支持动态更新以及可以支持几百 K 大小的数据集等特点. 该设计可以在需要高速实时表查找的各种场景, 包括在新型网络架构 NDN(Named Data Networking, 命名数据网络)得到应用[17], 解决软件实现的 Hash 函数查找速度无法满足大吞吐量的服务器以及骨干网中数据处理速度要求的问题. 另外对利用 SRAM 构建 CAM 电路中的存储单元提供方法参考.

表 4 本文提出方法与现有其他硬件 Hash 函数对比

	Cuckoo hashing [7]	Ficara [8]	Extended BF [13]	LogLog[15]	Gianni Antichi[16]	本方法
Hash 函数存储空间(bits/key)	N/A	2～4	50	3.5～5.6	4.0～4.7	2.8～5.6
硬件逻辑复杂度	视被选择的 Hash 函数的复杂度而定	简单	中等	简单	中等	简单
负载因子	最高 0.5	0.5 左右	0.25 左右(表压缩时可达 1.0)	～1.0	～1.0	0.7～1.0
可扩展性	视 Hash 函数而定	受限	好	中等	受限	好
是否支持新 key 持续插入	视条件而定(无限循环时. 重新 Hash)	否	是(Hash 表没压缩的情况下)	支持	否	是

参考文献

1 Dharmapurikar S, Krishnamurthy P, Sproull TS, et al. Deep Packet Inspection Using Parallel Bloom Filters. IEEE Micro, 2004, 24(1): 52–61.

2 杜飞,董治国,苗琳,等.基于无冲突哈希表和多比特树的两级 IPv6 路由查找算法.计算机应用,2013,33(5):1194–1196.

3 Xu Y, Liu ZB, Zhang ZY, et al. High-throughput and memory-efficient multimatch packet classification based on distributed and pipelined hash tables. IEEE/ACM Trans. on Networking, 2014, 22(3): 982–995.

4 Baker ZK, Prasanna VK. A computationally efficient engine for flexible intrusion detection. IEEE Trans. on VLSI Systems, 2005, 13(10): 1179–1189.

5 Alicherry M, Muthuprasanna M, Kumar V. High speed matching for network IDS/IPS. Proc. of the IEEE Int. Conf. on Network Protocols. Washington, DC. IEEE CS Press. 2006. 187–196.

6 Sourdis I, Pnevmatikatos D, Wong S, et al. A reconfigurable perfect-hashing scheme for packet inspection. Proc. of the IEEE Int. Conf. on Field Programmable Logic and Applications. Washington, DC. IEEE CS Press. 2005. 644–647.

7 Pagh R, Rodler FF. Cuckoo hashing. Journal of Algorithm, 2004, 51(2): 122–144.

8 Ficara D, Giordano S, Kumar S, et al. Divide and discriminate: Algorithm for deterministic and fast hash lookups. Proc. of ACM/IEEE ANCS. Los Alamitos, CA. IEEE CS Press. 2009. 133–142.

9 Kumar S, Turner J, Crowley P. Peacock hashing: Deterministic and updatable hashing for high performance networking. Proc. of IEEE INFOCOM. Washington, DC.

IEEE CS Press. 2008. 101–105.

10 István Z, Alonso G, Blott M, et al. A flexible hash table design for 10Gbps key-value stores on FPGAs. Proc. of the 23rd Int. Conf. on Field Programmable Logic and Applications (FPL 2013). Washington, DC. IEEE CS Press. 2013. 1–8.

11 Bando M, Artan NS, Chao HJ. Flashlook: 100-gbps hash-tuned route lookup architecture. Proc. of the Int. Conf. on High Performance Switching and Routing. Washington, DC. IEEE CS Press. 2009. 1–8.

12 Bloom B. Space/time trade-offs in hash coding with allowable errors. Communications of ACM, 1970, 13(7): 422–426.

13 Lu Y, Prabhakar B, Bonomi F. Perfect hashing for network applications. IEEE Symp. on Information Theory. Washington, DC. IEEE CS Press. 2006. 2774–2778.

14 Song H, Dharmapurikar S, Turner J, et al. Fast hash table lookup using extended bloom filter: An aid to network processing. ACM SIGCOMM. New York. ACM. 2005. 181–192.

15 Bando M, Artan NS, Chao HJ. Highly memory-efficient loglog hash for deep packet inspection. IEEE GLOBECOM. Washington, DC. IEEE CS Press. 2008. 1–6.

16 Antichi G, Ficara D, Giordano S, et al. Blooming trees for minimal perfect hashing. IEEE GLOBECOM. Washington, DC. IEEE CS Press. 2008. 1–5.

17 张良,刘敬浩,李卓.命名数据网络中基于 Hash 映射的命名检索.计算机工程,2014,40(4):108–111.