Recursive Lattice Search: Hierarchical Heavy Hitters Revisited

Kenjiro Cho
IIJ Research Laboratory
Tokyo, Japan
kjc@iijlab.net

ABSTRACT

The multidimensional Hierarchical Heavy Hitter (HHH) problem identifies significant clusters in traffic across multiple planes such as source and destination addresses, and has been widely studied in the literature. A compact summary of HHHs provides an overview on complex traffic behavior and is a powerful means for traffic monitoring and anomaly detection. In this paper, we present a new efficient HHH algorithm which fits operational needs. Our key insight is to revisit the commonly accepted definition of HHH, and apply the Z-ordering to make use of a recursive partitioning algorithm. The proposed algorithm produces summary outputs comparable to or even better in practice than the existing algorithms, and runs orders of magnitude faster for bitwise aggregation. We have implemented the algorithm into our open-source tool and have made longitudinal datasets of backbone traffic openly available.

CCS CONCEPTS

• **Networks** → **Network monitoring**; *Packet classification*;

KEYWORDS

hierarchical heavy hitters, flow aggregation algorithm, Z-order

ACM Reference Format:
Kenjiro Cho. 2017. Recursive Lattice Search: Hierarchical Heavy Hitters Revisited. In *Proceedings of IMC '17, London, United Kingdom, November 1–3, 2017,* 7 pages.
https://doi.org/10.1145/3131365.3131377

1 INTRODUCTION

Network-wide activities are often involved with many individual flows, and better presented by means of aggregated flows by their 5-tuple attributes. Identifying significant flow aggregates in traffic, known as the Hierarchical Heavy Hitter (HHH) problem, provides a powerful means for traffic monitoring as well as valuable components for anomaly detection to identify attacks and scans.

Algorithms for finding HHHs have been extensively studied in the literature. Nonetheless, they are not satisfactory for practical applications based on our experience developing and using HHH-based tools over the years [3, 12]. First, the performance is not good enough for multidimensional bitwise aggregation. Second, theoretical HHHs are not always relevant to operational needs, as reported HHHs include many broad and redundant ones (e.g., '128.0.0.0/4' and '128.0.0.0/2' are overlapping broad subnets). Third, we found it highly useful for interactive analysis to re-aggregate results for a coarser result (e.g., producing a daily result from 5-minute-long results) but most methods do not consider re-aggregation.

In this paper, we revisit the multi-dimensional HHH problem by introducing a new definition of HHH. Contrary to common practice, we apply the Z-ordering [16] to HHH so as to use an efficient recursive partitioning algorithm. Our contributions are (1) the proposed efficient algorithm for bitwise aggregation that matches operational needs and supports re-aggregation, and (2) the open-source traffic monitoring tool and the open dataset for the community. More broadly, the key contribution is to transform the existing hard problem into a tractable one by revisiting the commonly accepted definition.

2 BACKGROUND AND RELATED WORK

IP addresses are hierarchical and a node in the hierarchy represents an address range or a specific subnet. A node and its associated counts (e.g., packets or bytes) can be aggregated to a more generic (superset) node in the hierarchy, where they are called 'descendant' and 'ancestor'. An HHH for a total count N and a threshold $\phi \in (0, 1)$ is an aggregate with count $c \geq \phi N$.

Multi-dimensional aggregation has a rich history in database research (e.g., iceberg-cubes [2]). In networking, unidimensional HHH was introduced in early 2000s [3, 6, 8]. Extending HHH to multiple dimensions was considered by Estan *et al.* [8], and then, more formally explored by Cormode *et al.* [7].

In one-dimension, each node in the hierarchy has only one parent node, and HHHs can be uniquely determined by depth-first traversal: aggregating small nodes from lower positions in the hierarchy until the count of an aggregate exceeds the threshold. In n-dimensions, however, each node has n-parent nodes and there are many possible ways to aggregate. As a result, identifying multi-dimensional HHHs is much harder than unidimensional HHHs.

For simplicity, we focus on 2-dimensional HHH, using IPv4 source-destination address pairs in this paper. We use $[l_0, l_1]$ to denote a prefix length pair, $(p_0/l_0, p_1/l_1)$ for a prefix pair, and '$*$' to represent wildcards ('0' for prefix length, '0/0' for prefix).

When a l-bit space has g-bit granularity, it is divided into $h = l/g$ subspaces by $h' = h + 1$ hyperplanes (including ones at both ends). For IPv4 addresses ($l = 32$) with 8 bit granularity ($g = 8$), $h = 4$ and $h' = 5$. Possible aggregations with different prefix length

IMC '17, November 1–3, 2017, London, United Kingdom

ACM ISBN 978-1-4503-5118-8/17/11…$15.00
https://doi.org/10.1145/3131365.3131377

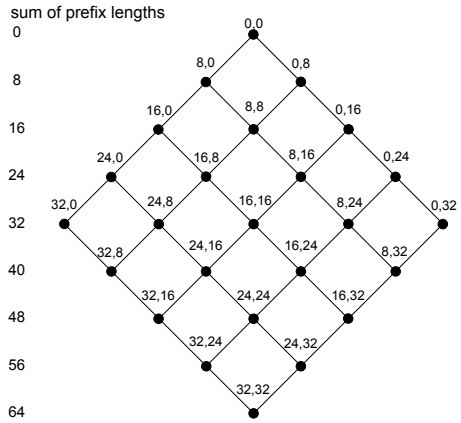


Figure 1: Lattice for a IPv4 prefix length pair $[l_0, l_1]$ with 8-bit granularity ($g = 8$)

pairs can be represented by a lattice ordered by the sum of prefix lengths [7]. Figure 1 shows the lattice for an IPv4 address pair with $g = 8$, having the size of the lattice $h' \times h' = 5 \times 5 = 25$. (when $g = 1$, $h' \times h' = 33 \times 33 = 1089$). For example, (1.2.3.4, 5.6.7.8) with $[32, 32]$ at the bottom can be aggregated to any other node in the lattice: (1.2.3.4/32, 5.6.7.0/24) with $[32, 24]$, (1.2.3.0/24, 5.6.7.8/32) with $[24, 32]$, ... , (1.2.3.0/24, 5.6.7.0/24) with $[24, 24]$, ... , up to (0.0.0.0/0, 0.0.0.0/0) or $(*, *)$ with $[0, 0]$.

There exist different definitions and the corresponding algorithms introduced by Cormode *et al.* [5–7] and subsequently by many others. Here, we briefly review the relevant definitions.

Most of the existing methods employ 'discounted HHH' where descendant HHHs are not double-counted in their ancestors' counts so as to make outputs concise and compact. Otherwise, all ancestors of an HHH become HHHs, which is too redundant. We also employ the discounted HHH.

The discounted count c' for node i is the sum of its non-HHH direct children's discounted counts and can be computed from the bottom of the lattice:

$$c'_i = \sum_j c'_j \qquad where \quad \{\, j \in child(i) \mid c'_j < \phi N \,\} \tag{1}$$

A naive algorithm in 2-dimensions goes through all possible prefix length pairs in the lattice in the decreasing order of the sum of prefix lengths. For each node in the lattice $[l_0, l_1]$, it goes through every input (p_0, p_1) making the corresponding aggregate $(p_0/l_0, p_1/l_1)$ and accumulating the count for the aggregate. After processing all the inputs, it extracts aggregates as HHHs when $c' \geq \phi N$ and then their corresponding inputs are removed for 'discount' (instead of keeping track of direct children's counts). The cost is $O(h'^2 N \log N)$; $O(h'^2 N)$ for going through the entire inputs for every node in the lattice, and $O(\log N)$ to find $(p_0/l_0, p_1/l_1)$, though the latter can be optimized to $O(1)$. Hence, it is costly to use $g = 1$ for IPv4 address pairs, and it becomes even worse for IPv6.

Another factor is the rollup rule: how to roll up counts to parents. Cormode *et al.* classify rollup rules into 2 categories: overlap and split. The former allows double-counting among nodes if they do not have ancestor-descendant relationship, and rolls up counts to both parents in order to identify all possible HHHs. The latter preserves the counts and splits a count between its parents using some split function (e.g., first found, even split). The overlap rule produces more HHHs than the split rule; for a threshold ϕ, the number of discounted HHHs is at most $1/\phi$ for the split rule but A/ϕ for the overlap rule where A denotes the length of the longest width of the lattice (e.g., $A = h' = 33$ for $g = 1$) [5].

Our requirement is to preserve counts because our tool is designed to re-aggregate outputs to produce coarser grained outputs and for interactive analysis, and double-counting distorts re-aggregated results. Thus, we use a simple split rule that rolls up counts to the first found ancestor HHH, depending on the aggregation ordering. Almost all of the existing methods use the sum of prefix lengths for ordering. We will revisit this commonly accepted ordering in the next section.

The idea behind the HHH problem is that inputs are skewed and sparse in space. To this end, various data structures are devised to efficiently keep track of inputs such as grid-of-trie, rectangle-search and cross-producting [13, 20, 24], or exploiting TCAM or FPGA [11, 17, 18, 21]. For example, the cross-producting method first aggregates inputs along each dimension separately to form a compact n-dimensional matrix. The inputs are then mapped to the corresponding entry in the matrix by longest prefix matching along each dimension, and finally to make a summary, it aggregates entries smaller than the threshold. In contrast, our algorithm is simple space partitioning and uses no elaborate data structure.

Most of the theoretical studies investigate streaming approximation algorithms and their error bounds [1, 5–7, 10, 15]. There are well known streaming algorithms to find frequent items using a limited number of counters, and they are extended to apply to the HHH problem by using $1/\phi$ counters for each node in the lattice. Recently, Ben Basat *et al.* tackle faster online HHH computation by applying randomized sampling for updating counters in exchange for slower convergence [1], whose motivation is common to one of ours. In Section 4, we compare our method with the Space-Saving algorithm [15] by Mitzenmacher *et al.* as a baseline.

All these algorithms are bottom-up, probably due to the fact that the bottom-up approach is natural for unidimensional HHH and all the algorithms extend unidimensional HHH to multi-dimensions. To the best of our knowledge, ours is the first top-down HHH algorithm, albeit flow partitioning itself is hardly new (e.g., [23]).

Our insight is to revisit the definition of HHH. The aggregation order in almost all of the existing HHH algorithms employ the sum of prefix lengths, which is intuitive for sorting aggregates from more specific to less specific but does not always match operational needs. For example, $[32, *]$ and $[16, 16]$ are at the same level in the prefix length sum order. From the operational view, however, the former is more important as a specific source sending to diverse destinations (e.g., scanning) while the latter only identifies broad address space for sources and destinations and does not require immediate attention from operators. Also, the existing methods tend to produce broad aggregates with very short prefix lengths falling under the upper lattice.

We introduce a different order by redefining $child(i)$ in Equation 1 to take advantage of the underlying multi-dimensional hierarchical structure. It allows top-down recursive partitioning, while making the full prefix length (/32 for IPv4) higher in the order.

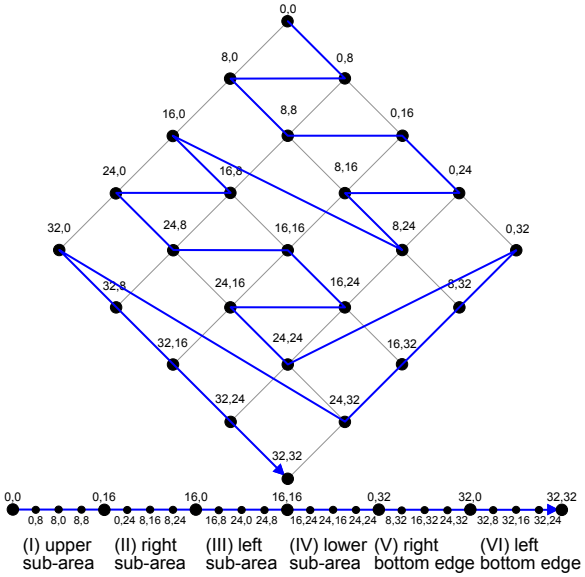


Figure 2: Z-order on the IPv4 prefix length pair lattice

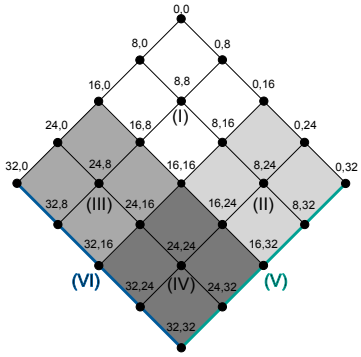


Figure 3: Recursive Lattice Search with 6 Regions

3 RECURSIVE LATTICE SEARCH

Our new algorithm is Recursive Lattice Search (RLS). The specific order to use in our algorithm is the Z-order introduced by Morton in 1966 [16]. The Z-order is an ordering along a space filling curve while preserving locality, preferring the largest value across all dimensions. The Z-value is simply calculated by interleaving the binary representations of two prefix length values. When applied to an IPv4 prefix length pair $[a, b]$, each dimension needs 5 bits for [0...32]: $a = a_4a_3...a_0$ and $b = b_4b_3...b_0$. The Z-value is 10 bits, $z = a_4b_4a_3b_3...a_0b_0$, for ordering the nodes in the lattice.

The Z-order on the 2-dimensional lattice with $g = 8$ is shown in Figure 2. It looks slightly different from a standard Z-curve at the bottom edges, because the maximum prefix length is 32 and it does not have the entire 5 bit space. As a result, the uncovered space is collapsed onto the bottom edges. It has a favorable effect that having a full prefix length in either dimension becomes higher in the order, which meets the operational bias for detecting DDoS attacks and scanning. The line in the bottom of the figure shows how the lattice nodes are ordered; descendant nodes are placed

Algorithm 1 Recursive Lattice Search

```
procedure LatticeSearch(parent, l0, l1, Δ, pos)
    doAggregate ← TRUE, doRecurse ← TRUE
    if pos = UPPER then
        doAggregate ← FALSE                          ▷ already aggregated by the caller
    if Δ = minGranularity and not next to the very bottom node then
        doRecurse ← FALSE
    if doAggregate = FALSE and doRecurse = FALSE then
        return                                       ▷ nothing to do
    if doAggregate then
        aggregateList ← Aggregate(inputsOf(parent), l0, l1)
    else
        aggregateList ← InheritFrom(parent)
    if doRecurse then
        if Δ ≠ minGranularity then
            Δ ← Δ/2                                  ▷ halve granularity
        for all f in aggregateList do
            if count(f) ≥ thresh then
                if not on bottom edge then
                    LatticeSearch(f, l0+Δ, l1+Δ, Δ, LOWER)    ▷ recurse DOWN
                if not on left bottom edge then
                    LatticeSearch(f, l0+Δ, l1, Δ, LEFT)       ▷ recurse LEFT
                if not on right bottom edge then
                    LatticeSearch(f, l0, l1+Δ, Δ, RIGHT)      ▷ recurse RIGHT
                LatticeSearch(f, l0, l1, Δ, UPPER)            ▷ recurse UP
    if doAggregate then
        for all f in aggregateList do
            if count(f) ≥ thresh then                ▷ check again for discount rule
                hhh ← f                              ▷ extract f as HHH
                discount inputs for f

                                  ▷ Starting the recursive lattice search for IPv4 address pairs
root ← all inputs
LatticeSearch(root, 32, 0, 32, RIGHT)                ▷ visit left bottom edge
LatticeSearch(root, 0, 32, 32, LEFT)                 ▷ visit right bottom edge
LatticeSearch(root, 0, 0, 32, LOWER)                 ▷ visit sub-areas
```

close to their ancestors. The ordering can be divided into 6 regions from (I) through (VI), the first four regions for the internal nodes and the last two for the nodes on the bottom edges.

The Z-order changes the parent-child relationship in the lattice from a binary tree to a quadtree [9, 19], which transforms the HHH problem into simple space partitioning of a quadtree. To obtain finer granularity, it suffices to recurse the partitioning until the desired granularity, making the previous $O(h'^2N)$ algorithm into $O(N \log h)$. Differing from the other methods, the algorithm is deterministic, requires no parameter other than ϕ, and produces a unique result without any approximation.

To aggregate inputs following the Z-order, the algorithm visits the regions in Figure 3 in the reverse order from (VI) to (I). When aggregating on the left bottom edge (VI), the algorithm first tries to aggregate inputs with [32, 0], and detects all HHHs having the full prefix length on the first dimension. Then, it recursively subdivides each detected HHH along the second dimension, using [32, 16],[32, 24], [32, 32]. It works similarly on the right bottom edge (V) in the order [0, 32], [16, 32],[24, 32], and at this point finds all HHHs having a full prefix length in either dimension. For the remaining internal nodes from (IV) to (I), it partitions the two-dimensional space by recursively subdividing it into four quadrants, from the lower quadrant (IV) with [16, 16], the left quadrant (III) with [16, 0], the right quadrant (II) with [0, 16] to the remaining upper quadrant (I). Each quadrant can be further recursively subdivided into quadrants.

Note that the algorithm recurses only for nodes larger than the threshold, and the subdivision is only on the constituent inputs of the caller. The caller performs threshold checking again after the recursions, since its count could have been decreased by the

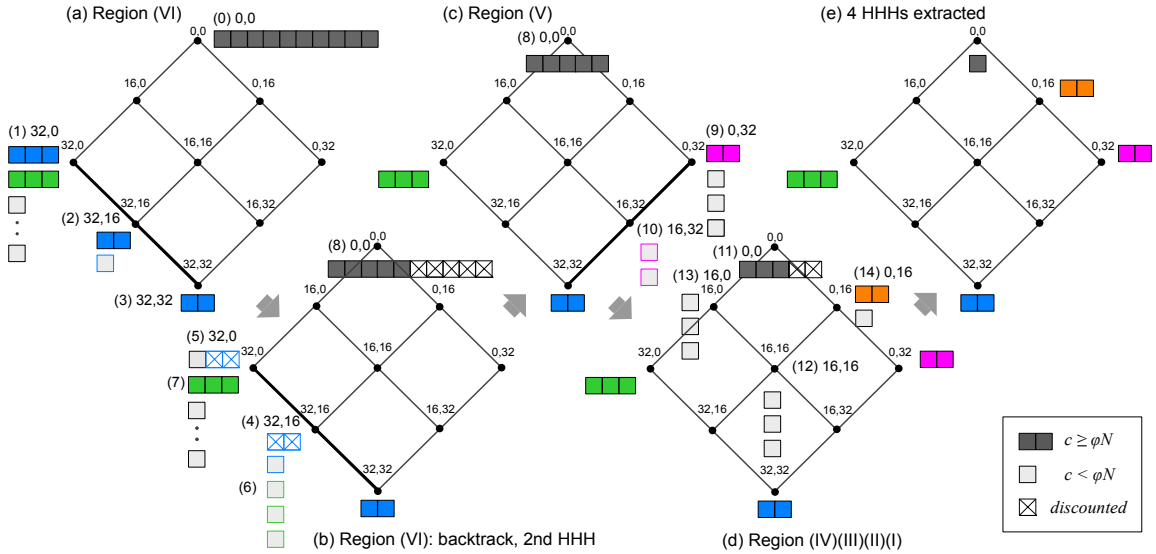


Figure 4: Recursive Lattice Search illustrated: a step-by-step example using a 3×3 lattice for HHHs with $c' \geq 2$

(a) Initially, all 10 inputs are at the root [0,0] (step 0). RLS starts with Region (VI), the left bottom edge. The initial aggregation at [32,0] finds 2 HHHs (blue, and green); each count is 3 (step 1). The blue HHH is further aggregated with [32,16] (step 2), and then, further again with [32,32] (step 3). This HHH is extracted. (b) On the return path, the blue HHH at [32,16] and [32,0] were already discounted (step 4-5). Similarly, the green HHH is tried with [32,16] but no HHH is found (step 6) so that the green HHH is extracted at [32,0] (step 7). (c) Next, RLS visits Region (V), the right bottom edge. At this point, 5 inputs remain at the root (step 8). The pink HHH is found at [0,32] (step 9) but it cannot be further aggregated at [16,32] (step 10) so that the pink HHH is extracted at [0,32]. (d) Third, RLS visits internal nodes with 3 remaining inputs at the root (step 11). No HHH is found in Region (IV), the lower sub-area at [16,16] (step 12), or in Region (III), the left sub-area at [16,0] (step 13). The orange HHH is found in Region (II), the right sub-area at [0,16] (step 14). (e) Finally, the search terminates with 4 HHHs and 1 remaining input at the root.

descendants for the discount rule. The pseudo code of RLS is given in Algorithm 1. Figure 4 presents a step-by-step example using a 3×3 lattice, where 4 HHHs ($c' \geq 2$) are extracted from 10 inputs.

The Z-ordering, however, introduces a bias for the first dimension, and the counts along the second dimension could be undercounted as a result. Still, this bias does not significantly affect results for real traffic as we will see in Section 4.

It is possible to extend the algorithm for higher dimensions. For 3-dimensions, the lattice becomes a cube, also known as an octree [14]. For partitioning, it first visits the 3 bottom edges, then the 3 bottom faces, and finally the internal cubic space to be partitioned into 8 sub-cubes. For n-dimensions, the number of binary subspaces grows with 2^n but the depth of recursions required for space partitioning remains $\log h$.

4 EVALUATION

For evaluation, we have ported our RLS code implemented in our tool to the simulation code for the Space-Saving algorithm by Mitzenmacher *et al.* [15] [1], because the code is publicly available and written in C, the programming language we used to implement the tool. We use a packet trace from the WIDE MAWI archive [4], a 15-minute-long packet trace taken on October 20, 2016 at 14:00 JST, containing about 73 million IPv4 packets[2]. The IP addresses in the packet trace are anonymized in a prefix-preserving manner so that HHHs are preserved. The simulator uses only packet counts for IPv4 source-destination address pairs.

First, we observe the sensitivity in the source-destination order in the Z-order in Table 1, by comparing outputs aggregated by (src, dst) and (dst, src), with $\phi = 0.05$ (5%), $N = 10^6$ and $g = 1$. The rightmost column shows each HHH's count share, c'/N, in percent. Both report 15 HHHs that are very similar with only minor differences: both report identical HHHs from (1) through (12) with identical percentage. The differences highlighted by the bold fonts are only HHHs from (13) through (15) in Region (I) with very short prefix lengths. We have similar results with other traces or with varying ϕ and N. The result confirms that the bias introduced by the source-destination order is negligible in practice.

Next, we compare the output of RLS to that of Space-Saving (SS) as a baseline that was already compared with other algorithms in [15]. Because we use a different definition for HHH, this is not intended to make head-to-head comparison with other algorithms but provided only to illustrate major differences. Note that, other than the ordering, SS is a streaming approximation algorithm, using the overlapping rollup rule (double-counting), and the code is not optimized especially for bitwise aggregation. The simulation code for SS has bitwise aggregation only for one-dimension so that we have modified the code for 2-dimensional bitwise aggregation. Again, we use $\phi = 0.05$, $N = 10^6$ and $g = 1$, along with the SS error bound parameter $\epsilon = 0.01$.

The output of RLS is compared to that of SS in Table 2. The 15 HHHs reported by RLS correspond to ones for (src, dst) in Table 1,

[1] The modified simulator is at https://github.com/kjc0066/hhh/
[2] http://mawi.wide.ad.jp/mawi/samplepoint-F/2016/201610201400.html

Table 1: Sensitivity of source-destination order

region	no	aggregated by (src,dst) src	dst	c'/N(%)
VI	(1)	112.31.100.1/32	163.229.97.230/32	16.5
V	(2)	64.0.0.0/2	202.203.3.13/32	5.2
	(3)	128.0.0.0/1	202.203.3.13/32	5.8
	(4)	*	202.26.162.46/32	6.0
III	(5)	163.229.96.0/23	*	5.0
	(6)	203.179.128.0/20	*	6.8
II	(7)	*	202.203.3.0/24	5.9
	(8)	*	203.179.140.0/23	5.7
	(9)	*	163.229.128.0/17	5.1
I	(10)	0.0.0.0/1	202.192.0.0/12	5.3
	(11)	202.192.0.0/12	*	6.7
	(12)	*	202.0.0.0/7	7.6
	(13)	**128.0.0.0/4**	*	**5.0**
	(14)	**128.0.0.0/2**	*	**6.0**
	(15)	*	**128.0.0.0/2**	**5.4**
-		*	*	**2.0**
				100.0
		aggregated by (dst,src)		
	(1)-(12)	identical to (src,dst)		
I	(13)	**128.0.0.0/2**	**0.0.0.0/2**	**5.7**
	(14)	*	**128.0.0.0/3**	**5.3**
	(15)	**128.0.0.0/1**	*	**6.4**
-		*	*	**1.0**

Table 2: Output of RLS compared to SS

no	RLS(%)	SS(%)	missing SS HHHs with their c'/N(%)
(1)	16.5	16.5	-
(2)	5.2	5.2	-
(3)	5.8	5.8	-
(4)	6.0	6.0	-
(5)	5.0	5.0	-
(6)	6.8	6.8	-
(7)	**5.9**	**16.9**	-
(8)	5.7	5.7	-
(9)	5.1	5.1	-
(10)	**5.3**	-	(96/**3**,202.203/**16**):5.4 (0/**2**,202.203/**16**):5.6 (112/**4**,202.192/**12**):5.2 (64/**2**,202.192/**12**):9.0
(11)	6.7	6.7	-
(12)	**7.6**	-	(0/**1**,203.179.128/**20**):6.0 (128/**2**,202.203/**16**):5.5 (192/**4**,202/**8**):5.1 (*,202.192/**12**):25.5 (16/**4**,202/**7**):5.4 (128/**1**,202.128/**9**):10.6 (64/**2**,202/**7**):15.5 (128/**1**,202/**7**):17.7
(13)	**5.0**	**5.2**	-
(14)	**6.0**	-	(163.229/**16**,0/**1**):6.0 (144/**4**,128/**1**):5.3 (128/**2**,96/**3**):5.0 (128/**3**,0/**1**):5.3 (160/**3**,128/**1**):7.0 (128/**2**,0/**2**):5.7 (128/**2**,0/**1**):11.4
(15)	**5.4**	**33.1**	(128/**1**,160/**6**):5.0 (192/**4**,128/**2**):5.2 (0/**1**,128/**2**):22.7 (*,128/**3**):7.1
-	**2.0**	-	(202/**7**,0/**2**):5.4 (192/**8**,128/**1**):5.6 (202/**8**,0/**1**):5.7 (202/**7**,128/**1**):6.0 (192/**3**,200/**5**):10.5 (128/**1**,112/**6**):5.1 (112/**5**,128/**1**):21.8 (200/**5**,*):17.0 (192/**4**,128/**1**):13.6 (128/**1**,16/**4**):6.2 (*,200/**5**):42.4 (64/**3**,128/**1**):6.0 (96/**3**,128/**1**):29.7 (128/**1**,64/**2**):10.4 (0/**1**,128/**1**):46.7 (128/**1**,*):53.3 (*,128/**1**):78.3

while SS reports much larger 52 HHHs due to its overlap rollup rule resulting in the inflated total count of $\sum c'/N = 6.84$ by double-counting. Even though the two methods differ in the definition (ordering and rollup rules), the results are comparable. Both methods have the identical HHHs for (1)-(9), with one exception that (7) has different percentage due to the different rollup rules. The HHHs only appearing in SS are listed in the rightmost column under the corresponding ancestor in RLS. Among 40 HHHs found only by SS, 35 fall into Region (I), four into Region (II) and one into Region (III). Most of them have very short prefix lengths, and do not add much information for operational purposes. Although some have longer prefixes, they are due to double-counting by the overlap rule and their related (more representative) HHHs can be found in the RLS output (e.g., (8) for (0/1, 203.179.128/20)). Generally, HHHs in Region (I) are not so informative; we will introduce heuristics to further suppress such HHHs in Section 5.2.

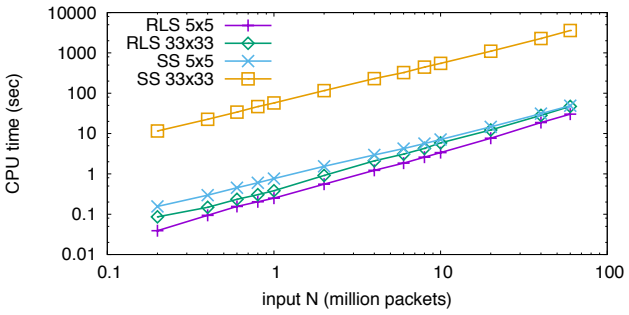
Overall, capturing noteworthy HHHs is not so sensitive to subtle differences in the aggregation ordering or the rollup rule. The overlap rule of SS produces a lengthy and redundant summary for bitwise aggregation, while concise and compact reports by RLS better meet needs for traffic monitoring and anomaly detection.

Finally, we compare the CPU time and memory usage of RLS and SS for bytewise (5×5) and bitwise (33×33) aggregations in Figure 5, using a standard desktop PC with a 4-core CPU (Intel Core-i7 3770K 3.5GHz) and 16GB DRAM. The CPU time grows linearly with the input size for all cases. The difference between bytewise and bitwise is less than a factor of 2 for RLS while a factor of 70 for SS. RLS is about twice faster than SS for bytewise aggregation, and about 100 times faster for bitwise aggregation. RLS can process more than 2M packets per second for bitwise aggregation, corresponding to 10Gbps with mean packet size of 512 bytes. The memory usage of RLS is proportional to the input size as a non-streaming algorithm that starts with all inputs buffered, while the SS uses a fixed size of memory. However, several GB of memory usage is not an issue for modern PCs and, if needed, we can use a shorter aggregation period and re-aggregate the results for a summary report as described in Section 5.3.

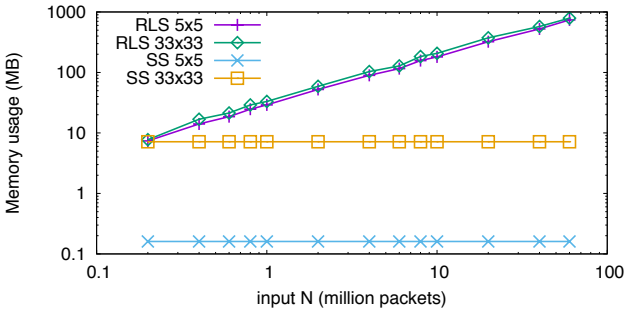
5 IMPLEMENTATION

In this section, we briefly cover our implementations related to the algorithm. We have developed an HHH-based traffic monitoring tool named *agurim* [12], and we are using it to monitor the WIDE [22] backbone traffic since 2013. *Agurim* extensively uses re-aggregation. The primary aggregation creates a rudimentary list of aggregated flows by efficiently processing raw traffic data such as pcap, NetFlow and sFlow. The secondary aggregation re-aggregates its own (primary and secondary) outputs to update coarser hourly and daily records. For visualizing time-series, records with an appropriate temporal granularity for the plotting period are selected and further re-aggregated. A user can dynamically switch views based on traffic volume or packet counts, address or protocol attributes, with different temporal and spatial granularities on the Web user interface. In addition, we have made anonymized datasets openly available to provide broader access to backbone traffic for the networking community[3].

[3] http://mawi.wide.ad.jp/~agurim/ (add 'dataset/' for raw data)



(a) CPU Time



(b) Memory Usage

Figure 5: Performance Comparison: RLS vs. SS

We started using *agurim* for traffic monitoring since February 2013. Our original implementation [12] employed a variant of the cross-producting method for the primary aggregation and a variant of the naive algorithm for the secondary aggregation. We switched to the new RLS algorithm for the secondary aggregation since May 2016, and for the primary aggregation since December 2016.

5.1 2-Level HHH

In the networking context, it makes sense to have two planes: one for address pairs and the other for port pairs. Although the HHH algorithm can be extended for four dimensions, the search space grows exponentially with the dimension. Thus, *agurim* employs two levels of 2-dimensional HHH: the first level for the main attribute pairs (source-destination addresses) and the second level for the sub-attribute pairs (source-destination ports) under each main attribute pair. *Agurim* allows one to swap the main attribute and sub-attribute for the protocol-port view in which flows are aggregated first by port pairs and then by address pairs within each port pair.

5.2 Protocol Specific Heuristics

RLS allows one to control aggregation granularity for part of the hierarchy by limiting the depth of recursions. Using protocol specific knowledge, we have added heuristics to suppress entries not so useful for operation and to make concise summaries. For IPv4, when a prefix length is shorter than 16, the granularity is reduced from $g = 1$ to $g = 8$ so as not to produce HHHs with short prefix lengths. Similarly, for IPv6, when a prefix length is shorter than 32, the granularity is reduced to $g = 16$. In addition, the prefix length in the range of [65, 127] is not aggregated, as the lower 64 bits of an IPv6 address are used for an interface ID and are not hierarchical. For ports, we found that, even though some applications use certain port ranges, they are rare in occurrence and often buried in noise, so that we use only a wildcard for aggregation.

5.3 Online Processing

Our original motivation was to use RLS for *agurim*'s secondary aggregation whose inputs are limited in number. We realized, however, that RLS is fast enough to be used for the primary aggregation, and using the same algorithm makes the code simpler and outputs more consistent. For online processing, we employ multi-threading and double-buffering to take advantage of a multi-core CPU; one thread keeps reading raw inputs, switches input buffers at the end of aggregation periods and wakes up another thread that aggregates the inputs in the buffer and produces a summary report.

The current bottleneck in *agurim* is not the aggregation algorithm but the cost to maintain the inputs. We use a hash table to keep track of input packets by their 5-tuple. The larger the hash table grows, the higher the cost becomes for search.

To control the number of inputs in the hash table as well as to reduce memory usage, a user can optionally set the aggregation period to a fraction of the summary period. Then, inputs are aggregated in a shorter cycle (e.g., every 3 seconds) and a summary is produced by re-aggregating the intermediate results in a longer cycle (e.g., every 30 seconds). The same technique is used for DoS resilience; when the buffered packet count exceeds a predefined limit, early aggregation is invoked.

6 CONCLUSION

In this paper we have introduced a new efficient HHH algorithm. Our key insight is to revisit the commonly accepted definition of HHH, and apply the Z-ordering to make use of a recursive partitioning algorithm. The Z-order makes the ordering consistent with ancestor-descendant relationship in the hierarchy, and it transforms the HHH problem into simple space partitioning of a quadtree.

The proposed algorithm produces concise and compact summaries capturing HHHs, satisfies our operational needs, and runs faster than the existing methods by orders of magnitude for bitwise aggregation.

This work is part of our ongoing effort to provide practical tools and datasets for traffic monitoring and networking research. The proposed algorithm has been integrated into our traffic monitoring tool and used for operation. The source code of the tool is available along with open longitudinal dataset starting from 2013 that can be browsed on the Web[4].

ACKNOWLEDGMENTS

We thank Midori Kato and Arthur Carcano for contributing to the *agurim* development, and Yuji Sekiya and Ryo Nakamura for operational support. We would like to thank Kensuke Fukuda, Romain Fontugne, the anonymous IMC reviewers and our shepherd, John Byers, for their valuable feedback and comments on the paper.

[4]More information is available at http://mawi.wide.ad.jp/~agurim/about.html

REFERENCES

[1] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo C. Luizelli, and Erez Waisbard. 2017. Constant Time Updates in Hierarchical Heavy Hitters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 127–140. https://doi.org/10.1145/3098822.3098832

[2] Kevin Beyer and Raghu Ramakrishnan. 1999. Bottom-up Computation of Sparse and Iceberg CUBE. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99)*. ACM, New York, NY, USA, 359–370. https://doi.org/10.1145/304182.304214

[3] Kenjiro Cho, Ryo Kaizaki, and Akira Kato. 2001. Aguri: An Aggregation-Based Traffic Profiler. In *Proceedings of the Second International Workshop on Quality of Future Internet Services (COST 263)*. Springer-Verlag, London, UK, UK, 222–242. http://dl.acm.org/citation.cfm?id=646462.693721

[4] Kenjiro Cho, Koushirou Mitsuya, and Akira Kato. 2000. Traffic Data Repository at the WIDE Project. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '00)*. USENIX Association, Berkeley, CA, USA, 51–51. http://dl.acm.org/citation.cfm?id=1267724.1267775

[5] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding Frequent Items in Data Streams. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1530–1541. https://doi.org/10.14778/1454159.1454225

[6] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. 2003. Finding Hierarchical Heavy Hitters in Data Streams. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29 (VLDB '03)*. VLDB Endowment, 464–475. http://dl.acm.org/citation.cfm?id=1315451.1315492

[7] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. 2004. Diamond in the Rough: Finding Hierarchical Heavy Hitters in Multi-dimensional Data. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*. ACM, New York, NY, USA, 155–166. https://doi.org/10.1145/1007568.1007588

[8] Cristian Estan, Stefan Savage, and George Varghese. 2003. Automatically Inferring Patterns of Resource Consumption in Network Traffic. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '03)*. ACM, New York, NY, USA, 137–148. https://doi.org/10.1145/863955.863972

[9] R. A. Finkel and J. L. Bentley. 1974. Quad Trees a Data Structure for Retrieval on Composite Keys. *Acta Inf.* 4, 1 (March 1974), 1–9. https://doi.org/10.1007/BF00288933

[10] John Hershberger, Nisheeth Shrivastava, Subhash Suri, and Csaba D. Tóth. 2005. Space Complexity of Hierarchical Heavy Hitters in Multi-dimensional Data Streams. In *Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '05)*. ACM, New York, NY, USA, 338–347. https://doi.org/10.1145/1065167.1065211

[11] Lavanya Jose, Minlan Yu, and Jennifer Rexford. 2011. Online Measurement of Large Traffic Aggregates on Commodity Switches. In *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE'11)*. USENIX Association, Berkeley, CA, USA, 13–13. http://dl.acm.org/citation.cfm?id=1972422.1972439

[12] Midori Kato, Kenjiro Cho, Michio Honda, and Hideyuki Tokuda. 2012. Monitoring the Dynamics of Network Traffic by Recursive Multi-Dimensional Aggregation. In *Presented as part of the 2012 Workshop on Managing Systems Automatically and Dynamically*. USENIX, Hollywood, CA. https://www.usenix.org/conference/mad12/workshop-program/presentation/Kato

[13] Yunqi Li, Jiahai Yang, Changqing An, and Hui Zhang. 2007. Finding Hierarchical Heavy Hitters in Network Measurement System. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC '07)*. ACM, New York, NY, USA, 232–236. https://doi.org/10.1145/1244002.1244061

[14] Donald Meagher. 1982. Geometric Modeling Using Octree Encoding. *Computer Graphics and Image Processing* 19 (1982), 249–270.

[15] M. Mitzenmacher, T. Steinke, and J. Thaler. 2012. Hierarchical Heavy Hitters with the Space Saving Algorithm. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments (ALENEX '12)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 160–174. http://dl.acm.org/citation.cfm?id=2790265.2790281

[16] G. M. Morton. 1966. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. Technical Report. IBM Ltd.

[17] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2014. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 419–430. https://doi.org/10.1145/2619239.2626291

[18] Diana Andreea Popescu, Gianni Antichi, and Andrew W. Moore. 2017. Enabling Fast Hierarchical Heavy Hitter Detection Using Programmable Data Planes. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. ACM, New York, NY, USA, 191–192. https://doi.org/10.1145/3050220.3060606

[19] Hanan Samet. 1984. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.* 16, 2 (June 1984), 187–260. https://doi.org/10.1145/356924.356930

[20] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. 1998. Fast and Scalable Layer Four Switching. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '98)*. ACM, New York, NY, USA, 191–202. https://doi.org/10.1145/285237.285282

[21] Da Tong and Viktor Prasanna. 2015. High Throughput Hierarchical Heavy Hitter Detection in Data Streams. In *Proceedings of the 2015 IEEE 22Nd International Conference on High Performance Computing (HiPC) (HIPC '15)*. IEEE Computer Society, Washington, DC, USA, 224–233. https://doi.org/10.1109/HiPC.2015.30

[22] WIDE Project 2017. WIDE Project web page. (2017). Retrieved September 28, 2017 from http://www.wide.ad.jp/

[23] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. 2007. ProgME: Towards Programmable Network Measurement. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '07)*. ACM, New York, NY, USA, 97–108. https://doi.org/10.1145/1282380.1282392

[24] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick Duffield, and Carsten Lund. 2004. Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Applications. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement (IMC '04)*. ACM, New York, NY, USA, 101–114. https://doi.org/10.1145/1028788.1028802