

Fast Data Stream Algorithms using Associative Memories*

Nagender Bandi Ahmed Metwally Divyakant Agrawal Amr El Abbadi
Department of Computer Science
University of California, Santa Barbara
{nagender, metwally, agrawal, amr}@cs.ucsb.edu

ABSTRACT

The primary goal of data stream research is to develop space and time efficient solutions for answering continuous on-line summarization queries. Research efforts over the last decade have resulted in a number of efficient algorithms with varying degrees of space and time complexities. While these techniques are developed in a standard CPU setting, many of their applications such as click-fraud detection and network-traffic summarization typically execute on special networking architectures called Network Processing Units (NPU). These NPUs interface with a special type of associative memories, known as Ternary Content Addressable Memories (TCAMs), to provide gigabit rate forwarding at network routers. In this paper, we describe how the integrated architecture of NPU and TCAMs can be exploited towards achieving the goal of developing high-speed stream summarization solutions. We propose two TCAM-conscious solutions for the frequent elements problem in data streams and present a comprehensive evaluation of these techniques on a state-of-the-art networking platform.

Categories and Subject Descriptors

H.4 [Information Systems Applications]: Miscellaneous

General Terms

Algorithms, Design, Performance

Keywords

Data Streams, Hardware, TCAMs

1. INTRODUCTION

During the last decade, researchers in the data stream community have presented diverse solutions for answering continuous summarization queries over data streams. Some

*Partly supported by NSF grants IIS 02-20152, IIS 02-23022 and CF 04-23336.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD'07, June 12–14, 2007, Beijing, China.

Copyright 2007 ACM 978-1-59593-686-8/07/0006 ...\$5.00.

of the popular summarization problems are finding the most frequent elements [18, 10, 8, 20], and quantile summarization [19, 14] of a data stream. Given a continuous data stream, the frequent elements problem involves finding the data elements whose frequency of occurrence in the data stream is greater than a user-defined threshold. Quantile summarization provides other statistics over the data stream such as the median and histograms. These problems have a wide range of applications such as click fraud detection, network-traffic summarization, online analysis of stock market data etc. For example, a network administrator who wishes to know all the network flows which contribute more than 1% of network link capacity at a particular router would issue the frequent element query with a threshold of 1%. Similarly, a stock market analyst would issue a quantile query to get approximate representation of the stock transactions on a particular day.

As data stream algorithms operate over an infinite stream of elements where an element is accessed only once, they are designed to be efficient in terms of both space and time complexity. Typically, these techniques maintain data structures such as binary search trees, heaps and priority queues, which aid in executing only a bounded number of operations per stream element. However these bounds are typically logarithmic or amortized $O(1)$ cost (with a relatively large constant) in terms of the stream size or the data space. Furthermore, these data structures are built using pointers which are known to be inefficient in terms of memory access cost. With these basic data stream algorithms forming the core, a number of data stream processing systems such as STREAMS [25], TelegraphCQ [6], Aurora [3], Gigascope [9] etc. have been proposed. Although these systems achieve good stream processing throughput, with the ever increasing generation rates of real life data streams, there is a need for faster solutions.

We observe that many applications such as click-fraud detection and network-traffic summarization execute on network routers which operate on non-conventional hardware called Network Processing Units (NPUs). While the conventional processor architectures have not changed much over the last decade, networking hardware has evolved significantly. In particular, the memory hierarchy of NPUs features a Ternary Content Addressable Memory (TCAM) which enables constant time lookup for an element among a large collection of elements. Off-the-shelf TCAMs provide *constant time* searches at speeds of 100 million searches per second and provide this throughput irrespective of the size of data repository. In this paper, we describe how NPUs

and TCAMs can be exploited to provide faster solutions for data stream problems. In particular, we describe how some of the well known algorithms proposed for conventional processor architectures can be adapted to the Associative Memory model of a TCAM towards providing highly efficient implementations.

The main contributions of our paper are :

- Although NPU and the integrated TCAM architecture is widely used in the networking world for providing fast packet forwarding, to the best of our knowledge, we are the first to use this architecture for providing data stream summarization solutions.
- We present two TCAM-conscious techniques for finding the most frequent elements in a data stream. These techniques incur a constant number of TCAM operations per stream element unlike their software counterparts which require a significantly higher number of operations per element.
- We compare our techniques with another TCAM-based technique called the Range Adaptive Profiling (RAP) [22] used in computer architecture for profiling most frequently used blocks of code. Through our cost analysis, we show that not only do our techniques optimally use the memory space but are also efficient compared to RAP in terms of the number of TCAM operations.
- To the best of our knowledge, this is the first attempt to implement TCAM-conscious data stream algorithms on a real NPU system. Through our experimental and analytic evaluation, we not only present several interesting characteristics of the TCAM-conscious techniques but also show that they are at least 3 times faster than their software counterparts.

In Section 2 we describe the architecture of an NPU followed by a description of TCAMs. In Section 3, we present an overview of the frequent elements problem, summarize various proposed solutions and their implementation issues, and motivate the need for more efficient solutions. In Section 4, we present two TCAM-based techniques for frequent elements and present a comparison with the only known TCAM based solution for finding frequent elements, the RAP technique. Section 5 presents the experimental evaluation of the techniques on synthetic zipfian data.

2. ADVANCED NETWORKING ARCHITECTURE

In this section, we describe the architectures of a Network Processing Unit (NPU) and Ternary Content Addressable Memories (TCAMs) and discuss the integrated setup of NPU and TCAMs which we use to develop high speed data stream solutions. While the conventional processor architecture has not changed much over the past decade, networking hardware has witnessed significant advances both in terms of processor and memory architectures. Networking software which powers routers was initially developed to operate on conventional machines. As the traditional hardware was not built to be optimal for networking applications, custom application specific hardware was built which can operate at multi-gigabit speeds [1]. However such systems offer limited configurability and software programmability thereby

limiting the reuse of existing systems. In order to enable reusability, high-performance microprocessors called the Network Processing Unit (NPU) have been developed with a very flexible software programming capability.

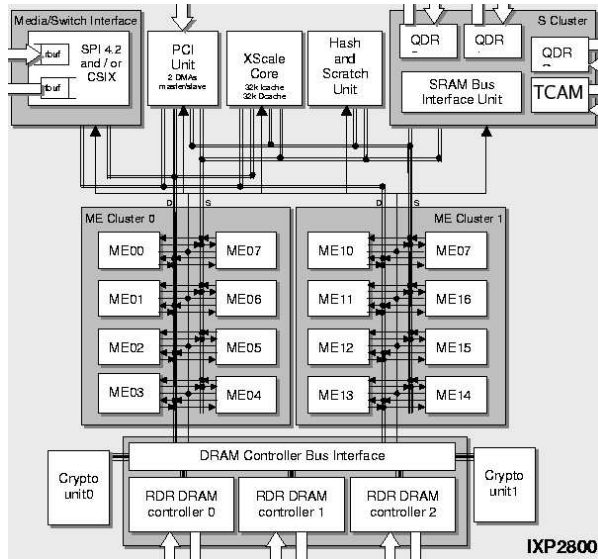


Figure 1: Intel's IXP2800 NPU

A typical network processor, such as the Intel IXP2800 [1], has one control plane processor (CPP) and 16 data plane processors on a single die (Figure 1). The CPP is a standard 32-bit low-power high-performance Xscale processor which runs an embedded operating system such as Monta Vista Linux. The data plane processor, also referred to as a micro-engine (ME), is a 32-bit low-power RISC processor with 8 thread contexts thus providing the IXP2800 NPU with 128 (16*8) parallel threads of raw computation power. One of the main differences between a conventional multiple processor architecture and the NPU is the method of communication between the MEs. Apart from the traditional method of shared bus and memory sharing, the MEs are organized in a chain-like fashion which is similar to the classical ring topology. Each ME can communicate with two of its neighbors through shared registers. This makes the NPU ideal for streaming applications as it allows an application to be componentised across multiple processors (MEs and their threads) in a pipe-lined fashion. Different components operate on different pieces of the data stream thereby increasing the processing throughput.

Both the CPP and all of the 16 MEs share a variety of resources: Dynamic RAM (DRAM), Static RAM (SRAM), hashing unit, cryptography unit and the TCAM unit. The requirement for different kinds of memory can be explained with the help of an example such as IPV4 forwarding. In any typical router, when a packet arrives, its destination information, stored in the packet header, is used to lookup in the forwarding table and is forwarded to the destination based on the lookup value. In this process, the router needs to operate only on the header information and hence this information is stored in a faster memory such as the SRAM. On the other hand, the less relevant payload information is stored inside the slower DRAM which offers better access speed for large chunks of data such as the payload. Although

SRAM and DRAM exist in conventional architectures in the form of L1/L2 cache and main memory respectively, they are more clearly delineated in terms of the control given to a program running on the CPP or ME to explicitly place the data according to the requirements.

Further, an IPV4 forwarding application takes advantage of a special kind of associative memory called the TCAM for providing constant time lookups. The routing table is stored inside the TCAM and when a packet arrives, the NPU looks-up the destination address in constant time inside the TCAM and forwards the packet accordingly. A TCAM stores the data in an array-like fashion and given a key word, it compares the key with the entire array parallelly in a SIMD-like fashion. If there is a match with the key word, the address of the top-most match is returned. In case there are multiple matches, a special multiple hit flag is set. Commodity TCAMs store hundreds of thousands of data entries and support searches of up to 100 million searches per second. They support search keys of width up to 572 bits. A TCAM is usually associated with an SRAM to store additional information regarding each TCAM entry. For example, in the IPV4 application, the next hop information of the routing table is typically stored in an associated SRAM whereas the destination information is stored in the TCAM.

TCAMs also support selective searching, where certain bits can be masked from being included in the search. This is enabled by two levels of masking: global and local. Global masking masks certain bits of all the data words to be masked out while the Local masking precludes certain bits of a particular data word from the search. For each data word, there is an associated mask word which enables local masking. The masking feature of the TCAM enables the representation of ranges and range matching inside the TCAM. For example, an IP address range from 128.111.1.0 to 128.111.1.255 can be represented as 128.111.1.* where the last 8 bits are masked out using the local mask bits. Given any address in this range, say 128.111.1.100, it matches this range as the last bits are not involved in the search. This constant time range-matching feature of the TCAM enables high speed packet classification [23, 17] at routers.

We observe that the integrated setup of NPU and TCAMs has been used predominantly for building routing applications. Recently there has been significant interest in exploiting NPUs and TCAMs for developing non-routing applications, e.g. giga-bit rate intrusion detection[12], giga-bit rate pattern-matching [26], linear time sorting [24] and relational database joins [13, 5]. However, this setup has not been exploited for developing efficient traffic summarization solutions, a popular data stream application. Using the state-of-the-art NPU platform, the IXDP2801, which feature an IXP2800 NPU and using TCAMs from Integrated Devices Technology, we built an integrated setup for developing our solutions. In the rest of this paper, we discuss how we use our setup towards developing and evaluating efficient summarization techniques on data streams.

3. FREQUENCY COUNTING

In this section, we describe the frequent elements problem over data streams and the proposed solutions. In particular, we discuss the implementation issues of these solutions and motivate the need for alternative solutions. The data stream model of computation involves answering continuous queries over an infinite stream of data elements. Some exam-

ples of data streams are the network traffic flowing through routers, click stream data flowing through the Internet Service Providers (ISPs), financial data from a stock exchange, etc. A popular query is the frequent elements query which lists all the elements that have a frequency of more than a certain threshold. For example, a network administrator may want to keep track of all the network flows passing through a router which independently contribute more than 1% of the entire traffic. Another closely related query is the *top-k* query which lists k most frequent elements in the stream. The frequent elements problem can be formally stated as follows: Given an alphabet, A, a frequent element, e_i , is an element whose frequency, f_i , in a stream S of a given size N, exceeds a user-specified support ϕN , where $0 \leq \phi \leq 1$; whereas the top-k elements are the k elements with highest frequencies.

Since an exact solution for this problem basically stores the entire stream, several approximate solutions have been proposed. The Hot Elements problem [8] is a special case of the frequent elements problem that lists up to k elements each with a frequency $f_i \geq N/(k+1)$. The most popular variation of the frequent elements problem, ϵ -Deficient Frequent Elements [18], asks for all the elements with frequency greater than ϵN . The FindCandidateTop(S, k, l) proposed in [7] lists l elements among which the top-k elements are concealed, with no guarantees on the rank of the remaining $l-k$ elements. The FindApproxTop(S, k, ϵ) [7] is a more practical approximation for the top-k problem which lists all k elements such that every element, e_i , in the list has $f_i \geq (1-\epsilon)f_k$, where ϵ is a user-defined error, and $f_1 \leq \dots \leq f_{|A|}$, such that e_k is the element with the kth rank. Several algorithms [18, 10, 21, 16, 7, 8] have been proposed to solve the frequent elements, top-k problems, and their variations. These techniques can be classified into counter-based, and sketch-based techniques.

3.1 Counter-Based techniques

Counter-based techniques keep an individual counter for each element in the monitored set, a subset of A. The counter of a monitored element, e_i , is updated when e_i , occurs in the stream. If there is no counter kept for the observed element, it is either disregarded, or some algorithm-dependent action is taken. For solving the ϵ -deficient frequent elements, algorithms Sticky Sampling, and Lossy Counting were proposed in [18]. Demaine *et al.* proposed the Frequent algorithm to solve the Hot Elements problem in [10]. Their algorithm, a re-discovery of the algorithm in [21] and independently proposed by Karp *et al.* [16], uses exactly k counters and outputs all elements with frequency more than $N/(k+1)$. However, it always outputs k elements and does not provide any guarantees with regard to non-hot elements in the output. Other techniques include the Space Saving solution [20] and the Adaptive Space Partitioning [15] which will be discussed in detail later.

3.2 Sketch-Based techniques

Unlike the Counter-based techniques, Sketch-based techniques do not monitor a subset of elements. They provide, with less stringent guarantees, frequency estimation for all elements using bitmaps of counters. Each element is hashed into the space of counters using a family of hash functions, and the hashed-to counters are updated for every hit of this element. Those representative counters are then queried for

the element frequency with less accuracy, due to hashing collisions. Some of the popular sketch-based solutions are the CountSketch [7], the GroupTest [8], Multistage filters [11].

3.3 Implementation issues

Sketch-based techniques are not only known to have a large space complexity but also do not offer any information about elements' frequencies [20]. Although Multistage filter [11] provides the worst case guarantees, it has unbounded space complexity which is undesirable. Furthermore, they monitor all elements in the data stream where a hit entails expensive calculations thus making the sketch-based solutions not ideal for faster implementations. Therefore, we consider only counter-based solutions in this paper. Although all the counter-based solutions for finding the frequent elements were designed to be both space and compute efficient, they incur a logarithmic or amortized constant number of operations (in terms of stream size) per stream element [20]. The data structures used in these techniques are typically implemented as trees, priority queues, heaps, hash tables and operations over these data structures contribute for the above mentioned cost. These data structures are used to efficiently store the data stream summary and facilitate lookups for incoming stream elements in the current summary. On the other hand, we observe that TCAMS provide constant time lookups over large amount of data. These TCAMs are readily available on an NPU where several applications of the frequent elements problem are typically used. In the rest of this paper, we explore whether this constant time lookup feature of a TCAM can be exploited for faster implementations.

4. TCAM-CONSCIOUS ALGORITHMS

In this section, we discuss some popular solutions for the frequent elements problem and present their TCAM-conscious implementations. In particular, we discuss the Lossy Counting [18] and Space Saving [20] techniques and discuss various issues which arise in their adaptation to the TCAM model. The Space Saving technique [20] is superior to the technique proposed by Demaine [10] in terms of both space and time requirements. Hence, we do not discuss the latter technique separately. We note that TCAMs have been recently proposed by computer architects [22, 15] to be used for dynamically profiling hot ranges of program code at execution time. This solution, which has only been proposed as a model and not implemented on hardware, can be used for finding frequent elements. In this section, we analyze the TCAM-conscious techniques and show that our techniques are simpler to implement and efficient in terms of space usage.

4.1 Lossy Counting using TCAMs

The Lossy Counting technique [18] divides the stream into rounds. The algorithm maintains a data structure G which contains entries of the form $\langle e, f, \Delta \rangle$ where e is an element in the stream, f is its frequency and Δ is the error in estimation of the frequency of the element. Given a user-defined error bound ϵ , all the elements present in G satisfy the condition $f \leq f_e \leq f + \Delta$ where f_e is the exact frequency of the element e in the data stream. The algorithm maintains this condition by dividing the data stream into rounds of size $1/\epsilon$. Each round is associated with a round number r which is set to $\lceil N/\epsilon \rceil$ where N is the total length of the data stream seen

Algorithm 1 TCAM-conscious Lossy Counting

```

/* This code assumes all components  $\langle e, f, \Delta \rangle$  to be stored
in a TCAM entry and does not use SRAM */
mask1 register masks the element component
for each element  $e$  in stream do
    hit = LOOKUP ( $e, \text{mask}_1, \text{addr}$ )
    /* if hit, the variable  $\text{addr}$  contains the matching address */
    if hit then
        READ( $\text{addr}, f$ ); /* read the frequency */
        WRITE( $\text{addr}, f+1$ ); /* update the frequency */
    else
        address = LEARN
        WRITE ( $\text{addr}, \langle e, 1, \text{round} \rangle$ )
    end if
    roundcount++;
if (roundcount = roundsize) then
    round = round + 1
    for each element  $e_i$  in  $G$  do
        READ( $i, f, \Delta_i$ );
        if  $f+\Delta \leq \text{round}$  then
            DISABLE ( $i$ )
        end if
    end for
    roundcount = 0;
end if
end for

```

so far. In each round, each element is checked for occurrence in G . If it is present in G , its frequency f is incremented appropriately. Otherwise, a new entry $\langle e, 1, r \rangle$ is added to G . At the end of each round, all the entries $\langle e, f, \Delta \rangle$ with $(f+\Delta) \leq r$ are removed from G . Intuitively, this step, which we refer to as the *zeroing step* removes all the elements with lower frequencies. For example, at the end of the first round, it removes all the elements with frequency 1. At the end of round r , this technique ensures that any element not present in G has an exact frequency $f_e \leq r$. This algorithm has a proven upper bound on space usage of $O(1/\epsilon \log(N * \epsilon))$ [18].

For each element in the stream, this algorithm incurs a cost of looking up the element in the data structure G in addition to the possible cost of zeroing the element at the end of each round. Naively, searching for each element in G incurs $|G|$ operations per element. In [18], all the elements in each round are sorted and then inserted into G . This leads to a cost of $O(1/\epsilon \log(1/\epsilon))$ per round which amounts to $O(\log(1/\epsilon))$ per stream element. Further, this algorithm unnecessarily allocates space for insignificant elements which is released at the end of each round during the zeroing phase.

4.1.1 TCAM-Conscious Lossy Counting

The Lossy Counting technique is very well suited to be adapted to the TCAM model (pseudo-code shown in Algorithm 1). Given an entry $\langle e, f, \Delta \rangle$ in the data structure G , the component e can be stored in the TCAM and the other two components $\langle f, \Delta \rangle$ can be either stored in an associated SRAM (integrated with the TCAM) or in a standard SRAM of the NPU or along with e component inside the TCAM itself. When a new stream element e' arrives, it is matched in constant time against all the e 's stored inside the TCAM by masking out the remaining components and selectively including only the e component in the search. If there is

a match, the counter’s frequency is incremented appropriately. Otherwise, a TCAM entry corresponding to the new counter is added to the table by using the TCAM WRITE command. Initially, we can assume that the new counter is allocated a TCAM entry towards the end of the table stored in the TCAM.

However, there are several issues which need to be addressed in this adaptation. One such issue is the garbage collection of TCAM space. At the end of each round, several counters will be zeroed which means that the TCAM space occupied by these entries should no longer be valid and should be available for future allocation. This can be done by disabling the VALID bit associated with each TCAM entry. This operation precludes the entry from getting involved in any search command issued from then on. However disabling TCAM entries leaves G , stored in the TCAM, as an array of valid counters with holes in between. These holes correspond to the disabled entries at the end of the prior round. In the current round, if an element, which is not present in G arrives, it would be inefficient to add a new entry towards the end of table inside the TCAM. This is because, the table size will keep increasing indefinitely as holes created in the previous rounds are not used. This can be addressed by the LEARN feature provided by the TCAM. The LEARN command is a hardware accelerated mechanism which returns the first free entry inside the TCAM in constant time. Whenever a new counter needs to be allocated, a LEARN followed by a WRITE command allocates the entry for the new element.

It can be seen that Lossy Counting incurs one TCAM LOOKUP per every stream element. If the entry is already present and assuming all the components of an entry are stored in the TCAM, it incurs a TCAM READ and WRITE for updating the frequency. Otherwise, it incurs a TCAM LEARN followed by a TCAM WRITE per new element. Additionally, this technique incurs a potential DISABLING (zeroing) cost for each element in G which also requires another TCAM READ for retrieving the frequency information. Intuitively, if an element is infrequent, it incurs the cost of DISABLING the entry during the zeroing phase. On the other hand, if the element is very frequent then it does not incur the DISABLING cost. Further, since for any frequent element there is only one counter for all the element’s occurrences, we can also assume the TCAM READ cost in the zeroing phase to be zero by amortization. Assuming that all the TCAM operations have similar cost, any element incurs a total cost of 3 to 5 TCAM operations.

4.2 Space Saving using TCAMs

We now discuss the Space Saving technique [20] which provides stricter space bounds independent of the stream size. The data structure used in this technique is similar to the one used in Lossy counting. Each entry has a data stream element, its frequency and the associated error. The algorithm monitors only a fixed number of elements m , which is a function of the error bound and the size of the alphabet. If an element that is monitored arrives, its counter is incremented appropriately. If the new element e_{new} is not monitored, it is given the benefit of doubt and the element e_{min} that currently has the least hits, min , is assigned to e_{new} . The frequency $count_{new}$ is set to the value $min + 1$. For each element e_i , its maximum over-estimation, ϵ_i , resulting from the initialization of its counter when it was inserted

into the list is maintained. That is, when starting to monitor e_i , set ϵ_i to the counter value that was evicted. The algorithm assures that if an element is frequent, then its frequency count will be high and thus it will not be replaced. Only the least frequent elements are likely to be replaced by the incoming element.

In [20] the authors showed that by using only $O(1/\epsilon)$ counters, the algorithm lists all the ϵ -deficient frequent elements. More precisely, if the size of the alphabet is $|A|$, the maximum number of counters required is $min(|A|, 1/\epsilon)$. However, this technique also incurs $|G|$ operations to search for a stream element in G . Using hashing, this cost could be reduced to constant time [20]. In order to maintain the information regarding the minimum element, this technique requires G to be sorted on the frequency of the counters. When the current minimum element occurs in the stream again, maintaining G in a sorted order enables calculating the new minimum in a constant number of operations. G is organized as a doubly linked list with the head of the list pointing to the most frequent element and the tail pointing to the least frequent element. When a hit occurs, the appropriate counter is located and incremented. This requires the sorted list to be updated every time a new element arrives, thereby incurring expensive pointer re-arrangement cost for each element.

Algorithm 2 TCAM-conscious Space Saving

```

/* This code assumes all components  $\langle e, f, \Delta \rangle$  to be stored
in a TCAM entry and does not use SRAM */
mask1 register masks the element component
mask2 register masks the frequency component
for each element  $e$  in stream do
  if  $e = e_{min}$  then
    multihit = SEARCH( $f_{min}$ , mask1, addr)
    /* if hit, the variable addr contains the top matching
address */
    if (multihit) /* there is another min freq element */
    then
      WRITE(indexmin,  $e_{min}$ ,  $f_{min} + 1$ , errmin)
      hit = SEARCH( $f_{min}$ , mask1, addr)
      indexmin = addr
      ( $e_{min}$ ,  $f_{min}$ ) = READ(indexmin)
    else
      WRITE(indexmin,  $e_{min}$ ,  $f_{min} + 1$ , errmin)
    end if
  else
    hit = SEARCH( $f_{min}$ , mask2, addr)
    if (hit) then
      ( $e_{addr}$ ,  $f_{addr}$ , erraddr) = READ(addr)
      WRITE(addr,  $e_{addr}$ ,  $f_{addr} + 1$ , erraddr)
    else
      WRITE(tablesize,  $e_{tablesize}$ , 1, 0)
      tablesize = tablesize + 1
    end if
  end if
end for

```

4.2.1 TCAM-Conscious Space Saving

Unlike the Lossy Counting which has a fairly straight forward mapping, Space Saving does not have an intuitive mapping. Space Saving requires G to be sorted on the frequency of the counters in order to efficiently retrieve the minimum

element. We now show that maintaining the information regarding the minimum element can be efficiently done using TCAMs.

As shown in Algorithm 2, $\langle e, f \rangle$ of each counter are stored inside the TCAM. At any stage in the algorithm’s execution, we ensure that the current minimum frequency element (e_{min}) and its frequency (f_{min}) are known. When a new element arrives, the element is searched in the current summary by masking out the bits corresponding to the frequency information. There are several possibilities for the new element. If the element is not present, then e_{min} is replaced. If the element is already present in G and it is not e_{min} , then e_{min} retains its status. However, if the new element is e_{min} , then some other element in G will become the new minimum only if its frequency is same as f_{min} . This condition can be checked for by searching the TCAM with f_{min} as the lookup key and masking out the element component of the counter information. If there is another element with the same frequency, then the search results in the multi-hit flag being set. In this case, we update the frequency of the current minimum element. We can locate the new minimum element by again searching with a key value set to the old minimum frequency but with the element component masked out. Since we already updated the frequency of the old e_{min} , it will not be returned as the search result thereby returning only the new minimum frequency element. Intuitively, the idea of selective masking can be adapted to provide an efficient software implementation for Space Saving. By using separate hash tables for elements and frequency, Space Saving can avoid the expensive pointer rearrangement for each element.

For each incoming element, if the element is not the current minimum element, the Space Saving algorithm incurs one LOOKUP for checking if the element is in the summary. If it is not present, it incurs one TCAM write for replacing the minimum element. If it is present, it incurs one TCAM READ and WRITE for updating the frequency. If it is the current minimum element, then it still incurs one LOOKUP cost for searching for same frequency elements. If there are none, it incurs one TCAM write for updating the current minimum element. If there is another minimum element, it still incurs only one TCAM WRITE for updating the older minimum frequency element. On the whole, the Space Saving technique incurs one TCAM LOOKUP, one TCAM READ and one TCAM WRITE per stream element.

4.3 Adaptive Space Partitioning

In [15, 22], the authors propose a solution for finding frequent ranges, a more general form of the frequent elements problem. This technique indirectly solves the frequent elements problem by assuring that it maintains a separate counter for each stream element with a frequency greater than $\epsilon * N$. In this technique, instead of maintaining the frequency information for individual stream elements, counters are used to represent the frequency of ranges in the data space of the stream elements. That is, the value associated with each counter corresponds to the number of elements belonging to this range which occurred in the data stream. For each stream element, the range in which it falls into is found and the frequency is appropriately incremented. Whenever a counter’s value exceeds a certain threshold, the range is split into two halves and two new counters are created to count the frequency of the two newly created ranges. The

frequency count associated with the old counter is retained. This partitioning of the ranges goes on until no further partitioning is possible. At this point, the range represents a unique stream element. Given the data space $[0, R]$, the partitioning of a given binary range stops in $\log(R)$ steps. It can be intuitively seen that this partitioning forms a tree structure with the root representing $[0, R]$ and the maximum depth of a node in the tree is $O(\log(R))$. This tree is referred to as the Adaptive Space Partitioning (ASP) tree. The error in estimating the frequency of any range is basically the sum of errors in estimation of all its parent nodes. Given a user-defined error ϵ , by setting the split threshold to be $O(\log(R)/\epsilon)$, the error in estimating the frequency of any node (range) in the tree is limited to $\epsilon * N$. Henceforth, given any element with a frequency greater than $\epsilon * N$, it is bound to have a leaf node. As the threshold increases with N , a previously split range may no longer need to be maintained as split ranges. Based on this observation, the authors propose to merge nodes either periodically or using priority queues over the counter values.

This algorithm is shown to have a space bound of $O(1/\epsilon)$. For each stream element, it requires looking up a matching range which requires $O(|\log(R)|)$ operations to traverse the tree. In addition to this cost, the algorithm also requires merging of nodes to keep the memory usage within the bound. This involves storing the counter values of the tree nodes in a priority queue. Since the number of nodes is $1/\epsilon$, the priority queue operations require $O(\log(1/\epsilon))$ cost per element.

4.3.1 TCAM-Conscious ASP

In [22], the authors propose to use the Adaptive Space Partitioning technique for profiling program execution and list the blocks of program code which execute the most. This technique called Range Adaptive Profiling (RAP) is a simple adaptation of the ASP technique to the TCAM model. For each node in the ASP tree, which corresponds to a range, a TCAM entry corresponds to the range is added. For example, if the data space is 8 bits, then the range $[0, 63]$ can be represented by a prefix string 00^{*****} . Given any element in this range, it matches with this range in constant time thus reducing the searching cost for an element $O(1/\epsilon)$ to $O(1)$. When a new stream element arrives, the range which it falls into is found in constant time and updated accordingly. For each stream element, it takes a TCAM LOOKUP followed by a TCAM READ and TCAM WRITE cost for reading and updating the frequencies. Further, a node in the ASP tree represents a range that is a super set of all its children. If the TCAM entry corresponding to the parent node precedes the child node in the ASP tree, then a TCAM lookup returns the parent node whereas the ASP technique requires it to be the child node. In order to address this problem, larger prefixes (small ranges) are placed before smaller prefixes (large ranges). For example, if the data space is 8 bits, all the 1-bit prefixes such as 0^* (representing 0-127) are stored before the 2-bit prefixes such as 00^* (representing 0-63). This requires the TCAM space to be partitioned with empty spaces left in between to accommodate new nodes. If the data space is represented by b bits, the TCAM space is divided into b partitions to store different partitions. When a new node is added, it is added to the partition depending on the length of prefix. Each partition needs to have additional space for accommodating new

entries. This leads to wastage of TCAM space as it leaves holes between the partitions.

4.4 Cost Analysis

We now present a discussion on how TCAM-conscious Lossy Counting and Space Saving compare with RAP, the TCAM-conscious ASP technique. As summarized in Table 1, the TCAM-conscious Lossy Counting requires 3 to 5 TCAM operations per element. Although this technique reuses the holes created, it still suffers from fragmentation. This is because, holes which are created by zeroing and not re-allocated, can possibly be located before valid elements in the TCAM towards the end of the table. This implies that the size of the TCAM table can be more than the number of entries in the actual data structure. The TCAM-conscious Space Saving is optimal in terms of space utilization and takes exactly 3 TCAM operations for each stream element.

TCAM-conscious Technique	Operations per element	Space complexity
Lossy Counting	3 to 5	$O(1/\epsilon \log(1/\epsilon))$
Space Saving	3	$\min(A , 1/\epsilon)$
ASP (RAP)	3 + amortized merge + block movement cost	$O(1/\epsilon) + b * (\textit{extraspacespace})$

Table 1: Summary of TCAM-conscious techniques

RAP also requires 3 TCAM operations per element. However, it requires the partitioning of the TCAM space and extra space has to be allocated in each partition to accommodate new nodes created at run-time. This partitioning results in the fragmentation of the TCAM space resulting in wastage of valuable TCAM space. Furthermore, when the space allocated to partition is insufficient as the number of nodes in the partition overflows, it incurs the cost of moving the data inside TCAM to resize the partitions. If resizing is not used, then enough space has to be pre-allocated to accommodate for the maximum number of nodes possible in each partition. In this case, the fragmentation problem worsens with the number of bits used for representing stream elements. If the data space requires b bits, then there are 2^b possible b-bit prefixes. So for each partition, the amount of space which needs to be allocated is $\min(1/\epsilon, 2^b)$. Furthermore, ASP needs periodical merging of nodes to keep the space consumption within the bounds resulting in an undesirable stall in the stream pipeline. This merging involves traversing the TCAM using the logical tree looking for the frequency of a node and its children which implies using pointers thus making the periodic traversal an expensive phase. Hence, the RAP solution is not so ideal for the frequent elements problem.

5. EXPERIMENTAL EVALUATION

In this section, we present the experimental evaluation of the two TCAM-conscious techniques presented in the previous section. This evaluation not only demonstrates the implementation feasibility of our proposed techniques but also shows their performance advantage over their software implementations. Regarding the TCAM-conscious ASP, we note that the technique is quite complex in data structure maintenance, and even in the original paper [22] was never implemented. Henceforth, instead of implementing it, we demonstrate in this section, using our cost analysis (Sec-

tion 4.4) in conjunction with experimental results of TCAM-conscious Lossy Counting and Space Saving, that the TCAM-conscious ASP technique could not be superior to the other techniques. We evaluated the techniques on IXDP2801, Intel’s platform for developing networking applications. This platform features IXP2800 [1], Intel’s latest NPU which has 16 micro-engines and 1 control plane processor. Each of these 16 micro-engines has 8 thread contexts thus providing 128 parallel threads of execution. This system has 32MB of SRAM and 768MB of DRAM. It contains one TCAM that contains 128K entries of 72-bits each. The TCAM interfaces with the processors through the SRAM bus. For this purpose, the TCAM comes with a controller which interfaces with the TCAM on one side through a 72-bit LA-1 interface and the processors on the other side through an SRAM interface. Currently, the controller is designed so that each of the 128 threads from the microengines can access the TCAM independently.

The Content Plane Processor (CPP) runs Monta Vista Linux, an embedded operating system and the micro engines run Teja Network Operating System [2]. Programming is done using Teja C, a version of Intel C enhanced for NPU. Both the TCAM-conscious techniques are developed on MEs. For comparison purposes, we implemented efficient software implementations for Lossy Counting and Space Saving as described in [18, 20]. These solutions are developed on a 2.8GHz Pentium 4 machine with 1GB of system memory. We also present a brief comparison of the performance of the TCAM-conscious techniques with the reported throughput results of Gigascope [9], a state-of-the-art stream processing system meant for networking environments.

5.1 Datasets and TCAM cost analysis

Evaluation of the two TCAM based techniques and their software counterparts is done with respect to the following metrics: query processing time, space utilization and accuracy. For experimentation, we use synthetic zipfian data with zipfian factors ranging from 0 to 3. We chose to use zipfian data as most real life datasets since zipfian distribution is visible in most real life datasets [20] and varying zipfian factors help analyze the performance characteristics of the TCAM-conscious techniques with respect to the data skew. For all the experiments, we used a stream of a size 1,000,000 and an alphabet A of size of 100000. For all the experiments, we set the threshold to 0.01 and the error factor ϵ to 0.001.

First, we present an analysis of various TCAM operations: READ, WRITE, SEARCH, LEARN, DISABLE. Table 2 shows the processing cost for a million consecutive executions of each of these from a single micro-engine thread. It can be seen that we were able to achieve a throughput of 2.5 million searches per second. However, the TCAM can provide a much higher throughput of 100 million searches per second. The lower performance in the current prototype is due to the controller which interfaces with the TCAM and the micro-engines. Currently, for each TCAM command, the micro-engine thread has to write the necessary command into the controller’s memory which in turn interfaces with the TCAM and executes the command on the micro-engine’s behalf. This results in significant overhead in terms of the command execution time and hence the lower throughput. In this paper, we show the results of our TCAM-conscious techniques with this overhead included. However, it should

be noted that if a TCAM interfaces directly with an ASIC device over its native LA-1 interface instead of an NPU through a controller and the SRAM bus, each of the TCAM operations can be speeded up by 40 times. Implementation of TCAM-based algorithms on an ASIC device is a standard practice [22, 17, 26] for providing high speed networking solutions, as these devices can fully utilize the capacity of TCAMs.

TCAM Operation	Time (seconds) for million operations
READ	0.366
WRITE	0.34
SEARCH	0.34
LEARN	0.34
DISABLE	0.308

Table 2: Statistics of TCAM operations

5.2 Processing cost

Figure 2 presents the performance of TCAM-conscious Lossy Counting, TCAM-conscious Space Saving and their software counterparts with zipfian factors varying from 0 to 3. We refer to the TCAM-conscious Lossy Counting, TCAM-conscious Space Saving, software Lossy Counting and software Space Saving as TLC, TSS, SLC and SSS respectively. These results show that the TCAM versions of both techniques perform at least 3 times faster than their software counterparts. This is despite the fact that the software techniques are evaluated on a faster and architecturally superior Pentium 4 processor. If the software solutions are developed on a micro-engine where the TCAM-conscious techniques are developed, they will take more time than a Pentium 4 implementation.

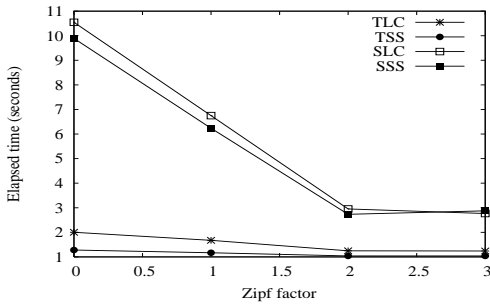


Figure 2: Processing time

It can be seen that while TSS's performance does not change much with zipfian factor, TLC shows significant difference in processing cost. This can be explained using the analysis of TLC shown in Section 4.1.1. We observed that when an element is infrequent, it incurs 5 TCAM operations as opposed to 3 operations for the frequent elements. At lower zipfian factors, the data becomes more uniform which means the frequency of each element of the alphabet is small compared to the size of dataset. Hence, it incurs 5 operations per element. At higher zipfian factors, it incurs only 3 operations per element. Using the costs of TCAM operations shown in Table 2, we estimate the lower and upper bound TCAM operation costs of Lossy Counting in Table 3. This table supports our above argument as the estimated costs at higher zipfian factors is similar to the lower bound

cost while the lower zipfian cost is closer to the upper bound estimate. It also suggests that the TCAM operations contribute for most of the overall cost. Estimated cost shown in Table 3 and the actual processing time of Space Saving also shows that estimated and actual costs are very similar. This analysis can be used to estimate the minimum processing time for the TCAM-conscious ASP technique. Since, this technique also incurs a minimum of 3 TCAM operations, it will also take at least as much time as the Space Saving. This is in addition to the periodic merging and space adjustment cost which implies that it cannot be more efficient in terms of processing time when compared to Space Saving.

At this point, it is worth making a comparison with the results reported by other stream processing systems. In Gigascope [9], a stream processing system specific to network data, the authors report a throughput of around 1 million elements per second. This compares with the above reported performance of the TCAM-conscious solutions. However, the TCAM-conscious techniques are evaluated on an NPU setting which exploits only 2.5% of the TCAM's processing capacity. If the same techniques were implemented on a standard ASIC device interfacing directly with the TCAM [22, 17, 26], the performance can be potentially improved by 40 times. This argument is supported by the fact that, the performance of TCAM-conscious techniques is a direct multiple of the number of TCAM operations per element. In an ASIC device implementation of these techniques, where the speed of each component TCAM operation can be improved by 40 times, overall performance improvement of 120 times over software counterparts is not unrealizable.

Technique	Lower Bound	Upper Bound
TLC	1.02	1.7
TSS	1.02	1.02

Table 3: Estimated cost of TLC and TSS

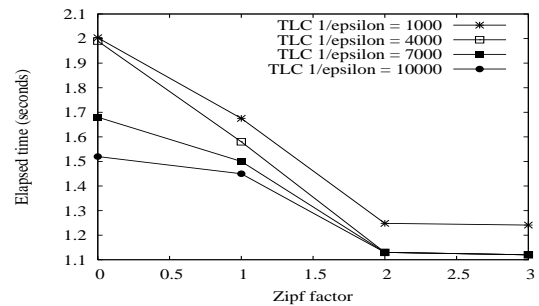


Figure 3: TLC Processing cost with varying ϵ

We now analyze the performance of these techniques with respect to *varying* ϵ , the error factor. While keeping the same stream size, we vary the ϵ from 0.001 to 0.0001 and present the results for TCAM-conscious Lossy Counting in Figure 3. We do not show the results for TCAM-conscious Space Saving, as its cost practically remains the same at all the error bounds. This behavior is expected as it incurs 3 TCAM operations per stream element irrespective of the error bound. It can be seen that the performance of the TLC improves with decreasing ϵ . These results present a counter-intuitive behavior of TLC as opposed to its software imple-

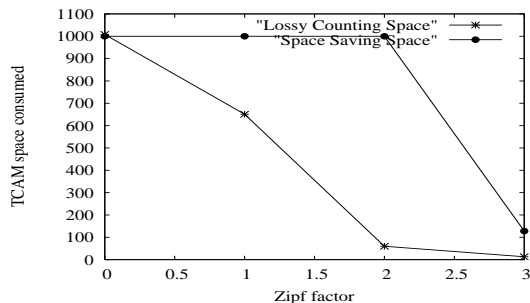


Figure 4: TCAM space consumed

mentation [18]. Intuitively, at stricter error bounds (lower ϵ), the round size ($1/\epsilon$) becomes large and it takes more time for looking up an incoming element. Hence, it takes more time and space to provide lower error bounds. However, TCAM takes the same amount of time for looking up a given element whether the round size is 1000 or 10000. Hence, it should take the same amount of processing time. However, the results also show that the time consumed actually decreases with lower error bounds. This can be attributed to the lesser number of zeroing round operations as we explain below. At lower ϵ , the round sizes become large which implies lower number of rounds for a given stream size. For a given element with multiple occurrences, the possible number of times its corresponding counter will be zeroed is directly related to the number of rounds. Hence when the number of rounds is fewer, the amortized cost reduces thus resulting in lower cost. Hence, TLC exhibits an interesting behavior of lower processing cost at stricter error bounds which is a very useful property.

5.3 Space cost

Figure 4 shows the results for the amount of TCAM space consumed by both the TCAM-conscious techniques for an error factor of 0.001. For space cost, we only discuss the TCAM-conscious techniques as their software counter-parts incur similar cost in terms of memory space usage. This analysis is important as these techniques have not been experimentally compared in terms of space usage. We refer to TLC and TSS simply as Lossy Counting and Space Saving. These results show a contradictory behavior to the theoretical bounds described in [18] and [20]. As described in the Section 3, given a stream of size N and an error bound ϵ , Lossy Counting has a space bound of $O(1/\epsilon * \log(N * \epsilon))$ and Space Saving has a strict upper bound of $\min(|A|, 1/\epsilon)$ respectively. So given an error bound of 0.001 and stream size of 1000000, these techniques have projected sizes of $O(3000)$ and 1000 entries respectively. However the results show that Lossy Counting exhibits better space usage compared to Space Saving at almost all zipfian factors. Since the alphabet size is 100000, at zipfian factors less than 3, there are more than 1000 unique elements in the stream. This explains the space consumption of 1000 entries by the Space Saving as the space is bounded by $1/\epsilon (=1000)$. At the higher zipfian factor of 3, there are only 130 unique elements which shows up as the space usage of Space Saving. However in the case of Lossy Counting, any element with a frequency less than $\epsilon * N$ has a good chance of not being included in the summary. This results in only the most fre-

quent elements to be involved in the summary. This shows that Lossy Counting exhibits very good space usage when the data has a high zipfian factor.

5.4 Accuracy

We now present an evaluation of the accuracy of the results returned by Lossy Counting and Space Saving. Accuracy is the percentage of the correct results out of the output results for a given technique. Here, we the threshold is set to 0.01 and the error bound to 0.001 thus retrieving all elements with frequency greater than 9000 (10000 - 1000). Results shown in Figure 5 show that both Lossy Counting and Space Saving have 100% accuracy for all the zipfian factors. This shows that the Lossy Counting, while consuming lesser TCAM space, can still accurately return the correct results thus showing that it has desirable performance and space characteristics in most cases. However, Space Saving may be desirable if the data distribution has not skewed and elements are less frequent.

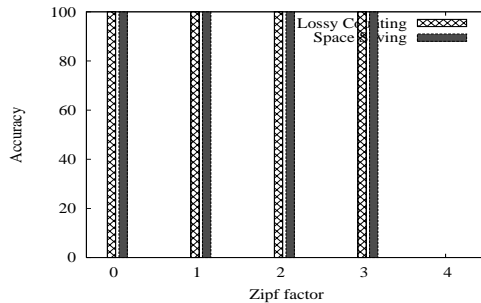


Figure 5: Accuracy of Results

In [20], the authors propose that if the zipfian factor α of the stream data is known *a priori*, the space complexity of Space Saving can be reduced to $(1/\epsilon)^{1/\alpha}$ while still providing the same error bound. Using this observation, we evaluated the Space Saving with zipfian factors of 1, 2 and 3 using TCAM space of 1000, $(1000)^{(1/2)}$, $(1000)^{(1/3)}$ respectively. Results in Table 4 confirm that this accuracy remains at 100% for all the above space sizes.

Zipf factor	TCAM Space	Accuracy
1	1000	100
2	31	100
3	10	100

Table 4: Accuracy of Zipf-conscious Space Saving

6. CONCLUSION

In this paper, we explored the frequent elements problem in data streams from a networking systems perspective. We proposed to exploit TCAMs, a special kind of memories found in NPUs, for providing efficient implementations for the frequent elements problem. We proposed two solutions, the TCAM-conscious Lossy Counting and TCAM-conscious Space Saving, and presented a theoretical cost analysis of each of these techniques. We implemented both these techniques on an Intel's IXDP2801 NPU platform and evaluated them over synthetic zipfian data. Experimental results show that the TCAM-conscious techniques are at least 3

times faster than their software counter-parts. By using a standard networking ASIC implementation that can increase the TCAM usage to 100%, these results project a potential for orders of magnitude speedup over the software implementations. Experimental evaluation not only supports the theoretical analysis but also demonstrates several interesting characteristics of the TCAM-conscious solutions. We show that the performance of Lossy Counting improves with stricter error bounds, a contradictory characteristic from the software implementation. We also show that although TCAM-conscious Lossy counting takes slightly more processing time than Space Saving, it exhibits good space usage characteristics while providing good accuracy thus making it an ideal solution in most cases. In this paper, we exploited the basic matching capability of the TCAMs towards providing faster heavy hitter implementations. We have also been exploring if the range matching and sorting features of a TCAM can help provide faster solutions for other data stream problems in networking environments. In particular, we provide fast TCAM-conscious solutions for detecting heavy distinct hitters [4], which comprise another important class of the data stream summarization problems.

7. REFERENCES

- [1] Intel network processing units. <http://www.intel.com/>, 2006.
- [2] Teja networking systems. <http://www.teja.com/>, 2006.
- [3] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.
- [4] N. Bandi, D. Agrawal, and A. E. Abbadi. Fast algorithms for heavy distinct hitters using associative memories. In *International Conference on Distributed Computing Systems (ICDCS)*, 2007.
- [5] N. Bandi, S. Schnieder, D. Agrawal, and A. E. Abbadi. Hardware acceleration of database operations using content-addressable memories. In *DaMoN*, 2005.
- [6] S. Chandrasekaran. Telegraphcq: Continuous dataflow processing for an uncertain world, 2003.
- [7] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *ICALP'02*, pages 693–703, 2002.
- [8] G. Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. In *PODS '03*, pages 296–306, 2003.
- [9] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 647–651, New York, NY, USA, 2003. ACM Press.
- [10] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'02)*, volume 2461, pages 348–360, 2002.
- [11] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.*, 21(3):270–313, 2003.
- [12] R. H. K. Fang Yu. Efficient Multi-Match Packet Classification with TCAM. In *IEEE Hot Interconnects*, 2004.
- [13] B. T. Gold, A. Ailamaki, L. Huston, and B. Falsafi. Accelerating database operations using a network processor. In *DaMoN*, 2005.
- [14] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD Conference*, 2001.
- [15] J. Hershberger, N. Shrivastava, S. Suri, and C. D. Tóth. Adaptive spatial partitioning for multidimensional data streams. In *ISAAC*, pages 522–533, 2004.
- [16] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28(1):51–55, 2003.
- [17] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for advanced packet classification with ternary cams. In *SIGCOMM '05*, pages 193–204, New York, NY, USA, 2005. ACM Press.
- [18] G. S. Manku and R. Motwani. Approximate frequency counts over data streams, 2002.
- [19] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *SIGMOD'98*, pages 426–435, 1998.
- [20] A. Metwally, D. Agrawal, and A. E. Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, pages 398–412, 2005.
- [21] J. Misra and D. Gries. Finding repeated elements. *Sci. Comput. Program.*, 2(2):143–152, 1982.
- [22] S. Mysore, B. Agrawal, T. Sherwood, N. Shrivastava, and S. Suri. Profiling over adaptive ranges. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 147–158, 2006.
- [23] G. J. Narlikar, A. Basu, and F. Zane. Coolcams: Power-efficient tcams for forwarding engines. In *INFOCOM*, 2003.
- [24] R. Panigrahy and S. Sharma. Sorting and Searching using Ternary CAMs. In *IEEE Micro.*, 2003.
- [25] J. Widom and R. Motwani. Query processing, resource management, and approximation in a data stream management system, 2003.
- [26] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using tcam. In *ICNP '04*, pages 174–183, 2004.