# Fast Algorithms for Heavy Distinct Hitters using Associative Memories*

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

## Abstract

*Real-time detection of worm attacks, port scans and Distributed Denial of Service (DDoS) attacks, as network packets belonging to these security attacks flow through a network router, is of paramount importance. In a typical worm attack, a worm infected host tries to spread the worm by scanning a number of other hosts thus resulting in significant number of network connections at an intermediate router. Detecting such attacks amounts to finding all hosts that are associated with unusually high number of other hosts, which is equivalent to solving the classic heavy distinct hitter problem over data streams. While several heavy distinct hitter solutions have been proposed and evaluated in a standard CPU setting, most of the above applications typically execute on special networking architectures called Network Processing Units (NPUs). These NPUs interface with special associative memories known as the 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 to develop high-speed solutions for heavy distinct hitters.*

## 1 Introduction

During the last decade, researchers have developed diverse solutions for answering summarization queries over data streams. Given a continuous stream of elements, data steam techniques are essentially one-pass algorithms which answer a variety of summarization queries such as finding the most frequent elements [11, 12], quantiles [7] and super-spreaders [17]. The frequent element problem involves finding the data elements whose frequency of occurrence in the data stream is beyond a certain threshold. Quantile summarization provides other statistics over the data stream such as the median and histograms. The super-spreader problem, also referred to as the heavy distinct hitter, operates on a stream of pairs and outputs all the elements which are uniquely paired with more than a certain number of other elements. These problems have a wide range of applications such as click fraud detection, network-traffic summarization, on-line analysis of stock market data, detecting port scans etc.

In this paper, we focus on the heavy distinct hitter problem. Given a stream of pairs *(x, y)*, the heavy distinct hitter problem involves finding all the *x*'s which are paired with a large number of *distinct y*'s. This problem is motivated by the network security monitoring applications of detecting worms and Distributed Denial of Service attacks (DDoS). When a network host is infected by a worm, it tries to spread this worm by infecting other hosts. For this purpose, the compromised host does a fast scan for other machines thereby making an unusually large number of network connections. For example, a host infected by the Slammer worm [13] sent around 26,000 scans per second and hence these infected hosts are referred to as super-spreaders [17]. At a network router which sees network connections as (source, destination) pairs, detecting source hosts which contact significant number of destination hosts is referred to as the super-spreader problem which is essentially the problem of finding heavy distinct hitter problem in a stream of source-destination pairs. The converse of this problem is to find all the destination hosts ($y$'s) which are uniquely paired with more than a certain number of source hosts ($x$'s). This version of problem finds application in the Distributed Denial of Service attack, where a destination host is attacked by a number of source hosts. This problem can be extended to a the problem of finding heavy distinct hitters over a sliding window. Unlike the unrestricted version of the problem which finds heavy distinct hitters over the entire processed stream, the sliding window version of the problem returns all the heavy distinct hitters with-in the window of last *n* pairs.

Estan et al. [8] have proposed a series of bitmap based techniques for finding DDoS and worm attacks. Although these hashing-based solutions are efficient in terms of mem-

*This work is partly supported by NSF grants IIS 02-20152, IIS 02-23022 and CF 04-23336



IEEE COMPUTER SOCIETY

ory usage, they are prone to approximation errors due to the hashing collisions. Further, these solutions are not suited for finding heavy distinct hitters over a sliding window. Venkataraman et al. [17] formalized the problem of the heavy distinct hitters and proposed two hashing based solutions referred to as 1-level and 2-level filtering techniques. Their solutions adapt well to the problem of finding heavy distinct hitters over sliding windows. However, apart from the errors induced by hash-collisions, they incur implementation overheads such as bucket-chaining and sorting-based implementations [8, 17]. On the other hand, widely deployed solutions, such as NetFlow [1] and Snort Intrusion Detection System [16], use a relatively simple approach. Each unique pair in the already processed stream is stored and compared with every incoming pair of elements. For each source, if the destination count increases beyond a certain threshold, it is output as a heavy hitter. Although this approach is accurate, it becomes computationally expensive as the number of unique pairs increases. On a network router, where the number of unique flows is in the order of a million, the solution proposes to use sampling to reduce the number of times a flow is compared with the existing flows. However, this process introduces estimation errors depending on the sampling rate. Ideally, it is desirable if the performance cost and the error is invariant to the number of flows and still achieves exact guarantees on the accuracy.

We observe that both the worm attack detection and the DDoS applications execute on network routers which operate on non-conventional hardware called the 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 heavy distinct hitter problem.

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 to develop several TCAM-conscious solutions for the heavy distinct hitter problem.
- We present a TCAM-conscious technique, referred to as the TCAM heavy distinct hitter (THDH), for finding the heavy distinct hitters in a data stream. We extend this technique to answer the heavy distinct hitter queries over sliding windows. These techniques incur a constant number of TCAM operations per stream element and provide bounded error guarantees unlike other software based techniques which do not provide any such exact guarantees.
- We also adapt a popular heavy distinct hitter algorithm, the 1-Level Filter, to the TCAM model and present unrestricted and sliding versions of this algorithm. These algorithms, although lacking the exact guarantees of the THDH technique, are relatively much faster.

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 heavy distinct hitter problem, summarize various proposed solutions and their implementation issues, and motivate the need for more efficient solutions. In Section 4, we present TCAM-based solutions for heavy distinct hitters over unrestricted and sliding windows. Section 5 presents the experimental evaluation of the techniques. Due to space constraints, we present a more detailed version of this paper in [5].

## 2 Advanced Networking Architecture

In this section, we describe the architectures of Network Processing Units (NPU) and Ternary Content Addressable Memories (TCAM) and discuss the integrated setup of NPUs 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 were built which can operate at multi-gigabit speeds [2]. However such systems offer limited configurability and software programmability thereby limiting the reuse of existing systems. In order to enable re-usability, high-performance microprocessors called the Network Processing Unit (NPU) have been developed with a very flexible software programming capability.

A typical network processor, such as the Intel IXP2800 [2], has one control plane processor (CPP) and 16 data plane processors on a single die. 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.

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. In addition to the widely used memories like SRAM and DRAM,


IEEE COMPUTER SOCIETY

NPUs feature TCAMs used for IPV4 forwarding application which routes network packets to the appropriate destination according to the routing table. 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 entire array in parallel 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.

TCAM also supports 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 the local masking. The masking feature of the TCAM enables the representation of ranges and range matching inside the TCAM which enables high speed packet classification [10] 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[9], giga-bit rate pattern-matching [18], linear time sorting [15]. In the context of data streams, we have shown that the NPU/TCAM integrated setup can be used to develop data stream algorithms for finding frequent elements which are significantly faster than software-based implementations [6]. 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 solutions for finding heavy distinct hitter over data streams.

## 3 Heavy Distinct Hitters: Background

In this section, we summarize existing solutions for finding heavy distinct hitters and motivate the need for TCAM-based solutions. As described before, a *heavy distinct hitter* algorithm finds all the elements which are uniquely paired with most other elements in the data stream. Similarly, the *heavy hitter* technique finds the most frequent elements in a data stream. Although the *heavy distinct hitter* problem appears very similar to the *heavy hitter* problem, the semantics of the two problems is mostly orthogonal. For example, a network host which accounts for a significant amount of traffic (i.e. a *heavy hitter*) need not be a *heavy distinct hitter* as the traffic might correspond to a large regular file transfer to just a single host. Experimental results from [17] reveal that some *heavy distinct hitters* account for only 0.004% of total traffic. This implies that the solutions (both software and TCAM-conscious) for finding heavy hitters are not applicable in the context of heavy distinct hitters.

Most of the widely deployed solutions for the heavy distinct hitter problem owe their origins to the port scan application. Popular solutions such as NetFlow [1] and Snort [16] use a naive approach: For each active connection, they maintain a record and for each source, a counter to count the number of connections. Although this technique provides accuracy, it is expensive to maintain both in terms of performance and memory utilization.

In [8], the authors present a series of bitmap algorithms which are updated for each incoming flow. For each source in the network traffic, they maintain a bit map which estimates the number of unique destinations it contacts. In [8], the authors proposed algorithms for finding $k$-super-spreaders, which are essentially heavy distinct hitters with a threshold $k$. They describe two hashing-based algorithms referred to as the 1-level and 2-level filtering techniques.

In the 1-level filtering technique, two hash tables $T_1$ and $T_2$ represent the state of the algorithm. $T_1$ stores $< s, d >$ pairs and $T_2$ stores connection count information for all the sources. For each incoming pair, the pair is hashed using a hashing function $h_1$ and only those pairs whose hash value fall in the range [0, $\frac{1}{r}$ ), where $r$ is the sampling rate, are chosen. Using this value, an index into the table $T_1$ is calculated and the corresponding location is checked for the given pair. If it is present, the algorithm proceeds to the next input pair. Otherwise, a pair is added at the particular location. The hash table $T_2$ contains the counter information of each source. If an input pair does not have a match in $T_1$, it means that the pair has not arrived before and hence the connection count for the respective source has to be incremented. The source component of the pair is hashed into $T_2$ and if the source is present, its count is incremented. Otherwise, a new bucket is added in the hash table to account for the new source. All sources in the table $T_2$ whose count is greater than $k$ are labeled as heavy distinct hitters. Using the core sampling approach, a more space-efficient and faster technique referred to as the 2-level filtering technique is proposed [17].

Next, the basic algorithm is extended to handle the case of finding heavy distinct hitters over sliding windows. In the sliding window version, only those sources which are



IEEE COMPUTER SOCIETY

heavy distinct hitters over the last $n$ pairs are required. That is, upon a new pair's arrival, an old pair needs to be discounted from the results. This is done by adding time stamp information to the pairs inside $T_1$. Apart from the source-destination information, each pair is assigned a time stamp which can be the actual arrival time stamp or the logical sequence number of a network packet. When a pair gets through the sampling process, its time stamp information is reset to reflect the latest arrival. After the addition of the new pair whose time stamp is $TS_{current}$, $T_1$ is searched for a pair whose time stamp is $TS_{current} - n$ and removed from the table. The connection count for the source in table $T_2$ is decremented to reflect this change.

We now discuss implementation issues of each of these techniques and motivate the need for TCAM-Conscious solutions. The naive approach used by Netflow and Snort does not scale well with the number of network flows. It incurs errors in estimation due to the proposed sampling approach. The bitmap algorithms are hashing based and provide probabilistic bounds on the accuracy of results. Further, this approach does extend to finding heavy distinct hitters over sliding window. The 1/2-level filtering techniques suffer from the bucket-chaining and duplicate sampling problem. If more than one pair maps to the same location in $T_1$, Estan et al. propose to use bucket-chaining which results in a performance hit. In the case of a large regular network flow, if the first packet gets sampled, then all of its successors get sampled. This requires looking up for the pair every time it occurs and can become expensive if the hit entails searching inside a bucket chain. In the sliding window version of the algorithm, it is necessary to search for the smallest time stamp after each packet's arrival. In order to expedite this search, a sorted ordering of all the pairs is recommended [17]. However this requires pointer re-arrangement at each arrival and is very expensive to implement. Ideally, it is preferable to have the accuracy of the non-sampling naive approach while providing the speed of the hashing based solutions. We observe that the TCAMs, which are used by many networking applications, enable high speed constant time searching irrespective of the size of the data repository.

# 4 TCAM-Conscious Heavy Distinct Hitter Algorithms

In this section, we present TCAM-Conscious algorithms for the heavy distinct hitter problem over unrestricted and sliding windows. In all cases, we start with abstract formulation of the algorithms, discuss various bottleneck issues in their software implementations, and present how TCAMs can be used to alleviate the bottleneck issues. First, we present the algorithm for unrestricted window size, which we refer to as Heavy Distinct Hitter algorithm (HDH), and show that searches contribute for most of the per-stream-element cost. We then show how this search cost can be efficiently reduced using the *constant-time* search capability of TCAMs. Next, we present Heavy Distinct Hitter algorithm over Sliding Windows (HDHSW), the sliding window version of HDH, and show how TCAMs can be exploited for an efficient implementation. We then describe the TCAM-adapated versions of the 1-Level Filtering technique for both unrestricted and sliding windows [17].

## 4.1 Heavy Distinct Hitter Algorithm for Unrestricted Windows

We first present the Heavy Distinct Hitter algorithm (HDH) as shown in Algorithm 1. Basically, the algorithm stores source-destination pairs and per-source-connection-count information in two separate tables. The source-destination pairs are stored to check if a given source-destination pair has already been seen in the data stream. Algorithm 1 is based on the intuition that a *heavy distinct hitter* with a threshold of $k$ is indeed a *heavy hitter* with a threshold of $k$ if only the source component of the flow is used in aggregation. For example, if a source $A$ makes 10000 distinct connections, then it corresponds to at least 10000 packets at the router whose source component is set to $A$. If this data stream is aggregated only on the source component, then $A$ becomes a heavy hitter with a threshold of at least 10000. By only storing the source-destination pairs for those flows whose source contributes network traffic which is more than a fraction of the heavy distinct hitter threshold ($k$), we can avoid storing information of all the flows whose source makes a small number of low-bandwidth connections. For example, if we store only those flows whose source contributes for more than 10% of the threshold, this avoids all the sources which contribute to traffic of less than 1000 packets.

**Algorithm 1** Heavy Distinct Hitter (HDH) algorithm

```
for each element ⟨ s, d ⟩ in stream do
  if T1 contains ⟨ s, d ⟩ then
    go to next pair
  else
    if T2 contains s then
      READ(s, frequency) from T2
      frequency = frequency + 1
      if frequency ≥ ε * k then
        add ⟨ s, d ⟩ to T1
      end if
    else
      add s to T2
    end if
  end if
end for
```



IEEE COMPUTER SOCIETY

HDH keeps a per-source-connection counter for all the sources seen till now. This counter is incremented initially for each packet seen irrespective of the destination. Once this counter crosses a certain threshold, all the source-destination pairs for that source are stored from then on. Once the threshold is crossed, the per-source-connection counter is incremented only upon the addition of a new pair. Formally, if a source is a heavy distinct hitter with a threshold $k$, then it occurs at least $k$ times in the data stream. By only storing source-destination pairs for a flow which has a packet count of more than $\epsilon * k$ where $\epsilon$ is the error threshold, the error in estimating the actual number of unique connections per source is $\epsilon * k$. That is, the actual number of distinct connections in the data stream for a source can be offset from the estimate from the per-source-connection counter by at most $\epsilon * k$.

It can be noted that the above approach generates false positives. A flow corresponding to a genuine large file transfer from a source $A$ can potentially result in HDH storing all the flows from $A$. However, this is acceptable as the number of distinct connections from a genuine source are typically small compared to an infected source thus wasting little space. This approach is a more general version of the naive approach used by Netflow [1] and Snort [16]. Instead of storing every distinct flow in the traffic, Algorithm 1 stores only those flows whose source contributes to more than a certain amount of traffic. However, this technique also suffers from the same scaling issues as the above mentioned techniques [8]. That is, searching for a given pair among the stored pairs becomes expensive as the number of stored pairs increases. Although hashing and indexing can be used to expedite the search, they suffer from their respective drawbacks. For example, indexing incurs logarithmic search cost while hashing suffers from unused memory problem and bucket-chaining. We now present the TCAM-Conscious version of Algorithm 1 which efficiently addresses the search issue.

The HDH algorithm has a direct mapping to the associative memory model of a TCAM. A TCAM can be configured to store multiple databases of different widths. The source-destination pairs are stored inside the TCAM in one of the databases while the other database contains the source-counter information. Using the tables stored inside the TCAM, the TCAM-Conscious algorithm as illustrated in Algorithm 2 is a straight-forward adaptation of the HDH algorithm. The search for a source-destination pair or a source for counter information is replaced by the *constant-time* SEARCH command on the TCAM. New entries to the tables are added by simply adding TCAM entries to the respective databases inside the TCAM.

It can be seen that the TCAM-conscious HDH incurs a constant number of TCAM operations per flow in the data stream. If the given flow is already accounted for, it incurs only one LOOKUP. If it is not being accounted for, it incurs another LOOKUP for finding the source's entry inside $T_2$. If the source is not encountered till now, it incurs one WRITE for adding a new entry. Otherwise, depending upon the current frequency, it incurs at most 2 WRITES. Thus each flow might incur costs varying from 1 TCAM operation to 5 TCAM operations. In the case of a typical flow which corresponds to a large file transfer, it only incurs 1 LOOKUP operation. In the case of a heavy distinct hitter, it incurs 5 TCAM operations. Since the heavy distinct hitter traffic is usually a rare phenomenon, this algorithm incurs a little more than 1 TCAM operations for most of the flows.

**Algorithm 2** TCAM-Conscious Heavy Distinct Hitter (THDH) algorithm

**for** each element $\langle s, d \rangle$ in stream **do**
  **if** LOOKUP($T_1$, $\langle s, d \rangle$, address_if_hit) == hit **then**
    go to next pair
  **else**
    **if** LOOKUP($T_2$, $\langle s \rangle$, address_if_hit) == hit **then**
      READ($T_2$, frequency, address_if_hit)
      WRITE(($T_2$, $\langle s, frequency + 1 \rangle$, address_if_hit)
      **if** frequency$\geq \epsilon * k$ **then**
        WRITE($T_1$, $\langle s, d \rangle$, next_free_address_$T_1$)
      **end if**
    **else**
      WRITE($T_1$,$\langle s, 1 \rangle$, next_free_address_$T_2$)
    **end if**
  **end if**
**end for**

## 4.2 HDH over Sliding Window (HDHSW)

In the rest of this section, we extend the HDH algorithm to address the problem of finding heavy distinct hitters over a sliding window of size $n$. As before, we first present the HDH algorithm over Sliding Window (HDHSW), discuss the implementation issues in software, and show how this technique can be efficiently adapted to the TCAM model. As shown in the Algorithm 3, the HDHSW algorithm is an augmented version of the HDH algorithm. Unlike the case of the unrestricted window problem where source-destination pairs are only added, the sliding window problem also needs to take care of deletion of pairs as the window moves. We use time stamps [17] to augment the HDH algorithm to manage the information regarding which pairs need to be stored or removed from the data structures. An entry in sliding window version of $T_1$ stores $\langle src, dst, timestamp \rangle$ where $timestamp$ may be the arrival time or the logical sequence number of a packet. In the case of multiple occurrences of the same pair with in the current window, the $timestamp$ of the pair is set to the timestamp of the latest pair.


IEEE COMPUTER SOCIETY

Upon the arrival of a new packet, the source-destination pair is either added to $T_1$ if pair is new or the timestamp of the pair is updated if the pair already exists. It needs to be checked if the pair corresponding the last packet in the moving window has to be dropped. It should be noted that since the windows can be large, storing all the elements in the current window is not feasible and hence we do not know which pair is actually being dropped. The only information we have is that the outgoing packet can potentially have a time stamp of $TS_{current} - n$. If there are no duplicate occurrences of the outgoing pair, then this pair will have a time stamp of $TS_{current} - n$ and hence it can be dropped. Therefore for each incoming pair, the HDHSW algorithm needs to check if there is another pair with the time stamp of $TS_{current} - n$. In [17], the authors propose to order all the pairs inside $T_1$ based on the time stamp so that this search can be done faster. Once the pairs are stored in a sorted list, a comparison with the least time stamp in this sorted list with $TS_{current} - n$ after each window moment reveals whether an old pair needs to be dropped. However, maintaining the sorted list requires a lot of pointer rearrangement which is not desirable. We now present an efficient adaptation of this algorithm to the TCAM model.

**Algorithm 3** Heavy Distinct Hitter (HDH) algorithm over Sliding Window of size $n$

```
for each element ⟨ s, d ⟩ in stream do
  if T1 contains ⟨ s, d ⟩ then
    Update ⟨ s, d, TS_old ⟩ with ⟨ s, d, TS_current ⟩
  else
    if T2 contains s then
      READ(s, frequency)
      frequency = frequency + 1
      if frequency≥ ε * k then
        add ⟨ s, d ⟩, TS_current to T1
      end if
    else
      add s to T2
    end if
    if search for ⟨ s+, d+, TS_current − n ⟩ in T1 is success then
      delete ⟨ s+, d+, TS_current − n ⟩ from T1
      update ⟨ s+, count⟩ in T1 to ⟨ s+, count − 1⟩
      dropping the tuple if count was 1
    end if
  end if
end for
```

In addition to the straight-forward mapping of the HDH algorithm to the TCAM model, the TCAM adaptation of HDHSW as shown in Algorithm 4 also efficiently searches for a flow with the minimum time stamp. This is achieved using the masking feature of TCAMs. The masking feature allows a TCAM to selectively include only the specified bits of each TCAM entry during a search. For example, if a SEARCH command with search key $\langle\ src, dst, timestamp\rangle$ and a mask with the source-destination component masked out is issued on the database $T_1$, it returns the first $T_1$ entry which has a matching time stamp while ignoring the source-destination component. By issuing such a SEARCH command with the time stamp set to $TS_{current} - n$, this feature enables *constant-time* search cost for finding the required pair. Based on this observation, this algorithm (see Figure 4) uses two global mask words $mask_1$ and $mask_2$ which mask out the time stamp and the source-destination components respectively. $mask_1$ is used for searching using the source-destination components while $mask_2$ enables the search for a particular time stamp.

**Algorithm 4** TCAM-conscious Heavy Distinct Hitter algorithm over Sliding Window (THDHSW) of size $n$

```
/* mask1, mask2 mask out ⟨ count⟩ and ⟨ src, dest⟩
components for the elements in T1 respectively */
for each element ⟨ s, d ⟩ in stream do
  if LOOKUP(T1, ⟨ s, d ⟩, mask1, address_if_hit) == hit
  then
    WRITE(T2, ⟨ s, d⟩, TS_current, address_if_hit)
  else
    if LOOKUP(T2, ⟨ s⟩, address_if_hit) == hit then
      READ(T2, frequency, address_if_hit)
      WRITE(T2, ⟨ s, frequency + 1⟩, address_if_hit)
      if frequency≥ ε * k then
        WRITE(T1, ⟨ s, d ⟩, next_free_address_T1)
      end if
    else
      WRITE(T2, ⟨ s, 1⟩, next_free_address_T2)
    end if
    if LOOKUP(T2, ⟨ s+, d+, TS_current − n ⟩, mask2,
    address_if_hit) == hit then
      DISABLE(T2, address_if_hit)
      UPDATE ⟨ s+, count⟩ in T1 to ⟨ s+, count − 1⟩
      disabling the entry in T2 if count is 1
    end if
  end if
end for
```

This algorithm incurs varying number of TCAM operations depending upon the incoming pair. If the incoming pair is already stored, then the THDHSW algorithm incurs a search and a write for updating the time stamp. On the other hand, if the incoming source-destination pair is not yet seen and if the packet is the first from the given source, the algorithm can incur an overall cost of 3 TCAM operations. In the sliding phase, irrespective of the type of traffic this algorithm incurs a lookup for the search of the out-going time stamp, followed by a decrement of source frequency


0-7695-2837-3/07 $20.00 © 2007 IEEE

count in the case of a match. In the case of regular traffic, search for a particular timestamp typically fails, thereby incurring only one TCAM operation in the sliding phase. Henceforth, the THDHSW algorithm incurs 3 TCAM operations for most of the traffic unlike the case of 1 TCAM operation for the unrestricted version.

### 4.3 TCAM-conscious 1-Level Filtering technique

The 1-Level Filtering technique also adapts well to the the TCAM model. When a new source-destination pair arrives, the 1-Level Filtering technique uses a hash table to find if the given element is present in the data structure $T_1$ which stores the source-destination pairs. While this requires only a constant time operation to lookup in the hash table, the hash table incurs a lot of space wastage when the pairs do not get mapped to any buckets. Further, in the case of hash collisions, this requires bucket chaining which is not desirable. We observe that this hash table can be efficiently mapped to a regular table inside the TCAM. When a new pair has to be added, we just add it at the next available entry inside the TCAM. A lookup for a source-destination pair in the hash table gets mapped to a LOOKUP command on the TCAM. Similarly the hash table $T_2$ can also be replaced with a table inside the TCAM. By replacing the hash tables $T_1$, $T_2$ with TCAM tables, we present the TCAM-conscious 1-Level Filtering technique, which we refer to as TCAM1LF. Due to space constraints, the pseudocode for this algorithm is given in [5]. This algorithm samples the packets and updates the data structures $T_1$, $T_2$ inside the TCAMs in the same way as a software implementation of the 1-Level Filtering technique does. Here, we only present the unrestricted window version of the algorithm. The sliding window version of this algorithm can be easily developed using the THDHSW algorithm described in this section. We refer to TCAM-conscious unrestricted and the sliding window versions of the algorithm to as TCAM1LF and TCAM1LFSW respectively.

## 5 Experimental evaluation

We now present an experimental evaluation of the TCAM-conscious techniques described in the previous section. We compare the TCAM-conscious techniques (THDH, THDHSW, TCAM1LF, TCAM1LFSW) with the software implementations of the 1-level Filtering technique [17] for unrestricted and sliding windows. We chose 1-Level Filtering technique because this is the only technique which is suitable for both unrestricted and sliding window problems. We evaluated the techniques on IXDP2801, Intel's platform for developing networking applications. This platform features IXP2800 [2], 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.

The Content Plane Processor (CPP) runs Monta Vista Linux, an embedded operating system and the micro engines run Teja Network Operating System [4]. Programming is done using Teja C, a version of Intel C enhanced for NPU. All the TCAM-conscious techniques (THDH, THDHSW, TCAM1LF, TCAM1LFSW) are developed on Micro Engines (MEs) running at 1.4GHz. For comparison purposes, we developed software implementations for unrestricted and sliding window versions of the 1-level Filtering technique [8]. Since a ME does not have L1 cache and is not ideally designed to handle programs with extensive pointer manipulations, as required by an efficient software implementation of these algorithms, we implemented these techniques on the CPP. This processor is a more conventional Xscale processor from Intel with 32K of L1 cache and operates at 700MHz accessing 768MB of DRAM and 32MB of SRAM. For comparison purpose, we also evaluated the same algorithms on a Pentium 4 processor running at 3GHz interfacing with 1GB of DRAM. It should be noted that Pentium 4 is much more sophisticated than the CPP and each of the MEs. Therefore, a direct comparison does not ideally reflect the true comparison with the TCAM-conscious techniques as they operate on a ME which has lesser computation power. However, we note that the NPU derives its tremendous computation power from multiple MEs (16MEs with 128 threads of execution).

### 5.1 Datasets and TCAM cost analysis

Evaluation of the three TCAM based techniques and the software implementations of the 1-Level Filtering technique is done with respect to the following metrics: query processing time and accuracy. We refer to the software implementation of the 1-Level Filtering technique on the Xscale and the Pentium platforms as Xscale1LF and Pentium1LF respectively. For experimentation, we use network traces from the NLANR repository [3] suitably injected with the heavy distinct hitter traffic. This experimental trace contains 684000 packets corresponding to 850 sources. We injected this traffic with 200 heavy distinct hitter sources which are associated with 500 different destinations. This injected data exhibits a uniform distribution. That is, we assign the threshold $k$ for a source to be defined as a super-spreader to be 500. We set the user-defined threshold factor $\epsilon$ for the THDH and THDHSW to be 0.1. Therefore, any source which contributes more than 50 packets can poten-


IEEE COMPUTER SOCIETY

tially be monitored by the THDH and the THDHSW algorithms. In order to test the accuracy of the 1-Level filtering technique, we also added sources with just under $k * \epsilon$ (=50) destinations. For the 1-level filtering algorithms, we monitor the performance for sampling factors ranging from 1 to 0.25. In all the cases, we report the performance time and the corresponding accuracy. We first report the results for the unrestricted windows followed by the restricted window version where the window size is set to 10000.

First, we present an analysis of various TCAM operations: READ, WRITE, SEARCH, LEARN, DISABLE. Table 1 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 potentially speed up by 40 times. Implementation of TCAM-based algorithms on an ASIC device is a standard practice [14, 10, 18] 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 1. Statistics of TCAM operations**

## 5.2 Unrestricted Window

Figure 1 presents the results for the THDH, TCAM1LF, Xscale1LF and Pentium1LF techniques. The x-axis represents various sampling factors used in the experiments while the y-axis represents the processing cost. The Xscale1LF, Pentium1LF and TCAM1LF techniques show a predictable linear scaling with the sampling rate as number of source-destination pairs accessed reduce linearly with the sampling rate. The results for THDH do not vary with the sampling factor, as THDH accesses every stream element. Figure 1 shows that the THDH and TCAM1LF algorithms perform much better than the Xscale1LF techniques at all the sampling factors. On the other hand, THDH performs comparably with the Pentium1LF implementation while TCAM1LF's performance is almost the same as the Pentium1LF at all sampling rates. When all the elements are sampled, the performance of THDH and TCAM1LF is almost the same as Pentium1LF. Sampling rate of 1 is of importance, because the 1-Level Filtering technique does not incur any false-negatives at this rate, as is always the case with the THDH technique. At lower sampling rates, where 1-Level Filtering technique can incur false negatives, Pentium1LF's and TCAM1LF's performance improves over the THDH algorithm. Furthermore, it should be noted that the results for THDH are from the evaluation on a prototype which can exploit only 2.5% of the processing capacity of the TCAM. In a more efficient ASIC implementation where TCAM operations can be potentially speeded up by 40 times, significant performance enhancement over the reported results of both THDH and TCAM1LF is plausbile.

Accuracy results show that THDH and the 1-Level Filtering technique achieve 100% accuracy in finding the heavy distinct hitters. Although these results are as expected in the case of the THDH algorithm, this high level of accuracy in the case of the 1-Level Filtering technique can be attributed to the distribution characteristic of the input data. The injected heavy distinct hitter traffic has a uniform distribution and hence hashing leads to uniform distribution of the elements. In a more realistic setting where the heavy distinct hitter traffic can be non-uniform, the 1-Level Filtering technique is known to have false negatives [17].

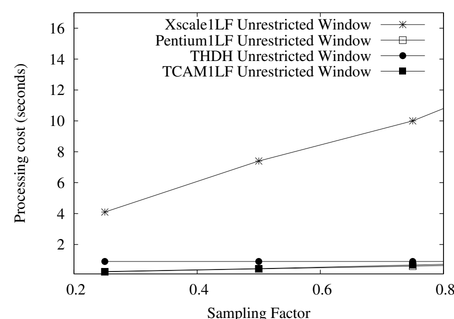


**Figure 1. Performance results**

## 5.3 Sliding Window

Figure 2 presents the performance results for the THDHSW, TCAM1LFSW, Xscale1LF, Pentium1LF techniques over a sliding window of size 10000. At this point, it should be noted that while Xscale1LF and Pentium1LF implementations use a sorted list implementation for window slide phase of the algorithm, TCAM1LFSW and THDHSW use TCAM-based approach for the same purpose. A comparison of Figure 2 and Figure 1 shows that the THDHSW

and TCAM1LF algorithms perform slower than the case of the unrestricted window versions. This is because both THDHSW and TCAM1LFSW algorithms have an additional *window slide* phase, which requires deleting the source-destination pair from the other edge of the window. On the other hand, the software implementations of the 1-Level filtering technique (Xscale1LF and Pentium1LF), which also have this *window slide* phase, perform relatively better than the unrestricted versions. This is because, the sliding window version maintains a much smaller data structure compared to the unrestricted version as it needs to take care of only the elements in the current window. Although the TCAM-conscious techniques also have to manage only few elements, TCAM operations incur the same cost irrespective of the size of the data structure. This shows that although TCAM-conscious techniques perform well in the case of unrestricted windows, they are not so ideal for sliding window cases which manage only a few elements.

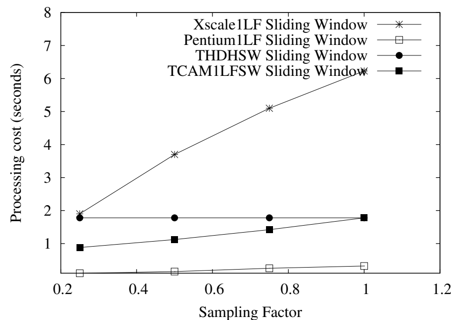


**Figure 2. Performance results**

## 6 Conclusion and Future Work

In this paper, we explored the heavy distinct hitters problem in data streams from a networking systems perspective. We proposed the HDH algorithm for finding heavy distinct hitters, which does not generate any false negatives, and applied it to both unrestricted and sliding window versions of the problem. We proposed to exploit TCAMs, a special kind of memories found in NPUs, for providing fast implementations for both the algorithms. We also adapted the unrestricted and sliding window versions of a popular heavy distinct hitter solution, the 1-Level Filter, to the TCAM model. While the TCAM-conscious HDH techniques provide exact accuracy guarantees, the faster TCAM-conscious 1-Level Filtering techniques provide probabilistic guarantees and are prone to false negatives. We implemented all the four TCAM-conscious techniques on an Intel's IXDP2801 NPU platform, and evaluated them over real network traces injected with heavy distinct hitter traffic. Even though the current prototype exploits only 2.5% of the processing capacity of the TCAM, experimental evaluation reveals that these techniques perform well in comparison to the existing software solutions. We plan to include real-life DDoS attack datasets to our experimental setup in the future.

## References


[1] Cisco netflow. http://www.cisco.com/warp/public/732/Tech/netflow.

[2] Intel npus. http://www.intel.com/, 2006.

[3] Nlanr archive. http://www.nlanr.net/, 2006.

[4] Teja networking systems. http://www.teja.com, 2006.

[5] N. Bandi, D. Agrawal, and A. El Abbadi. TCAM-conscious algorithms for data streams. In *UCSB Technical Report*, 2007.

[6] N. Bandi, A. Metwally, D. Agrawal, and A. El Abbadi. Fast algorithms for data streams using associative memories. In *SIGMOD*, 2007.

[7] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *ESA'02*.

[8] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. In *SIGCOMM'03*, 2003.

[9] R. H. K. Fang Yu. Efficient Multi-Match Packet Classification with TCAM. In *IEEE Hot Interconnects*, 2004.

[10] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for advanced packet classification with ternary cams. In *SIGCOMM '05*.

[11] G. S. Manku and R. Motwani. Approximate frequency counts over data streams, 2002.

[12] 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.

[13] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4), 2003.

[14] S. Mysore, B. Agrawal, T. Sherwood, N. Shrivastava, and S. Suri. Profiling over adaptive ranges. In *CGO '06*, 2006.

[15] R. Panigrahy and S. Sharma. Sorting and Searching using Ternary CAMs. In *IEEE Micro.*, 2003.

[16] M. Roesch. Snort - lightweight intrusion detection for networks. In *LISA '99: Proceedings of the 13th USENIX conference on System administration*.

[17] S. Venkataraman, D. X. Song, P. B. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreaders. In *NDSS*, 2005.

[18] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using tcam. In *ICNP '04*, pages 174–183, 2004.




IEEE COMPUTER SOCIETY