electronics

Article

ComPipe: A Novel Flow Placement and Measurement Algorithm for Programmable Composite Pipelines

Dengyu Ran [1,2], Xiao Chen [1,2,3] and Lei Song [1,2,*]

[1] National Network New Media Engineering Research Center, Institute of Acoustics, Chinese Academy of Sciences, Beijing 100190, China; randy@dsp.ac.cn (D.R.); xxchen@dsp.ac.cn (X.C.)
[2] School of Electronic, Electrical and Communication Engineering, University of Chinese Academy of Sciences, Beijing 100049, China
[3] Peng Cheng Laboratory, Shenzhen 518055, China
* Correspondence: songl@dsp.ac.cn

Abstract: Programmable networks comprise heterogeneous network devices based on both hardware and software. Hardware devices provide superior bandwidth and low latency but encounter challenges in managing large table entries. Conversely, software devices offer abundant flow tables but have a limited forwarding capacity. To overcome this limitation, some commercial switches offer implementations that combine both hardware and software devices. In this context, this paper presents the Composite Pipeline (ComPipe), an algorithm for high-performance and high-precision flow placement and measurement. ComPipe utilizes a multi-level hashing algorithm for the real-time identification of heavy hitters, incorporates a unique flow eviction strategy, and is implemented on commercial programmable hardware. For non-heavy flows, ComPipe employs sketch structures to accomplish a high-performance flow synopsis within limited memory constraints. This design allows to replace flow rules entirely in the data plane, ensuring the timely detection and offloading of heavy-hitter flows, and offering a unified interface to the controller. The ComPipe prototype has been implemented in both testbed and simulation environments. The results indicate that ComPipe is an effective solution for dynamic flow placement in programmable networks, distinguished by its low cost, high performance, and high accuracy.

Keywords: programmable networks; composite pipeline; heterogeneous network devices; flow placement and measurement; high performance and precision

check for updates

Citation: Ran, D.; Chen, X.; Song, L. ComPipe: A Novel Flow Placement and Measurement Algorithm for Programmable Composite Pipelines. *Electronics* **2024**, *13*, 1022. https://doi.org/10.3390/electronics13061022

Academic Editor: Christos J. Bouras

Received: 17 January 2024
Revised: 23 February 2024
Accepted: 27 February 2024
Published: 8 March 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

In contemporary heterogeneous network environments, devices leveraging programmable networks exhibit distinct advantages and constraints, driven by underlying hardware and software paradigms. Table 1 outlines these differences: hardware devices like programmable Application Specific Integrated Circuit (ASIC) based on the Reconfigurable Match-Action Table (RMT) architecture [1] provide high bandwidth, low latency, and reduced jitter. They also offer custom processing flexibility via P4 language-defined pipelines. However, their functionality is limited by scarce lookup resources, especially when it comes to managing large table entries. In contrast, software-based switches, such as Open vSwitch (OVS) [2], feature ample memory for storing flow tables but are hindered by a constrained forwarding capability, due to the high CPU clock cycle demands per packet. Consequently, many commercial network devices, including cloud gateways [3–5] and Smart Network Interface Cards (Smart-NICs) [6,7], are adopting composite pipeline structures. This approach leverages the strengths of both hardware and software, aiming to enhance performance and flexibility in diverse network settings.

Table 1. Comparison of programmable pipeline implementations.

Type of Implementation	Flow Table Capacity	Lookup Speed	Update Speed	Complexity
Hardware Pipeline	low	high	high	low
Software Pipeline	high	low	low	low
Composite Pipeline	high	high	high	high

In the composite pipeline structure, packet processing initiates with hardware-based tables. If a packet finds a matching entry, the corresponding actions are executed; if not, it gets rerouted to the CPU for software-based table matching. This design only resorts to software tables when hardware tables lack a matching rule, thus enhancing forwarding performance and reducing the burden typically associated with pure software forwarding. Furthermore, within the cloud network, this method reduces the processing headroom that network engineers reserve for each CPU core, thereby further lowering both Capital Expenditures (CapEx) and Operating Expenditures (OpEx) [3]. Yet, the dynamic nature of network traffic introduces a complex challenge in flow table placement [8]. With continually ever-changing traffic patterns, it becomes crucial to develop an intelligent and adaptable algorithm for managing flow rules across both hardware and software tables. Such an algorithm must dynamically respond to traffic fluctuations, strategically allocating flow entries between hardware and software based on their immediate utility and overarching performance objectives. This method seeks a balance between the efficiency of hardware processing and the scalability and flexibility of software solutions.

When designing a flow table placement algorithm, it is crucial to fully consider the skewed pattern of network traffic, a characteristic often mathematically modeled by Zipf's distribution [9–11]. Specifically, significant flows in the network, also known as heavy hitters or "elephant flows", constitute only a small portion of all network flows in terms of number but occupy a substantial amount of bandwidth resources. In contrast, the less significant flows, also referred to as "mouse flows", dominate in number but account for a relatively small proportion of the bandwidth. In the context of a composite pipeline, effectively utilizing the skewness of traffic—by dynamically identifying heavy hitters and offloading them to the hardware—holds the potential to enhance traffic processing capabilities while further reducing the CPU resource consumption of software switches.

Current methods primarily involve the controller periodically polling the data plane's counters to collect statistical information, followed by the execution of the placement algorithm [8]. However, this polling mechanism suffers from a lag in timeliness, impacting the effectiveness of flow table replacements. For instance, heavy-hitter characteristics might shift in the interval between controller actions. An alternative approach is a periodic replacement strategy, which involves swapping the flow tables between software and hardware when the software traffic exceeds the hardware traffic. This method, however, can result in frequent flow table replacements and not fully utilize the advantages of hardware forwarding. A more recent development is the Elixir [5], which utilizes a sampling method for flow placement. While this method eases hardware implementation and accommodates more intricate replacement algorithms, its fixed sampling rate may result in a scenario where heavy hitters are still partially processed by the software flow table. Furthermore, the substantial CPU and bandwidth requirements for processing sampled packets in software make high-rate sampling an impractical approach. Consequently, there is a pressing need for more sophisticated and adaptable strategies to overcome the challenges of timeliness and frequency in flow table replacements. Implementing such methods could lead to a more efficient use of hardware and software resources, significantly enhancing overall system performance.

In this study, we introduce Composite Pipeline (ComPipe), a novel methodology for flow placement and measurement. ComPipe is uniquely designed to identify heavy hitters in real-time, achieving this with minimal memory overhead on hardware. Concurrently, it manages less flows through software, ensuring high throughput overall. At its core,

ComPipe employs a multi-level, hash-based algorithm adept at detecting heavy hitters, coupled with an innovative flow 'eviction' strategy. This approach efficiently maintains a register table for heavy hitters within the hardware. For less heavy flows, ComPipe leverages sketch data structures to construct statistical flow synopsis, effectively optimizing performance even when faced with memory limitations.

This design empowers the data plane to independently manage flow table replacements, yielding multiple benefits. Firstly, it allows for the swift identification and hardware offloading of heavy hitters, facilitating rapid processing. Secondly, this approach conservatively utilizes CPU resources in the software pipeline, which is a critical advantage in cloud environments. Furthermore, it presents a unified southbound interface for the controller, simplifying network function development and data plane programming. This is achieved by using standard southbound interfaces such as OpenFlow [12], POF [13], and P4Runtime [14]. For the control plane, instead of distinguishing between software and hardware pipelines, the composite pipeline is managed as a single device, significantly reducing the complexity of network management.

We developed a ComPipe prototype and deployed it on a testbed platform comprising a DPDK-based POF Switch [15] and a Barefoot Wedge100BF-65X P4 Switch. To evaluate the performance of ComPipe, extensive experiments were conducted in a simulated environment, benchmarking ComPipe against leading-edge flow measurement algorithms. The results of these experiments reveal that ComPipe consistently delivers more precise measurements across two fundamental tasks of network flow detection.

In summary, this paper makes the following contributions:

- We present ComPipe, an innovative flow placement and measurement algorithm designed for the composite pipeline architecture. While its potential applications span numerous critical domains, this field has traditionally been underexplored in research.
- We have implemented the heavy-hitter detection algorithm entirely in the data plane, ensuring its full compatibility with the RMT architecture. This enhancement provides a more reliable and efficient mechanism for real-time identification of large traffic flows at the hardware.
- In the software pipeline, we have implemented the recording of less heavy flows, employing a multi-core approach to expedite this process. This strategy is aimed at significantly enhancing the overall measurement throughput of the software data plane.
- We conducted extensive experiments in both testbed environments and simulations to verify the high performance and precision of ComPipe. These experiments demonstrate the feasibility and effectiveness of ComPipe in practical applications.

The remainder of this paper is structured as follows: Section 2 offers a comprehensive review of related work in the realms of flow placement, heavy-hitter detection, and network flow recording. In Section 3, we detail the design and underlying algorithms of ComPipe, accompanied by a mathematical analysis. Section 4 elucidates how ComPipe facilitates multiple measurement tasks, along with the technical specifics of executing these tasks. Section 5 discusses practical implementation strategies for ComPipe on commercial hardware and software platforms. The evaluation of ComPipe is thoroughly examined in Section 6. Finally, Section 7 provides concluding remarks and suggests potential avenues for future research.

2. Related Work

Flow Placement: Numerous academic studies, based on the principle of traffic skewness, have delved into flow placement strategies in software and hardware data planes. TFO [16] employs controllers for monitoring traffic across varied time scales, segmenting flow tables based on traffic skewness. CacheFlow [17] constructs a directed acyclic graph model to identify and analyze dependencies between rules, employing either the Dependent-Set Algorithm or the Cover-Set Algorithm to determine which rules should be cached in hardware. LFP [18] proposes a method for network function offloading, blending sampling and machine learning to decide on hardware node offloading based

on initial flow packet predictions. OVS-CAB [7] presents an offloading mechanism for Open vSwitch on smart network interface cards, addressing rule overlap issues effectively. Sailfish [3], a pioneering cloud gateway utilizing programmable switches, employs a dual hardware–software design. It strategically places crucial table entries like VXLAN routing and VM-NC mapping in hardware, while retaining volatile and large stateful tables in software. However, it lacks dynamic migration capability for these tables based on traffic variations. Elixir [5] differentiates between heavy and burst traffic through static sampling and counters, decoupling flow rate identification from the replacement process. Nonetheless, its static sampling approach might compromise accuracy in low-speed traffic scenarios or be resource-intensive for high-speed traffic. Previous studies primarily concentrated on intelligent software algorithms for decision offloading in flow table partitioning. In contrast, ComPipe emphasizes programmable heterogeneous pipeline environments, exploiting hardware data plane programmability to measure large flows directly on ASIC, thus minimizing software–hardware communication overhead.

Heavy-Hitter Detection (HHD): Narrowly defined, a "Heavy Hitter" refers to network flows occupying bandwidth beyond a specific threshold. Broadly speaking, HHD techniques can be applied to other measurement objectives, such as identifying persistent flows, heavy change flows, and super spreaders. HashParallel and HashPipe [19] approximate the implementation of the Space-Saving algorithm [20] to meet the design constraints of the P4 language. This method involves dividing counters into multiple independent stages to adapt to the match-action structure of programmable data planes. However, these methods do not satisfy the design constraints of programmable switches under the RMT architecture. Building on the foundations of HashPipe, PRECISE [21] introduces a scheme that adeptly balances partial packet recirculation, facilitating high-speed processing in hardware programmable switches based on RMT. FlowRadar [22] introduces an innovative approach, where the data plane encodes network flows using XOR operations, with decoding managed by the control plane. This method employs joint data structures, such as heaps, for effective heavy-hitter identification and can be implemented on the data plane via P4. Conversely, HeavyKeeper [11] adopts a count-with-exponential-decay strategy to proactively filter out lower-frequency items, thereby enhancing the accuracy of heavy-hitter detection. In the realms of heavy-hitter and Top-k query accuracy, HeavyKeeper demonstrates remarkable effectiveness. Other algorithms used for heavy-hitter queries include classic methods under traditional network frameworks, such as Frequency [23], Lossy Counting [24], Unbiased Space Saving [25], and Augmented Sketch (AS) [26].

Record of Network Flows: Sketch algorithms constitute a series of classic methods for processing data streams. Their core objective is to record the frequency of all items while trading off some completeness in data stream recording for reduced memory usage in data structures. Representative algorithms in this category include Count Sketch (CS) [27], Count-Min Sketch (CMS) [28], and Conservative Update Sketch (CUS) [28]. These algorithms typically employ a data structure composed of a $d \times w$ array of counters, with each row functioning as a hash table of length w and a total of d rows. They map items onto d counters through multiple hash mappings. The primary distinction among them lies in their varying update strategies. Both CMS and CUS algorithms offer one-sided error guarantees, meaning the reported frequencies will not be lower than the actual values. In contrast, CS features two-sided error, indicating that query results might either overestimate or underestimate the true values. SALSA [29] proposes a method of dynamically adjusting counter sizes, merging counters with their neighbors upon overflow, thereby allowing more counters within a given space to enhance the accuracy of data stream analysis. However, this merging logic inevitably slows down the data insertion speed. TowerSketch [30] improves measurement accuracy by using counters of different sizes to record flows of varying magnitudes while maintaining simplicity in data insertion operations. TowerSketch's update strategy can be chosen from CS, CMS, or CUS. Elastic Sketch (ES) [10] is a general network measurement algorithm that distinguishes frequently from infrequently occurring items through a ostracism mechanism. ES consists of two parts:

the heavy part, a hash table for recording frequent items, and the light part, a CMS for infrequent items. ES is capable of addressing six network measurement tasks, including heavy-hitter detection. Additionally, there are other well-known sketch algorithms such as the LogLog Filter (LLF) [31], MV-Sketch [32], Pyramid Sketch [33], and OneSketch [34].

3. Design

3.1. Design Overview

As shown in Figure 1, ComPipe's method for segregating heavy-hitter flows from less heavy flows is illustrated. Within the hardware pipeline (HP), a multi-level hash structure is employed to accurately identify and record heavy hitters. Simultaneously, less heavy flows are redirected to the software pipeline (SP) through an eviction algorithm, aiding in sketch construction. This technique effectively addresses flow distribution skewness; the HP significantly improves frequency estimation accuracy by preemptively filtering out the most frequent items. In the SP, redundancy is minimized since items already recorded in the HP are not duplicated in the sketch, reducing the likelihood of collisions between non-heavy flows and heavy hitters. This not only decreases the potential for misclassifying low-frequency items as having high frequency but also substantially enhances the system's overall precision in traffic measurement and classification. Table 2 outlines the main notations used in Sections 3.2 and 3.3.

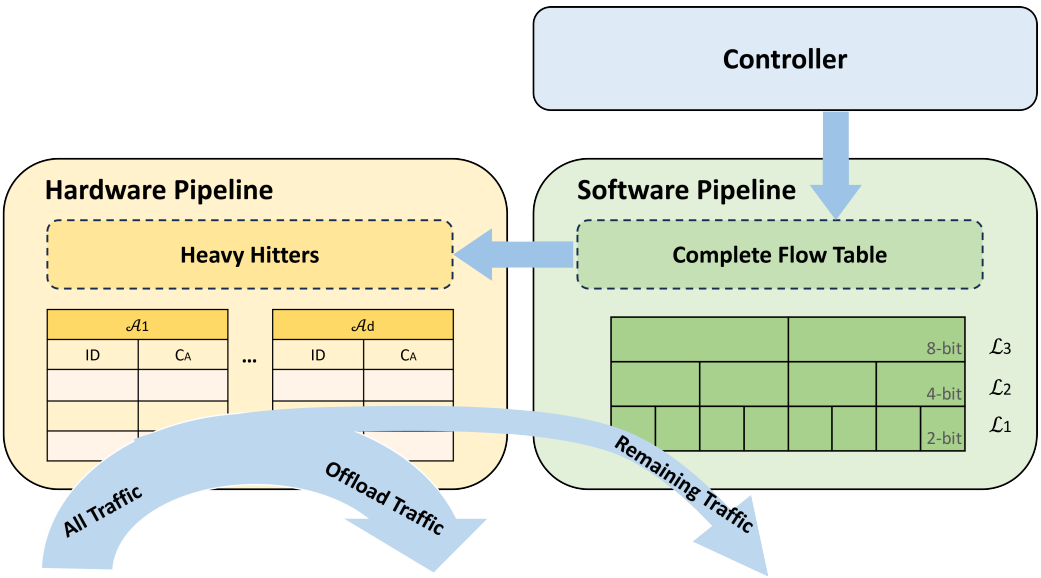

Figure 1. Data structure and work flow of ComPipe.

Table 2. Summary of notations.

Notation	Description
p, f	packet p with flow ID f
d	number of hash tables in hardware pipeline
K	Number of concurrent flows that can be accommodated in the hardware pipeline
$h_j(.)$	hash function of the jth table in hardware pipeline
$\mathcal{A}_j[i]$	the ith bucket in the jth table of a hardware pipeline
$\langle ID, C_A \rangle$	the two fields recorded in each bucket of hardware pipeline
w	number of 8-bit counters in software pipeline
$g_k(.)$	hash function of the kth array in software pipeline, where $k \in [1,3]$
$\mathcal{L}_k[t]$	the tth counter in the kth array of software pipeline, where $k \in [1,3]$

3.2. Data Structure and Algorithm Design of ComPipe

In this section, we will detail the design aspects of ComPipe. In Section 3.2.1, we will discuss the underlying data structures of ComPipe. In Sections 3.2.2 and 3.2.3, we will respectively elaborate on how to insert and query flows within ComPipe.

3.2.1. Data Structure of ComPipe

As mentioned above, the data structure is divided into two main parts: the HP and SP, designated for recording heavy hitters and less heavy flows, respectively. The HP is composed of d hash tables $(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_d)$ connected in series, each employing an independent hash function $h_j(.)$. Each table contains $e = K/d$ buckets, with each bucket comprising two fields: a flow identifier (ID) and a flow count (C_A). Thus, the entire structure totals K buckets, designated for tracking the Top-K flows with the highest frequency. Here, the quantity K also approximately equals the total number of flows that the HP can accommodate. This layered design effectively addresses the dynamism of network flows and the imbalance of traffic volumes.

In the SP, ComPipe employs a Count-Min version of the Tower Sketch [30] (CM-Tower). The CM-Tower is structured into three counting arrays: $\mathcal{L}_1$ with $4 \times w$ 2-bit counters, $\mathcal{L}_2$ with $2 \times w$ 4-bit counters, and $\mathcal{L}_3$ with w 8-bit counters. Each array is paired with its respective hash function $g_k(.)$ for $1 \leq k \leq 3$. The counters are designed to solely record the count of an item, escalating to overflow at thresholds of 3, 15, and 255 for the 2-bit, 4-bit, and 8-bit counters, respectively. Here, $\mathcal{L}_k[t]$ represents the tth counter in the kth array. The insertion process in CM-Tower mirrors that in CM-Sketch: the flow ID is first extracted, followed by the application of d hash functions to identify one counter in each array. These counters, termed as d hash counters, are then incremented. For querying, the smallest non-overflow counter value is retrieved from the d hash counters; if all have overflowed, the query defaults to 255. This architecture allows CM-Tower to adeptly record smaller flows and, through its varied bit-width counters, significantly improves upon the number of counters and reduces hash collisions, a notable advancement over the CM-Sketch.

3.2.2. Insertion of ComPipe

The pseudocode for ComPipe's insertion process is delineated in Algorithm 1. Initially, all flow IDs in the measurement structures are set to *NULL*, and their corresponding count fields are initialized to 0. Given a packet p with flow ID f, the ID_{min} and C_{min} fields in p's metadata are first set to *NULL* and *INT_MAX*, respectively. Then, by computing the hash function $h_j(f)(1 \leq j \leq d, 1 \leq h_j(f) \leq e)$, it is mapped to each bucket $\mathcal{A}_j[h_j(f)]$ in the HP. During the sequential access of each bucket, based on the information of $\mathcal{A}_j[h_j(f)]$, there are three possible cases:

Case 1: If $\mathcal{A}_j[h_j(f)].ID = NULL$, indicating an unoccupied bucket, ComPipe inserts f by setting $\mathcal{A}_j[h_j(f)].ID = f$ and $\mathcal{A}_j[h_j(f)].C_A = 1$ (lines 3–5 in Algorithm 1).

Case 2: If $\mathcal{A}_j[h_j(f)].ID = f$, implying f is already in the bucket, ComPipe increments $\mathcal{A}_j[h_j(f)].C_A$ by 1 (lines 6–8 in Algorithm 1).

Case 3: If $\mathcal{A}_j[h_j(f)].ID \neq f$, this indicates a hash collision. ComPipe then checks if f is stored in subsequent hash tables or if an empty slot is available. It tracks the minimum counter value and its ID in the metadata's C_A field and continues backward searching (lines 9–13 in Algorithm 1).

If the insertion process fails to complete after traversing all d hash tables, indicating a lack of available buckets in $\mathcal{A}j[h_j(f)]$, ComPipe then forwards the packet to the software pipeline for further processing. Here, it retrieves the records corresponding to the flow ID f. In the event that the measurement value of f exceeds the frequency of the candidate light flow, ComPipe initiates an eviction operation. This involves evicting $\langle p.ID_{min}, p.C_{min} \rangle$ and replacing it with $\langle f, 1 \rangle$. Subsequently, $p.ID_{min}$ is inserted into the CM-Tower, where the associated counters are incremented by $p.C_{min}$. Furthermore, the eviction operation also includes the crucial step of offloading the flow table of the newly identified heavy hitter to

the hardware. This enables all subsequent packets of this flow to be efficiently measured and forwarded along the hardware's fast path.

Algorithm 1: Insertion of ComPipe.

Input: Incoming packet p with flow ID f

Function `Insert(`p`)`**:**
- $p.ID_{min} \leftarrow NULL$; $p.C_{min} \leftarrow INT\_MAX$;
- **for** $j = 1$ *to* d **do**
  - **if** $\mathcal{A}_j[h_j(f)]$ *is an empty slot* **then**
    - $\langle \mathcal{A}_j[h_j(f)].ID, \mathcal{A}_j[h_j(f)].C_A \rangle \leftarrow \langle f, 1 \rangle$;
    - **return**
  - **else if** $\mathcal{A}_j[h_j(f)].ID = f$ **then**
    - $A_j[h_j(f)].C_A \leftarrow A_j[h_j(f)].C_A + 1$;
    - **return**
  - **else if** $\mathcal{A}_j[h_j(f)].ID \neq f$ **then**
    - **if** $p.C_{min} > \mathcal{A}_j[h_j(f)].C_A$ **then**
      - $p.C_{min} \leftarrow \mathcal{A}_j[h_j(f)].C_A$;
    - **end**
  - **end**
- **end**
- **if** `Query` $(f) + 1 >$ `Query` $(p.ID_{min}) + p.C_{min}$ **then**
  - ▷ See Section 3.2.3 for Query procedure
  - Evict $\langle p.ID_{min}, p.C_{min} \rangle$, replace with $\langle f, 1 \rangle$
  - Insert $\langle p.ID_{min}, p.C_{min} \rangle$ to $\mathcal{L}_k$
- **else**
  - Insert $\langle f, 1 \rangle$ to $\mathcal{L}_k$
- **end**

return

In Figure 2, we demonstrate three example cases of insertion within this structure.

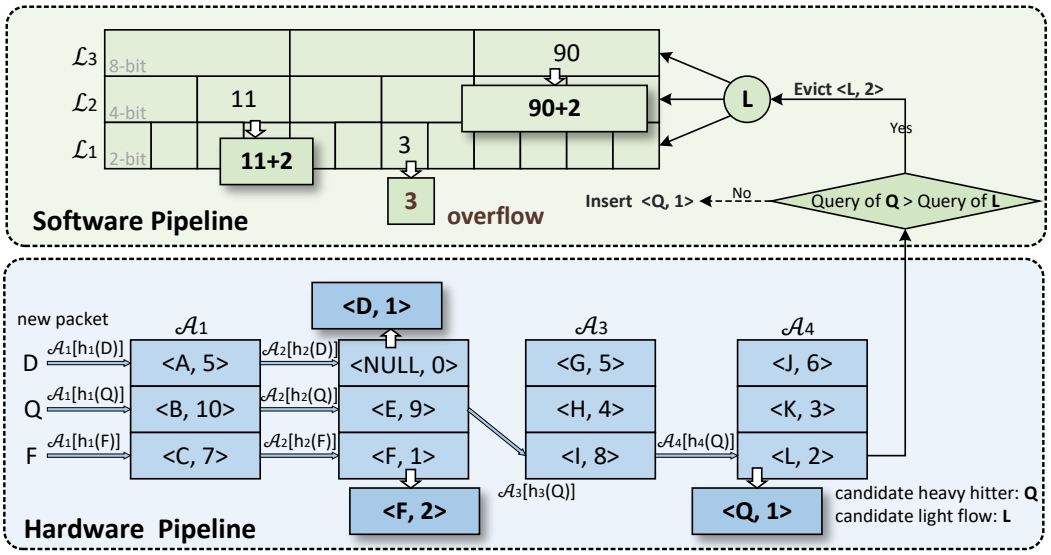

Figure 2. Insert examples of ComPipe.

(1) When inserting flow D, it is not matched in the first hash table, but in the second, it hashes to an unoccupied first bucket, allowing for the direct insertion of $\langle D, 1 \rangle$.

(2) For the insertion of flow F, it also finds no match in the first hash table. However, in the second table, F hashes to the second bucket where a matching element exists, prompting an increment in the corresponding C_A counter.

(3) Finally, when inserting flow Q, it demonstrates a scenario where no match is found across the four-level hash table. Here, the flow with the smallest counter value along the matching path is selected as a candidate light flow (in this example, L). In the SP, it is established that Q has a higher count than L, leading to the replacement of $\langle L, 2\rangle$ with $\langle Q, 1\rangle$. Consequently, Q's entry is transferred to the hardware pipeline, ensuring that subsequent packets from Q bypass the software, thereby reducing the hardware–software communication overhead.

This operation exemplifies the dynamic nature of the structure, which consistently removes the key with the smallest current count to insert a new one. This approach effectively ensures that only the Top-K flows with the highest counts are retained in the structure.

3.2.3. Query of ComPipe

To query a flow f in ComPipe, the process begins with an examination of the heavy part $\mathcal{A}_j$ in the HP. If f corresponds to a cell in $\mathcal{A}_j[h_j(f)]$, ComPipe reports the sum of the associated count C_A in HP and the count value obtained from querying f in the SP. In cases where f does not match any cell in HP, the query response is solely based on the count value identified in SP.

In the SP, the querying mechanism involves returning the smallest value among the counters $\mathcal{L}_k[g_k(f)]$ that f hashes to. Here, any counter that has overflowed is treated as having a value of $+\infty$. Should all counters associated with f in SP reach overflow, the query defaults to returning a value of 255.

3.3. Mathematical Analysis

The accuracy of flow measurements in HP is high, as the flow table entries precisely measure the matched flows. Consequently, the primary source of measurement error in ComPipe arises from the SP component. Without loss of generality, assume that there are N counting arrays $\mathcal{L}_k (1 \leqslant k \leqslant N)$, and the range of each counter array $[0, w \cdot 2^{k-1} - 1]$. Defining δ_k as $2^k - 1$, it follows that for any actual flow size a_i, the condition $\delta_{k-1} \leqslant a_i \leqslant \delta_k$ holds true. This condition indicates the range within which the actual flow size a_i falls.

According to CM-Tower [30], for $\forall \epsilon > 0$ and $\epsilon \cdot w \cdot 2^{k-1} > 1$, the estimated value $\hat{a_i}$ of f_i is bounded by:

$$\Pr\{\hat{a_i} \leqslant a_i + \epsilon \cdot a\} \geqslant 1 - \prod_{k=1}^{N} \left(\frac{1}{\epsilon \cdot w \cdot 2^{k-1}} \right) \tag{1}$$

where $a = \sum_{i=1}^{m} a_i$ is the total actual size of m flows measured by the software pipeline.

Based on the upper bound of the measurement error of CM-Tower, the upper bound of the measurement error of ComPipe can be derived as:

$$\Pr\{\hat{a_i} \leqslant a_i + \epsilon(a - a_r)\} \geqslant 1 - \prod_{k=1}^{N} \left(\frac{1}{\epsilon \cdot w \cdot 2^{k-1}} \right) \tag{2}$$

where $a_r = \sum_{i=1}^{d} \sum_{j=1}^{e} A_i[j].C_A$ is the total count of the counters in the HP. Given the skewness principle of network traffic, $a - a_r$, the difference between the total actual size and the HP count, is typically much smaller than a itself. Consequently, ComPipe achieves a more stringent error bound compared to CM-Tower. Moreover, according to the formula, a larger w value reduces errors for SP.

3.4. Discussion

Our approach, while inspired by HashPipe [19], introduces several key enhancements. First, HashPipe employs an 'Always insert in the first stage' strategy, ensuring consideration

for every key. However, this can lead to redundant IDs across different tables in the pipeline, inefficiently utilizing hardware memory. In contrast, our method, recognizing the inevitability of directing the first packet of a flow to the software for forwarding rules, opts for a different strategy. We allow packets to traverse all pipeline stages to identify candidate light flows, deferring the replacement or forwarding decision until the packet's return from the software. Additionally, previous studies have pointed out HashPipe's limitations in terms of compatibility with the current RMT programmable switch architecture, especially its challenges with in-stage branching and constraints on accessing memory in a single stage [21]. These limitations render it unsuitable for direct application to the commercial P4 pipeline. The implementation of ComPipe on commercial programmable hardware is detailed in Section 5.1.

In the software pipeline, when packets are forwarded from the hardware, there are generally two typical processing approaches. The first method involves deleting the count value in the software pipeline before inserting it in the hardware. This approach reduces collisions between large and small flows in the CM-Tower but may underestimate the count for other elements mapped to the same position, violating the one-side error guarantee of the measurement system. An alternative approach, suggested by OneSketch [34], merges the counts of light flows into the heavy ones while retaining a backup in the light part. However, this strategy introduces complexity in insertion and can decrease throughput due to the necessity of duplicate control strategies. Our approach contrasts with these by replacing $< ID_{old}, C_A >$ with $< ID_{new}, 1 >$ in the hardware pipeline. This method upholds the one-side error guarantee of the measurement and reduces comparison times relative to OneSketch. The trade-off, however, is that heavy-hitter queries must now also include the software pipeline.

4. Multiple Measurement Application

4.1. Flow Size Estimation

ComPipe efficiently measures any flow f. As highlighted in Section 3, when f is identified as a heavy hitter in HP, its primary measurement occurs through HP. Additionally, as flow f may have prior measurements in the SP, a combined HP+SP approach is necessary for accurately representing the measurement value of any flow f.

4.2. Heavy-Hitter Detection

For detecting heavy hitters, we utilize only the flow IDs recorded in the HP as feasible candidates for heavy hitters. By summing their count values with those from the SP, flows exceeding a certain threshold are identified as heavy hitters. Alternatively, similar to the flow size estimation process, flow sizes can be obtained from the HP+SP, reporting those surpassing a predefined threshold as significant. This approach enhances the accuracy in identifying important flows.

4.3. Flow Size Distribution

The distribution for various flows is directly queried from the HP, while the MRAC algorithm [35] estimates small flow size distributions in SP. The overall distribution is then obtained by combining these two results.

4.4. Heavy Change Detection Estimation

ComPipe facilitates heavy-change detection by constructing snapshots of heavy hitters for two adjacent time windows. Heavy changes are marked when either a substantial decrease in flow volume occurs between the windows exceeding a threshold, or a new heavy hitter emerges in the latter window.

4.5. Cardinality Estimation

Cardinality estimation in ComPipe aggregates the count of distinct flows in both HP and SP. While HP directly reads the C_A in the hash table, SP employs the linear counting

method [36] to calculate minor flow volumes. The cardinality is then estimated by summing these counts.

5. Implementations

5.1. Hardware Pipeline Implementations

We implemented ComPipe using P4 and compiled it on the programmable Tofino [37] ASIC to evaluate the resource consumption on the hardware pipeline. Considering the limited resources of ASICs and the need to support various network functions, ComPipe's resource efficiency is pivotal for its practical application in real-world scenarios.

To realize the hash tables in ComPipe, we employed the register tables available in commercial switches and utilized the Stateful Arithmetic and Logical Units (SALUs) in each stage to check and update entries within these register tables. This approach has the advantage of allowing modifications to these tables entirely within the data plane, without the need to involve the host CPU. However, SALUs are subject to resource limitations: branching operations are expensive, and the RMT pipeline limits the number of branching operations per stage (to a maximum of 2). As the ComPipe, outlined in Algorithm 1, necessitates four branching operations, we redesigned the data structure to fit this constraint.

Our tailored ComPipe implementation is optimized for line-rate operations within P4 switches. As depicted in Figure 3, each hash table is split into two register tables across two stages, each 16 bits wide and of length w, to store flow IDs and their counts. In the first stage, we determine whether $\mathcal{A}_1[h_1(f)].ID$ is equal or NULL, and if not, we proceed to the second stage for counter comparison and packet metadata update. Utilizing the RMT parallel matching capabilities, we simultaneously ascertain $\mathcal{A}_2[h_2(f)].ID$ in the second stage. This approach effectively minimizes the number of required pipeline stages, limiting it to at most $d + 2$ for a d-level hash implementation.

Table \ Stage	1	2	3	4	5
1	M_0	A_0/A_1			
2		M_1	A_0/A_1		
3			M_2	A_0/A_1	
4				M_3	A_0/A_1

Figure 3. Tailored ComPipe implementation for RMT architecture.

Prior to forwarding packets to the SP, ComPipe necessitates the insertion of specific metadata into the packet header. These metadata include the packet's ingress port number, the current and candidate flow IDs, candidate flow count, and operation type. These fields enable the SP to make informed decisions and operations. The operation type is encoded; for instance, "00" signifies a new heavy hitter inserted into the HP, requiring the SP to offload its flow table without recording frequency, while "01" indicates a software-driven decision to offload flow tables based on ComPipe's algorithmic comparison between the current and candidate flow.

Given the hardware limitations of ASICs and insights from our experimental analysis in Section 6.3, we recommend employing 4 hash tables in the hardware pipeline, necessitating only 4 + 2 = 6 pipeline stages. For $d = 4$ and $K = 1024$, we calculated the resource consumption of ComPipe as shown in Table 3. In the table, we also list a switch.p4 program [38] that implements typical top-of-rack (TOR) switch functions, such as L2/L3 forwarding, VLAN, QoS, etc.

Table 3. Hardware resources used by ComPipe.

Resource [1]	Percentage		
	switch.p4	**ComPipe**	**switch.p4 + ComPipe**
Hash Bits	30.5%	8.9%	34.3%
SRAM	66.7%	15.0%	78.5%
Map RAM	23.1%	6.7%	25.5%
TCAM	45.5%	3.3%	46.5%
Match Crossbar	39.9%	15.2%	42.3%
Stateful ALUs	37.5%	40.0%	65.4%
VLIW Actions	22.7%	6.25%	24.0%
Stages	100.0%	50%	100.0%

[1] SRAM: Static Random Access Memory; TCAM: Ternary Content Addressable Memory; VLIW: Very Long Instruction Word; ALUs: Arithmetic Logic Units.

Our evaluation revealed that ComPipe predominantly utilizes SRAM and Stateful ALUs, accounting for 15.0% and 40.0% of total resources, respectively. Notably, adding extra logic to the ASIC does not genuinely affect the ASIC processing throughput, as long as it can be accommodated within the ASIC resource constraints. Therefore, we can integrate ComPipe into the switch ASIC for packet processing at the line rate.

5.2. Software Pipeline Implementations

We have integrated the ComPipe software pipeline into a prototype software switch, a crucial component in virtualized software across contemporary public and private cloud environments. Numerous open-source solutions exist for software switch implementation; our work is based on the POFSwitch [15] on top of DPDK [39]. This particular version leverages kernel bypass technology to avoid frequent hardware interrupts and extraneous data copying, thereby allowing direct access to driver interfaces in user space for efficient packet handling. This design ensures high throughput and low latency in network packet processing and forwarding.

To scale the algorithm's performance, we adopted a multi-core approach. Utilizing the Receive-Side Scaling (RSS) algorithm and the NIC multi-queue technology, our system computes hash values from packet five-tuples for packet dispatch, facilitating balanced load distribution across various forwarding cores. Each core is equipped with a CM-Tower data structure as detailed in Section 3.2.1 and adheres to the principles of Algorithm 1 for insertion. This efficient design not only elevates the processing capability of the software pipeline but also aligns with the high-performance network processing requirements of modern network infrastructures.

6. Evaluation

6.1. Experimental Setup

6.1.1. Datasets

(1) MAWI Dataset. We use one-minute traces on ISP from the anonymized dataset MAWI [40] for our long-term testing. There are around 28 million packets and 20 million flows in this dataset.

(2) IP Trace Dataset. We use the Anonymized Internet Traces 2018 Dataset published by CAIDA [41], derived from 10G traces on the Equinix NYC monitor. We divide one hour's trace into 600 different ten-second datasets for separate testing. Each minute's trace contains 1.5–1.8 million packets from 0.9–1.1 million flows. Each item is identified by its source IP address.

(3) Web Dataset. We use the WebDocs dataset [42], which is a spidered collection of web HTML documents. The whole collection contains about 70 million items in this dataset.

(4) Data Center Dataset. We use data from UNI2 data centers studied in the IMC 2010 paper [43]. The trace contains 47 million anonymized packets from 18 million flows.

6.1.2. Evaluation Metrics

(1) Average Absolute Error (AAE). AAE is calculated as $\text{AAE} = \frac{1}{|\Psi|}\sum_{f_i' \in \Psi} |s_i' - s_j|$, where Ψ represents the set of flows, s_i' is the true size of flow f_i', and s_i is its estimated size. This metric measures the average magnitude of errors in the frequency estimates.

(2) Average Relative Error (ARE). ARE is given by $\text{ARE} = \frac{1}{|\Psi|}\sum_{f_i' \in \Psi} \frac{|s_i' - s_i|}{s_i}$, where Ψ denotes the set of flows, s_i' is the actual size of flow f_i', and s_i is the estimated size. This statistic assesses the average size of errors relative to the true frequencies.

(3) F1 Score. The F1 Score is defined as $F1 = \frac{2 \cdot PR \cdot RR}{PR + RR}$, where PR is the Precision Rate and RR is the Recall Rate. This formula calculates the harmonic mean of precision and recall, offering a composite reflection of these two metrics.

(4) Throughput. The update performance of the data structure is quantified by the number of packets processed per second (PPS). These metrics are tested in Testbed, and improvements are assessed through multi-core acceleration methods.

6.1.3. Simulation Setup

ComPipe was implemented using C++ language. The system employs 32-bit Bob Hash [44] with distinct seeds for different hash tables. Our experiments were conducted on a high-performance server, featuring dual 16-core Intel Xeon Silver 4208 CPUs @ 2.10 GHz and 128 GB of RAM. This server boasts a three-tier cache hierarchy, with each core equipped with 32KB L1 data and instruction caches, a dedicated 1024 KB L2 cache per core, and a shared 11 MB L3 cache.

For the simulation experiments, we initially assessed the performance of Flow Size Estimation (FSE) and Heavy-Hitter Detection (HHD) tasks, with detailed implementation aspects discussed in Section 6.2. Additionally, we compared ComPipe against nine advanced traffic measurement algorithms: Space-Saving (SS) [20], Unbiased Space-Saving (USS) [25], Count-min Sketch (CM) [28], Conservative Update Sketch (CU) [28], Augmented sketch (Augmented) [26], Elastic Sketch (Elastic) [10], SALSA [29], MV-Sketch (MV) [32] and LogLog Filter (LLF) [31]. For the ComPipe setup, we fixed the parameter d at 4 and adjusted e and w to suit different memory sizes, maintaining a 1:4 memory ratio between the HP and SP. This ratio was also used for algorithms with similar structures, such as Elastic Sketch, where we utilized eight-bit counters for positive and negative votes. For the light part of Elastic, CMS, CUS, SALSA, and MVS, three hash functions were used, following the recommendations in [45]. SS and USS each allocated 100 Bytes per bucket, scaled according to available memory. For LLF, we adopted the CU strategy, allocating 75% of the memory, with each register set to four bits, three hash functions, and a threshold of five.

6.1.4. Testbed Setup

Figure 4 presents the hardware–software data plane testbed employed in our study. The HP comprises a Wedge 100BF-65X programmable P4 switch equipped with an Intel Tofino chip. The data plane program, developed in $P4_{16}$, is installed on this chip. The SP operates on a Linux system, powered by an Intel Xeon Silver 4216 CPU @ 2.10 GHz, equipped with 64 GB RAM and an Intel E810-C 100G NIC, configured in 4 $\times$ 25 G mode. The system runs on CentOS 7.9.2009, with the software compiled using the -O3 optimization option. ComPipe is developed based on POFSwitch and DPDK 22.11. A separate server, mirroring the hardware specifications of the SP, hosts the controller. This controller manages the SP through the relevant southbound protocol. For traffic generation and analysis, the Keysight Ixia XGS12 network test platform is employed, enabling the assessment of forwarding throughput for both the hardware and software components.

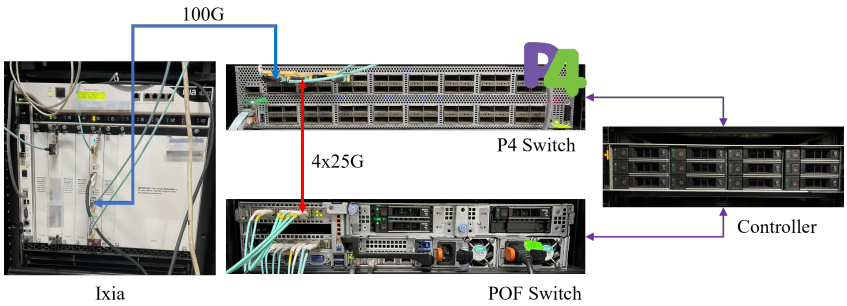

Figure 4. The testbed topology of the experiments.

6.2. Experiments on Measurement Tasks

In the following sections, we present the experimental results of the two most representative measurement tasks: FSE and HHD. We recognize that memory size is a critical parameter for all measurement schemes, as it impacts both measurement accuracy and performance. To evaluate the effect of memory size on all approaches, experiments were conducted under various memory size configurations.

6.2.1. Flow Size Estimation

As demonstrated in Figures 5 and 6, ComPipe significantly outperforms the other nine measurement algorithms across various memory scales. By offloading heavy hitters to the HP, as elucidated in our mathematical analysis, ComPipe effectively reduces counting errors and hash collisions compared to purely sketch-based algorithms. In contrast to algorithms like Elastic that utilize multi-level hash structures, ComPipe reduces the memory resources required for each bucket on the HP. Simultaneously, in the SP, with the advantage of the CM-Tower, ComPipe can accommodate more counters, thus notably enhancing measurement accuracy, especially in low-memory environments.

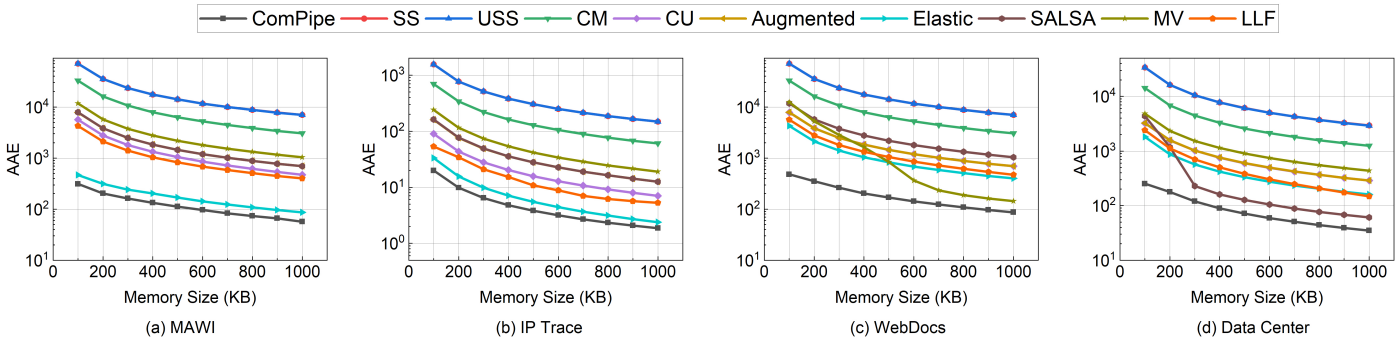

Figure 5. AAE of frequency estimation for flow size.

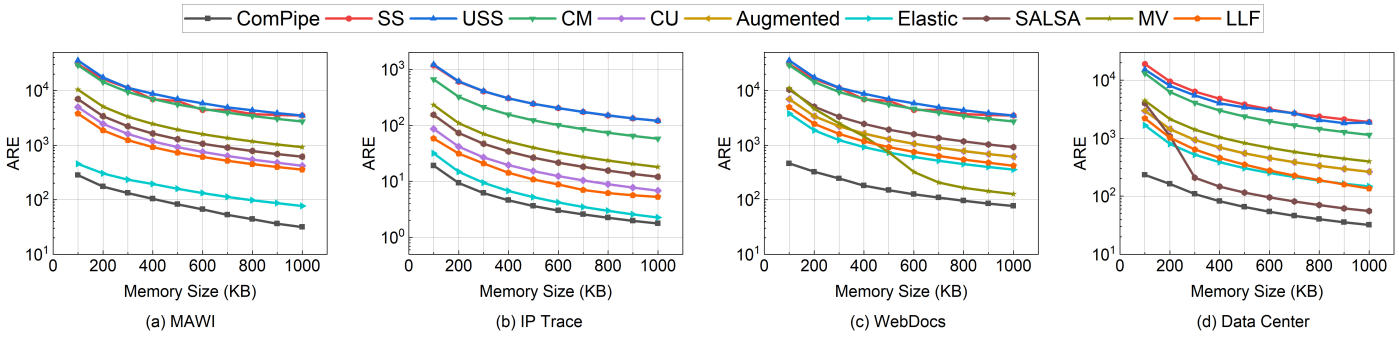

Figure 6. ARE of frequency estimation for flow size.

When evaluating the AAE, ComPipe's accuracy surpasses that of the nine comparative algorithms across four distinct datasets. However, it is notable that ComPipe's AAE is approximately 3.9 times lower than that of Elastic Sketch, the best-performing algorithm in this category. Additionally, ComPipe's AAE is about 52.8 and 10.1 times lower than traditional algorithms like CM and CU, respectively.

As for the ARE, ComPipe demonstrates substantial superiority over the nine comparison algorithms across these datasets. The ARE values for ComPipe are markedly lower, being 70.2, 73.1, 53.8, 10.3, 12.5, 3.9, 13.5, 17.4, and 7.2 times lower than the other algorithms, respectively. This indicates a significant enhancement in accuracy, showcasing ComPipe's effectiveness in diverse network environments.

6.2.2. Heavy-Hitters Detection

The threshold for identifying heavy hitters in ComPipe is set at 2×10^{-5} of the total flow size. ComPipe exclusively considers flows within the HP as heavy-hitter candidates, integrating these with count values from the SP to ascertain the overall flow frequency. Owing to its multi-level hash structure and sophisticated eviction algorithms, ComPipe efficiently retains large flows within the HP, dynamically accommodating high-volume flows as traffic patterns evolve. Unlike Elastic, which directly executes eviction decisions within the hardware data plane, ComPipe defers such judgments until after consulting the SP. This approach enables ComPipe to detect and manage heavy hitters more accurately and dynamically in real-time, ensuring that heavy hitters are maintained in the hardware for longer durations.

As illustrated in Figures 7–9, ComPipe's performance on IP Trace datasets is slightly inferior to LLF and Elastic Sketch when memory sizes exceeded 800 KB. However, in MAWI, WebDocs and data center scenarios, it surpasses the other nine algorithms, reporting heavy hitters more accurately.

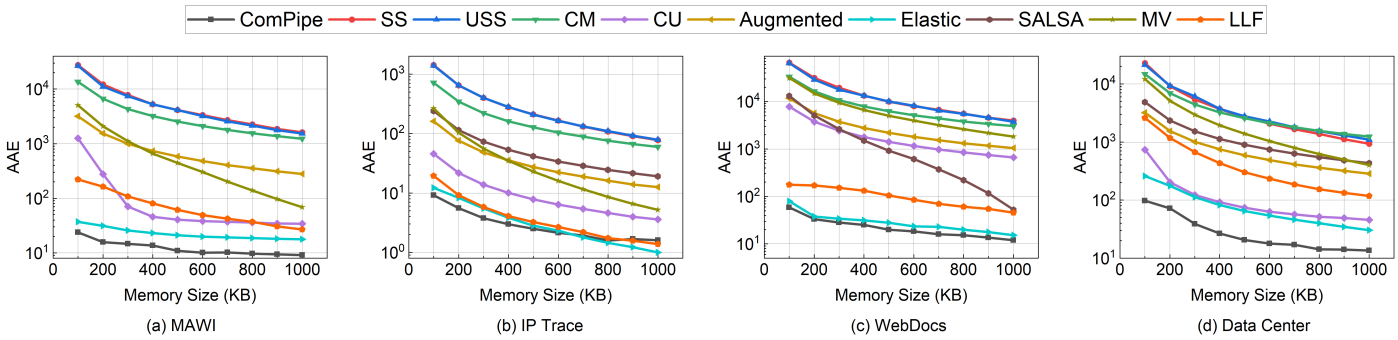

Figure 7. AAE of frequency estimation for heavy hitters.

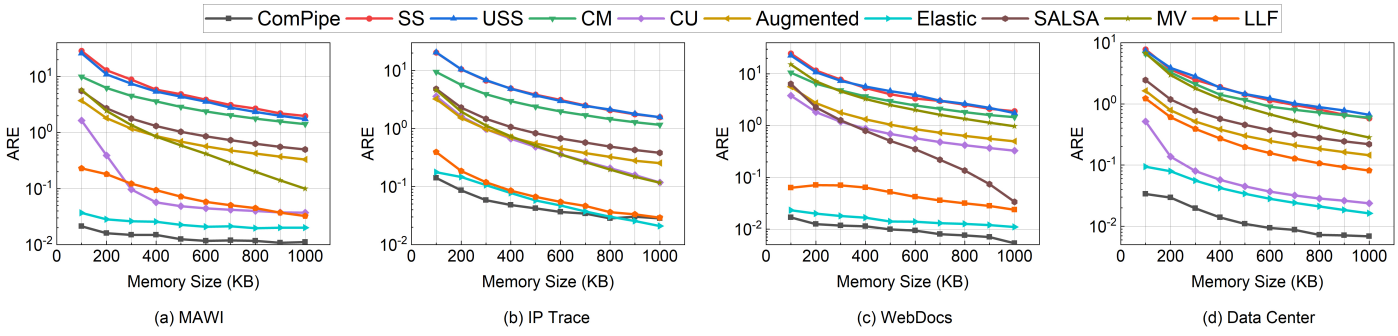

Figure 8. ARE of frequency estimation for heavy hitters.

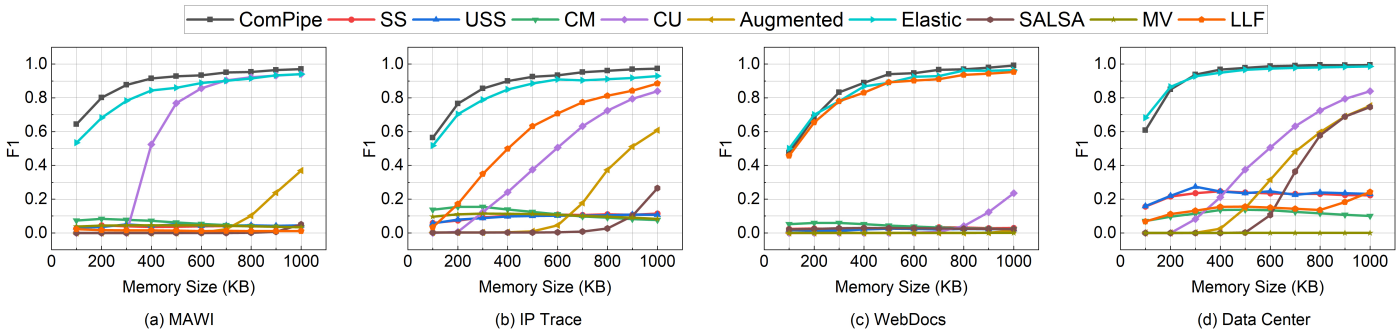

Figure 9. F1 Score of frequency estimation for heavy hitters.

Regarding AAE, the performance of ComPipe substantially exceeds that of the nine comparison algorithms across the four datasets. On average, the ComPipe AAE is lower by factors of 374.6, 364.2, 217.9, 28.1, 61.7, 1.8, 40.8, 128.1 and 7.6 compared to each of these algorithms.

In terms of ARE, ComPipe maintains its significant lead over the nine algorithms, with average ARE values lower by factors of 366.2, 347.8, 207.2, 36.5, 70.6, 1.9, 76.5, 153.9 and 8.9, respectively.

Concerning the F1 Score, which balances precision and recall, ComPipe's performance is notably higher. Its F1 Scores average 79.6%, 79.6%, 81.1%, 52.6%, 75.5%, 3.3%, 81.2%, 85.7%, and 50.3% more than the corresponding scores of the nine comparative algorithms.

6.3. Experiments on Parameter Settings

We evaluated how varying levels of hash (denoted as d) within a multi-level hash structure affect the accuracy of HHD tasks. Using the IP Trace dataset, we experimented with d values ranging from 4 to 64, keeping other parameters constant, to gauge its impact on measurement precision. As depicted in Figure 10, we observed that at lower memory configurations, increasing the value of d tends to enhance measurement accuracy. Conversely, in scenarios with larger memory sizes, a smaller d value appears more advantageous for accuracy.

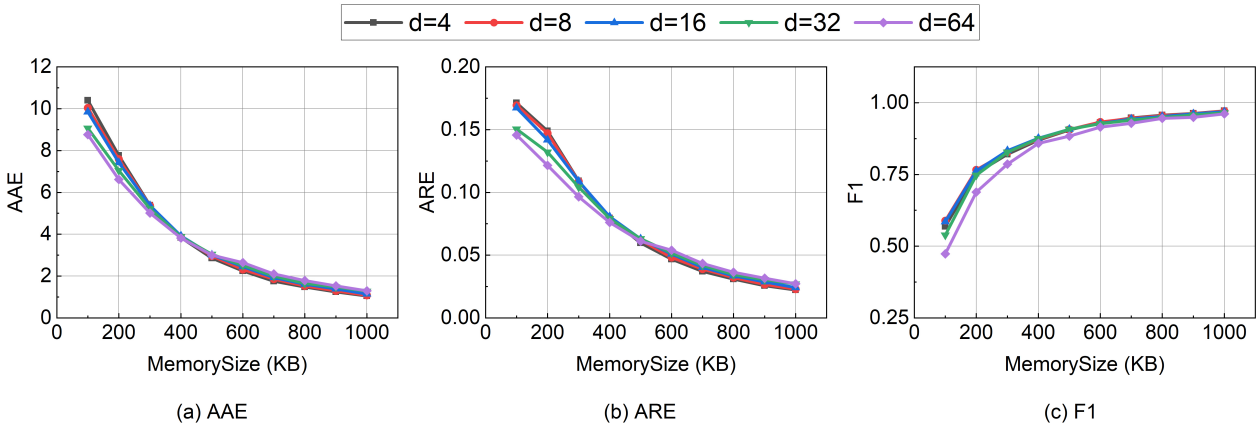

Figure 10. Parameter settings of frequency estimation for heavy hitters.

A key observation for hardware implementations is that while higher d values do not diminish the processing throughput, they do require more pipeline stages. This factor becomes critical when integrating ComPipe with other network applications in a hardware environment, as it can pose challenges in optimal resource allocation given the hardware limitations. Therefore, it is essential to strike a balance between accuracy, memory efficiency, and hardware resource utilization in designing multi-level hash structures. This balance is necessary to ensure adaptability to diverse network conditions and application requirements.

6.4. Experiments on Throughput

We analyzed the throughput performance of ComPipe, CU, and CM within a consistent hardware setup. As depicted in Figure 11, we increased the number of CPU cores allocated to the switch and recorded the forwarding rates. For context, we also measured the throughput of the original POFSwitch. The findings indicate that with a single CPU core, the ComPipe throughput is on par with that of CM and CU. However, due to the use of two-bit and four-bit counters, which necessitate arithmetic operations like shifting, its performance demonstrated a downward trend as the CPU cores increased. Despite an average decrease of about 9.4% compared to CU and about 20.4% compared to CM, this performance drop is offset by the significant increase in accuracy as previously mentioned, achieving improvements by factors of 52.8 and 10.1, respectively.

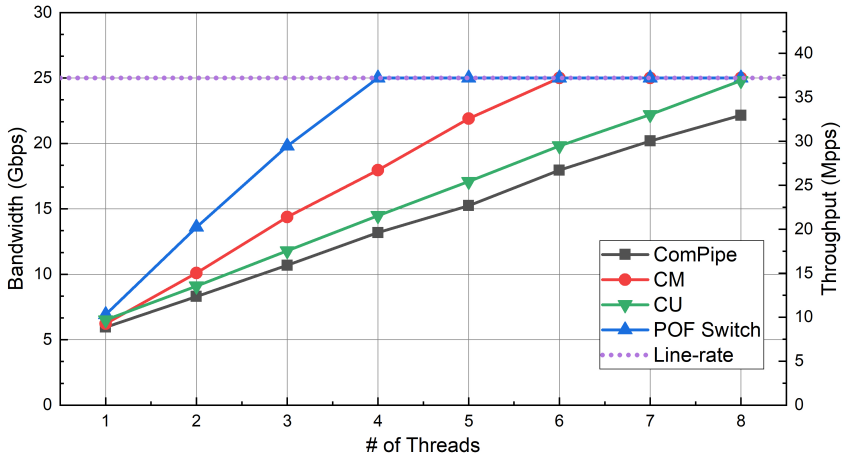

Figure 11. Throughput.

Furthermore, under the real-world workloads, ComPipe's distinct software–hardware separated measurement approach means that only about 20–30% of the total traffic volume is processed by the software switch. This results in ComPipe surpassing pure software-based forwarding solutions in terms of processing efficiency, thereby necessitating fewer CPU cores for comparable performance.

7. Conclusions

This study delves into the challenges of flow placement and measurement within programmable composite pipelines. To achieve an algorithm design that is both precise and efficient, we introduced the ComPipe scheme, which integrates a unique collaborative mechanism of hardware and software pipelines. In the hardware pipeline, we established a multi-level hash structure for heavy-hitter detection. This method enables the real-time, reliable, and efficient identification of heavy hitters. Subsequently, in the software pipeline, we employed a network flow recording technique based on the sketch algorithm, coupled with multi-core processing to enhance the overall forwarding capacity of the software pipeline. Following this, we successfully implemented ComPipe on hardware and software platforms and supported five typical traffic measurement tasks. To evaluate ComPipe's performance, we conducted extensive experiments, comparing it with current state-of-the-art traffic measurement solutions. The results demonstrate that ComPipe achieves higher measurement accuracy while reducing resource consumption, thereby confirming its performance. ComPipe extends the capabilities of the data plane, enabling it to handle large-scale table entries while maintaining high performance.

Author Contributions: Formal analysis, D.R. and X.C.; Methodology, D.R., X.C. and L.S.; Project administration, L.S.; Software, D.R.; Supervision, X.C.; Writing—original draft, D.R.; Writing—review and editing, D.R., X.C. and L.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work is funded by Strategic Priority Research Program of Chinese Academy of Sciences: Research on Information Collaborative Service and Data Sharing Technology (Project No. XDA031050100).

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Acknowledgments: The authors would also like to express thanks to Chentong Xiang at Microsoft as well as Zhiyuan Ling at the University of Chinese Academy of Sciences.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

HP	Hardware Pipeline
SP	Software Pipeline
VXLAN	Virtual eXtensible Local-Area Network
VM	Virtual Machine
FSE	Flow Size Estimation
HHD	Heavy-Hitter Detection
RMT	Reconfigurable Match-Action Table
ASIC	Application Specific Integrated Circuit
P4	Programming Protocol-Independent Packet Processors
POF	Protocol Oblivious
SRAM	Static Random Access memory
TCAM	Ternary Content Addressable Memory
VLIW	Very Long Instruction Word
ALUs	Arithmetic Logic Units
ISP	Internet Service Provider

References

1. Bosshart, P.; Gibb, G.; Kim, H.S.; Varghese, G.; McKeown, N.; Izzard, M.; Mujica, F.; Horowitz, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Comput. Commun. Rev.* **2013**, *43*, 99–110. [CrossRef]
2. Pfaff, B.; Pettit, J.; Koponen, T.; Jackson, E.; Zhou, A.; Rajahalme, J.; Gross, J.; Wang, A.; Stringer, J.; Shelar, P. The Design and Implementation of Open vSwitch. In Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), Oakland, CA, USA, 4–6 May 2015; pp. 117–130. [CrossRef]
3. Pan, T.; Yu, N.; Jia, C.; Pi, J.; Xu, L.; Qiao, Y.; Li, Z.; Liu, K.; Lu, J.; Lu, J. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In Proceedings of the 2021 ACM SIGCOMM 2021 Conference, New York, NY, USA, 23–27 August 2021; pp. 194–206. [CrossRef]
4. Qian, K.; Ma, S.; Miao, M.; Lu, J.; Zhang, T.; Wang, P.; Sun, C.; Ren, F. Flexgate: High-performance heterogeneous gateway in data centers. In Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019, New York, NY, USA, 17–18 August 2019; pp. 36–42. [CrossRef]
5. Wang, Y.; Li, D.; Lu, Y.; Wu, J.; Shao, H.; Wang, Y. Elixir: A High-performance and Low-cost Approach to Managing Hardware/Software Hybrid Flow Tables Considering Flow Burstiness. In Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), Renton, WA, USA, 4–6 April 2022; pp. 535–550.
6. Radhakrishnan, S.; Geng, Y.; Jeyakumar, V.; Kabbani, A.; Porter, G.; Vahdat, A. SENIC: Scalable NIC for end-host rate limiting. In Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), Seattle, WA, USA, 2–4 April 2014; pp. 475–488. [CrossRef]
7. Gao, P.; Xu, Y.; Chao, H.J. OVS-CAB: Efficient rule-caching for Open vSwitch hardware offloading. *Comput. Netw.* **2021**, *188*, 107844. [CrossRef]
8. Mimidis-Kentis, A.; Pilimon, A.; Soler, J.; Berger, M.; Ruepp, S. A Novel Algorithm for Flow-Rule Placement in SDN Switches. In Proceedings of the 2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft), Montreal, QC, Canada, 25–29 June 2018. [CrossRef]
9. Estan, C.; Varghese, G. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst. (TOCS)* **2003**, *21*, 270–313. [CrossRef]
10. Yang, T.; Jiang, J.; Liu, P.; Huang, Q.; Gong, J.; Zhou, Y.; Miao, R.; Li, X.; Uhlig, S. Elastic sketch: Adaptive and Fast Network-wide Measurements. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, Budapest, Hungary, 20–25 August 2018. [CrossRef]

11. Yang, T.; Zhang, H.; Li, J.; Gong, J.; Uhlig, S.; Chen, S.; Li, X. HeavyKeeper: An Accurate Algorithm for Finding Top-*k* Elephant Flows. *IEEE/ACM Trans. Netw.* **2019**, *27*, 1845–1858. [CrossRef]
12. McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.; Rexford, J.; Shenker, S.; Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Comput. Commun. Rev.* **2008**, *38*, 69–74. [CrossRef]
13. Li, S.; Hu, D.; Fang, W.; Ma, S.; Chen, C.; Huang, H.; Zhu, Z. Protocol Oblivious Forwarding (POF): Software-Defined Networking with Enhanced Programmability. *IEEE Netw.* **2017**, *31*, 58–66. [CrossRef]
14. P4Runtime Specification. Available online: https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html (accessed on 23 February 2024).
15. POFSwitch v1.0. Available online: https://github.com/ProtocolObliviousForwarding/pofswitch (accessed on 23 February 2024).
16. Sarrar, N.; Uhlig, S.; Feldmann, A.; Sherwood, R.; Huang, X. Leveraging Zipf's law for traffic offloading. *ACM SIGCOMM Comput. Commun. Rev.* **2012**, *42*, 16–22. [CrossRef]
17. Katta, N.; Alipourfard, O.; Rexford, J.; Walker, D. CacheFlow: Dependency-Aware Rule-Caching for Software-Defined Networks. In Proceedings of the Symposium on Software Defined Networking (Sdn) Research (Sosr'16), Santa Clara, CA, USA, 14–15 March 2016. [CrossRef]
18. Durner, R.; Kellerer, W. Network function offloading through classification of elephant flows. *IEEE Trans. Netw. Serv. Manag.* **2020**, *17*, 807–820. [CrossRef]
19. Sivaraman, V.; Narayana, S.; Rottenstreich, O.; Muthukrishnan, S.; Rexford, J. Heavy-hitter detection entirely in the data plane. In Proceedings of the Symposium on SDN Research, Santa Clara, CA, USA, 3–4 April 2017; pp. 164–176. [CrossRef]
20. Metwally, A.; Agrawal, D.; El Abbadi, A. Efficient computation of frequent and top-k elements in data streams. In Proceedings of the International Conference on Database Theory, Edinburgh, UK, 5–7 January 2005; pp. 398–412. [CrossRef]
21. Ben Basat, R.; Chen, X.; Einziger, G.; Rottenstreich, O. Designing Heavy-Hitter Detection Algorithms for Programmable Switches. *IEEE/ACM Trans. Netw.* **2020**, *28*, 1172–1185. [CrossRef]
22. Li, Y.; Miao, R.; Kim, C.; Yu, M. FlowRadar: A Better NetFlow for Data Centers. In Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), Santa Clara, CA, USA, 16–18 March 2016; pp. 311–324. [CrossRef]
23. Karp, R.M.; Shenker, S.; Papadimitriou, C.H. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst. (TODS)* **2003**, *28*, 51–55. [CrossRef]
24. Manku, G.S.; Motwani, R. Approximate frequency counts over data streams. In Proceedings of the VLDB'02: Proceedings of the 28th International Conference on Very Large Databases, Hong Kong SAR, China, 20–23 August 2007; pp. 346–357. [CrossRef]
25. Ting, D. Data sketches for disaggregated subset sum and frequent item estimation. In Proceedings of the 2018 International Conference on Management of Data, Houston, TX, USA, 10–15 June 2018; pp. 1129–1140. [CrossRef]
26. Roy, P.; Khan, A.; Alonso, G. Augmented sketch: Faster and more accurate stream processing. In Proceedings of the 2016 International Conference on Management of Data, San Francisco, CA, USA, 26 June 2016; pp. 1449–1463. [CrossRef]
27. Charikar, M.; Chen, K.; Farach-Colton, M. Finding frequent items in data streams. In Proceedings of the International Colloquium on Automata, Languages, and Programming, Malaga, Spain, 8–13 July 2002; pp. 693–703. [CrossRef]
28. Cormode, G.; Muthukrishnan, S. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms* **2005**, *55*, 58–75. [CrossRef]
29. Basat, R.B.; Einziger, G.; Mitzenmacher, M.; Vargaftik, S. SALSA: Self-adjusting lean streaming analytics. In Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE), Chania, Greece, 19–22 April 2021; pp. 864–875. [CrossRef]
30. Yang, K.; Long, S.; Shi, Q.; Li, Y.; Liu, Z.; Wu, Y.; Yang, T.; Jia, Z. Sketchint: Empowering int with towersketch for per-flow per-switch measurement. *IEEE Trans. Parallel Distrib. Syst.* **2023**, *34*, 2876–2894. [CrossRef]
31. Jia, P.; Wang, P.; Zhao, J.; Yuan, Y.; Tao, J.; Guan, X. LogLog Filter: Filtering Cold Items within a Large Range over High Speed Data Streams. In Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE), Chania, Greece, 19–22 April 2021. [CrossRef]
32. Tang, L.; Huang, Q.; Lee, P.P.C. MV-Sketch: A Fast and Compact Invertible Sketch for Heavy Flow Detection in Network Data Streams. In Proceedings of the IEEE INFOCOM 2019—IEEE Conference on Computer Communications, Paris, France, 29 April–2 May 2019. [CrossRef]
33. Yang, T.; Zhou, Y.; Jin, H.; Chen, S.; Li, X. Pyramid Sketch: A Sketch Framework for Frequency Estimation of Data Streams. *Proc. VLDB Endow.* **2017**, *10*, 1442–1453. [CrossRef]
34. Fan, Z.; Wang, R.; Cai, Y.; Zhang, R.; Yang, T.; Wu, Y.; Cui, B.; Uhlig, S. OneSketch: A Generic and Accurate Sketch for Data Streams. *IEEE Trans. Knowl. Data Eng.* **2023** , *35*, 12887–12901. [CrossRef]
35. Yoon, M.; Li, T.; Chen, S.; Peir, J.K. Fit a spread estimator in small memory. In Proceedings of the IEEE INFOCOM 2009, Rio de Janeiro, Brazil, 15–25 April 2009; pp. 504–512. [CrossRef]
36. Whang, K.Y.; Vander-Zanden, B.T.; Taylor, H.M. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst. (TODS)* **1990**, *15*, 208–229. [CrossRef]
37. Tofino Programmable Ethernet Switch ASIC. Available online: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html (accessed on 23 February 2024).
38. Open-Source p4 Implementation of Features Typical of an Advanced l2/l3 Switch. Available online: https://github.com/p4lang/switch (accessed on 23 February 2024).
39. Data Plane Development Kit. Available online: http://doc.dpdk.org/ (accessed on 23 February 2024).

40. MAWI Working Group Traffic Archive. Available online: http://mawi.wide.ad.jp/mawi/ (accessed on 23 February 2024).
41. The Caida Anonymized Internet Traces. Available online: http://www.caida.org/data/overview/ (accessed on 23 February 2024).
42. Frequent Itemset Mining Dataset Repository. Available online: http://fimi.uantwerpen.be/data/ (accessed on 23 February 2024).
43. Benson, T.; Akella, A.; Maltz, D.A. Network traffic characteristics of data centers in the wild. In Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, New York, NY, USA, 1–30 November 2010; pp. 267–280. [CrossRef]
44. The Source Code of Bob Hash. Available online: http://burtleburtle.net/bob/hash/evahash.html (accessed on 23 February 2024).
45. Goyal, A.; Daumé, H., III; Cormode, G. Sketch algorithms for estimating point queries in nlp. In Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, Jeju Island, Republic of Korea, 12–14 July 2012; pp. 1093–1103. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.