Journal of Information Processing Vol.26 590–600 (Sep. 2018)

[DOI: 10.2197/ipsjjip.26.590]

Regular Paper

Abnormal Traffic Detection Circuit with Real-time Cardinality Counter

Shuji Sannomiya[1,a)] Akira Sato[1,b)] Kenichi Yoshida[2,c)] Hiroaki Nishikawa[3,d)]

Received: November 10, 2017, Accepted: June 8, 2018

Abstract: To identify abnormal traffic such as P2P flows, DDoS attacks, and Internet worms, this paper discusses a circuit design to realize real-time abnormal traffic detection in broadband networks. Real-time counting of cardinality is the key feature of the circuit. Although our previous study showed that cardinality counting is effective for detecting various types of abnormal traffic, the slowness of DRAM access prevented us from deploying cardinality counting in backbone networks. To address the problem of DRAM access time, this paper proposes a new algorithm for cardinality counting. By changing the order of the cardinality counting process, the proposed algorithm enables parallel accesses of DRAM circuits, which hides the slow DRAM access time through a pipeline circuit. In addition, we propose a new hashing function that also hides the DRAM access problem. It partially replaces scattered addresses with successive addresses, in order to use a faster DRAM burst access. We also report the accuracy of the cardinality counting of the new algorithm, and describe the estimated processing performance based on a pipeline tact level circuit simulation. Our experimental results show that the use of the self-timed pipeline circuit can help realize cardinality counting at rates up to 100 Gbps.

Keywords: real-time traffic analysis, simple frequent-itemset-mining, self-timed pipeline

1. Introduction

The explosive growth of Internet traffic has led to data rates of 100 Gbps or higher in backbone networks. Unfortunately, abnormal traffic such as peer-to-peer (P2P) flows, distributed denial of service (DDoS) attacks, and Internet worms that disrupt smooth and safe communications are also increasing. To address this situation, network management techniques that detect and control the abnormal traffic mixed into vast numbers of packet streams are essential to ensure Internet reliability.

Previously, we proposed a cardinality counting method [1] to realize automatic abnormal traffic detection. This previously proposed method searches for header field combinations that are frequently found in the packet streams of abnormal traffic, as well as variations of those combinations. For instance, different destination IP addresses are combined with the same source IP address to transfer malicious packets. A worm-infected client that is searching for a new vulnerable server will send such packets.

Our previously proposed method uses a counting routine to keep track of these frequently found combinations, and to track the number of variations in the non-frequent parts of the combinations. This original counting procedure and its offsprings were collectively termed cardinality counting, because of its similarity to mathematical cardinality, which denotes the number of elements in a given set. Another advantage of this cardinality counting method over conventionally studied methods is its relatively small memory requirements. Our proposal also has a mechanism to count cardinality using a given small fixed-size memory area, to satisfy the design constraints of abnormal traffic detection systems [1].

The effectiveness of the cardinality counting method for actual Internet traffic was demonstrated using a software implementation [1]. However, this software implementation failed to achieve the processing capability required for traffic speeds of 100 Gbps or higher. This is because the approach requires dynamic random access memory (DRAM) for storing all the possible combinations of addresses, ports, and protocols. To store this information, the resulting access pattern becomes randomized. Thus, the long latency associated with random access on DRAMs becomes the bottleneck.

In this study, a self-timed pipeline is introduced to parallelize the memory access and reduce access latency. The self-timed pipeline is a global clock-less pipeline, whose stages process data asynchronously without any additional controls. In the deployment of the cardinality counting algorithm over the self-timed pipeline, the processing order of the original algorithm is changed. The original order results in a backward data transfer over the pipeline. Because this backward data transfer prevents the reduction of the number of DRAM accesses, we developed a new algorithm that does not require backward data transfer. Moreover, to increase the worst-case throughput for enhancing the robustness of the cardinality counting, the hash function of

[1] Faculty of Engineering, Information and Systems, University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
[2] Faculty of Business Sciences, University of Tsukuba, Bunkyo, Tokyo 112–0012, Japan
[3] Headquarters for International Industry-University Collaboration, University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
a) san@cs.tsukuba.ac.jp
b) akira@cc.tsukuba.ac.jp
c) yoshida@gssm.otsuka.tsukuba.ac.jp
d) nisikawa@cs.tsukuba.ac.jp

the original algorithm that calculates the DRAM addresses to be used is modified to produce successive addresses when applicable. The use of successive addresses enables the use of a burst DRAM access with a short latency [2].

The changed processing order explores the longest combinations first; in the original order, the shortest combinations were processed first. This difference may affect the counting result. In addition, the DRAM access latency varies at run-time owing to refreshing operations. This variation may become a bottleneck that degrades the cardinality counting throughput. In this study, the counting result is quantitatively evaluated by using actual traffic data to show the sufficiency of the counting accuracy. The throughput of the self-timed pipeline circuit is also evaluated by a pipeline tact level simulation, using an actual memory model provided by a memory vendor. Finally, the processing capability needed to achieve a throughput of 100 Gbps is discussed based on the simulation results.

The early stage of this study was presented in Refs. [2], [3]. Although Refs. [2], [3] reports basic ideas, this paper confirms the adequacy of those ideas by providing experimental results; these are described in Section 4. The remainder of this paper is organized as follows. Section 2 summarizes previous studies and Section 3 explains the circuit design. Section 4 reports experimental results and Section 5 summarizes our findings.

2. Related Work

Massive flows of abnormal traffic disrupt smooth and safe communications. Several methods for detecting such mass flows have been proposed or studied. In contrast with those methods, the simple frequent-itemset-mining method [1] focuses on the itemset of every packet and monitors its frequency and variety simultaneously. Itemset refers to a combination of packet header items such as the source IP address and the destination port number; the frequency of the itemset indicates the massiveness of the flow, and the variety characterizes the type of flow.

In this section, we discuss the advantages of the simple frequent-itemset-mining method by comparing it with other proposed methods. Next, we explain the process of simultaneously counting the frequency and variation using the cardinality counting algorithm. Finally, we discuss the self-timed pipeline circuit implementation in terms of the processing capability needed for analyzing the traffic at speeds of 100 Gbps or higher.

2.1 Abnormal Traffic Analysis Method

One major abnormal traffic is the DDoS attack because of the massiveness of its flow. For example, the response traffic against the fumbling requests generated by Internet worms searching for a vulnerable server appears as a mass flow from the target servers to a victim client.

Depending on the location at which attacks are to be detected, Beitollahi and Deconinck classified the defense methods against DDoS attacks into two types: victim server and router [4]. The device or program used for detecting the attacks in both of these methods is called a sensor. In the victim server type, the sensor is installed on all the servers to be protected. In the router type, the sensor is installed on the router above the servers to be protected. Obviously, the victim server type is more expensive because the number of sensors required scales with the number of servers to be protected; hence, we focus on the router type defense in our work.

In recent years, the volume of traffic in the Internet backbone and the subnetworks underneath the backbone has increased tremendously. It has been reported that the connection bandwidth between the Internet backbone and internet service providers (ISPs) is over 100 Gbps [5]. Hence, to protect the servers under an ISP, the sensor should be able to analyze traffic at speeds of 100 Gbps or higher.

Based on the reaction time, router type methods are further categorized into two groups: proactive and reactive [4]. Proactive methods aim to prevent the occurrence of DDoS attacks, while reactive methods come into effect after the DDoS attacks occur. Owing to safe side filtering, proactive methods may detect a normal traffic as abnormal, i.e., false positive detection may occur. To guarantee a normal traffic, the reactive methods are indispensable.

Within the reactive group, there are two main approaches: change-point detection techniques and statistical techniques. Change-point detection techniques [6], [7] count the number of packets in the time series, and detect DDoS attacks based on the temporal change in the packet count. For instance, the method proposed in Ref. [6] counts the number of packets received at each port in a router. The method in Ref. [7] counts the number of packets transiting through the inbound and outbound ports of the router. Using either of these methods, the router port transferring the DDoS attacks can be detected and identified. However, it is still difficult to detect or identify the victims because it is impossible to count the number of packets included in all possible sessions using a limited memory. Statistical techniques [8], [9] measure various statistical properties of specific fields in the packet headers during normal conditions in order to detect abnormalities that may indicate a DDoS attack; however, the identification of the victims is not mentioned.

In contrast to these methods, simple frequent-itemset-mining can detect and identify victims despite a limited memory, and it works regardless of the location. Thus, it can be implemented in a sensor in the router. Moreover, the wide range coverage for the abnormal traffic is also the feature of the cardinality counting based on the simple frequent-itemset-mining, and P2P flows and scanning to search a vulnerable host can also be detected [1]. The P2P flows may be concerned with fair use but they may disturb the other flows; therefore, the detection of the P2P flows becomes a help to not only illegal acts (e.g., piracy by file sharing) but also traffic controls. This detection capability can be complementary to the existing methods, e.g., the cardinalities may be used instead of packets in the Change-point detection techniques.

In contrast to the massive flows, scanning to search vulnerable hosts at a low traffic rate can be performed and its detection method has already been studied [10]. Although the cardinality counting with the previously studied algorithm can also detect such scanning by using a large amount of memory to store long-term cardinalities, it is difficult to analyze a high speed traffic of more than 100 Gbps. This is because the memory access time

tends to increase as the size of memory increases and such a high throughput cannot be achieved with the previously studied algorithm.

Traffic monitoring methods using the cardinality have already been studied. The simple frequent-itemset-mining method tolerates counting errors in order to count the cardinality with a fixed-size memory. Other researchers also tolerate such errors and give priority to memory resource requirements rather than precision topics. For example, Ishibashi et al. also proposed stochastic algorithms with good memory efficiency for similar purposes [11]. Their method is probabilistic, and thus it is assumed that the algorithms' results have probabilistic errors.

2.2 Simple Frequent-itemset-mining Method

The frequency of an itemset is used to detect mass flow in this technique. For instance, assume that a DDoS attacker attempts to deplete a target computer resource by sending a large number of packets. In this case, the destination IP address (DIP) and destination port number (DPT) are frequently found in the sent packets, and can be taken as an itemset. This is shown in **Fig. 1** (a). Moreover, variations in the itemset are also important and can be utilized to classify the detected mass flow. For example, itemsets that have the same source IP address (SIP) but different DPTs are frequently found in the packets sent by P2P software. This is because the P2P software generates service ports with randomly distinct numbers, as shown in Fig. 1 (b). In other words, the SIP in Fig. 1 (a) and the DPT have a large cardinality.

In order to detect flows with a large cardinality, the CPM algorithm was proposed [1], which counts both the frequency and variation in the itemset. **Figure 2** shows the CPM algorithm. The Itemsets function is called recursively to count the cardinality for every possible itemset, and only unique itemsets are enumerated during the recursion in order to avoid multiple counts on each itemset. A dagger mark is syntactically unmeaning but for indicating a relation with a figure explained later. **Figure 3** shows an example of the function call process for the itemsets consisting of the SIP, DIP, and DPT. In the figure, the numbers next to the arrows indicate the recursion sequence. The parentheses "()" show the value of the variable "items," while the curled parentheses "{}" show the value of the variable "rests."

In each recursion, we need to check whether the itemset is new or already present. To implement this, a quasi-associative memory function that only stores frequently found itemsets is realized by using a hash function, Hash2, as shown in **Fig. 4**. Hash2 first calculates "n" hash values of the input item, and then generates "n" indexes from the calculated values. If the input item is new, the input item is not equal to any of the cache memory contents referred to by the indexes. In this case, Hash2 selects an index corresponding to the cache memory content with the lowest frequency. As a result, only frequently found itemsets are stored in the cache memory. According to previous experimental evaluation results, "n" is set to 4 [1].

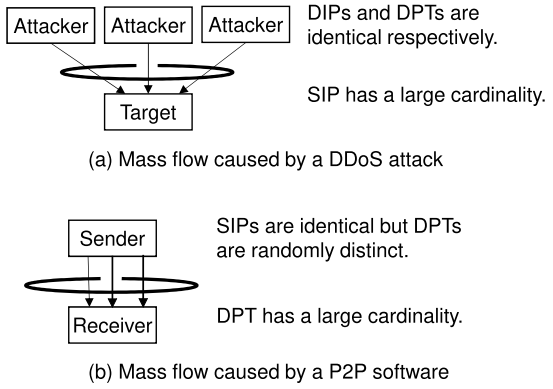

Fig. 1 Examples of mass flows derived from an abnormal traffic.

```
Algorithm CPM
Variable
    Cache[]: fixed-size table
begin
    Create empty cache;
    while (input Transaction)
        for each item in Transaction
            Itemsets(item, rest of items in Transaction);
end

Function Itemsets(items, rests)
Variable
    items[]: items in the current itemsets
    rests[]: other items in the transaction
begin
    i = index of items in cache; (calculated by hash2)
    if (i is new index)
        increment cache_diff[index of original items] by 1; †
    increment cache_cnt[i] by 1;
    for each item in rests
        Itemsets(items+item, rest of items in rests);
    if (cache_cnt[i] ≥ threshold)
        report statistics;
        cache_cnt[i] = 0;
        cache_diff[i] = 0;
end
```

Fig. 2 CPM algorithm.

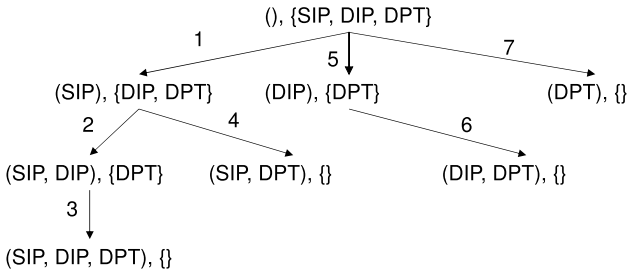

Fig. 3 Example of a recursive call.

```
Function Hash2
Input
    Item: Data to be stored in Cache
Variable
    Hash[]: Table of Hash Values
    Idx[]: Table of Cache Index
begin
    Calculate "n" hash values from Item
                and store them into Hash[]
    Idx[] = Hash[] % Cache Size
    if (one of entry refereed by Idx[] stores Item)
    then    return Idx that refers the entry
    else    Select Idx that refers least frequent entry
            cache_cnt[Idx] = 0
            return Idx
end
```

Fig. 4 Hash function for quasi-association.

Table 1 Structure of cache table.

ID	cache	1st item (e.g., SIP)		2nd item (e.g., SPT)		n-th
	_cnt	cache_diff	value	cache_diff	value	item
1	16	-	198.51.100.5	-	80	...
2	152	-	203.0.113.19	8	-	...
3	2	2	-	-	443	...
4	40	-	192.0.2.46	-	110	...
5	20	-	198.51.100.5	2	-	...
6	20	2	-	-	80	...

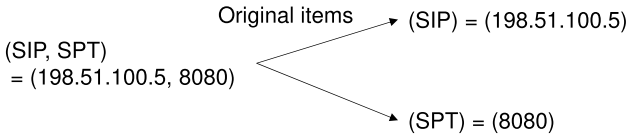

Fig. 5 Example of original items.

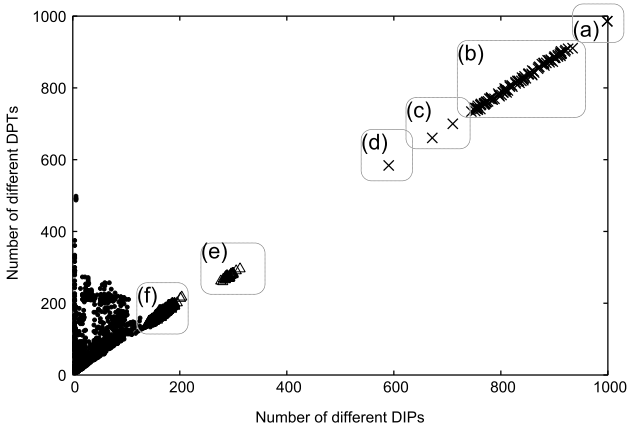

Fig. 6 Typical example of cardinality counting result.

The frequency and variations in the itemsets are stored in a fixed-size cache table. **Table 1** shows an example of the cache table. The frequency and variation are represented by cache_cnt and cache_diff, respectively. Generally, the number of columns depends on the number of items to be considered. However, only five columns are shown in the table because only two items, SIP and source port number (SPT), are considered in order to simplify the description.

Each entry in the cache table corresponds to a distinct value in the itemsets. The cache_cnt is incremented when the itemset is already stored in the cache memory. The values of the cache_diff are cleared to zero when the corresponding itemset is newly found, and they are incremented when the supersets of the corresponding itemset are newly found. One itemset has cache_diff for every item that is not included in the itemset but exists in the supersets of the itemset, and thus it may have plural cache_diff's. When the superset of an itemset is newly found, the itemset's cache_diff for an item that is included only in the superset is incremented. For example, both the cache_diff of the SPT column in the itemset (SIP) entry and the cache_diff of the SIP column in the itemset (SPT) entry are incremented when a new itemset (SIP, SPT) is found. This is shown in **Fig. 5**. The subsets are called "original items," and they are generated by subtracting one item from the newly found itemset. The cache_diff's of each original item are simply termed cache_diff, for simplification. For example, the itemset (DIP) has cache_diff's for SIP and DPT respectively in the case where three items, SIP, DIP and DPT, are focused on, and the cache_diff of itemset (DIP) indicates both the cache_diff for SIP and the cache_diff for DPT. The count for the cache_diff is maintained by the single-underlined steps in Fig. 2.

Some examples are shown in Table 1. The first entry (ID=1) shows that there are 16 packets with an SIP of "198.51.100.5" and an SPT of "80." The second entry (ID=2) shows that there are 152 packets with an SIP of "203.0.113.19," and that the number of packets with different SPTs is at least eight; i.e., there are at least eight packets with an SIP of "203.0.113.19" designated with distinct SPTs.

The value of the cache_cnt is checked, and the statistics of every itemset whose cache_cnt reaches a given threshold are reported. Concretely, the value, cache_cnt and cache_diff of the itemset are simply read out from the cache table and output, after that, the value is cleared and the cache_cnt and cache_diff are set to zero. This operation is realized by the double-underlined steps in Fig. 2. In the outside of the cardinality counting, the output values are checked with criteria that discriminate between abnormal and normal traffic.

Using this algorithm, in a fixed-size memory, only frequently found itemsets are enumerated along with the number of variations in the non-frequent parts of the itemsets. The effectiveness of the CPM has already been investigated, and **Fig. 6** shows a typical example of the cardinality counting result of the original algorithm (CPM) [1]. In this figure, four items (SIP, SPT, DIP, DPT) are focused on, and each plotted point represents the reports of an itemset (SIP, SPT) and the horizontal axis denotes the reported cache_diff for DIP while the vertical axis denotes the reported cache_diff for DPT. Those cache_diff's are reported every time when the cache_cnt of the itemset (SIP, SPT) reaches 1,000. Figure 6 (a), (b), (c), (d), (e) and (f) represent the points for the same SIP or host, respectively. As a result of the analysis on an actual traffic data, it is proven that the four hosts (a), (b), (c) and (d) are under attack or trying to find vulnerable hosts (i.e., they are concerned with DDoS or scanning) and they are clearly distinguished from the others. As the figure shows, this detection capability can be maintained even if the reported cache_diff values have a 10% or 20% error. Based on these facts, the throughput improvement of the original algorithm is discussed by introducing a self-timed pipeline and the effectiveness of the proposed algorithm is evaluated in this paper.

2.3 Self-timed Pipeline Implementation

An abnormal traffic should be detected in real time to minimize its adverse effects, particularly for traffic rates of 100 Gbps or higher. Moreover, the implementation cost of the CPM algorithm should be as low as possible in order to deploy cardinality counting widely over the Internet. However, it is difficult to implement the CPM algorithm for real-time detection over high-speed networks in a cost-effective manner.

The CPM algorithm is implicitly designed for software implementation as shown in Fig. 2. However, a circuit implementation is expected to achieve a higher throughput because it can eliminate extrinsic controls such as fetching and decoding instructions. Therefore, in this paper, we discuss the design and the effectiveness of the circuit implementation of the CPM algorithm.

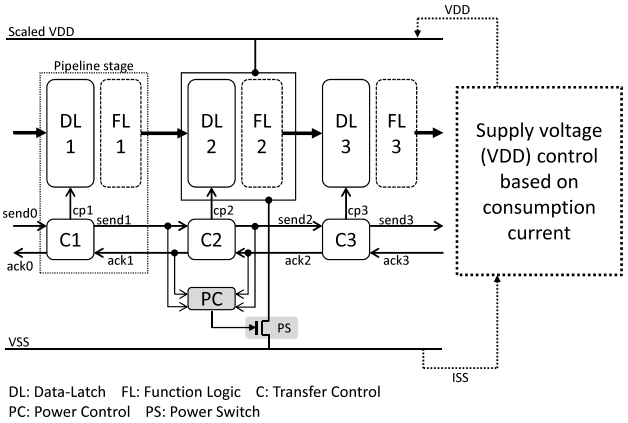

Fig. 7 Self-timed pipeline with power control mechanism.

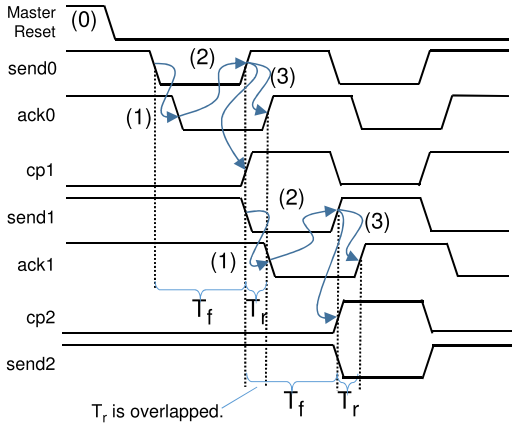

Fig. 8 Timing chart of handshake.

As shown in Fig. 2, the dominant operations of the CPM algorithm are memory reads and writes; these implement the quasi-associative memory function and the update of the cache table. From the previous study, we know that the memory size should be on the order of 1 GB to support the processing of the Gbps class of traffic [1]. To lower the implementation cost, a general-purpose memory should be utilized as long as it satisfies the requirements. Unfortunately, a static random access memory (SRAM) and content addressable memory (CAM) chips cannot be used because of their limited capacity. DRAM chips provide sufficient memory size, but they suffer from a long latency during random access. Random access is required for both the quasi-associative memory function and the cache table update because the effective addresses are calculated by hashing. Consequently, the key requirement for the circuit implementation is to reduce the DRAM access latency.

Generally, the amount of traffic that can be processed depends on the throughput, which is the number of packets processed per unit time. To increase the throughput, we implement pipelining. In this technique, the target algorithm is divided into small parts called pipeline stages that are temporally executed in parallel. Moreover, the spatially parallel execution of atomic parts that cannot be divided into the pipeline stages can also be implemented to increase the throughput.

To realize the temporal and spatial pipelining of the DRAM access, we introduce a self-timed pipeline. The self-timed pipeline is not an exclusive circuit architecture to realize the pipelining of a cardinality counting algorithm. Although conventional clock-synchronized pipeline can also be used to the pipelining, the flexibility and power saving feature of the self-timed pipeline make it possible to implement the cardinality counting algorithm with low power consumption as described later. **Figure 7** illustrates the basic structure of the self-timed pipeline [12]. In the self-timed pipeline, pipeline stages with valid data are driven exclusively by a localized data transfer called a handshake. As shown in Fig. 7, each pipeline stage consists of a data-latch (DL), a functional logic (FL), and a transfer control unit (C). The self-timed pipeline is a type of asynchronous bundled data pipeline that employs a four-phased handshake [13]. Based on the handshake, the valid data in the self-timed pipeline is transferred between adjacent stages according to the following procedure.

- (0) Reset: After the assertion of the reset signal, C negates both its send signal (representing a transfer request) and its ack signal (representing acknowledge).
- (1) C asserts its ack signal after its send signal is asserted.
- (2) After the assertion of the ack signal, the preceding C negates its send signal.
- (3) After negation of the send signal, C asserts both its gate open signal (cp) and its send signal. Concurrently, it negates its ack signal if the ack signal from the succeeding C is negated. As a result, the data is latched in the stage to which the succeeding C belongs.
- The succeeding C repeats the above steps in the same manner as the current C, as shown in **Fig. 8**.

T_f and T_r denote the send signal propagation time and the ack signal propagation time, respectively. T_f is adjusted to the critical path of the corresponding FL, while T_r is set to the set-up hold time of the corresponding DL. As a result of the handshake, T_r is contained within T_f and thus no extrinsic processing time is added as long as the occupancy of the pipeline stages is kept within a design target.

This handshake also lowers power consumption by directing dynamic consumption current into pipeline stages with valid data. On the other hand, empty pipeline stages can be powered off by the power control and power switch to reduce the leakage current through the empty stages [14]. Moreover, the signal propagation delay of DL, FL, and C are changed at an equal rate according to the supply voltage. Thus, the supply voltage of the self-timed pipeline can be scaled at run time while the rate of change of the voltage is moderate enough to guarantee transistor switching. In other words, to lower the power consumption, the throughput and the processing time can be changed during the execution of the target algorithm. The self-timed pipeline can implement not only dedicated circuits but also processors; its feasibility and effectiveness are analyzed in Refs. [12], [15], [16].

To exploit the parallelism inherent in the target algorithm exhaustively, there must be flexibility in the deployment of the pipeline stages to implement data dependencies among the operations in the target algorithm. The self-timed pipeline can be expanded freely by using (1) a merge (M) stage that accepts data from two preceding stages in the order of arrival and then transfers the accepted data to a succeeding stage, and (2) a branch

(B) stage that transfers each set of accepted data to one of the succeeding stages selectively. The flexibility of the self-timed pipeline structure is discussed in Ref. [17].

Consequently, the self-timed pipeline is a promising circuit architecture to realize the temporally and spatially parallel DRAM access required for cardinality counting.

3. Cardinality Counting Circuit Design

Although the CPM algorithm can be deployed over the self-timed pipeline that parallelizes DRAM access, serially plural random accesses for updating the cache table remain because of the processing order of the itemsets. Aggregating such plural random accesses into a single access is necessary to reduce the total DRAM access latency and to achieve a high throughput.

In the next section, we describe how the processing order inherent in the CPM algorithm is an obstacle to the aggregation in the circuit implementation. We also present a new algorithm whose processing order enables the DRAM access aggregation and the self-timed pipeline circuit implementation.

3.1 Algorithm

As shown in Fig. 2 and Table 1, the value, frequency (cache_cnt), and variety (cache_diff) of an itemset is stored at an address calculated by hashing the itemset. In other words, the entries of one itemset can be stored into a discrete DRAM chip independently from those of the other itemset without any collisions, in this case, each DRAM chip plays a role of a cache table dedicated to one itemset and there are no needs to combine the entries stored in different DRAM chips. Based on this fact, the cache table can be divided into distinct spaces that are respectively assigned to discrete DRAM chips. On the other hand, to minimize overhead or additional controls, the pipeline should be deployed along with the data flow inherent in the target algorithm. The unfolding of the iterations of the Itemsets function over the self-timed pipeline is shown in **Fig. 9** (a). The cache table is divided into distinct spaces based on the itemset, and each space is allocated to the corresponding iteration part. By implementing this pipelining, an itemset's value and cache_cnt can be updated by a single read from and a single write to the DRAM at each iteration. Moreover, the cache_diff updates denoted by dotted lines in Fig. 9 (a) can be combined with this single read and write by postponing the update timing until the execution proceeds to the corresponding original item's iteration part. In contrast, the cache_diff updates denoted by solid and backward lines cannot be aggregated into the other updates. This is because their corresponding itemsets may differ from the itemset processed in the source pipeline stages, and thus the calculated effective addresses in the source pipeline stages may differ from those in the destination pipeline stages.

To realize the aggregation of the cache_diff updates, the processing order of the itemsets is changed by modifying the recursive call of the Itemsets function. This is shown in **Fig. 10**; the modified part is marked with wavy underlines. The new algorithm in Fig. 10 is named CPM2. The original items of an itemset are a subset of the itemset and must be shorter than the itemset. Based on this fact, CPM2 processes longer itemsets before shorter

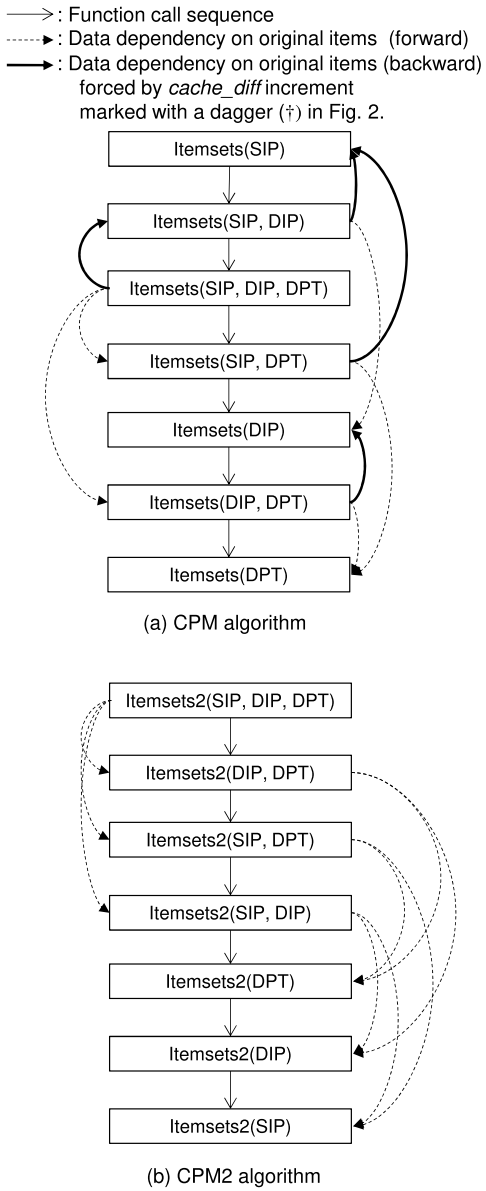

Fig. 9 Pipelining of itemsets function's iterations.

ones. By introducing a Boolean flag *new*, the original item's cache_diff update is postponed until the corresponding itemsets are processed. Figure 9 (b) illustrates the pipelining using CPM2. This shows that DRAM accesses required for all cache_diff updates are aggregated, and that each itemset is processed with a single read from and a single write to the DRAM.

The original CPM was realized on the basis of the well-known frequent itemset mining program Apriori [18]. Apriori explores shorter itemsets first, followed by the longer itemsets based on the found/shorter itemsets. In other words, Apriori uses the "shorter first" strategy to explore frequent itemsets. The original CPM follows this process, and this results in the backward data flow shown in Fig. 9 (a). In contrast, CPM2 explores the longest itemsets first. Here, Hash2 is used to store frequent itemsets and discard less frequent itemsets. This change in the processing order eliminates the backward data flow, and allows for the realization of parallel processing with aggregated plural random access using self-timed pipeline circuits. This aggregation results in throughput increase because it reduces the total DRAM access time that

```
Algorithm CPM2
Variable
    Cache[]: fixed-size table
begin
    Create empty cache;
    while (input Transaction)
        for each items in Transaction
            Itemsets2(items, FALSE);
end


Function Itemsets2(items, new)
Variable
    items[]: items in the current itemsets
    new: boolean value indicating that the superset is new
begin
    i = Hash3(items);
    if (new)
        increment cache_diff[i] by 1;
    increment cache_cnt[i] by 1;
    if (i is new index)
    then  new = TRUE;
    else  new = FALSE;
    for each item in items
        Itemsets2(items-item, new);
    if (cache_cnt[i] ≥ threshold)
        report statistics;
        cache_cnt[i] = 0;
        cache_diff[i] = 0;
end
```

Fig. 10 Pipelining-oriented CPM algorithm.

is the bottleneck of the pipeline. As shown in the Fig. 9 (a), the cache_diff of itemset (SIP) is incremented in addition to the increment of its cache_cnt when a new itemset (SIP, DIP) or a new itemset (SIP, DPT) is found, and thus three times of DRAM accesses are performed at most for one packet while these access are aggregated to one DRAM access with the CPM2; therefore the throughput of the CPM2 becomes 3 times that of the original CPM at the worst-case. Moreover, the maximum number of DRAM accesses required for one packet is governed by the maximum number of cache_diff increments, and thus it increases as the number of items in the target itemset increases. For instance, four times of DRAM accesses are required at most for 4-tuple itemset (SIP, SPT, DIP, DPT), e.g., one cache_cnt increment is required and three cache_diff increments performed by new itemsets (SIP, SPT), (SIP, DIP) and (SIP, DPT) against an itemset (SIP). In this case, the worst-case throughput of the CPM2 becomes 4 times that of the original CPM.

Furthermore, we introduce a new memory management function named Hash3 in CPM2 to improve the throughput. This is shown in **Fig. 11**. It is well-known that the DRAM imposes not only a long latency on the random access but also a shorter latency on a burst access for successive addresses. The original Hash2 imposes "n" times a random access in the worst case. From the viewpoint of guaranteeing the robustness of cardinality counting, the worst case throughput should be improved. Hash3 makes the "n" effective addresses successive and "n" entries are checked or updated at a time. Obviously, this burst access reduces the DRAM access latency and thus improves the cardinality counting throughput.

3.2 Self-timed Cardinality Counting Circuit

To exploit the parallelism of the CPM2 algorithm, the recur-

```
Function Hash3
Input
    items: itemset to be stored in Cache
Variable
    hash: hash value
    index: cache index
begin
    Calculate hash value from items;
    index = hash % (cache size / n) * n;
    if (one of cache entries referred by successive indexes,
            index, index+1, ···, index+n-1,
            stores itemset)
    then  return the index value referring to the entry;
    else  select index value referring to the least frequent entry;
            cache_cnt[the index value] = 0;
            return the index value;
end
```

Fig. 11 DRAM-oriented memory management.

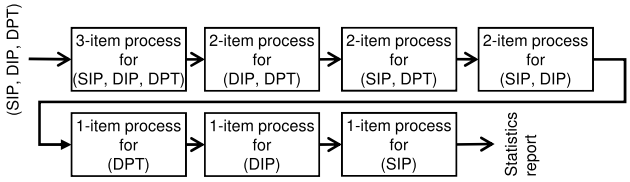

Fig. 12 Pipelining of Itemsets2 function.

sion of the Itemsets function is unfolded and the iterative parts are deployed linearly (or temporally) as shown in **Fig. 12**. In contrast to this linear deployment, a concurrent deployment in which some iterative parts are performed concurrently after their precedent parts can be realized. As for the concurrent deployment, an additional mechanism that results in the circuit area/power increase is required to detect the completion of all the spatially parallelized parts and to gather sets of data that correspond to each iterative part among the output data from its precedent parts. This is because the completion timing of each iterative part may be different from that of the others due to the DRAM access time's run-time variation caused by refreshing operations. To avoid such area/power overhead, the linear deployment is adopted. In each iterative part, to conceal the access latency of the DRAM, the read and write required to realize the quasi-associative memory function and cache table update are parallelized spatially by dividing the memory space of the cache table. This spatial division is realized by using a specific part of the hash value calculated from the itemset for selecting one of the DRAM chips. Moreover, the "n" itemsets are stored to successive addresses indexed by the calculated hash value, and they are read from and written to the DRAM in a burst. These techniques eliminate the one-by-one memory read and write accesses in the pipeline stages for checking whether the input itemset is already stored and writing back the updated values.

Figure 13 illustrates the block diagram of the self-timed pipeline that implements the temporally and spatially parallel execution of each Itemsets recursion. Although the spatially divided DRAM accesses can be assigned to parallel pipeline stages, they are deployed temporally into linear pipeline stages. This is because a set of signal lines that are connected to I/O pins one-on-one is required for each DRAM chip and the temporal deployment leads to a reduction in the number of signal lines per an LSI chip by assigning each Itemset process to a discrete LSI chip connected to a DRAM chip. In contrast, the spatial deployment

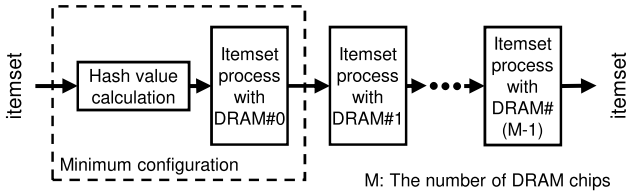

Fig. 13 Pipelining of DRAM accesses in an N-item process.

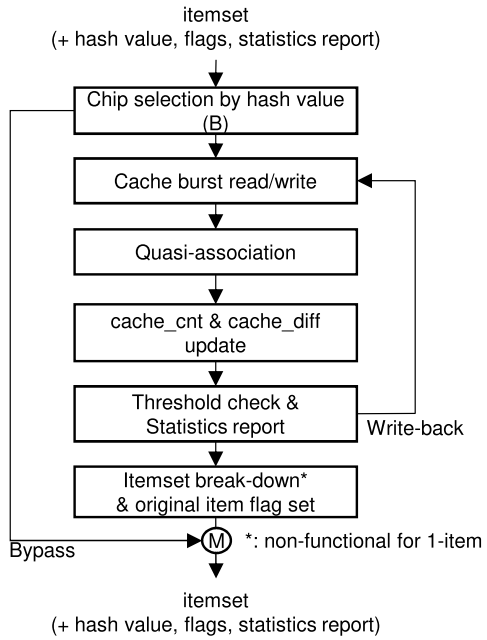

Fig. 14 Pipelining of itemset process.

into spatially parallel pipeline stages inherently requires parallel signal lines to plural Itemset processes or DRAM chips, and thus it increases the number of I/O pins per an LSI chip in proportion to the number of the DRAM chips used. The numbers of I/O pins are strictly limited and should be lowered to reduce the development cost.

The body of the recursion is finely divided and parallelized using the self-timed pipeline, as shown in **Fig. 14**. First, the "n" itemsets and their cache_cnt's and cache_diff's are read from the DRAM in a burst during the "Cache burst read/write" stage. They are compared to the input itemset to realize the quasi-association by which one itemset with the lowest frequency is discarded in the "Quasi-association" stage. In accordance with the result of the quasi-association, the values of the cache_cnt and cache_diff are updated in the "cache_cnt & cache_diff update" stage. The updated values are compared with the given threshold values in the "Threshold check & Statistics report" stage. In this stage, if the updated values exceed the threshold, the corresponding statistics are added to the output data. To realize the zero-clear of cache_cnt and cache_diff and perform a cache table update, a write-back path is added. At the last stage ("Itemset break-down & original item flag set"), the itemsets for the next recursion are produced by subtracting one of the items from the input itemset. A flag representing the original item's cache_diff for the subtracted item is added to the produced itemset if the input itemset is newly found. This flag is evaluated in the "cache_cnt & cache_diff update" stage of the next recursion part, and the corresponding itemset's cache_diff for the subtracted item is incremented if the flag is set.

Thus, the temporal and spatial parallelism inherent in the CPM2 algorithm is deployed using the self-timed pipeline. The DRAM access latency is reduced to 1/M theoretically, where M denotes the number of DRAM chips utilized.

4. Evaluation of Practicality

4.1 Performance of Cardinality Counting

To enable parallel processing, the processing order is changed in our algorithm. In the previous algorithm, the shortest itemsets are stored first, and extended/longer itemsets are stored later. In contrast, in the new algorithm, the longest itemsets are stored first, and subset/shorter itemsets contained in the longest itemsets are stored later.

As a result of the hashing described in Fig. 11, an itemset may be assigned to an address at which a previously processed distinct itemset is stored. When two distinct itemsets are assigned to the same address, the itemset with the lowest frequency count is removed. If the two distinct itemsets have the same frequency count, the itemset that was processed first is removed. Thus, the change in the processing order may also change the cardinality counting result.

To analyze the difference between the two algorithms, we count cardinalities in the actual Internet traffic. For this purpose, we downloaded traffic data from the MAWI Internet Traffic Archive [19] in Dec. 2016. As shown in the Fig. 10, the CPM2 reports the itemsets whose cache_cnt reaches a given threshold value. After the counting, the existence or non-existence of the abnormal traffic is checked by investigating the cache_diff's values of the reported itemsets [1]. In the investigation, the abnormal traffic existence is indicated if the value of any one of the cache_diff's is high. For example, the Fig. 6 shows that the abnormal traffic exists if the value of either the cache_diff for DIP or the cache_diff for DPT is more than 500. Based on these facts, we compare the cache_diff's values of the itemsets reported by both algorithms.

Figure 15 shows the results. In this figure, "number of itemsets" denotes the number of unique itemset instances that are reported, and "reported cardinalities" means the number of unique itemset instances that have cache_cnt reaching 1,000. The horizontal axis denotes the value ranges for the highest value in the cache_diff's of each reported itemset. As the value range becomes higher, the number of itemsets reported commonly from both algorithms decreases from approximately 7,000 to 1,000, as shown in Fig. 15 (a). The ratio "the reported cardinality of each algorithm/the number of itemsets reported commonly from both algorithms" approaches 1.0 as the value range becomes higher, as shown in Fig. 15 (b). In the previous study, it was found that cache_diff's whose value is more than 500 are useful in detecting the abnormal traffic [1]. Based on this, we expect that a ratio close to 1.0 within the value ranges of more than 500 is acceptable for practical network monitoring.

4.2 Throughput of Cardinality Counting

To show the practicality of the proposed circuit implementa-

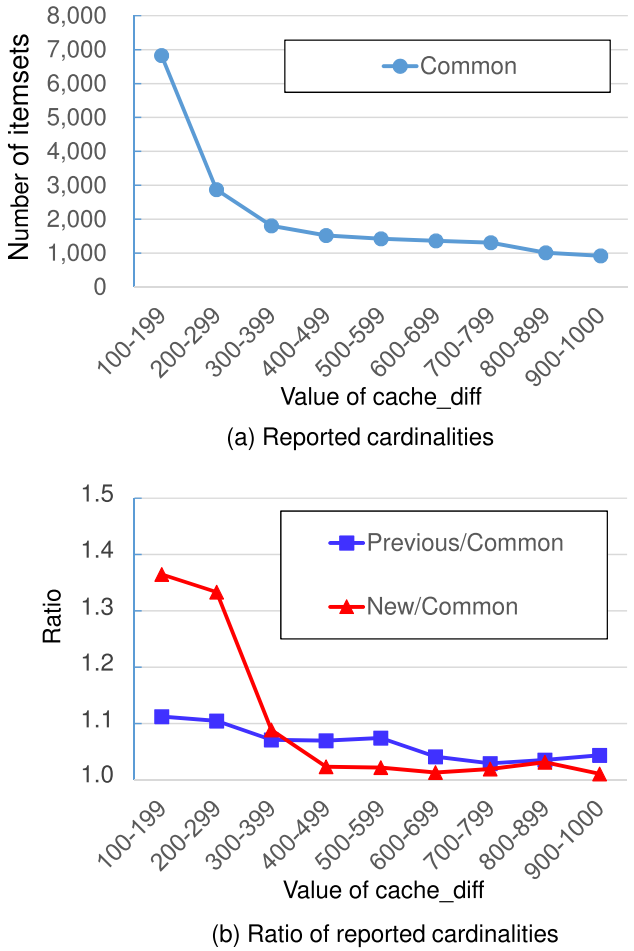

Fig. 15 Comparison of cardinality.

tion, the throughput of the designed circuit is evaluated from the viewpoint of achieving 100 Gbps. As shown in Fig. 12, the cardinality counting circuit is a series of pipeline blocks that are structured as shown in Fig. 13, where the minimum configuration is composed of a hash value calculation and an itemset process with one DRAM chip. The maximum throughput of pipelines is determined by the longest pipeline stage, which will have the longest processing time in the entire pipeline. This is defined to be $1/T_{max}$ [packet/sec.], where T_{max} denotes the longest processing time. In the cardinality counting circuit, the FL's of the pipeline stages except for "Cache burst read/write" stage can be realized by combinational circuits because they need no states inherently, and thus they can be divided into more than 2 finer parts that are housed in different successive pipeline stages in order to shorten the processing time of each pipeline stage; therefore, T_{max} is governed by the processing time of the "Cache burst read/write" stage with the longest itemset, and T_{max} can be reduced to a design target value by parallelizing the number of Itemset processes.

In this evaluation, to determine the design target of T_{max}, we assume that the number of packets is 50M ($= \frac{100G}{250\times8}$) per second for 100 Gbps traffic, and assume a packet length of 250 Bytes. In other words, T_{max} should be equal to or less than 20 ns ($= 1/50M$) to realize a real-time traffic analysis for a data rate of 100 Gbps or higher. Moreover, the design target of the other pipeline stages' processing times (i.e., the pipeline tact) should be less than 20 ns.

In the self-timed pipeline, the pipeline tact is defined by $(T_f + T_r)$; the difference $(T_{max} - (T_f + T_r))$ is used to absorb a part of the processing time of the next pipeline stage [20], and thus the $(T_f + T_r)$ of each pipeline stage is set to the minimum value by tuning the circuit design. However, it is difficult to retain the $(T_f + T_r)$ defined in the circuit design phase through a circuit implementation process that depends on design tools and the fabrication environment; therefore, the $(T_f + T_r)$ may vary. Based on these facts, to evaluate the worst-case throughput, the $(T_f + T_r)$ is set to 20 ns (the same as for T_{max}) and a four-item process for (SIP, SPT, DIP, DPT) with a minimum configuration is simulated to estimate the required number of DRAM chips.

The cardinality counting is finely pipelined into simple operations. Apart from the burst DRAM accesses, the dominant operations are (1) comparisons that check the equivalence and exceedance of two values, (2) summation and multiplication for hash value calculations, and (3) incrementation. The processing time for one of these operations may be shorter than 10 ns. This is because more complex operations, such as compound manipulation and accumulation implemented on the latest processor prototype LSI [16] using a self-timed pipeline for a 65 nm-CMOS process, take less than 10 ns to complete.

Although the pipeline stages shown in Fig. 14 can be directly implemented or pipelined finely as discussed above, overwriting the entries of the cache table may occur and it changes the counting results slightly. Stochastically, distinct instances of an itemset correspond to different hash values. On the other hand, the same itemset instances may be input or produced and they correspond to the same hash value. For example, two itemset instances that have the same value (192.0.2.46) for (SIP) are produced by subtracting the SPT from two distinct itemset instances that are (192.0.2.46, 80) and (192.0.2.46, 443) for (SIP, SPT). Moreover, even distinct itemset instances may have the same hash value infrequently because the hash value space is limited. When plural itemset instances that correspond to the same hash value are processed simultaneously, they may exist in consecutive pipeline stages in the pipeline, and thus the same address of a DRAM may be read consecutively in the "Cache burst read/write" stage. In this case, the memory read for an itemset instance may occur before the processing result of the precedent itemset instance is written back, and the processing result of the precedent itemset instance may be overwritten by that of the itemset instance. In the evaluation of this paper, to eliminate the possibility of the overwriting, the pipeline stages from the "Cache burst read/write" stage to the "Threshold check & Statistics report" stage are integrated into one pipeline stage.

As a result of the pipelining, the "Hash value calculation" and "Itemset process with DRAM#0" are realized in 23 and 10 stages, respectively.

The T_{max} is the processing time of the integrated pipeline stage because this stage includes the "Cache burst read/write". The "Cache burst read/write" stage is composed of DRAM and a memory controller necessary to broker data transfers between the self-timed pipeline circuit and the DRAM. We set 20 ns to each processing time for the "Quasi-association," "cache_cnt & cache_diff update," and "Threshold check & Statistics report"

stages for the worst-case evaluation. That is, the T_{max} is composed of 60 ns (= 20 × 3), the DRAM access time and the processing time of the memory controller. As a practical DRAM, Micron's DDR3 SDRAM with 2-Gbit capacity and 16-bit data width is assumed and its circuit simulation model provided by Micron is used. Although the designed circuit can be realized by an application specific integrated circuit (ASIC) with circuit tuning, a field-programmable gate array (FPGA) device, Intel's Stratix V GS, is assumed in the simulation for easy prototyping, because the circuit simulation model of a memory controller that is necessary to broker data transfers between the self-timed pipeline circuit and the DRAM is provided in the Intel's circuit libraries. These circuit simulation models recreate the actual processing time, and thus the T_{max} can be estimated precisely for the worst-case evaluation. By utilizing these simulation models and the designed circuit, a circuit simulator (ModelSim) is used to conduct RTL (register-transfer-level) simulation to simulate the pipeline tact level behavior of the entire cardinality counting circuit.

To measure the worst-case throughput, 1000 itemsets with randomly generated IP addresses and port numbers are input by changing the input rate, which is the number of itemsets input per second. For every input itemset, two DRAM accesses are performed to read and update the cache table in the circuit. **Figure 16** shows the measured result; it shows that the throughput is proportional to the input rate within the maximum value, and that the maximum achievable throughput is approximately 4.6 M packet/sec. This result also indicates that a 100 Gbps traffic can be analyzed in real-time by providing 11 (= ⌈50/4.6⌉) parallel DRAM accesses, because T_{max} has an average value of 217 ns ($\simeq 1/4.6M$) and will decrease to less than 20 ns (= 217/11).

Consequently, the proposed circuit with 11 DRAM components is expected to handle 100 Gbps traffic. By increasing the number of DRAM components, the proposed circuit can handle a faster network traffic. This conclusion is based on an assumption that the input itemsets amount of one Itemset process becomes stochastically the same as that of the other Itemset process as a result of the selection based on the hash value of the input itemset in the pipelining illustrated in Fig. 13. On the other hand, the itemsets to each Itemset process may be unequally input depending on traffic patterns, even in such cases, the expected throughput can be easily guaranteed by adding extra DRAM's and Itemset processes as a margin of processing capability because the self-timed pipeline circuit can be easily extended by virtue of its localized wiring.

Although all possible itemsets are generally surveyed in the cardinality counting and the effectiveness of this exhaustive survey has already been shown [1], some item processes may be skipped in the pipelining illustrated in Fig. 12 in practical use. In such cases, those redundant processes can be easily skipped by bypassing them with additional branch (B) and merge (M) pipeline stages without any degradation on the throughput of each pipeline stage in the proposed circuit while the aggregation of the DRAM accesses still increases the throughput; therefore, the throughput of the proposed circuit can be greater than that of a cardinality counting circuit with the original algorithm as discussed in Section 3.1.

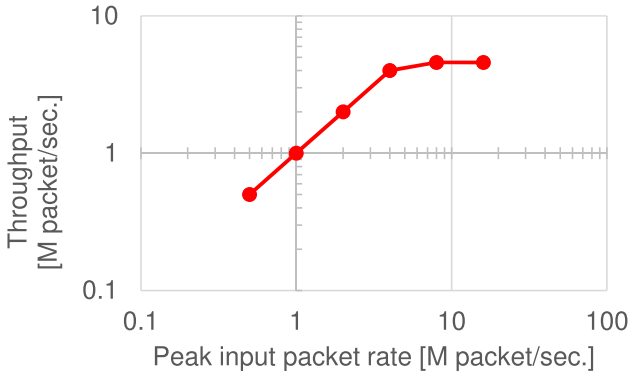

Fig. 16 Achieved throughput.

5. Conclusion

This paper proposes a new cardinality counting algorithm that enables the parallel handling of data with a reduced number of memory accesses. This algorithm and the design of a self-timed pipeline circuit to realize real-time cardinality counting are the main contributions of this paper. In summary:

- In contrast to the previous algorithm, the newly developed algorithm makes it possible to aggregate plural DRAM accesses by changing the processing order.
- The self-timed pipeline circuit can reduce the DRAM access duration by parallelizing the process. With 11 parallel DRAM accesses, the developed circuit is expected to handle network traffic rates of up to 100 Gbps.
- Although the change in processing order makes the cardinality counting results inaccurate, the experimental results show that the inaccuracy is within an acceptable range.

In the cardinality counting algorithm, the frequency (cache_cnt) of itemset is focused on and compared to a threshold because it is essential to detect a large amount of packet streams that disturb smooth and safe communications. On the other hand, the itemset's variation (cache_diff) is not checked with the threshold value. The detection capability of an expanded algorithm whose cache_diff has a threshold will be investigated as a future work. Moreover, the integration of the designed circuit into actual traffic monitoring systems remains as a future work.

References

[1] Shomura, Y., Watanabe, Y. and Yoshida, K.: Analyzing the number of varieties in frequently found flows, *IEICE Trans. Communications*, Vol.E91-B, No.6, pp.1896–1905 (2008).
[2] Sannomiya, S., Sato, A., Yoshida, K. and Nishikawa, H.: FPGA implementation of cardinality-based abnormal traffic detection algorithm, *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications* (*PDPTA*), pp.252–258 (July 2017).
[3] Sannomiya, S., Sato, A., Yoshida, K. and Nishikawa, H.: Cardinality counting circuit for real-time abnormal traffic detection, *Proc. 2017 IEEE 41st Annual Computer Software and Applications Conference* (*COMPSAC*), pp.505–510 (July 2017).
[4] Beitollahi, H. and Deconinck, G.: Analyzing well-known countermeasures against distributed denial of service attacks, *Computer Communications*, Vol.35, No.11, pp.1312–1332 (June 2012).
[5] Urushidani, S., Abe, S., Yamanaka, K., Aida, K., Yokoyama, S., Yamada, H., Nakamura, M., Fukuda, K., Koibuchi, M. and Yamada, S.: New directions for a Japanese academic backbone network, *IEICE Trans. Information and Systems*, Vol.98, No.3, pp.546–556 (2015).
[6] Chen, Y., Hwang, K. and Ku, W.-S.: Collaborative detection of DDoS

attacks over multiple network domains, *IEEE Trans. Parallel and Distributed Systems*, Vol.18, No.12, pp.1649–1662 (2007).
[7] Wang, H., Zhang, D. and Shin, K.G.: Change-point monitoring for the detection of DoS attacks, *IEEE Trans. Dependable and Secure Computing*, Vol.1, No.4, pp.193–208 (2004).
[8] Feinstein, L., Schnackenberg, D., Balupari, R. and Kindred, D.: Statistical approaches to DDoS attack detection and response, *Proc. DARPA Information Survivability Conference and Exposition*, Vol.1, pp.303–314 (Apr. 2003).
[9] Toledo, A.L. and Wang, X.: Robust detection of MAC layer denial-of-service attacks in CSMA/CA wireless networks, *IEEE Trans. Information Forensics and Security*, Vol.3, No.3, pp.347–358 (June 2008).
[10] Yoon, M. and Chen, S.: Detecting Stealthy Spreaders by Random Aging Streaming Filters, *IEICE Trans. Communications*, Vol.94, No.8, pp.2274–2281 (2011).
[11] Ishibashi, K., Mori, T., Kawahara, R., Hirokawa, Y., Kobayashi, A., Yamamoto, K. and Sakamoto, H.: Estimating top N hosts in cardinality using small memory resources, *Proc. 22nd International Conference on Data Engineering Workshops* (*ICDEW '06*), p.29 (Apr. 2006).
[12] Terada, H., Miyata, S. and Iwata, M.: DDMP's: Self-timed super-pipelined data-driven processors, *Proc. IEEE*, Vol.87, No.2, pp.282–296 (1999).
[13] Myers, C.J.: Asynchronous circuit design, University of Utah John Wiley & Sons, Inc. (2001).
[14] Miyagi, K., Sannomiya, S., Iwata, M. and Nishikawa, H.: Low-powered self-timed pipeline with variable-grain power gating and suspend-free voltage scaling, *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications* (*PDPTA*), pp.618–624 (July 2013).
[15] Nishikawa, H.: Design philosophy of a networking-oriented data-driven processor: CUE, *IEICE Trans. Electronics*, Vol.E89-C, No.3, pp.221–229 (2006).
[16] Sannomiya, S., Aoki, K., Iwata, M. and Nishikawa, H.: Power-performance verification of ultra-low-power data-driven networking processor: ULP-CUE, *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications* (*PDPTA*), pp.465–471 (July 2012).
[17] Komatsu, K., Sannomiya, S., Iwata, M., Terada, H., Kameda, S. and Tsubouchi, K.: Interacting self-timed pipelines and elementary coupling control modules, *IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences*, Vol.E92-A, No.7, pp.1642–1651 (2009).
[18] Agrawal, R., Imielinski, T. and Swami, A.: Mining association rules between sets of items in large databases, *Proc. 1993 ACM SIGMOD International Conference on Management of Data*, pp.207–216 (May 1993).
[19] MAWI working group traffic archive, available from ⟨http://mawi.wide.ad.jp/mawi/⟩.
[20] Sannomiya, S., Omori, Y. and Iwata, M.: A macroscopic behavior model for self-timed pipeline systems, *Proc. 17th Workshop on Parallel and Distributed Simulation* (*PADS*), pp.133–140 (June 2003).

Shuji Sannomiya received his B.E. and M.E. degrees in Information Systems Engineering from Kochi University of Technology, Kochi, Japan, in 2002 and 2004, respectively. He received his Ph.D. degree in engineering from Kochi University of Technology in 2009. Now he is an assistant professor of Faculty of Engineering, Information and Systems, University of Tsukuba. Currently he has interests in networking processor architecture and its VLSI implementation.

Akira Sato received his Ph.D. from University of Tsukuba in 1998. He is an associate professor in Department of Information Engineering, Academic Computing and Communications Center at University of Tsukuba. His current research interest is an operation of academic networks. He is a member of IPSJ.

Kenichi Yoshida received his Ph.D. from Osaka University in 1992. In 1980, he joined Hitachi Ltd., and is working for University of Tsukuba from 2002. His current research interest includes application of Internet and application of machine learning techniques.

Hiroaki Nishikawa received his B.E., M.E. and Ph.D. degrees in electronic engineering from Osaka University in 1976, 1981 and 1984, respectively. After being with Osaka University, he is presently a Professor at University of Tsukuba. He was Dean of School of Informatics from 2013 to 2015 and Vice President for Academic Intelligence from 2016 to 2017. He was also worked as a visiting scientist at Laboratory for Computer Science, MIT in 1994, 1995, 1997, 1998 and a visiting professor in the Electrical Engineering and Computer Science Department at the University of Southern California in 1988. His current research interests include user-friendly system specification and verification environment, ultra-low-power data-driven processor architecture and hyper-distributed system. He received IASTED Best Paper Award in the area of Processor Architecture in PDCS, WORLDCOMP'10 Outstanding Achievement Award, WORLDCOMP'16 Leadership & Visionary Award, respectively. Dr. Nishikawa is a member of IPSJ, IEICE and a senior member of IEEE.