# FPGA Implementation of Cardinality-Based Abnormal Traffic Detection Algorithm

**Shuji SANNOMIYA[1], Akira SATO[1], Kenichi YOSHIDA[2] and Hiroaki NISHIKAWA[1]**
[1]Faculty of Engineering, Information and Systems, University of Tsukuba, Ibaraki, Japan
[2]Faculty of Business Sciences, Univeristy of Tsukuba, Otsuka, Bunkyo City, Tokyo, Japan

**Abstract**— *To keep the Internet being a safe and secure social infrastructure, abnormal traffic such as P2P, DDoS and Internet worms should be detected and controlled robustly among huge amount of packets streams. Already, a cardinality-based abnormal traffic detection method has been studied, and its hardware-oriented algorithm has been proposed to use self-timed pipeline for parallelizing and pipelining the accesses of DRAM that is indispensable to implement the algorithm within acceptable cost. The key process of the proposed algorithm enumerates only frequent combinations of the input packet's header fields and their variations. In this paper, the proposed algorithm is improved to avoid the throughput fluctuation due to hash-based and scattered DRAM accesses in the enumeration, for guaranteeing and enhancing the worst-case performance, and the performance is also estimated based on an FPGA circuit simulation.*

**Keywords:** abnormal traffic detection, frequent-itemset-mining, self-timed pipeline

## 1. Introduction

With the continual inventions of its devices and applications, the Internet is becoming one of indispensable infrastructures of our society. On the other hand, abnormal traffic that disrupts safe and secure communications is also increasing. The typical constituents of the abnormal traffic are Peer-to-Peer (P2P) flows, Distributed Denial of Service (DDoS) attacks and Internet worms and they should be detected and controlled among huge amount of packets streams. Already, a cardinality-based traffic analysis method [1] and its hardware-oriented algorithm for real-time traffic analysis [2] have been proposed to realize such abnormal traffic detection.

The cardinality-based traffic analysis method focuses on the frequently found combinations of header fields and their variations observed in the packet streams to distinguish abnormal traffic from normal one. The frequency and variations of such combinations indicate the abnormal traffic flow. For instance, distinct destination IP addresses are combined together with the same source IP address in the headers of packets conveying malicious contact information from a worm-infected client that is searching for a vulnerable server. In contrast to other already studied methods, this method can enumerate only frequent and meaningful combinations by using a given small fixed-size memory that results in the acceptable design constraint of hardware implementation [1]. The core enumeration process of this method is termed cardinality counting, because of its similarity to the cardinality, the number of elements of a given set, in mathematics.

As for the implementation of this method, the cardinality counting algorithm requires GByte order of memory to store all the possible combinations along with their frequency and variation counts, and such memory should be realized by Dynamic Random Access Memory (DRAM) in order to keep the implementation cost acceptable. However, the DRAM access imposes long latency, and it may become the bottleneck to achieve high speed traffic analysis. To make the cardinality counting real-time against such high speed traffic for reducing the adverse effects of the abnormal traffic, a hardware-oriented cardinality counting algorithm is proposed based on self-timed pipeline that is a global clockless pipeline, and the proposed algorithm reduces the number of memory accesses and enables parallel and pipelined memory access for concealing the latency of some memory accesses under that of the others [2].

In this paper, the hardware-oriented cardinality counting algorithm is improved from the viewpoint of robustness. The cardinality counting's throughput may change because several memory accesses may be required additionally to check whether the combination of the input packet's header fields is newly found or not. On the other hand, to assure the abnormal detection capability, the cardinality counting should be robust against the abnormal traffic, i.e. the throughput of the cardinality counting should be guaranteed regardless of the instances of the combination. To guarantee and enhance the worst case throughput, an addressing function that utilizes the burst access capability of the DRAM is introduced. Moreover, the throughput is quantitatively estimated based on the FPGA circuit simulation.

## 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 cardinality-based abnormal traffic detection method [1] focuses on the itemset of every packet and

ISBN: 1-60132-466-9, CSREA Press ©

monitors its frequency and variety simultaneously. Itemset refers to the combination of a packet's header items such as source IP address and destination port number, the frequency of the itemset indicates the massiveness of the flow, and the variety characterizes the kind of flow.

In this section, we discuss the advantages of the cardinality-based abnormal traffic detection method by comparing it with other proposed methods. Next, we explain the process of simultaneous counting of both the frequency and variation using the cardinality counting algorithm. Finally, we discuss the self-timed pipeline circuit implementation.

## 2.1 Abnormal Traffic Analysis Method

Abnormal traffic can be treated as a 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 appear 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 [3]. The device or program used for detecting the attacks in both 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 the sensors required scales with the number of servers to be protected, and hence we focus on the router type defense in our work.

In recent years, the traffic volume in the Internet backbone as well as the subnetworks underneath the backbone has increased tremendously. It has been reported that the connection bandwidth between the Internet backbone and Internet Service Provider (ISP) is over 100 Gbps [4]. Hence, to protect the servers under an ISP, the sensor should be able to analyze traffic of 100 Gbps or more.

Based on the reaction time, router type methods are further categorized into two groups: proactive and reactive [3]. Proactive methods aim to prevent the occurrence of DDoS attacks, while reactive methods come into effect after the DDoS attacks occur. Proactive methods may detect normal traffic as abnormal due to safe side filtering, i.e., false positive detection may occur. To guarantee 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 [5], [6] count the number of packets in the time series, and detect DDoS attacks based on temporal change in the packet count. For instance, the method proposed in [5] counts the number of packets received at each port in a router. The method in [6] counts the number of packets transiting through in-bound and out-bound 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 limited memory. Statistical techniques [7], [8] measure various statistical properties of specific fields in the packet headers during normal conditions in order to be able to detect DDoS attacks, but the identification of the victims is not mentioned.

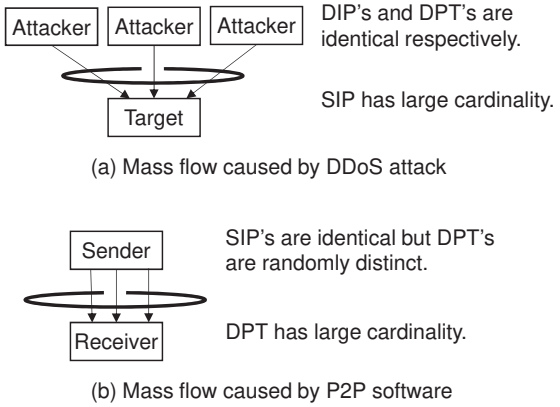


Fig. 1: Examples of mass flows derived from the abnormal traffic.

In contrast to these methods, the cardinality-based abnormal traffic detection method can detect and identify victims despite limited memory, and it works regardless of the location. Thus, it can be implemented in a sensor in the router. Moreover, cardinality counting makes it possible to detect P2P flows in addition to the DDoS attacks [1].

## 2.2 Cardinality-Based Abnormal Traffic Detection 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, variation in the itemset is also important and can be utilized to classify the detected mass flow. For example, itemsets that have same source IP address (SIP) but different DPT are frequently found in the packets sent by a 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 large cardinality.

In order to detect flows with large cardinality, a cardinality counting algorithm was proposed [1], which counts both the frequency and variation in the itemset and is originally named CPM algorithm. Fig. 2 shows the latest CPM algorithm that is derived by modifying the original one for

```
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 = index of items in cache; (calculated by hash2);
    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] ≥ thresh_hold)
        report statistics;
        cache_cnt[i] = 0;
        cache_diff[i] = 0;
end
```

Fig. 2: Pipelining-oriented CPM algorithm.

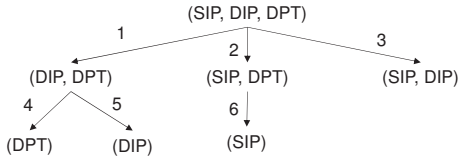


Fig. 3: Example of recursive call.

reducing the number of memory accesses [2]. The Itemsets2 function is called recursively to count the cardinality for every possible itemset. Fig. 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."

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 least frequency. As a result, only frequently found itemsets

```
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 _cnt | 1st item (e.g. SIP) | | 2nd item (e.g. SPT) | | n-th item |
|---|---|---|---|---|---|---|
| | | cache_diff | value | cache_diff | value | |
| 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 | ... |

are stored in the cache memory. According to previous experimental evaluation result, "n" is set to 4 [1].

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 for providing a simplified description.

Each entry of the cache table corresponds to a distinct value of 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. For example, both the cache_diff of the SPT column in the entry of the itemset (SIP) and the cache_diff of the SIP column in the entry of the itemset (SPT) are incremented when a new itemset (SIP, SPT) is found. This is shown in Fig. 5. The subsets are called "original items." The count for the cache_diff is maintained by the steps wavy-underlined in the Fig. 2.

Some examples are shown in table 1. The first entry (ID=1) shows that there are 16 packets whose SIP is "198.51.100.5" and SPT is "80." The second entry (ID=2) shows that there are 152 packets whose SIP is "203.0.113.19" and the number of packets with different SPT

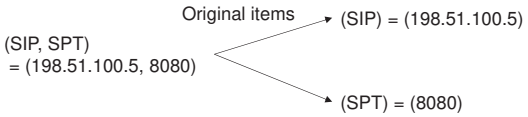


Fig. 5: Example of original items.

is at least "8."

The values of the cache_cnt and cache_diff are checked, and the statistics of the corresponding itemset are reported. These values are set to zero when the values exceed the threshold values that discriminate between abnormal and normal traffic. This operation is realized by the straight-underlined steps in Fig. 2

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 cardinality counting algorithm has already been investigated in [1].

## 2.3 Self-Timed Pipeline Implementation

Abnormal traffic should be detected at real-time to minimize its adverse effects. Moreover, the implementation cost of the CPM2 algorithm should be as low as possible in order to deploy cardinality counting widely over the Internet. A circuit implementation is expected to achieve higher throughput because it can eliminate extrinsic controls such as fetching and decoding of instructions. Therefore, in this paper, we discuss the design and effectiveness of the circuit implementation of the CPM2 algorithm.

As shown in Fig. 2, the dominant operations of the CPM2 algorithm are memory read and write; these implement the quasi-associative memory function and update of the cache table. From the previous study, we know that the memory size should be in the order of GByte in order to support the processing of Gbps class of traffic [1]. To lower the implementation cost, general-purpose memories should be utilized as long as they satisfy the requirement. Unfortunately, Static Random Access Memory (SRAM) and Content Addressable Memory (CAM) chips cannot be used because of their small capacity. DRAM chips provide enough memory size, but they suffer from long latency for random access. Random access is required for both the quasi-associative memory function and 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 executed in parallel temporally. Moreover, spatially parallel execution of atomic parts that cannot be divided into the pipeline stages can also be implemented to increase the throughput.

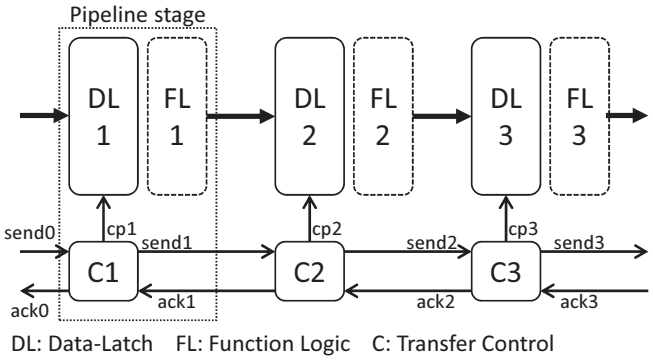


Fig. 6: Self-timed pipeline with power control mechanism.

To realize temporal and spatial pipelining of the DRAM access, we introduce a self-timed pipeline that enables flexible pipelining. Fig. 6 illustrates the basic structure of the self-timed pipeline [9]. In the self-timed pipeline, pipeline stages with valid data are driven exclusively by a localized data transfer called handshake. As shown in Fig. 6, each pipeline stage consists of a data-latch (DL), functional logic (FL), and transfer control unit (C). The self-timed pipeline is a kind of asynchronous bundled data pipeline that employs a four-phased handshake [10]. 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 transfer request and 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 similar to the current C, as shown in Fig. 7.

$T_f$ 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.

The self-timed pipeline can implement not only dedicated circuits but also processors; its feasibility and effectiveness are analyzed in [9], [11], [12].

To exploit the parallelism inherent in the target algorithm

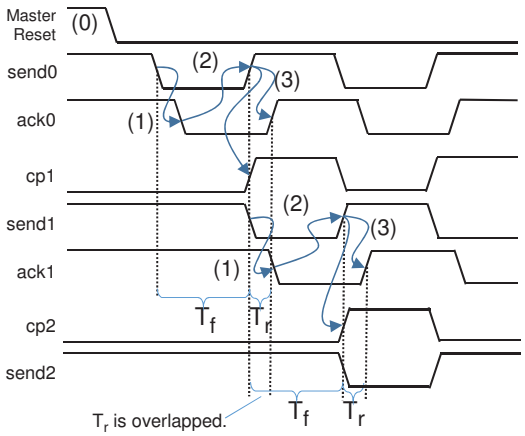


Fig. 7: Timing chart of handshake.

```
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 >> 2);
    if (one of cache entries referred by successive indexes,
            index, index+1, index+2, and index+3,
            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. 8: DRAM-oriented memory management (n = 4).

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 a merge (M) stage that accepts data from two preceding stages in the order of arrival and transfers the accepted data to a succeeding stage and a branch (B) stage that transfers each data to one of the succeeding stages selectively. The flexibility of the self-timed pipeline structure is discussed in [13].

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

# 3. Cardinality-Based Abnormal Traffic Detection Circuit Design

In this section, we describe that the Hash2 function may fluctuate the effective throughput of the CPM2 algorithm, and then present a robust cardinality counting algorithm and its self-timed circuit implementation for guaranteeing and enhancing the worst-case throughput.

## 3.1 Algorithm Modification

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. This information can be stored in a DRAM different from that of the other itemsets. Based on this fact, the cache table can be divided into distinct spaces that are deployed over separate 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 CPM2 algorithm makes it possible to reduce the number of DRAM accesses by unfolding the iterations of the Itemsets2 function over the self-timed pipeline, and by introducing a Boolean flag *new*, the original item's cache_diff update is postponed until the corresponding itemsets are processed [2].

In the pipelining, 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 set of read and write of the DRAM at each iteration.

Furthermore, we introduce a new memory management function named Hash3 in CPM2 to guarantee and enhance the worst-case throughput. This is shown in Fig. 8. It is well known that the DRAM imposes not only long latency on the random access but also shorter latency on burst access for successive addresses. The original Hash2 imposes "n" times random access in the worst case. From the viewpoint of guaranteeing the robustness of cardinality counting, the worst case throughput should be predictable and lowered. 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 recursion of the Itemsets2 function is unfolded and the iterative parts are deployed linearly (or temporally) as shown in Fig. 9. 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 out 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 used for checking whether the input itemset is already

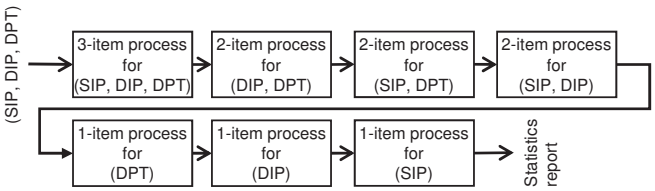


Fig. 9: Highly parallel pipeline of Itemsets2 function.

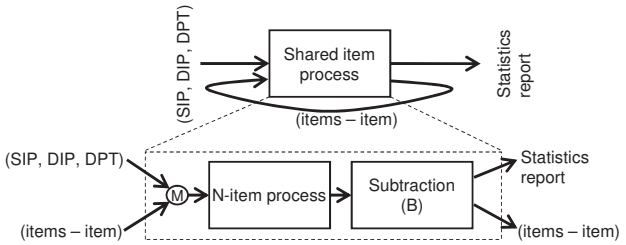


Fig. 10: Area-saving pipeline of Itemset2 function.

stored or not, and to write back the updated values.

As for the circuit area saving, the CPM2 algorithm can also be implemented on a circular pipeline as shown in Fig. 10. The hashing function can be used to divide the memory space according to the itemset's value even with different length. To exploit such hashing's functionality, one item process module can be shared to process every itemset for saving the circuit area at the expense of throughput. This pipelining is useful in the case where the prime design constraint is the circuit area but the required throughput is moderate.

Fig. 11 illustrates the block diagram of the self-timed pipeline that implements the temporally and spatially parallel execution of the item process module. Although the spatially divided DRAM accesses can be assigned to parallel pipeline stages, they are deployed temporally into linear pipeline stages. This is because temporal deployment leads to the reduction of the concurrent signal lines that may increase the number of I/O pins for LSI chips. The numbers of I/O pins are strictly limited and should be lowered to reduce development cost.

The body of the recursion is finely divided and parallelized using the self-timed pipeline as shown in Fig. 12. First, the "n" itemsets and their cache_cnt's and cache_diff's are read out from the DRAM as a burst in the "Cache burst read/write" stage. They are compared to the input itemset to realize the quasi-association by which one itemset with the least 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 "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-

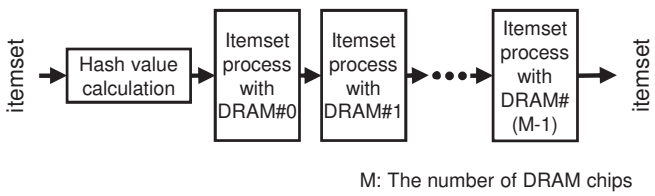


Fig. 11: Pipelining of DRAM accesses in N-item process.

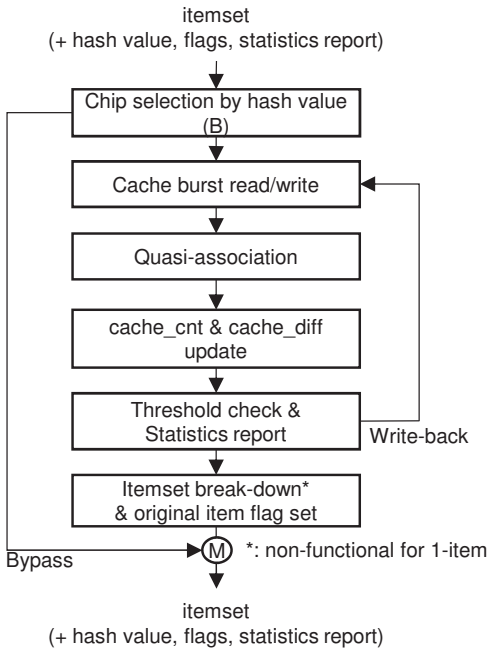


Fig. 12: Pipelining of itemset process.

clear of cache_cnt and cache_diff and perform cache table update, the 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 of the input itemset. A flag representing the original 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 cache_diff of the corresponding itemset 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.

## 3.3 Throughput Estimation

To show the practicality of the proposed circuit implementation, the throughput of the designed circuit is evaluated using a Register Transfer Level (RTL) circuit simulation.

Although the designed circuit can be realized using Application Specific Integrated Circuit (ASIC) with circuit tuning, Field-Programmable Gate Array (FPGA) device is assumed

in this estimation because of its easiness of prototyping. Intel's Stratix V GS which is one of high-end devices is used as the target device, and Micron's DDR3 SDRAM with 2-Gbit capacity and 16-bit data width is used as the DRAM. To execute the RTL circuit simulation, Intel's Quartus Prime are used. The DRAM should be refreshed periodically to retain its data, and some controller circuit is required to schedule and conduct the refreshing in addition to the read and write accesses. To realize such controller, a memory controller circuit library provided by Intel is used.

The number of packets to be processed to perform real-time analysis for a given traffic speed depends on the length of the packets. To estimate the performance of the circuit design quantitatively, we suppose that the number of packets is 50M ($= \frac{100G}{250\times 8}$) per a second for 100 Gbps traffic and a packet length of 250 Bytes. On the other hand, the maximum throughput of the pipeline is determined by the pipeline stage whose processing time is the longest in the entire pipeline. This is defined to be $1/T_{max}$ [packet/sec], where $T_{max}$ denotes the longest processing time. In other words, $T_{max}$ should be equal to or less than 20 ns ($= 1/50M$) to realize real-time traffic analysis for a data rate of 100 Gbps or more.

As mentioned previously, $T_{max}$ is determined by the pipeline stage that performs burst DRAM access. The burst DRAM access is performed twice for every packet because the cache table update requires burst read and burst write in the "Cache burst read/write stage" as shown in Fig. 12. Consequently, $T_{max}$ is defined by $\frac{T_{latency}\times 2}{M}$, where $T_{latency}$ denotes the average time of the burst DRAM access.

$T_{latency}$ is estimated using the RTL simulation. The actual $T_{latency}$ value varies depending on the DRAM refresh timing of the memory controller, and so an average value is calculated from the simulation results of approximately 2000 times read and write accesses. The results show that $T_{latency}$ is approximately 100 ns. On the other hand, the processing time of the other pipeline stages are expected to be less than 20 ns. This is because the dominant operations apart from the burst DRAM access are comparisons for checking the equivalence and exceedance for two values, and increment and hash value calculation; the processing time for one of them can be shorter than 20 ns. Actually, more complex operations such as multiplication and accumulation implemented on the latest processor prototype LSI [12] using self-timed pipeline for a 65 nm-CMOS process take less than 10 ns to complete.

Consequently, it is expected that 100 Gbps traffic can be analyzed in real-time by providing 10 parallel DRAM accesses. In this case, $T_{max}$ is approximately 20 ns ($= \frac{100n\times 2}{10}$). In other words, the proposed N-item process circuit with 10 DRAM components can handle 100 Gbps traffic. By increasing the DRAM components, the proposed circuit can handle faster network traffic.

# 4. Conclusion

In this paper, we proposed a robust cardinality counting circuit using self-timed pipeline for guaranteeing and enhancing the worst-case throughput. In contrast to the previous algorithm in which "n" memory accesses are required at most to check whether a newly incoming itemset is new or not, the proposed circuit makes it possible to aggregate the memory accesses into a series of burst DRAM accesses with predictable and low latency. Moreover, it is revealed that the developed circuit can handle up to 100 Gbps of traffic by providing 10 parallel DRAM accesses. The functionality of the designed circuit is verified through RTL circuit simulation. Performance tuning of fine pipelining of the circuit and integration with actual traffic monitoring systems remain as future works.

# References

[1] Y. Shomura, Y. Watanabe, and K. Yoshida, "Analyzing the number of varieties in frequently found flows," IEICE Transactions on Communications, Vol.E91–B, No.6, pp. 1896–1905, June 2008.

[2] S. Sannomiya, A. Sato, K. Yoshida, H. Nishikawa, "Cardinality counting circuit for real-time abnormal traffic detection," in Proc. of the IEEE Computers, Software, and Applications Conference, July 2017 (to appear).

[3] H. Beitollahi and G. Deconinck, "Analyzing well-known countermeasures against distributed denial of service attacks," Computer Communications, vol. 35, no. 11, pp. 1312–1332, June 2012.

[4] S. Urushidani, A. Shunji, K. Yamanaka, A. Kento, S. Yokoyama, H. Yamada, M. Nakamura, K. Fukuda, M. Koibuchi, and S. Yamada, "New directions for a japanese academic backbone network," IEICE Transactions on Information and Systems, vol. 98, no. 3, pp. 546–556, Mar. 2015.

[5] Y. Chen, K. Hwang, and W.-S. Ku, "Collaborative detection of ddos attacks over multiple network domains," IEEE Transactions on Parallel and Distributed Systems, vol. 18, no. 12, pp. 1649–1662, Dec. 2007.

[6] H. Wang, D. Zhang, and K. G. Shin, "Change-point monitoring for the detection of dos attacks," IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 4, pp. 193–208, Oct.-Dec. 2004.

[7] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred, "Statistical approaches to ddos attack detection and response," in Proc. of DARPA Information Survivability Conference and Exposition, vol. 1, pp. 303–314, Apr. 2003.

[8] A. L. Toledo and X. Wang, "Robust detection of mac layer denial-of-service attacks in csma/ca wireless networks," IEEE Transactions on Information Forensics and Security, vol. 3, no. 3, pp. 347–358, June 2008.

[9] H. Terada, S. Miyata, and M. Iwata, "Ddmp's: self-timed super-pipelined data-driven processors," Proceedings of the IEEE, Vol.87, No.2, pp.282–296, Feb. 1999.

[10] C. J. Myers, "Asynchronous circuit design," Univ. of Utah John Wiley & Sons, Inc., 2001.

[11] H. Nishikawa, "Design philosophy of a networking-oriented data-driven processor: CUE," IEICE Transactions on Electronics, Vol.E89–C No.3, pp. 221–229, Mar. 2006.

[12] S. Sannomiya, K. Aoki, M. Iwata, and H. Nishikawa, "Power-performance verification of ultra-low-power data-driven networking processor: ulp-cue," in Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 465–471, July 2012.

[13] K. Komatsu, S. Sannomiya, M. Iwata, H. Terada, S. Kameda, and K. Tsubouchi, "Interacting self-timed pipelines and elementary coupling control modules," IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Vol.E92–A, No.7, pp. 1642–1651, July 2009.