ABSTRACT

LAI, YU-KUEN. Packet Processing on Stream Architecture. (Under the direction of Dr. Gregory T. Byrd).

Stream processing architectures have been proposed as efficient and flexible platforms for network packet processing. This is because packet processing shares many of the same characteristics of media and image processing that motivate stream architectures: little global data reuse, abundant data parallelism, and high computational complexity.

This dissertation explores the SIMD (Single Instruction, Multiple Data) stream architecture for network packet processing with several security-related applications. The implementations are based on the stream programming model on the Imagine media processor, which consists of three tiers of memory hierarchy and eight VLIW clusters operating in SIMD mode.

The applications explored are listed as follows: the Advanced Encryption Standard (AES) in parallel operation modes with key agility, the Multilinear Modular Hash (MMH) message authentication code, Bloom-filter-based content inspection engine for signature-based intrusion detection, and the sketch update for Internet traffic analysis. Some novel methodologies are also presented as applications being transformed and implemented on the stream architecture.

The thesis characterizes the processing throughput of these applications and explores the tradeoffs on different configurations of stream architecture. Moreover, the sketch update application is also implemented on the Intel IXP network processor, in order to explore the difference between Imagine and a traditional architecture. The SIMD operation simplifies the access to shared data structure without explicit synchronization and arbitration overhead. As a result, the system achieves efficient utilization of maximum memory bandwidth.

The architecture demonstrates the flexibility to support computation-intensive packet processing tasks at high performance. Applications such as hash and statistical based tasks are best fit into the stream programming model with an abundance of producer and consumer locality: portions of values computed and stored in the stream register file (SRF) are used for calculating a new set of values recursively. With a 500-MHz clock, the stream processor is capable of processing packets up to multi-gigabit-per-second throughput with outstanding power efficiency.

Although packet processing over the SIMD stream architecture exhibits control flow and load balancing issues due to packet size variation, the analysis indicates that the multi-core, multi-SIMD architecture improves the performance and efficiency. Further explorations are proposed as

promising directions for future research.

Packet Processing on Stream Architecture

by

Yu-Kuen Lai

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh

2006

Approved By:

Dr. Khaled Harfoush	Dr. Yan Solihin
Dr. Gregory T. Byrd Chair of Advisory Committee	Dr. W. Rhett Davis

To my loving parents and wife . . .

I could not have completed my study without their love, support and encouragement . . .

Biography

Yu-Kuen Lai was born in Fong-Yuan, Taiwan. He received his Master degree in Electrical and Computer Engineering from North Carolina State University in 1997. He was a Sr. ASIC design engineer at Delta Networks Inc. and Applied Micro Circuits Corporation (AMCC) from 1997 through 2002 at Research Triangle Park (RTP), North Carolina.

Yu-Kuen started his Ph. D. study in the Center for Embedded Systems Research (CESR) lab at NC State University in 2002. He also worked as a Sr. Design Engineer Intern at Internet Specific Integrated Circuit Corp. (ISIC) in 2003 at Cary, North Carolina. His research interests include network processor architecture, computer networking, network security and digital system design. He is currently a member of IEEE.

Acknowledgements

First and foremost, I am very grateful and indebted to my parents Mr. Yuan-Lian Lai and Mrs. Shu-Yuan T. Lai, and my loving wife Yu-Chen. I could not have completed my study without their love, support and encouragement.

I am very fortunate to have had Dr. Gregory T. Byrd as my advisor. I would like to express my profound gratitude to him for his constant guidance, inspiration and encouragement throughout this research. Dr. Byrd is not only my Ph. D. advisor, but also a mentor giving me comments, feedbacks and showing me the way of doing research.

I would like to express my sincere appreciation to Dr. Davis, Dr. Harfoush and Dr. Solihin for serving as committee members and helping me through my research with constructive criticism.

Special thanks are given to Dr. Rom-Shen Kao at Infineon and his family. I am very lucky to have such a good friend backing me up while away from home.

My decision to pursue the degree was very much influenced by Dr. Wen-Yao Chung at Chung-Yuan Christian University and Dr. J. Morris Chang at Iowa State University. I am very thankful to them for their constant encouragements and advices.

I am also very thankful to Mr. Kehao Chen at Hifn and Dr. Ho-Yen Chang at Ericsson IP Infrastructure for their valuable comments and assistance. I really enjoyed the short walk around the Lake Raleigh with Ho-Yen brain storming creative research ideas.

I have to acknowledge the support that I received from Dr. Abhishek Das and Dr. Ujval Kapasi at Stanford University for providing and supporting the Imagine simulation and compilation tools. I also acknowledge the funding support from the Center for Advanced Computing and Communication (CACC) at North Carolina State University.

Finally, I would like to thank my colleagues Mine Altunay, Mu-Huan Chian, Salil Pant and Jathin Rai for their helping hands during my study at the CESR lab.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Emerging network applications are shifting from routing and traffic management to those requiring inspection and modification of packet contents. Examples include content-based billing, quality of service, layer-7 switching, and network security. Such applications act on data streams, and the requirement to process every byte of a packet exceeds the already-substantial processing capabilities of high-speed networking equipment. Hardware (ASIC) implementations are possible point-solutions to these problems, but they do not offer the flexibility needed to address new applications or adapt to changing network infrastructure. Programmable solutions based on conventional microprocessor architectures do not provide enough performance; their memory hierarchies are optimized for the spatial and temporal locality present in desktop or workstation applications, but they are not well-suited to data streaming, which exhibits little or no data reuse. Network processors (NPs) are application-specific processors that are optimized to bridge the gap between the performance of ASIC and the programmability of general-purpose processors. Most current NPs employ multiple optimized cores, operating independently in MIMD (Multiple Instruction stream, Multiple Data stream) fashion, usually with multiple threads per processor to hide latency to memory. Performance on traditional network applications (routing, queue management, traffic shaping) is impressive, but we believe that these architectures are not well-suited for the emerging packet applications. They are limited by insufficient processing power in the cores, inadequate bandwidth to memory, and complex programming models.

Stream architectures are specifically designed to apply a consistent set of complex operations to each of a sequence (stream) of elements. Packet processing shares many of the same

characteristics of media and image processing that motivate stream architectures: little global data reuse, abundant data parallelism, and high computational complexity. Therefore, stream architectures have been proposed as efficient and flexible platforms for network packet processing [28].

However, to the best of our knowledge, there is no research or publication available on the networking applications based on the SIMD stream architecture. Therefore, as part of a broader investigation into stream-based network processors, we have begun to study and identify several emerging networking applications and transform them into stream programming model on Imagine SIMD stream architecture [55]. This thesis presents several emerging, security-centric packet processing applications in stream programming model. Those implementations include AES encryption [66, 95], message authentication based on a family of almost-universal hash functions [68], Bloom-filter-based content matching engine [67] for deep packet inspection and the sketch-based Internet traffic analysis over streaming data model.

1.1 Thesis Organization

The AES encryption algorithm is one of the most important building block for the secure network operation. It has the characteristics of rich instruction level parallelism (ILP) and data level parallelism (DLP) in some operation modes. A key-agile AES implementation is demonstrated in Chapter 2. One of the best performances published [72, 3] in non-feedback mode for a 32-bit architecture is 232 cycles per block (16-byte block). In contrast to the typical measurement setup [3] where most of the data is in the level-one cache, the simulation results in this chapter do include the data movement from the memory to the processor itself. The performance can reach up to 32 cycles per block in a stream size of 96 blocks. For a system clock of 500 MHz, the throughput of the AES encryption can reach up to 2 Gbps. The simulation results of OCB-AES operation also demonstrate compelling performance. We also describe architectural enhancements and the performance impact of different packet sizes and more complex encryption modes.

The collision property of the universal hash function serves as the foundation for the applications (message authentication, content inspection and traffic analysis) demonstrated in the later part of this report. Therefore, in Chapter 3, we provide a detailed discussion on the universal class of hash functions as well as some implementation examples .

The implementation of MMH, a family of almost-universal hash functions for message authentication, is shown in Chapter 4. By using eight VLIW clusters, the operation is performed in

a Multi-SIMD fashion, achieving multi-Gigabit-per-second throughput with a collision probability on the order of 2^{-120}. The best and worst case throughput of MMH producing a 128-bit pre-tag is 7.14 Gbps and 2.23 Gbps with packet sizes of 1536 bytes and 128 bytes, respectively. The pre-tag represents the hash value before XORing with the one-time pad. According to the performance results of MMH (200MHz Pentium-Pro), the best case throughput (message in cache) for generating the 32-bit and 64-bit output are 1080 and 500 Mbps respectively [44]. As the throughput decreases approximately linearly, we estimate the throughput of producing a 96 and 128-bit results are roughly in the range of 375 and 250 Mbps, respectively. On the other hand, a speculated throughput of 300Mbps is estimated on a 200 MHz Pentium-Pro processor [44, 9].

A Bloom-filter-based content matching engine is presented in Chapter 5. By arranging multiple processors in a pipelined fashion, the system is capable of processing patterns extracted from the rules of the Snort distribution and achieving a throughput of 400 Mbps for 1500-byte packets. The packet processing, i.e., hash computation over an entire packet, is best fit into the stream programming model with an abundance of producer and consumer locality: portions of the the hash values computed and stored in the stream register file (SRF) are used for calculating a new set of hash values recursively. We also demonstrate the flexibility and performance of the stream architecture supporting the realization of the universal class of hash functions for the Bloom filter. This chapter explores the implementation of important data structures in the stream architecture, which may potentially benefit many emerging networking applications.

The accurate statistics collection and measurement of Internet traffic serve as the basis for infrastructure planning, network provisioning, capacity forecasting and accounting [127, 65, 35]. In Chapter 6, we discuss methodologies for IP traffic analysis on the stream architecture based on the sketch algorithm [19, 60]. Sketch [15, 1, 80] is a powerful yet compact data structure capable of synopsizing substantial numbers of data elements without keeping its stateful information. The sketch algorithms can be applied to many applications such as estimating frequent items, finding the top-k items and identifying the significant differences for anomaly detection. We explore the data structure through the *Indexed SRF* accesses on the Imagine stream processor as a continuing effort from the previous chapter. The simulation shows the processor is capable of supporting sketch update at 10.8 Gbps throughput for minimum-sized IP packets.

In Chapter 7, we implement the unique sketch data structure on a different type of processor architecture. We briefly introduce Intel's Internet eXchange Architecture (IXA) framework. Based on the IXP2800 network processor, we focus on the implementation of sketch update since it is regarded as the bottleneck of the sketch algorithm. We make a brief comparison with the simu-

lation results of Imagine and discuss the pros and cons of these two approaches. For sketch update on 40-byte packets, the simulation shows a throughput of 13 Gbps with 16 microengines running at 1.4 GHz system clock. The IXP2800 achieves approximately 22% higher throughput, however it consumes 7 times more power than that in the Imagine.

Finally, we summarize some of the excellent aspects of the stream architecture supporting these networking applications. For example, the architecture is capable of handling computation intensive tasks and achieve high performance with low power consumption. The estimated power consumption of this processor is less than four Watts. It exploits the abundant parallelism and stream locality effectively. The SIMD architecture provides efficient vector style processing and simplifies the memory access to share data structure without explicit synchronization overhead. Moreover, the programming model provides the great flexibility achieving high processing performance.

In the end, we provide analyses of performance tradeoffs for packet processing on stream architecture. We discuss the issues of architecture scaling for these applications and conclude with ideas for future work.

1.2 Contributions

While contributions are presented in each chapter, we summarize the major ones we've made in this thesis as follows.

- We survey and identify several emerging network applications (encryption, authentication, packet inspection and traffic analysis), successfully transform them into SIMD stream programming model and compare to state-of-the-art implementation of these applications.
- Provide the details of implementation analysis and algorithm mapping on several applications over the SIMD stream architecture.
- Explore the limitations and overheads on the Imagine stream architecture. The shortcomings of the architecture with regard to these networking applications are revealed as well as improvements suggested.
- Propose novel methodologies attacking the deficiency of algorithmic transformation in stream programming model. Examples such as the dual-core implementation for better utilization and throughput improvement (Chapter 2), the Multi-SIMD operation for authentication tag

generation (Chapter 4), the reduced-pattern length with central-weighted scheme on pattern matching (Chapter 5) and the time multiplexed sketch operation over SRF (Chapter 6).

- Study the feasibility of the traffic analysis in streaming data model over the SIMD stream architecture. Explore the data structure of counting Bloom filter over the SRF on stream architecture and the IXP network processor. Detailed analysis on the sketch update process between two different approaches is provided, which, we believe, will benefit a lot of networking applications based on the same data structure.

1.3 Imagine Stream Processor

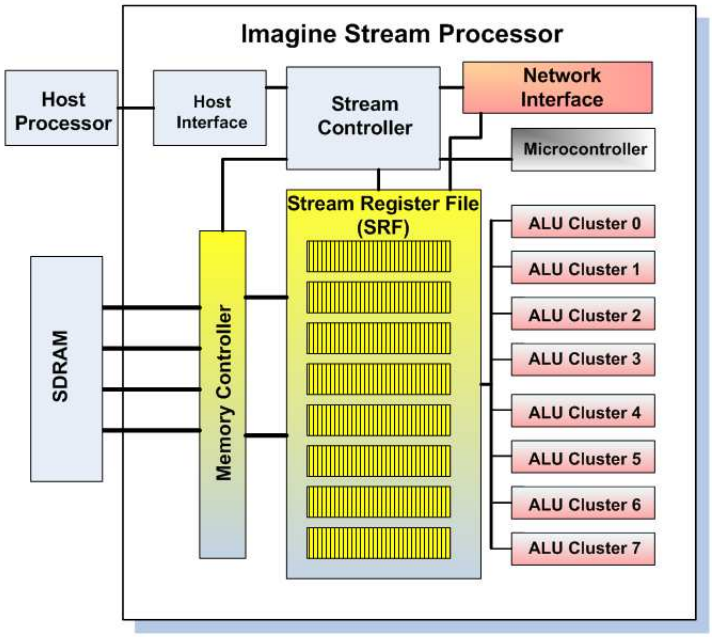

Figure 1.1: The Imagine architecture block diagram. [55]

The Imagine Stream Processor [101, 27, 109] is designed and optimized for image processing as a co-processor. The processor contains eight ALU (Arithmetic Logic Unit) clusters, which receive VLIW (Very Long Instruction Word) instructions broadcast from the on-chip microcontroller in a SIMD fashion. With a system clock of 500 MHz, the processor reaches a peak performance

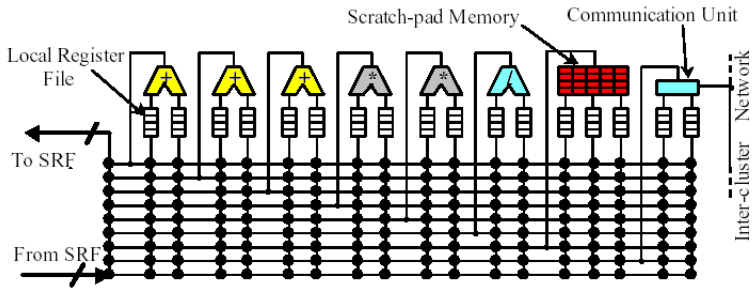

Figure 1.2: The block diagram of arithmetic cluster for Imagine processor [55].

of 40 GOPS (Billion Operations Per Second). A key feature of the architecture is a three-level memory hierarchy, which consists of the main memory, the Stream Register File (SRF) and the Local Register File (LRF) within the clusters. The hierarchy captures the characteristics of media processing applications' demands on memory bandwidth as well as the producer-and-consumer locality between kernels. Figure 1.1 shows the block diagram of the Imagine Processor.

In each cluster, as shown in Figure 1.2, there are 3 adders, 2 multipliers, 1 divider, a 256-entry scratchpad and an inter-cluster communication unit.

The programming model consists of two levels, the kernel and stream level. The kernel level is programmed in KernelC where the computation on the stream of data is specified. KernelC is compiled by the Kernel Scheduler, IScd [30]. At run-time, kernels are loaded into the Microcode Store within the on-chip Microcontroller through the SRF. The VLIW instructions are later dispatched to the clusters. The stream level program, written in StreamC, is run on the host processor to coordinate and orchestrate the flow of streams as well as the invocation of the kernels.

The detailed stream operations and programming examples are illustrated and provided in the next chapter as we transform the AES encryption algorithm into a program for the stream architecture.

Chapter 2

AES Packet Encryption

2.1 Introduction

Cryptography has become one of the most important requirements for networked applications as the Internet grows exponentially. Many Internet activities and transactions rely heavily on privacy and authentication services, which demand a lot of computational resources. Therefore, application-specific integrated circuits (ASICs) are widely used as building blocks for secure gateways and routers. However, due to the cost and time for ASIC development, Network Processor (NP) technologies are gaining momentum — flexibility is one of the crucial factors in a world of rapidly evolving networking protocols.

Several NP architectures exploit Packet Level Parallelism (PLP) in a MIMD (Multiple Instruction stream, Multiple Data stream) fashion. But typically the same operations are applied to every packet [110], which implies a SIMD (Single Instruction stream, Multiple Data stream) mode of processing. Many cryptographic algorithms, such as the AES candidates [81], exhibit an abundance of both Instruction Level Parallelism (ILP) [17, 125] and Data Level Parallelism (DLP). Several studies [3, 70, 71] show that the performance can be improved significantly by leveraging the special SIMD extensions for media processing on existing general-purpose processors such as Intel's MMX technology.

The encryption algorithm chosen for this study is Rijndael [25], which was named the Advanced Encryption Standard (AES) in the year 2000 by the National Institute of Standards and

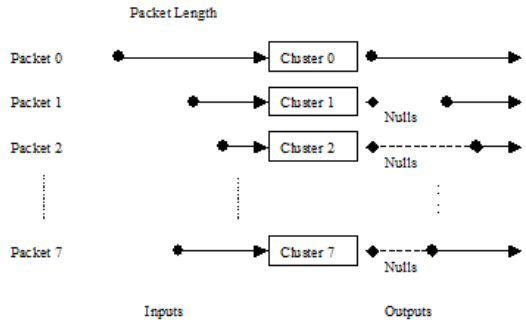

Figure 2.1: Each incoming packet is processed by a different cluster.

Technology (NIST). Since its selection, Rijndael has been implemented on all kinds of platforms [81], and it has proven to be the one of the fastest and most versatile algorithms. It has been applied to a variety of network protocols, including IPSec [52] and iSCSI [64].

In this paper, the AES algorithm is realized using the Stream Programming Model [101] and simulated on the Imagine Stream Architecture platform. The simulation results show promising performance. We first analyze the SIMD mode of computation as applied the variety of packet sizes present in a real-world Internet packet trace. Then, the implementation of the encryption algorithm in the two-level Stream Programming Model is introduced. Several experiments are presented for encryption in ECB mode with key agility (i.e., a different key for each packet). OCB mode [106] is also implemented, and its performance analyzed.

2.2 System Level Analysis

The SIMD architecture achieves the highest speedup and efficiency when working on regular and purely data-parallel structures. One of the major challenges in applying SIMD processing to packet encryption is dealing with the control variations introduced by varying IP packet lengths.

Assuming there are no inter-packet dependencies among the incoming packets 0 to 7, shown in Figure 2.1, each cluster can independently encrypt its packet at the same time. However, due to the packet size variation, the efficiency of processing valid data is degraded. The shorter packets have to wait until the end of processing the longest one, since each cluster operates on a single instruction stream. Applying load balancing techniques is a good way to improve the

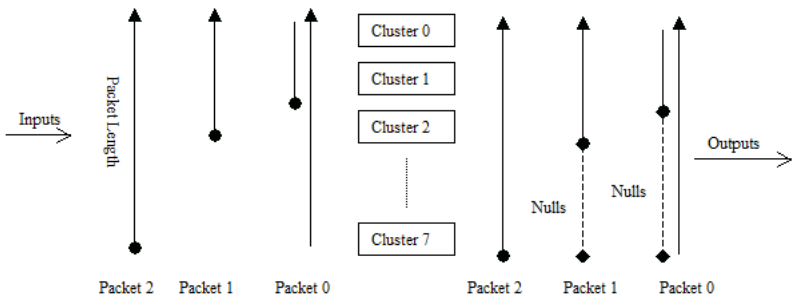

Figure 2.2: Distributing the data in the same packet to different clusters.

efficiency and processing time. For example, cluster 2 in Figure 2.1 can process part of the data in packet 0. However, a complicated reorder mechanism has to be applied on the interleaved data among each cluster to reconstruct the packets where the load balance is applied. Therefore, extra latency is expected as well. Furthermore, the packet data has to be either laid out in an interleaved order or a special index formulation needs to be constructed to form the input data stream.

Another way of encrypting the incoming packets is shown in Figure 2.2, where blocks (16 bytes) of data from a single packet are distributed to each cluster. In this way, the order of the incoming packet sequence can be maintained without extra effort. The preservation of packet sequence without degrading performance of parallel processing is a key technical challenge [12] in network processor design.

However, clusters are idle if packet data does not have enough blocks and it gets worse as the number of clusters increases. Therefore, the performance scalability by increasing the clusters is restricted due to the packet size variations if no load balance technique is applied.

A quick, first degree analysis on the packet trace[1] is shown in Figure 2.3 and Figure 2.4 labeled as _Mixed in the suffix. The analysis is based on a simple model: given a packet size, the processing time is decreased proportional to the number of clusters provided. In other words, the processing time T for a single cluster on a 1K-byte packet will be decreased to T/2 if two clusters are provided. Therefore, the speedup is doubled. The efficiency is defined as the average ratio of the number of clusters processing valid data over the total number of clusters involved (the clusters working on valid data + the clusters working on null data), as shown in Figure 2.2.

[1]TXG-1054945463-1, where 21.3% of the packets are less then 64 byte, and 32% of the packets are larger than 1000 bytes; total number of packets=1,412,513. National Laboratory for Applied Network Research (NLANR). http://moat.nlanr.net/pma

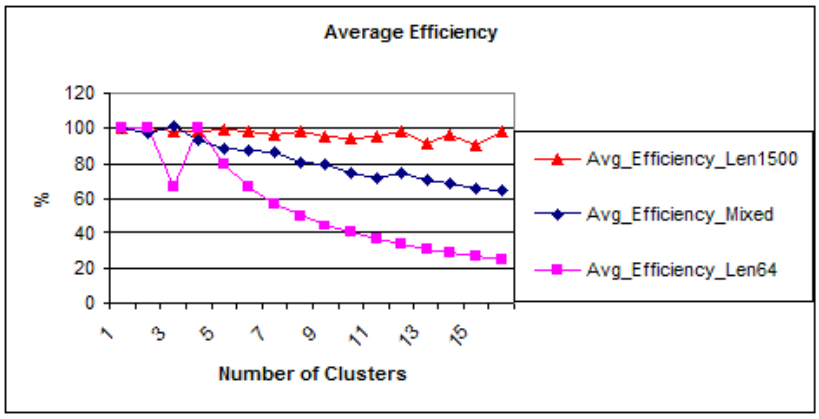

Figure 2.3: The average efficiency with different number of clusters.

The average efficiency drops below 80% if more than 8 clusters are provided, and there is only ~25% additional speedup when increasing from 8 clusters to 16 clusters. These figures also show the worst-case and best-case scenarios where every packet is 64 or 1500 bytes, respectively.

Load balance techniques can be applied to concatenate each packet as a single data stream instead of multiple ones. A relaxed SIMD architecture, the XIMD [124] where multiple instruction streams are allowed to be executed simultaneously in the machine is proposed to tackle the control flow variation issue. Another way is to explore Task Level Parallelism (TLP) by using multiple processors instead of scaling the clusters internally. Further explorations on load balance are proposed as promising directions for future work.

2.3 Implementation

2.3.1 AES Encryption Algorithm

Rijndael has the versatility of taking three different sizes (128, 192 and 256 bits) of data and key. The standard adopts the data block size of 128 bits, while the key size can be any one of the three sizes. For this work, we choose the size of 128 bits for both the key and the data block. The main loop of the cipher body consists of a certain number of rounds, which is a function of data block and key size, as shown in Table 2.1. The round number will be 10 given the size of 128 bits

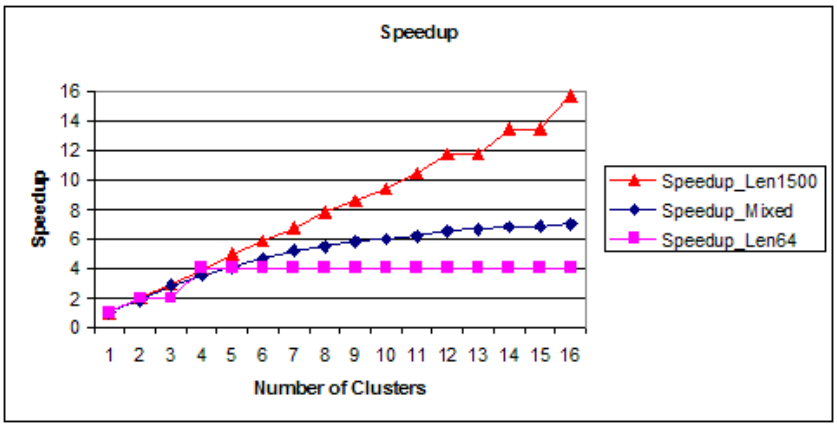

Figure 2.4: The speedup of different number of clusters.

Table 2.1: The number of rounds as a function of the block and key length. Nb represents the block length (32-bit words) and Nk is key length (32-bit words).

Number of Rounds	Nb=4	Nb=6	Nb=8
Nk=4	10	12	14
Nk=6	12	12	14
Nk=8	14	14	14

for both the data block and the key in this implementation.

There are four major functions within each round: *SubBytes*(), *ShiftRows*(), *MixColumns*() and *XorRoundKey*(). Following the main loop is the final round where only *SubBytes*(), *ShiftRows*() and *XorRoundKey*() is applied. There is a very efficient way of implementing the cipher by using a lookup table, known as the T-table [4], on a 32-bit processor. The T-table is the result of one complex transformation on *SubBytes*(), *ShiftRows*(), *MixColumns*() and *XorRoundKey*(). Hence, the main loop (without the final round) of encryption process can be done in a table lookup fashion as shown in the following equation.

$$\begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} = T_0\left[S_{0,c[0]}\right] \oplus T_1\left[S_{1,c[1]}\right] \oplus T_2\left[S_{2,c[2]}\right] \oplus T_3\left[S_{3,c[3]}\right] \oplus W_{round \times N_b + c} \qquad (2.1)$$

```
...
while_VARIABLE( i<ITERATION) {
loopIter();
if_VARIABLE (go_flag==0) {
      cout << "Compute a new round key set!\n";
      aes_key=aes_key_mem(i, i+8 , im_var_pos);
       key_expansion(aes_key);
      go_flag=1;
 }
      ...
cout << "Processing Packet# "<<i<< "/" <<ITERATION<<"\n";
packet_data=data_block(i*packet_data_size, (i+1)*packet_data_size, im_var_pos);
core(packet_data, key_source_index, core_out);
final_round(core_out, data_out);
streamCopy(data_out, data_to_mem(i*packet_data_size, (i+1)*packet_data_size));
i=i+1;
}//while
```

Figure 2.5: An excerpt from the StreamC code for the encryption process. Kernel invocations are shown in bold-italic.

The parameter of $S_{r,c}$ represents the cipher state with the row number r and column number c, where $0 \leq r < 4$ and $0 \leq c < Nb$. Each T-table (T_i) is a rotated version of the previous T-table (T_{i-1}). Therefore, with the expense of an extra rotation operation, storing only one T-table is enough. A detailed derivation of the T-table is shown in the proposal [25]. For the final round, the S-box (substitution table used in the *SubBytes*() function) has to be used instead of the T-table, due to the absence of the *MixColumns*() operation [25]. The S-box is not implemented in our work in order to save space in the scratchpad. Rather, the value S-box can be derived by an extra mask operation on the T-table value.

2.3.2 Programming Model

The stream-programming model consists of two major levels, i.e., the Stream level and the Kernel level. At the Stream level, StreamC is used to orchestrate the flow of data streams as well as the invocation of kernels. A snapshot of the StreamC code is listed in Figure 2.5.

The operation flow diagram is shown in Figure 2.6. Both the input key and the data stream consist of a collection of records. Each record serves as the building block of the stream and is defined as a data type consisting of four 32-bit words. The input key stream and data stream have to contain a number of records that is a multiple of the number of the clusters. In other words, the minimum number of records in the input key stream is eight for a system with eight clusters. Given

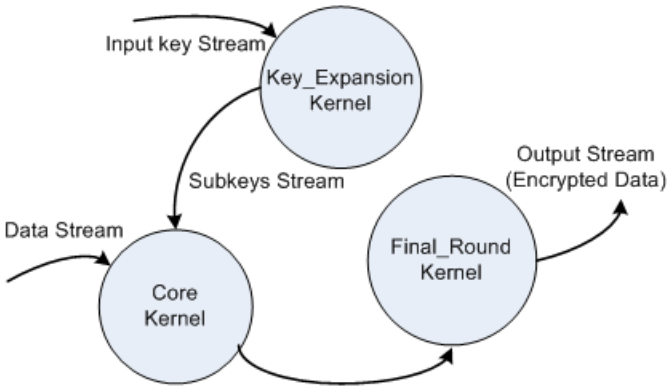

Figure 2.6: The stream level diagram of the encryption module.

```
kernel core(istream<BLOCK>in_1, uc<int>& key_source_idx, ostream<BLOCK>data_out) {

    target2_T0=     Ttable1[(shuffle(b_words.y,3)& 0xff)];
    target2_T1=rot(Ttable1[(shuffle(b_words.z,2)&0xff)],-8);
    target2_T2=rot(Ttable1[(shuffle(b_words.w,1)&0xff)],-16);
    target2_T3=rot(Ttable1[(shuffle(b_words.x,0)& 0xff)],-24);
    data_out_tmp.y = (target2_T3 ^ target2_T2 ^ target2_T1 ^ tar-
    get2_T0^ key_hold[idx+2]);

}
```

Figure 2.7: An excerpt of the core kernel code, written in KernelC.

a key stream with eight records, the subkey stream will contain 88 records[2] in an interleaved form after the key expansion process.

The major computation is done in the Kernel level, where KernelC is used and compiled as VLIW instructions for clusters. Figure 2.7 shows a fragment of the code in the core kernel.

The IScd scheduler provides two hints [30] for loop optimization at the kernel level. The UNROLL(n) command will instruct the scheduler to unroll the loop body n times. The modulo software pipeline command PIPELINE(startII) is a technique where a loop is divided into n stages and different stages of n iterations are executed at once. The code size expands proportionally as the amount of unrolling increases. Therefore, the time for the on-chip microcontroller to load

[2]More details are provided in Section 2.5.

the microcode from the host processor increases as well. The microcode store [27] is limited to 1024x512 bits. If the space is not enough for holding all the kernels, then the microcode has to be reloaded again during runtime, and the performance will be degraded due to the extra latency of the loading process.

2.4 The *Core* and *Final_Round* Kernels

The AES encryption operation contains two major kernels. The core kernel consists of the inter-cluster communication for subkeys, T-table lookup, and the arithmetic operations to encrypt a block. The core kernel will take the subkey stream and store eight sets of the subkeys in the scratchpad of each cluster. Extra inter-cluster communications are needed to transfer the subkeys if each cluster is encrypting the data block with the same set of subkeys. The final_round kernel is implemented such that an extra rotate and mask instruction is applied to the T-table to derive the S-box value for the byte substitution transformation. Following the *ShiftRows()* and *XorRoundKey()* operations, the encrypted data will be sent out as a data stream. Originally, on the Imagine Processor, each cluster contains a single 256-word scratchpad register file, so that each cluster has the capability of supporting coefficient storage, short arrays, small lookup tables and some local register spilling [101]. For our simulations, the size of the scratchpad is changed to 512 words, in order to accommodate the T-table and the other array variables used in the kernels. The core kernel consumes 72.5% of the whole encryption cycle. In the core kernel, 216 read operations are found out of 252 scratchpad accesses. The scratchpad has one output and three input units, which allows simultaneous read and write access [27]. However, the ratio for read and write accesses to the scratchpad in the core kernel is 6 to 1, since only read access is needed in the main round operation. Among the 216 scratchpad read operations, 180 of them are located in Basic Block 4 of the core kernel, where the T-table lookup is performed. As shown in the first row of Table 2.2, the cycle counts in the critical path are simply saturated, regardless of the number of adders provided. The critical path can be reduced up to 15% by adding an additional scratchpad to allow concurrent reads of the T-table. Therefore, a second scratchpad is implemented and added into the machine description file to hold the second T-table, such that two simultaneous read accesses can be provided. The entire simulations in this chapter are based on the configuration of two 512-word scratchpads.

A certain number of overhead cycles have to be paid to setup the constants and loop initialization inside a kernel. Therefore, if more data blocks can be processed with a fixed overhead,

Table 2.2: The critical path of Basic Block 4 in the core kernel with different machine configurations.

	Adder 3	Adder 4	Adder 6
1 Scratchpad	25	25	25
2 Scratchpads	21	17	17
4 Scratchpads	n/a	13	12

then the performance can be increased while more ILP and DLP can be exploited. The dual version of the core and final_round kernels are implemented such that each cluster is capable of encrypting two data blocks at the same time if enough hardware resources are available. The speedup of the dual version over the single one is shown in Figure 2.16 and is discussed in Section 2.6.

2.5 The *Key_Expansion* Kernel

The *key_expansion* kernel is based on the AES Key Schedule algorithm [25]. For a block size and key size of 128 bits, the number of rounds is equal to 10 while the number of columns of the *Cipher Key* N_k and *State* N_b is equal to 4. The operations of the AES Key Schedule algorithm are shown as pseudo code in Figure 2.8.

The round keys are based on the initial input key, denoted by W[0], W[1], W[2] and W[3] in a sequential fashion. The function of *RotByte()* will generate the result of word(b, c, d, a) from the original word(a, b, c, d). The *SubByte()* function is the Rijndael S-box transformation where each byte of the original input word will be replaced.

Due to the sequential nature of the algorithm, the *key_expansion* kernel is implemented such that each cluster can take one key for processing. Therefore, with eight clusters, the processor can generate up to eight different sets of subkeys at the same time. As shown in Figure 2.9, the kernel

```
for (i=4;i<44;i=i+1) {
   temp=W[i-1];
   if ( i mod 4 ==0)
      temp=SubByte(RotByte(temp)) xor Rcon[i/4];
   W[i]=W[i-4] xor temp;
}
```

Figure 2.8: The pseudo code of the AES key schedule algorithm for block size and key size equal to 128 bits.

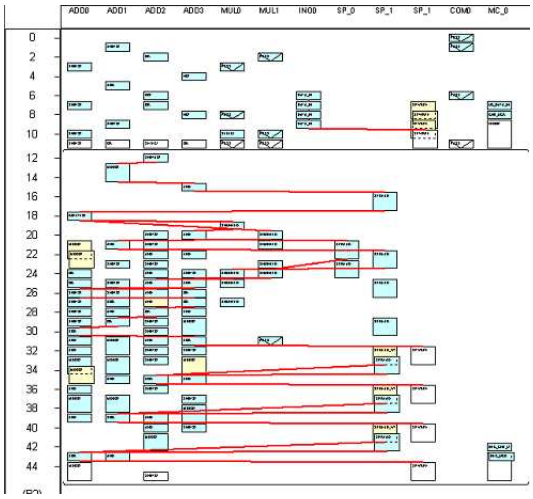

Figure 2.9: The scheduling result of the key expansion kernel with the critical path being highlighted in lines.

is compiled by the IScd Kernel Scheduler [109] with the critical path highlighted in a sequence of lines. The kernel consists of two basic blocks. The first basic block saves the incoming key stream into the scratchpad. The main key expansion loop is in the second block. The effective parallelism achieved in basic block1 is only 2.79, with total run time of 379 cycles. The effective parallelism is defined as the ratio of the total number of instructions per block to the number of cycles in the critical path [17]. As indicated by the effective parallelism, the kernel does not fully utilize the ALU resources provided.

Given the same hardware configuration with eight clusters in the Imagine, the ILP can be increased by simply processing 2 different keys at the same time in a cluster. A dual version of the key_expansion kernel is implemented, in which there are up to 16 different sub-keys calculated at the same time. In Figure 2.10, the IScd scheduling result shows that the processing capability is doubled with a 24.2% increase in kernel run time (4-adder configuration) while achieving an effective parallelism of 4.5. As expected, given the 6-adder configuration, there is no cycle count increase compared to that of the single version with 4 adders. The IScd compile result with 6-adder configuration is shown in Figure 2.11.

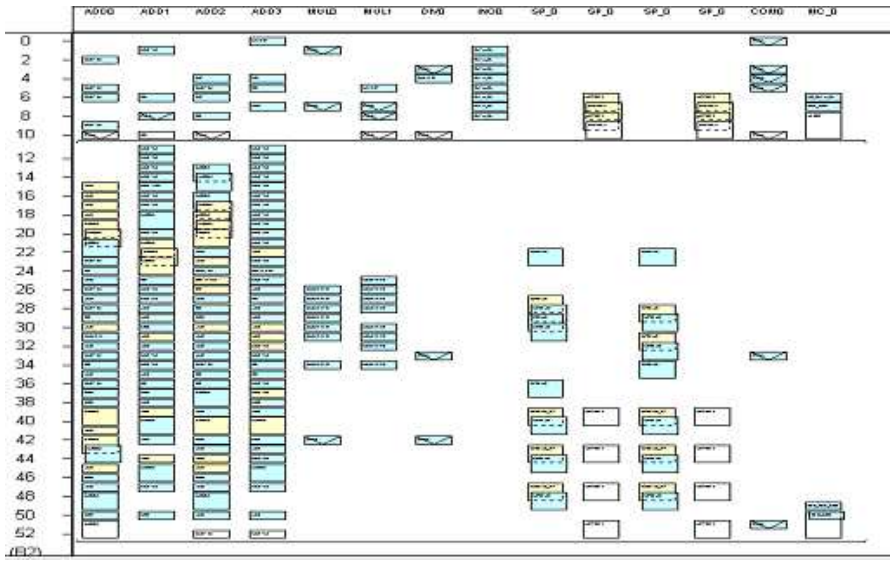

Figure 2.10: The IScd compile result with 4-adder configuration for dual key expansion kernel.

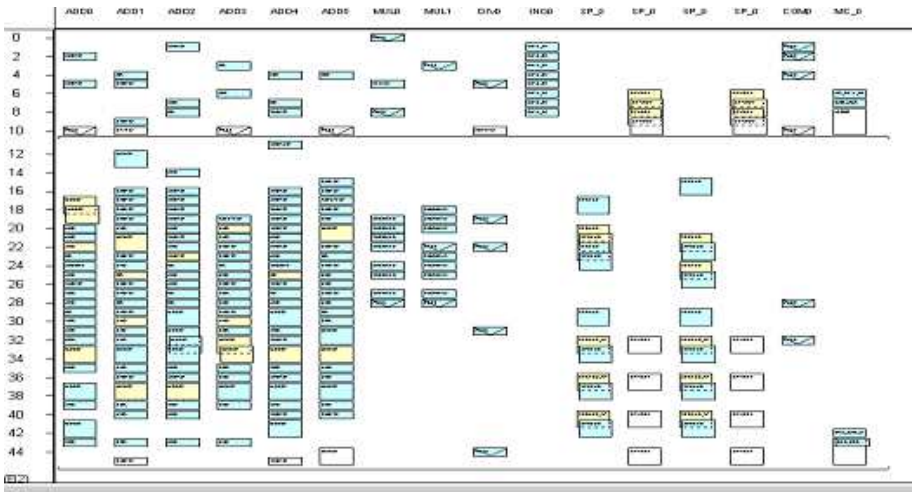

Figure 2.11: The IScd compile result with 6-adder configuration for dual key_expansion kernel.

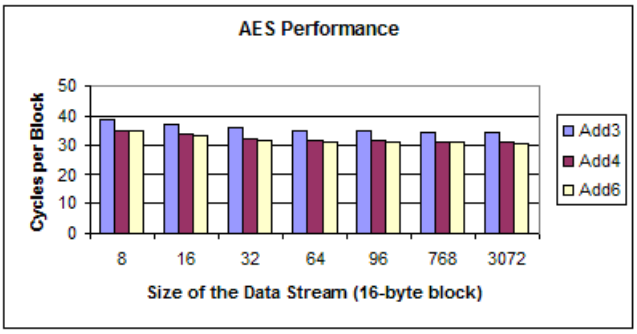

Figure 2.12: The performance of the AES encryption on a single stream of data. The size of the stream ranges from 8 to 3072 blocks (128 to 48k bytes).

2.6 Experiments and Discussions

The first set of simulation experiments encrypts a single stream with different sizes. As shown in Figure 2.12, the stream size ranges from 8 to 3072 blocks (128 to 48k bytes). A key stream consisting of eight identical (128-bit) keys is sent to the key_expansion kernel. The output of the key stream is then directed to the core kernel for encrypting the data stream. All eight clusters use the same subkeys to encrypt the data stream. The cycle counts for encryption are measured by subtracting the time for loading the microcode and key expansion from the total cycles. Three different machine configurations are applied during the simulation. Add3 is the original Imagine machine description file, which has three adders in each cluster. The add4 and add6 configurations increase the number of adders to four and six, respectively. For all three configurations, there are two 512-word scratchpads.

The best performance for a stream size of 96 blocks (1536 bytes), as shown in Figure 2.12, is 31.5 cycles per block and the throughput is 2.02 Gbps with a system clock of 500 MHz. The performance with a small data stream suffers from the short stream effect [92]. A fixed amount of cost has to be paid before and after the main loop inside a kernel; therefore, if the size of the stream is short, the fixed cost cannot be amortized among the run time. The cost is associated with the variable initialization, constant setup, etc.

Another set of simulations is conducted such that multiple numbers of packets in the size ranging from 8 to 96 blocks is sent into the kernel. The total amount of data is 61,440 blocks (960K bytes), which is 7.5 times larger than the stream register file, meaning that packet data must be transferred from DRAM to the SRF during the calculation. Therefore, if 16 blocks are picked as

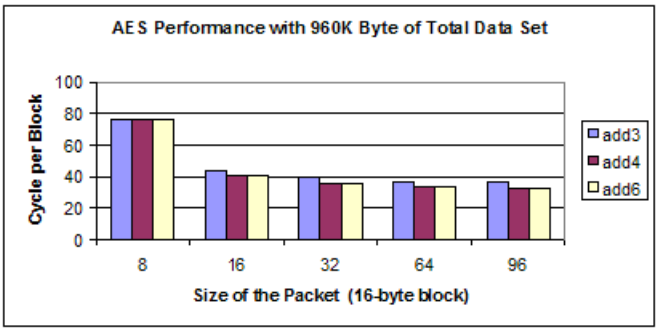

Figure 2.13: The AES performance with multiple fixed-size packets.

the size of a packet, then the total number of packets being processed will be 3840. The simulation results are shown in Figure 2.13 where the size of 96 blocks has the best performance. The results are close to those for encrypting a single packet, which shows that the DRAM access is effectively hidden by computation.

In order to conduct the key agility simulation, the original kernel codes are modified to be able to transfer the subkeys within clusters. A fixed cost for inter-cluster communication is imposed on the core kernel. Therefore, due to the communication overhead and the short-stream effect described earlier, the performance further decreases with the packet size of 8 blocks. Compared to those in Figure 2.12, another factor of the performance decrease is due to the stream derivation and stream copy operations where another layer of overhead at the stream level is imposed.

The purpose of this setup is to have a full duplex stream flow occurred between the SRF and the main memory. Therefore, the effectiveness of hiding the latency with the kernel computation can be observed. Figure 2.14 demonstrates the ratio of the kernel run time over the total run time. The total run time consists of the stream operations, stalls and kernel run time. For the packet size of eight blocks, the kernel takes only 60% of the total run time. However, as the packet size increases, the kernel runtime can take up to 98% of the total run time.

Figure 2.15 shows the occupancy of the functional units encrypting a single 16-byte block of data for two different architecture configurations. As more adders are provided, the total execution time decreases. Therefore, the percentage for the scratchpad, multiplier, divider and communication unit increases. The occupancy for adders decreases simply due to the instructions being distributed to the extra adder provided. The multiplier units take the instructions of *select* and *shuffled*. The divider unit and the two multiplier units can be replaced with adders which also provide

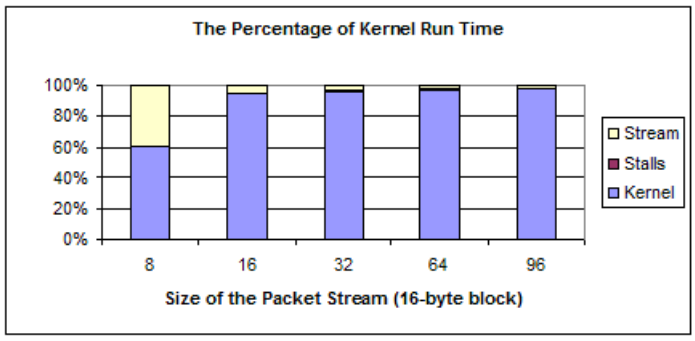

Figure 2.14: The percentage of the kernel run time within the total run time.

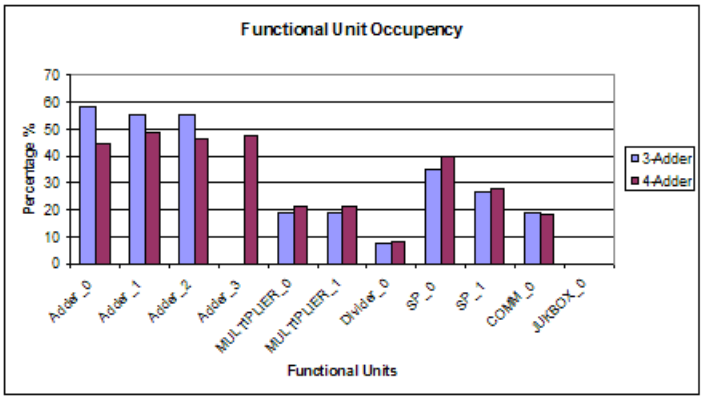

Figure 2.15: Occupancy of the functional units.

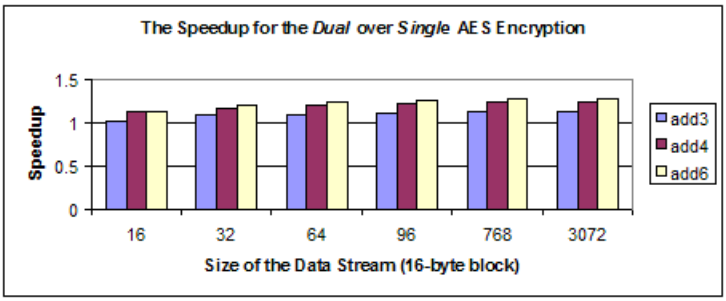

Figure 2.16: The speedup of dual over single AES with different sizes of data stream.

the same operations so the area can be saved. The scratchpad utilization is not symmetric. This is mainly due to some array variables used other than the main T-table lookup.

Figure 2.16 shows the speedup of *dual* over *single*. The speedup can reach beyond 1.2 if the size of the data stream is larger than 64 blocks in the 4-adder or 6-adder configuration.

2.6.1 Key Agility

For a security gateway router, where the encryption processes have to serve for multiple sessions of users, there exists a worst-case scenario that every incoming packet has to be encrypted by a different key. Therefore, the ability for a system to efficiently handle the key changes without degrading performance is a critical performance factor.

One of the commonly used schemes [108] is to compute the round key expansion on-the-fly in pipelined fashion. This is done with dedicated hardware to process the key expansion in time less than or equal to that of processing the minimum-sized packet. Therefore, the latency can be hidden without affecting the overall throughput while achieving high key agility. Another scheme is to pre-compute the round keys in advance, as soon as the security parameters for a flow are established, before the actual messages arrive. However, the drawback is that the memory storing these expanded round keys has to be increased in proportion to the ratio of the expansions. Furthermore, the memory bandwidth has to be expanded as well.

The easiest way is to do the pre-computation. Once the flow is established through the key management protocol, the round key expansion process can be started. The input key stream will first be loaded into the SRF and later on processed by the key_expansion kernel. The subkey output

stream will reside in the SRF temporarily and then spill out to the DRAM. Afterwards, the subkeys can be loaded by an index stream for the encryption on the incoming packets. The latency of loading the sub-key stream from the memory can be overlapped with the core kernel computations. Based on the AES standard, 10 rounds are needed for a key size and block size of 128 bits. Hence, the output stream from the key_expansion kernel, which contains the expanded subkeys, is 10 times larger than the size of the original input stream. The storage of the sub-keys definitely has a huge impact on the system cost, and there is increased demand on memory bandwidth, as well. On-the-Fly subkey computation means to calculate the subkeys needed for a particular round just before using them in the round [81]. Due to the SIMD architecture, all the clusters are processing the blocks from the same packet based on the same key. There is no reason to implement the on-the-fly subkey computation inside the core kernel calculating the same subkeys. Furthermore, all the hardware in the clusters is dedicated to executing the VLIW instructions broadcast from the microcode store. Therefore, one kernel at a time is being executed. There is no way to take the pipeline scheme where the key_expansion and the core kernel are being executed at the same time as describe above.

Another way is to expand only the sets of subkeys that are going to be used soon. Based on the assumption of a store-and-forward architecture, in which the incoming packet will be stored in the data memory, it is possible for a host processor to look-ahead into the control memory to identify the next eight packets that are going to be processed. Similar to the previous discussion, the host processor can initialize the key stream, which contains eight different keys, to the key_ expansion kernel before the packet encryption begins. After the keys are expanded, all the sub-keys are stored inside the Scratch Pad of cluster 0 to 7, where cluster 0 has the first set of sub-keys, cluster 1 has the 2nd set of sub-keys, and so forth. Using the inter-cluster communication network, each set of sub-keys can be broadcast to all the clusters; therefore, all the clusters can process the blocks of the same packet with the same subkeys. After the end of processing the 8th packet, the key_expansion kernel will be executed again to calculate the next 8 subkeys for the packets to be processed. The key_expansion kernel takes about 381 cycles to process 8 different sets of round keys. As discussed in the Section 2.5, the dual version of the key_expansion kernel can process up to 16 different sets of subkeys at no extra cycle increase given enough hardware resources (the 6-adder configuration). Therefore, the key_expansion kernel needs to be executed only once every 16 packets instead of eight.

The simulation is setup such that a fixed amount of data (61440 blocks) is given. A packet stream is derived with the sizes ranging from 8 to 96 blocks, as shown in Figure 2.17. The *key_expansion* kernel is executed once every eight packets since eight different keys can be

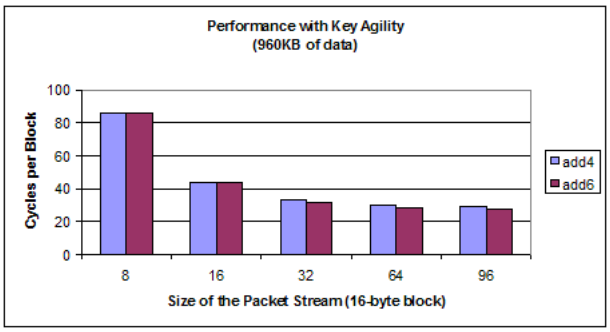

Figure 2.17: Key Agility Performance. (in 4-adder, and 6-adder configurations, code optimized).

Table 2.3: The characteristic matrix of AES encryption.

	Simulation Result
Memory_BW (GBps)	1.12
SRF_BW (GBps)	2.23
LRF_BW (GBps)	239.90
GOPS	20.66
Operation per Mem Reference	73.94

expanded at the same time. Therefore, for a packet size of 8 blocks, there are 7680 128-byte packet streams being sent into the clusters, and 960 key streams (each containing eight 128-bit keys) are consumed by the *key_expansion* kernel. The worst-case scenario is for the packet size of eight blocks, since the *key_expansion* kernel has to be executed more frequently. The run time for the *key_expansion* kernel is 381 cycles; therefore, on average, an extra six cycles per block will be the overhead over encrypting with a single key. The core kernel consumes a fixed amount of time to transfer a set of subkeys (44 words) from the scratchpad in the cluster. As the packet size gets smaller, the overhead is obvious. This overhead is in addition to the short stream effect, discussed earlier.

Table 2.3 shows some standard Imagine performance metrics: bandwidth for each of the three levels of memory hierarchy, billions of operations per second (GOPS), and the number of operations performed per memory reference. The measurement is performed where 640 1536-byte packets are encrypted with a new key for each packet. The whole operation includes 80 key_expansion kernel invocations.

The best efficiency can be achieved only in the case where all eight clusters are processing 8 or 16 different keys. On the other hand, if only one new key is needed while the other 7 or 15 keys remain the same, the efficiency is the worst because the same calculation is repeated again. To add an extra layer of memory between the SRF and the Clusters to serve as a subkey cache is another way to improve the performance and the efficiency where the existing sub-keys can be re-used. However, this might need a large cache size to achieve a satisfactory hit rate [108].

2.6.2 The simulation of variable-sized packets

The main purpose of this experiment is to characterize the system performance with variable-sized packet streams based on the length distribution information from an existing Internet trace.

2.6.2.1 The Internet Trace

The trace (AIX-1054837521-1) [89] used for this simulation was collected from the NASA Ames Internet exchange (AIX) [88] in Mountain View, California. It is collected from one of four (now five) OC-3 ATM links that interconnect AIX and MAE-West in San Jose. As we can see in Figure 2.18, almost 50% of the packets are under the size of 128 bytes where only less than 6% of the total bandwidth is taken. On the other hand, there are almost 22% of the packets with the size of 1500 bytes taking more than 75% of the total traffic. A large proportion of this TCP traffic is generated by bulk transfer applications such as HTTP and FTP [90]. Consequently, the majority of the packets seen are the minimum packet size for TCP acknowledgements and the 1500 byte packets with the maximum Ethernet payload.

A set of Perl scripts were implemented to parse the Internet trace file obtained from NLANR [88] and generate two input files for the simulation. The packet data file is constructed with random payloads where proper paddings are inserted. Therefore, the size will be the multiple of eight blocks, which is the number of the clusters in the system. The length distribution for the simulation is shown in Figure 2.19. Another file consisting of the packet lengths is given to the host processor. The host processor derives the packet stream out of the packet data based on the length information and orchestrates the overall stream operation.

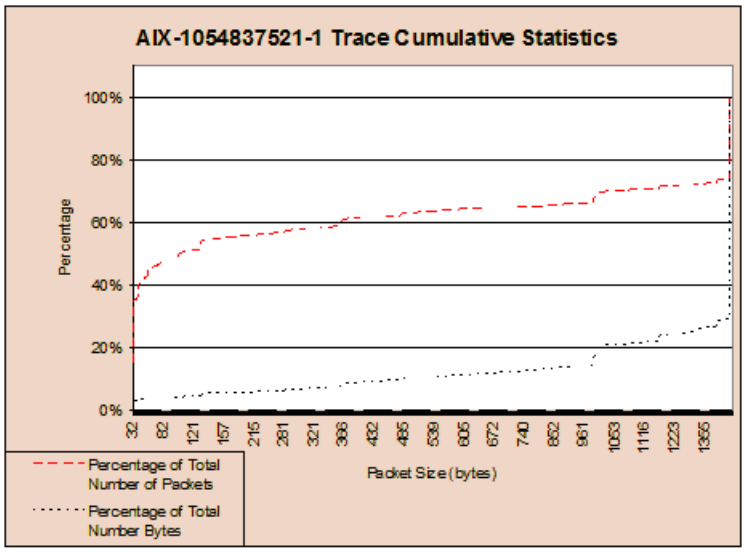

Figure 2.18: The accumulative statistics for the AIX Internet trace.

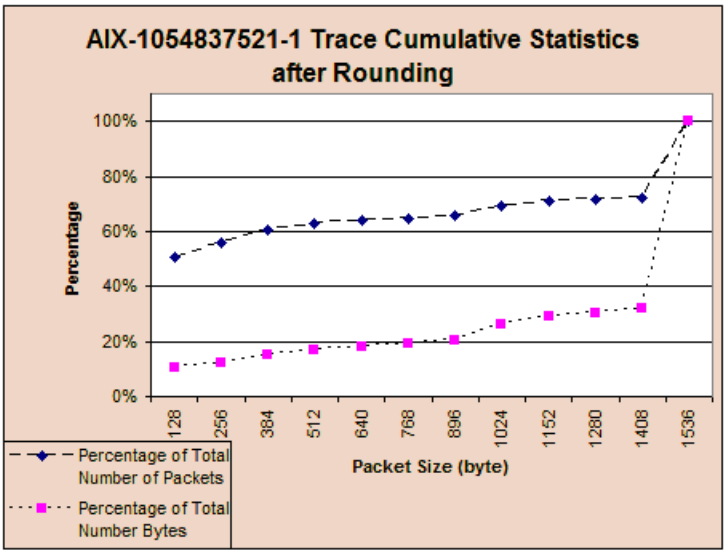

Figure 2.19: The Trace statistics after rounding to multiple of total cluster number.

```
#define MAX_PKT_SIZE 96 //set 96 blocks as the max pkt size (1536 bytes)
STREAMPROG(aes);// defining stream program
void aes(StreamSchedulerInterface& scd ,String args){
im_stream<BLOCK>NAMED(packet_data)=
  newStreamData<BLOCK>(MAX_PKT_SIZE,im_countup);
im_stream<BLOCK>NAMED(add_round_key_out)=
  newStreamData<BLOCK>(MAX_PKT_SIZE,im_countup);
im_stream<BLOCK>NAMED(data_out)=
  newStreamData<BLOCK>(MAX_PKT_SIZE,im_countup);

        ...

while_VARIABLE( i<ITERATION) {

        loopIter();
        packet_data_size=length_array[i]/16;
        cout << "Processing Packet# "<< i<< "/"<<ITERATION<<"\n";
        cout <<"packet_size="<<packet_data_size<<"blocks"<<"\n";
        streamCopy(data_block(delta,lengthpacket_data_size+delta),packet_data);
        core_s(packet_data, key_words_b,add_round_key_out);
        final_round_s(add_round_key_out, data_out);
        streamCopy(data_out,data_to_mem(delta,packet_data_size+delta,im_var_pos));
        delta=packet_data_size+delta;
        i=i+1;

}//while
}
```

Figure 2.20: The code fragment of the StreamC main loop.

2.6.2.2 The variable-sized stream

Since the system has no idea of the stream length at the beginning of the declaration phase, a straightforward way is to specify a count-up stream, im_countup with a maximum size of 96 blocks (1536 bytes). A maximum size of 96 blocks is given simply due to the MTU of the Ethernet protocol.

A count-up stream is declared where the end of a stream varies, depending on the number of records produced by a stream operation. It is a variable length stream which contains zero records initially. The size will be set once the dependency is resolved by stream operation or from the results produced by a kernel.

The packet length information is passed from the host processor to the Imagine Stream processor by using the array variable. As shown in Figure 2.20, the length of a packet is used to derive a stream from the main memory and sent to the following kernels for encryption. The output of the packet stream is sent to the SRF and stored back to the main memory finally.

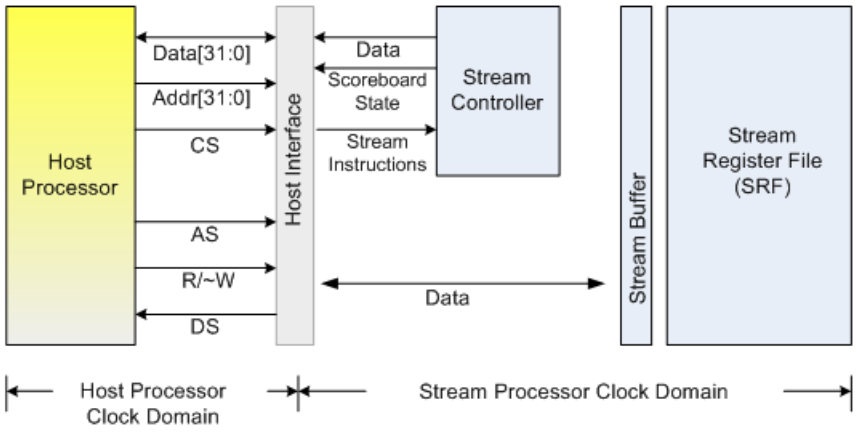

Figure 2.21: The block diagram of the Host Interface, SRF and Stream Controller [27].

The use of the count-up stream raises another issue where the performance is degraded due to the dependency resolution. This is because the memory space within the SRF can not be allocated until the size of the derived count-up stream is known. Unfortunately, the latency caused by the host processor operation can not be hidden effectively. Therefore, the system has to wait until the information is passed to the control logic for further operation. The extra time needed is approximately 400 cycles. The fixed overhead is associated with the transactions where the host processor issues the stream instructions to read and configure the Stream Description Registers (SDR) for the count-up stream.

As shown in Figure 2.21, the host processor issues several stream instructions such as move and write_imm listed in Figure 2.25, to the Stream Controller through the Host Interface. Since the data bus is only 32-bit wide, multiple Host Processor bus transactions are needed for issuing a single stream instruction. The buses at the Host Interface are operated at lower clock speed (200 MHz) compare to that in the system. The typical bus read and write transactions are illustrated in Figure 2.22 and Figure 2.23.

The stream instructions are stored in the Op Buffer within the Stream Controller. In addition, the Stream Controller also consists of pending instruction queue and control logics such as resource analyzer, completion detection unit and scoreboard for bookkeeping the resources and inter-instruction dependencies. Once the dependencies are resolved, the instructions will be issued by the issue and decode logic.

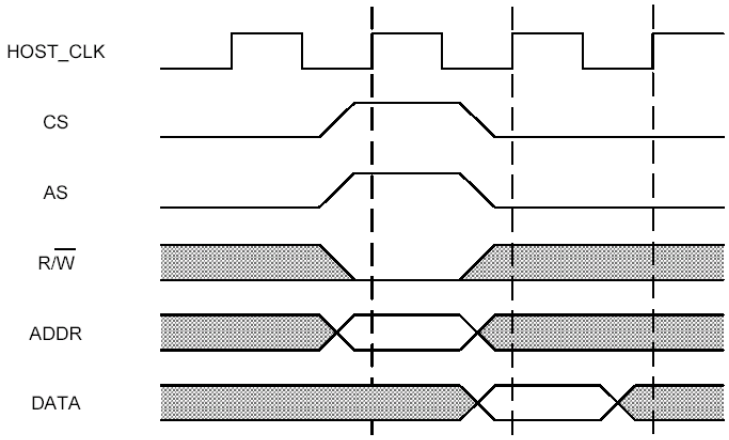

Figure 2.22: The Write operation in the Host Interface [27].

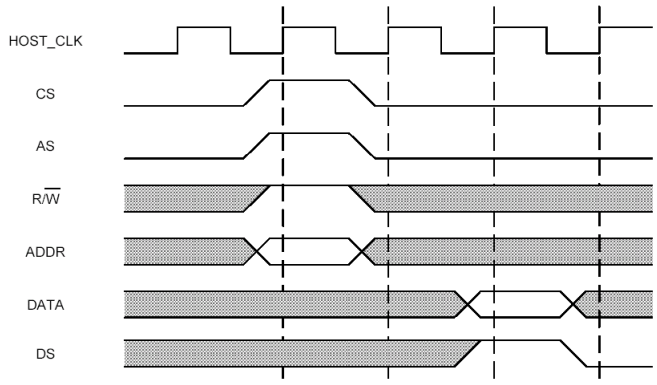

Figure 2.23: The Read operation in the Host Interface [27].

```
STREAMPROG(aes);// defining stream program
voidaes(StreamSchedulerInterface& scd ,String args) {
..
while_VARIABLE( i<ITERATION) {
  loopIter();
  packet_data_size=length_array[i]/16;
  cout << "Processing Packet# "<< i<< "/"<<ITERATION<<"\n";
  cout <<"packet_size="<<packet_data_size<<"blocks"<<"\n";
im_stream<BLOCK>NAMED(packet_data)=
  data_block(delta,packet_data_size+delta,im_var_pos)
im_stream<BLOCK>NAMED(add_round_key_out)=
  newStreamData<BLOCK>(packet_data_size,im_var_size);
im_stream<BLOCK>NAMED(data_out)=
  newStreamData<BLOCK>packet_data_size,im_var_size);
  core_s(packet_data,key_words_b,add_round_key_out);
  final_round_s(add_round_key_out,data_out);
  streamCopy(data_out,data_to_mem(delta,packet_data_size+delta,im_var_pos))
  delta=packet_data_size+delta;
  i=i+1;
  }//while
}
```

Figure 2.24: The code fragment without using the count-up stream.

There is another way to implement the stream operation by declaring the variable-sized stream within the main loop of the operation. Therefore, as shown in Figure 2.24, the stream length is available for three derived streams right at the beginning of the main loop. No extra cycles are paid due to the issue discussed before.

2.6.2.3 The Simulation Results

The simulation is similar to those shown in Figure 2.3 where the average efficiency analysis is done based on a larger Internet trace (TXG-1054945463-1). Because of the SIMD architecture, almost 10.1% of the byte counts are due to the padding and rounding process. Therefore, the performance calculation is based on the processing of effective blocks. Figure 2.26 shows the system can achieve 41 cycles per block for the AES encryption, compared to 72 and 35 in the fixed-size packet of 8 and 96 blocks, respectively. The ratio of kernel run time over the whole processing time is shown in Figure 2.27. A higher ratio means more effective hiding of the latencies (memory access and stream operation) with the real computation work. A 96.5% ratio is observed in the simulation.

operation	fields
All operations	*0 compl_mask 32 issue_mask 64 scorebd_assign 69 opcode 74*
move	74 dst_num 79 dst_type 82 src_num 87 src_type 90 uc_synch 91
write_imm	74 dst_num 79 dst_type 82 imm 143 uc_synch 144
barrier	--- (no arguments) 74
reset	--- (no arguments) 74
host_transfer	74 host_sdr 82
memop	74 data_sdr 82 idx_sdr 90 data_mar 93
load_ucode	74 pgm_sdr 82 mpc 93
clustop	74 sdr0 .. 130 sdr7 138 mpc 149
clust_restart	74 sdr0 .. 130 sdr7 138 uc_synch 139
synch_uc	--- (no arguments) 74
netop	74 net_sdr 82 net_sb 85 tag 90 vcmask 98 nrr 103
net_restart	74 net_sdr 82 net_sb 85

Figure 2.25: The format of Stream Instructions [27].

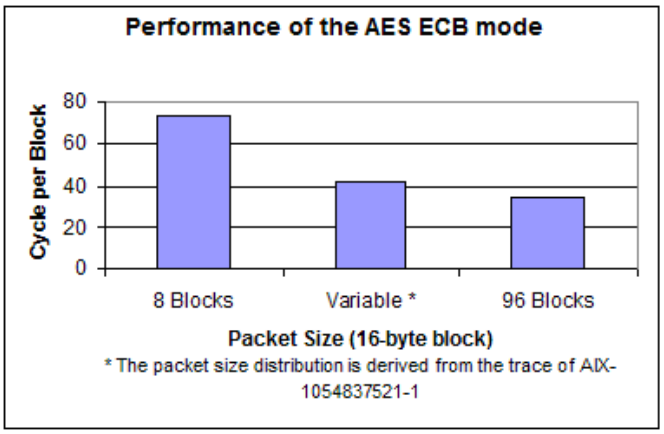

Figure 2.26: The performance of encrypting variable-size packets compare to those of the fixed-size ones.

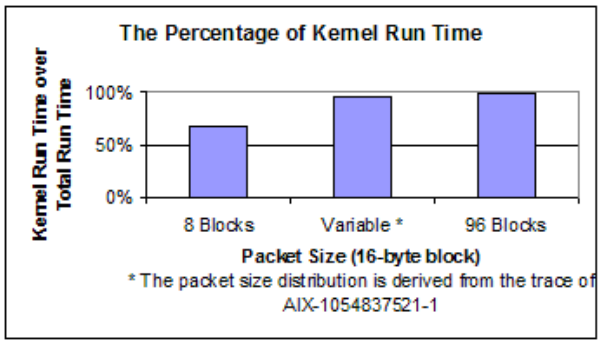

Figure 2.27: The percentage of kernel run time over total runtime.

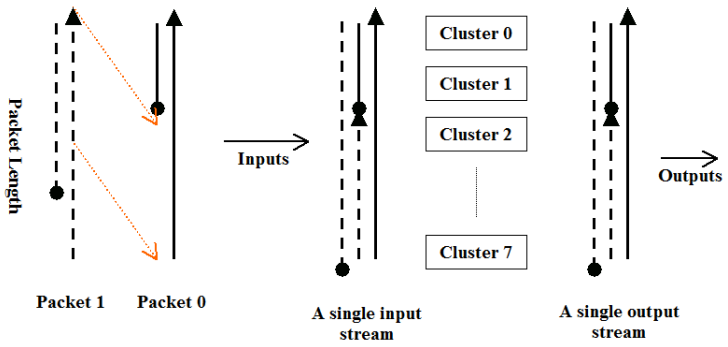

Figure 2.28: A load balancing scheme for small-sized packet encryption.

2.7 Load Balancing

As we can observe in the simulation results, the minimum-sized packet suffers the performance degradation due to the fixed overhead imposed on each incoming packet stream. In addition, the efficiency of the clusters is only 50% on processing a 4-block (64-byte) minimum sized packet.

The performance and efficiency can be improved by concatenating two or more packets as a single stream. Instead of sending each packet as a single stream, the fixed premium can be amortized and better performance can be achieved.

Figure 2.28 shows how two packets are merged as a single input stream. The system can encrypt the longer stream and avoid the short-stream effect which is mainly due to the nature of the stream language and the system operation.

Another major expense besides the short-stream effect is the sub-key distribution through the inter-cluster communication. As discussed earlier in the previous section, the process is to ensure each cluster can encrypt the packet based on the sub-keys associated with the incoming packets.

Figure 2.29 demonstrates the example where sub-keys have to be distributed for two packet encryption within a single stream. At the beginning of the encryption, eight sets of sub-keys are sent as a key stream into the kernel and each cluster will store a set of sub-keys into the scratchpad respectively. Then, based on the property of the incoming packet, each set of sub-keys will be broadcasted to the rest of clusters through the inter-cluster communication network. In Figure 2.29, the first set of sub-keys K0 is distributed to the working buffer B1 in all the clusters at iteration i=0. At the iteration of i=1, a second set of sub-keys K3 have to be distributed to the working buffer B2 so that cluster 3,4,5,6 and 7 can encrypt the second packet within the data stream. The working buffer B1 and B2 behaves like a ping-pong buffer holding the sub-keys currently used. Those are enabled based on the packet length provided through the stream variable issued from host processor passing into the kernels.

The size of the sub-keys is 44 words, based on the 128-bit key size for AES. Therefore, at least a total of 88 cycles (read and write access on scratchpad) are needed for the sub-key distribution, not counting the extra stream operations. This is almost 24.4% of the cycle time processing a minimize size packet.

Several extra operations are needed both at stream level and kernel level. The first is to provide the packet length information for the kernels through the microcontroller variable. Then, the kernels can identify the exact cluster ID and the processing iteration where the boundary of the packet within the data stream might be. These can be calculated easily by the packet length and the number of cluster. Therefore the distribution of the sub-keys can be conducted. Leveraging on the predicate instruction, select(), which is provided by the Imagine processor, the corresponding sub-key buffers can be used in different clusters for encryption.

The load balancing scheme can improve both the performance and efficiency in the stream level operation. However, counting on these extra operations, tradeoffs have to be taken care of such that the benefits gained by adopting the load balancing scheme will not be lost.

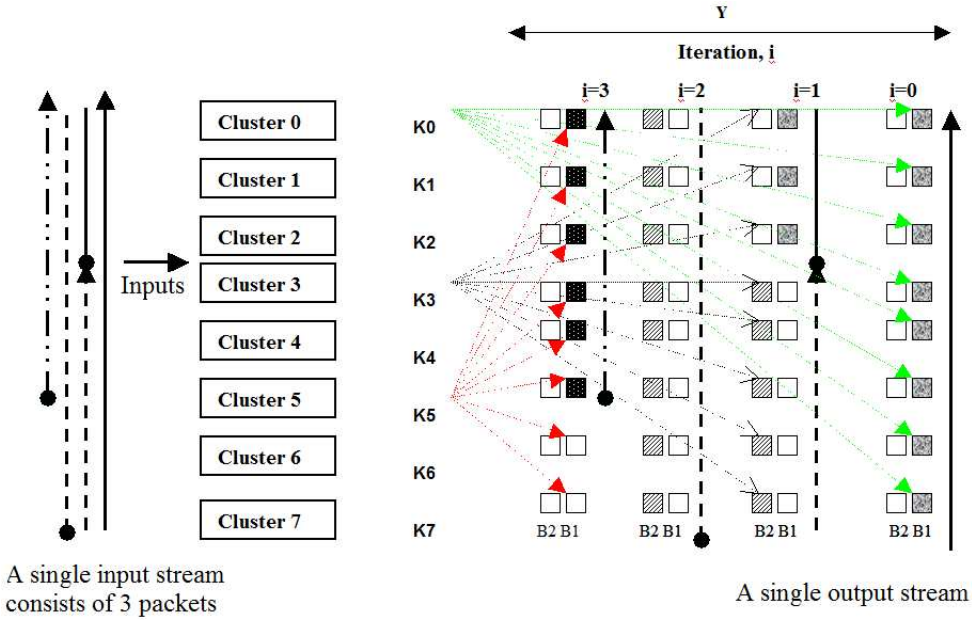

Figure 2.29: The sub-key distribution in the load balancing scheme.

2.8 The Mode of Operation

We have so far considered only the Electronic Codebook (ECB) mode of encryption. This allows each data block to be processed independently, in parallel. However, a particular plaintext will always be encrypted to the same ciphertext. Therefore a codebook can be obtained and the privacy will be compromised once the relation between the ciphertext and plaintext is known.

More sophisticated modes offer protection from repeated plaintext-ciphertext pairs, and some still allow packets to be processed in parallel. The Counter Mode (CTR) is one such mode that was recently added to NIST's approved list [85]. CTR mode demonstrates excellent efficiency in both of the hardware and software implementation [118]. Moreover, the security is well analyzed and proved as long as the counter block value is never reused with the same key. As proposed on the NISTs recent call for modes-of-operation [84] activity, the Offset Codebook mode (OCB) [106] and the Carter-Wegman + Counter dual-use mode (CWC) [56] can also be operated in parallel. We have implemented OCB mode and present the results in this section.

The implementation is based on the assumption that the maximum packet size is 1536 bytes (96 blocks). We further assume that the packet size is a multiple of 128-bit blocks with all 0s

being padded at the end of packet to form the total length in the multiple of eight blocks. The first step is to compute the *L(0)* and the first offset. The computation of the parameter *L* is based on the key and the function of *ntz(i)* where the number of trailing 0-bits in the binary representation of i is obtained. Eight of the *L* parameters (*L(-1),...,L(6)*) are precomputed since they are more than enough in our case. Since only 96 blocks of the offset is needed, the offsets are also precalculated along with the *L* parameters. The *offset_exp()* kernel is executed before the encryption process begins. The total of sixteen different sets of keys can be processed in the *offset_exp()* kernel. An indexed stream derived from the offset stream output from the *offset_exp()* kernel is then directed into the main kernel for encryption process. The offsets can also be calculated on-the-fly with the incoming message blocks. The loop for offset calculation is unrolled and optimized, as shown in Figure 2.33. The operations are distributed to eight clusters as shown in Table 2.4, with the bit position of 1 representing the value of *i* in *L(i)*. The regularity [105] can be observed immediately in Table 2.4. For example, the first column of Table 2.4 presents six iterations of *L(i)* parameters for the cluster zero. The parameters are separated by the underscore in two parts. The first part of it is indeed a gray code sequence while the second part has a repeated pattern of *001* and *101*. Therefore, for a given iteration and cluster number, the offset can be calculated based on the correct *L(i)* parameters.

The last block of the message is processed differently than the previous ones. Leveraging on the *select()* instruction provided by the Imagine Stream Processor, the original operation can be transformed as shown in Figure 2.30 for the SIMD operation. This is based on the fact that the packet length is available in advance, such that the position of the cluster containing the last block of message is known. Based on the same scheme, the operation on the checksum can be performed as in Figure 2.31.

At the very end of the iteration, one more step is needed to perform the tree-sum operation to XOR the partial checksum distributed among the eight clusters and the last block of offset. The tree-sum operation can be achieved by the inter-cluster communication instruction provided by the processor. Thus, the final tag can be obtained by directing the checksum data stream into the AES kernels.

Figure 2.32 shows the OCB_AES performance for 4-adder and 6-adder configurations (again, using two 512-word scratchpads). The performance is calculated based on the total run time (including the time for generating the tag) divided by the size of the packet stream. Since the time for tag generation is fixed, as the size decreases, the performance is degraded due to the fixed cost.

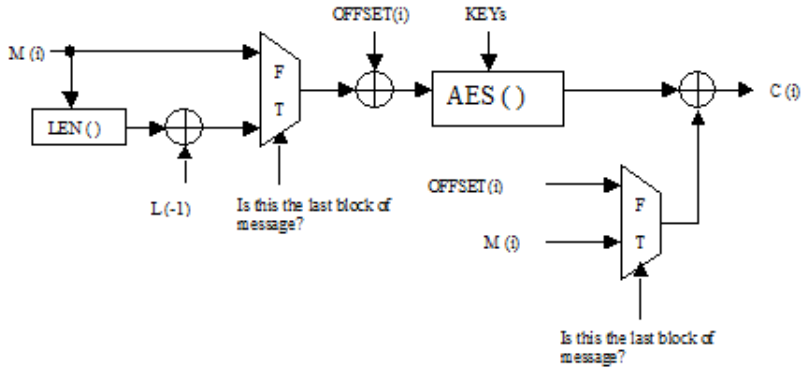

Figure 2.30: The implementation of OCB encryption algorithm for SIMD architecture.

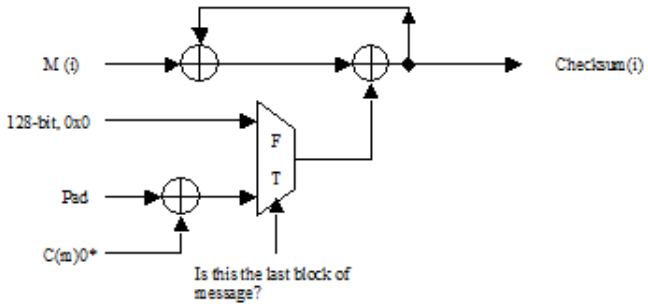

Figure 2.31: The implementation of checksum operation in the OCB encryption algorithm for SIMD architecture.

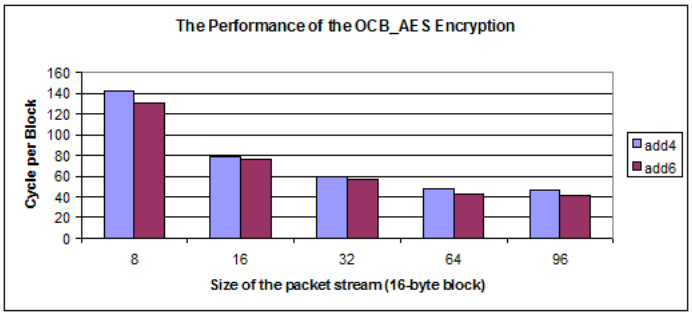

Figure 2.32: The performance of the OCB-AES encryption. (4 adder and 6-adder configurations)

```
m[1]  –>offset xor L(0)
m[2]  –>offset xor L(0) xor L(1)
m[3]  –>offset xor L(1)
m[4]  –>offset xor L(1) xor L(2)
m[5]  –>offset xor L(0) xor L(1) xor L(2)
m[6]  –>offset xor L(0) xor L(2)
...
m[53] –>offset xor L(0) xor L(1) xor L(2) xor L(3) xor L(5)
...
```

Figure 2.33: The corresponding value for offset calculation after loop unrolling and optimization.

Table 2.4: The distribution of the L for offsets calculation among eight clusters.

	C0	C1	C2	C3	C4	C5	C6	C7
0	1	11	10	110	111	101	100	1_100
1	1_101	1_111	1_110	1_010	1_011	1_001	1_000	11_000
2	11_001	11_011	11_010	11_110	11_111	11_101	11_100	10_100
3	10_101	10_111	10_110	10_010	10_011	10_001	10_000	110_000
4	110_001	110_011	110_010	110_110	110_111	110_101	110_100	111_100
5	111_101	111_111	111_110	111_010	111_011	111_001	111_000	101_000
6	101_001	101_011	101_010	101_110	101_111	101_101	101_100	100_100

2.9 Conclusions

The AES encryption algorithm has already been optimized and implemented in various processor platforms. One of the best performances published [72, 3] in non-feedback mode for a 32-bit architecture is 232 cycles per block (16-byte block). In contrast to the typical measurement setup [3] where most of the data is in the level-one cache, the simulation results in this chapter do include the data movement from the memory to the processor itself. The performance can reach up to 32 cycles per block in a stream size of 96 blocks. For a system clock of 500 MHz, the throughput of the AES encryption can reach up to 2 Gbps. The DRAM access is effectively overlapped by computation, as simulation shows more than 95% of the run time is taken by the kernel computation for a packet length larger than 16 blocks. The results also demonstrate compelling performance in the key agility simulation and OCB-AES operation. The encryption process utilized almost 50% of the theoretical memory and LRF bandwidth while only 7% are reached for the SRF.

In this chapter, the data blocks within a packet are distributed and processed among the clusters. It's a simple way to preserve the arriving packet sequence without having an extra re-ordering mechanism. However, based on the packet length distribution from a real Internet trace, there is only a limited speedup gained and the efficiency is down below 80% when doubling the number of clusters from eight to sixteen. This is because of the SIMD structure: the clusters are idle if packet data does not have enough blocks. Thus, the cluster utilization is low due to the packet length variation.

AES has been widely implemented in hardware as it was standardized in the year of 2000 by the National Institute of Standards and Technology (NIST). Therefore, the benefits of a high-performance programmable solution is not so obvious. However, as the first stepping stone into the exploration of the stream architecture for networking applications, we successfully demonstrate the processor's computation power and the unique programming model for encrypting the packet stream effectively.

In the next chapter, we are going to present and discuss an important data structure: the *universal class of hash functions*, which serves as a basic infrastructure for many networking applications presented in the following chapters.

Chapter 3

Hash Functions

The theory of universal class of hash functions is an important building block for several applications presented in this thesis (Chapters 4, 5 and 6). Generally, there are two main types of variant after the invention of universal class of hash functions due to Carter and Wegman: the one with the weaker collision property and the other with stronger one. Based on the needs of different applications, different functions are applied.

This chapter presents the general background of the universal hash function. We then discuss some of the variants based on the properties of the hash family. Finally, some implementation examples are discussed at the end of the chapter.

3.1 The Universal Class of Hash Functions

Carter and Wegman first described the idea of a universal class of hash functions [14] in 1979. Since then, this class has been widely used in many applications and regarded as "*one of the fundamental bag of tricks of every computer scientist*" [123]. The universal class of hash functions is a family of hash functions with a special randomized property. Given two keys, the probability of hashing these two keys into the same value is bounded as long as the function is randomly selected from the family.

The *universal property* [14] is defined as follows: Let H be a finite collection of hash functions which map a given universe of inputs U into domain B of range R. H is said to be

universal if for all distinct elements $x, y \in U$ where $x \neq y$, the number of hash functions selected from the family of H that yield collisions (i.e, $h(x) = h(y)$) is $\frac{|H|}{R}$. The symbol $|H|$ represents the number of functions in the collection of H. It has been proven that, if we randomly pick one hash function h from H, the probability of collision for x and y, where $x \neq y$ is exactly $\frac{1}{R}$.

$$Pr_{h \in H}[h(x) = h(y)] = \frac{1}{R} \tag{3.1}$$

3.2 Stronger Collision Properties

Wegman and Carter [122] later defined the *strongly universal*[1] hash family and utilized the properly for building unconditionally secure message authentication codes (MACs). The family of strongly universal hash functions has a stronger property: its collision probability is specified in a pairwise independent manner. For all $x \neq y \in U$ and $s, t \in B$, the probability is defined as follows:

$$Pr_{h \in H}[h(x) = s,\ h(y) = t] = \frac{1}{R^2}. \tag{3.2}$$

An example of such class of hash functions is H_1 [14]. The function maps an integer from a space $U = \{0, 1, 2..., p-1\}$, where p is a prime number, into a domain $B = \{0, 1, 2..., m-1\}$. The definition of H_1 is:

$$\begin{aligned} h_{a,b}(x) &= ((ax + b) \bmod p) \bmod m \\ H_1 &= \{h_{a,b}(x) | 0 < a < p,\ 0 \leq b < p\} \end{aligned} \tag{3.3}$$

The family of H_1 has the property of being *strongly universal*, which also implies universal. However, a universal class of hash functions is not necessarily strongly universal [115].

A family of hash functions H, is said to be *k*-universal if for every fixed sequence of *k* distinct keys $\{x_0, x_1, \cdots, x_{k-1}\} \in U$, the sequence of $\{v_0 = h(x_0), v_1 = h(x_1), \cdots, v_{k-1} = h(x_2)\}$ is equally likely to be any of the m^k sequences of length *k* with elements drawn from the hashing space $\{0, 1, \cdots, m-1\}$.

The universal class of hash functions can be generalized as *k*-universal defined in Equation 3.4, where $x < p$ and p is a prime number. The parameter a_i is selected randomly where $0 < a_i < p$.

[1] Also commonly denoted as *2-universal* or $universal_2$ due to the pairwise independent property.

$$h(x) = \sum_{i=0}^{k-1}((a_i x^i) \bmod p) \bmod m \tag{3.4}$$

$$Pr_{h\in\mathcal{H}}[h(x_i) = v_i, \forall\, i \in (0, 1, \cdots, k-1)] = \frac{1}{m^k}, where\, v_i \in [m]$$

In general, the family of *k*-universal hash functions has stronger property in terms of collision probability. For example, the *4*-universal has a lower variance of the expected number of collisions than that of the *2*-universal.

3.3 Weaker Collision Properties

Several families of the universal class of hash functions were proposed with weaker collision probabilities. The computation complexity is reduced, so these hash functions are better suited for various high-speed or low-power applications. The message authentication described in Chapter 4 is one such application.

Stinson [114] introduced a positive real number ε and formally defined new families of hash functions: the *ε-almost universal* and *ε-almost-strongly universal*. For the case of *ε-almost universal*, the constant ε represents the collision probability of the hash function where $\frac{1}{R} \leq \varepsilon \leq 1$. Therefore, Equation (3.1) can be expressed in more general way for those with this relaxed property.

A family of hash functions H is said to be *ε-almost universal* if for all $x \neq y \in U$ and the collision probability is less or equal to ε.

$$Pr_{h\in H}[h(x) = h(y)] \leq \varepsilon \tag{3.5}$$

In the case of *ε-almost-strongly universal*, the probability is defined as follows:

$$Pr_{h\in H}[h(x) = s,\ h(y) = t] \leq \frac{\varepsilon}{R}. \tag{3.6}$$

In other words, given the probability of $\frac{1}{R}$ of mapping $x \in U$ to s in the hashing space B, the conditional probability of hashing y to t is at most ε.

Rogaway [104] introduced the terminology of *ε-almost-xor universal* based on the same definition of *ε-otp-secure* from Krawczyk [57]. As $\oplus$ represents the bitwise exclusive-or operation, a family of hash functions H is *ε-almost-xor universal* if for all $x \neq y \in U$ and for any $s \in B$,

$$Pr_{h\in H}[h(x) \oplus h(y) = s] \leq \varepsilon. \tag{3.7}$$

Stinson [115] further generalized the above hash family to arbitrary Abelian groups and named it ε-*almost-*Δ *universal.* Assuming that B is an Abelian group and the group subtraction operation is denoted by "$-$", H is said to be ε-*almost-*Δ *universal* if for all $x \neq y \in U$ and for all $t \in B$,

$$Pr_{h \in H}[h(x) - h(y) = t] \leq \varepsilon. \tag{3.8}$$

Although not being *strongly universal*, the ε-*almost-*Δ *universal* family has stronger property [7] than that of ε-*almost universal*. Thus, by using Carter and Wegman's approach, a message being forged is bounded by ε if the hash function is selected from the ε-*almost-*Δ *universal* family [44].

3.4 Hashing Byte Strings for Packet Processing Applications

Based on different applications, packet processing needs to calculate the hash value on some particular attributes in the packet header. Sometimes, the calculation may need to be done over the entire packet as well. For example, if the traffic flow is defined as the tuple of source and destination IP address, the system needs to hash two 32-bit source and destination addresses for the sketch update, as described in Chapter 6, while maintaining the *k*-universal property.

An efficient way of hashing the data string is critical to the system performance. Dependent on the capability of the processing resources, it can be arranged as a sequence of bytes, 16-bit halfwords, or 32-bit words.

Generally, there are two main schemes to hash character strings [99]. The first approach is to directly reduce the long byte string to a shorter one by means of logical or arithmetic operations without the prime modulo operation. Ramakrishna proposed a class of such hash functions named *shift-add-xor* [99] for hashing 7-bit character strings. As the name suggests, the function utilizes only the simple and fast operations of *SHIFT*, *EXCLUSIVE-OR* and *ADD*. The construction is efficient and likely to be universal; simulation results suggest that the analytically-predicted performance can be achieved in practice by randomly choosing functions from this class. The pseudo code is shown in Figure 3.1 with parameters for processing byte strings.

The second is to convert the byte string into an integer, then a prime modulo operation is applied thereafter. The use of radix conversion of $\sum_{i=0}^{n-1}(byte_i \times base^i) \bmod prime$ is an example for the first type. However, care must be taken when using this scheme [79], because the selection of the *base* number and the *prime* number affect the performance significantly.

```
// The constants of L and R are pre-defined values
// we set the  L=6 and R=2 for processing
// 8-bit  strings
// m is the length of the string.

    h=seed;// randomly assigned a seed.
        for(i=0;i<m;++i) {
            h=h ⊕( (h<<L)+(h>>R)+c[i] );
           }
         h=h % T;

// The MOD operation can be replaced with
// bitwise AND for suitable values for T.
```

Figure 3.1: The pseudo codes of *shift-add-xor* hash function. [99]

In order to avoid the costly integer division (modulo) operation, several reduction techniques are proposed to speed up the process. One of the well-known schemes is the use of a Mersenne prime in the form of $2^p - 1$, where p is a prime number. It's also a widely-used technique for pseudo-random number generators. Crandall [24] proposed a clever reduction technique based on a so-called pseudo-Mersenne: $2^p - c$, where c is a small integer. The reduction techniques do impact the system design in many applications. Therefore, due to the limited scope of this paper, we refer to the discussions on more generalized Mersenne numbers in the literature [16].

The Mersenne prime is widely used to speed up the calculation of the hash function. The largest Mersenne prime available in a 32-bit integer is $2^{31} - 1$. As indicated in Equation 3.3, the key space is limited by the prime number used in the function. It means that we may not be able to maintain the 2-universal property by using the prime of $2^{31} - 1$ on a 32-bit key. Therefore, we need to use $2^{61} - 1$ as a prime number for hashing a 32-bit integer. However, some embedded processors may not support 64-bit arithmetic operations.

Instead of hashing a single 32-bit key, we can treat the key as a two 16-bit halfwords x_0x_1 and hash it by using the prime of $2^{31} - 1$. That is, the property of being 2-universal can be kept by doing the hash of $h_0(x_0) \oplus h_1(x_1)$, where $h_i()$ is randomly picked from a 2-universal class of hash functions [116].

We can extend this for hashing a n-halfword string of $x_0x_1 \cdots x_{n-1}$by selecting n hash functions $h_0, h_1, \cdots, h_{n-1}$ from a *2*-universal class of hash function H. The mapping of $x_0x_1 \cdots x_{n-1}$to $h_0(x_0) \oplus h_1(x_1) \oplus \cdots \oplus h_{n-1}(x_{n-1})$ is *2*-universal.

The tabulation method may be a good choice if the computation resource is limited

and low latency local memory lookup is available. For example, for hashing a n-byte string of $x_0x_1 \cdots x_{n-1}$, we need a $256 \times n$ 2-D array a_t. Each single column consists of 256 pre-calculated hash values by using a hash function randomly drawn from a *2*-universal hash family. The whole table is indexed by each byte value of x_i and position of i in the string. The hash process, shown in Equation 3.9, is done by XORing a sequence of values $a_t[x_i][i]$, where $i \in (0, 1, \cdots, n-1)$.

$$h_{tab}(x_0x_1 \cdots x_{n-1}) = a_t[x_0][0] \oplus a_t[x_1][1] \oplus \cdots \oplus a_t[x_{n-1}][n-1] \tag{3.9}$$

.

The above method is very attractive for applications that require *4*-universal hash functions due to its high computational complexity. However, Thorup and Zhang [117] indicate that the mapping of $x_0x_1$to $h_0(x_0) \oplus h_1(x_1)$ does not hold for *4*-universal hash functions. Instead, they prove that the following definition $hash(x_0x_1)$ holds the *4*-universal property if $h_0, h_1$and h_2 are independent *4*-universal functions.

$$hash(x_0x_1) = h_0(x_0) \oplus h_1(x_1) \oplus h_2(x_0 + x_1) \tag{3.10}$$

The time consuming *4*-universal hashing can be pre-calculated and stored in three different tables ht_0, ht_1 and ht_2 . The final hash is the XOR of the values read from these tables as shown in Equation 3.11.

$$hash_t(x_0x_1) = ht_0[x_0] \oplus ht_1[x_1] \oplus ht_2[x_0 + x_1] \tag{3.11}$$

There are several different types of universal class of hash functions for fast string hashing with arbitrary length. Those are widely used in message authentication applications. We discuss some of the newly proposed hashing functions designed for fast message authentication in the next chapter. For more details, we refer readers to the literature [116, 82, 99, 62].

Chapter 4

Message Authentication

4.1 Introduction

As the Internet grows rapidly there are strong demands for faster processing speed to protect network data streams at high speed [39]. Encryption alone can not guarantee the integrity of the data. Therefore, message authentication codes (MACs) play a very important role. The MAC is a one-way, keyed hash function designed for message authentication [75]. The construction of MACs can be based on block cipher (CBC-based MACs) or one-way hash functions. MD5 and SHA-1 are two very popular hash algorithms used in many applications. Due to the nature of these algorithms, the throughput is limited in both hardware [126] and software [6] implementations. In particular, it's difficult to process these hash functions at wire speed with software implementation on general purpose microprocessors.

However, there is a strong need for greater flexibility in the development stage and demand for code portability without much dependency on specific hardware. For these reasons, a software solution might be the most practical [45].

As the the technology advances, the attacking cost is expected to halve every 18 months. Dobbertin et al. [34] described in 1996 that "*a 128-bit hash-result does not offer sufficient protection for the next ten years*". Recent successful attacks on MD5 [119, 121] and SHA-1 [120] may be regarded as the proof of the prediction. In August 2002, NIST announced the approval of Secure Hash Standard, FIPS 180-2 that adds 3 additional hash algorithms: SHA-256, SHA-384, SHA-512,

designed for compatibility with increased security provided by AES.

Although these attacks do not directly affect the use of MD5 and SHA1 for MACs [86], demands for larger hash results and stronger hash functions are expected in the near future. The need for *larger* hash results and *stronger* hash functions means a need for more computational power for high throughput operation. A parallel architecture is a natural fit for such a requirement.

Carter and Wegman [122] proposed the use of a strongly universal hash function as the building block for unconditional message authentication. In other words, regardless of the adversary's computing power, his/her ability to alter the messages is no better than guessing with the probability of collision provided by the hash family. The computationally fast and mathematical strong properties of the hash functions open a new paradigm of MAC constructions.

The scheme is to first condense the long message body into a relatively short value by using the class of universal hash functions. Then, the hash value is encrypted with a One-Time Pad (OTP). This value is known as the "*TAG*" and sent to the communicating party along with the message itself. The Carter and Wegman construct is shown as follows:

$$TAG = (h_k(M) \oplus OTP). \tag{4.1}$$

There are several benefits for the universal-hash-and-encrypt paradigm. The first obvious one is the encryption is performed on the short hash value rather than the long message. Also, by using the universal hashing family, the computational complexity is much lower than that of the traditional cryptography-strength hash functions. Therefore, the speed of calculation is fast. Most of all, hashing performance is guaranteed by the mathematical property: the hash family's collision probability. The new paradigm has gained a lot of recent attention, and several new families of hash functions have been proposed for better performance [58, 59, 104, 112, 44, 7, 36, 62].

The brief definitions and terminologies of universal class of hash functions are provided in Chapter 3. In Section 4.2, the optimization evolution of two families of universal hash functions (MMH and NH) are presented. The general techniques related to practical implementation are then discussed.

We present the implementation of the Multilinear Modular Hash (MMH) over the programmable SIMD stream architecture for message authentication codes in Section 4.3. The algorithm can be easily parallelized and operated in SIMD fashion, and its data handling requirements match the tiered memory hierarchy. Thus, high throughput can be achieved effectively.

In Section 4.4, we demonstrate the simulation results of generating 128-bit hash values

for different sizes of packet. The best throughput achieves 7 Gbps with packet size of 1536 bytes. Finally we end this chapter with in-depth discussion over some issues and conclude with future work.

4.2 Universal Hash Functions for Message Authentication

In this section, we discuss two families of universal hash functions: MMH and NH, which are used for condensing long messages. The introductions and definitions of the universal hash functions are provided in Chapter 3.

4.2.1 MMH

Multilinear Modular Hashing (MMH) [44] was proposed by Halevi and Krawczyk, based on the well known universal hash construct denoted as MMH^* by Carter and Wegman. MMH^* is defined over the finite field of integers Z_p[1], where p is a prime number. That is, for any $X =< x_1, x_2, \ldots, x_k >$ and $M =< m_1, m_{2,}, \ldots, m_k >$ where $x_i, m_i \in Z_p$,

$$h_x(M) = M \cdot X \bmod p = \sum_{i=0}^{k} m_i x_i \bmod p. \tag{4.2}$$

The X is the set of keys that selects hash functions $h_x(\,)$ from the family H.

Based on MMH^*, the MMH^*_{32} shown in Equation (4.4) is initially modified and targeted for software implementation on 32-bit integers. The costly integer division operation is avoided by using modular reduction technique with a special prime of $2^{32} + 15$. Thus, it achieves good performance in calculation speed and maintains low collision probability. The family of MMH^*_{32} is ε-*almost-*Δ *universal* with $\varepsilon = 2^{-32}$.

$$\begin{aligned} h_x(M) &= M \cdot X \bmod (2^{32} + 15) && (4.3) \\ &= (\sum_{i=0}^{k} m_i x_i) \bmod (2^{32} + 15). && (4.4) \end{aligned}$$

The family of MMH^*_{32} is further optimized (denoted as MMH_{32} and shown in Equation (4.5)) to ignore the carry bit from the 64-bit inner product ($m_i x_i$) and the most significant bit in the

[1] Z_p consists of all integers between 0 and $p - 1$.

final output ($\sum_{i=0}^{k} m_i x_i \bmod p$) of the hash function MMH^*_{32}. Therefore, more speed-up is gained with an increase in the collision probability.

The family of MMH_{32} is ε-*almost-*Δ *universal* with $\varepsilon = 1.5 \times 2^{-30}$. The hash function is designed to work with fixed-size messages of 32 words, thus the k is 32. There is a tradeoff associated with the choice on the size of the message. Clearly, as shown in Equation (4.5), the longer the message being processed, the lower the average cost for the final modular reduction over the message. However, the longer the message, the larger the set of keys that must be kept at both communicating parties.

$$h_x(M) \quad = \quad [((\sum_{i=0}^{k} m_i x_i) \bmod 2^{64}) \bmod (2^{32}+15)] \bmod 2^{32} \tag{4.5}$$

4.2.2 NH

The New Hash function (NH) [7] was proposed by Black et al., based on the construction of NMH^* invented by Carter and Wegman [44]. The functions of the NMH^* family, shown in Equation (4.6), have fewer multiply instructions than additions. Therefore, the speed of hashing is improved for processor architectures with higher cost of multiplications.

$$h_x(M) = \sum_{i=1}^{k/2} (m_{2i-1} + x_{2i-1})(m_{2i} + x_{2i}) \bmod p \tag{4.6}$$

The hash function of the NH family shown in Equation (4.7) is derived by further removing the prime modulo operation from Equation (4.6). Thus, the hashing speed is greatly enhanced and very suited for hashing larger size of messages.

$$\begin{aligned} h_x(M) \quad = \quad & \sum_{i=1}^{k/2} [((m_{2i-1} + x_{2i-1}) \bmod 2^w) \times \\ & ((m_{2i} + x_{2i}) \bmod 2^w)] \bmod 2^{2w} \end{aligned} \tag{4.7}$$

The NH family is ε-*almost universal* with the ε equal to 2^{-w}. The w denotes the word size; thus, w equals to 32 for a 32-bit microprocessor.

The NH family has a weaker property than that of MMH_{32}. Therefore, the hash results can not be used directly XORing the one-time pad in the Carter and Wegman approach. Another level of hash (e.g., a strongly universal IPHash16 [63]) or pseudo-random function (PRF) [7] is required for generating the authentication tag.

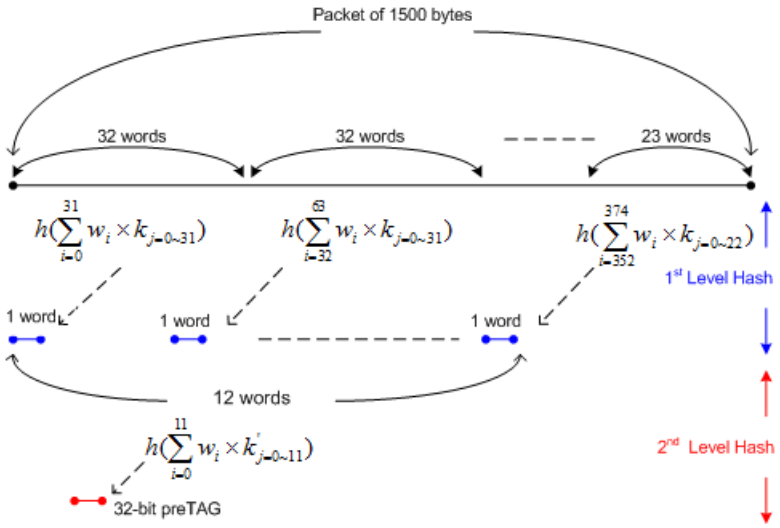

Figure 4.1: Two-level tree hash process over a packet size of 1500 bytes.

4.2.3 Arbitrary Lengths and Collision Probability

The hash function MMH_{32} works with fixed-size message blocks. Therefore, the *tree-hashing* scheme proposed by Carter and Wegman [122] is adopted for hashing messages with arbitrary lengths. A message is segmented into 32-word blocks, where the last block may contain fewer than 32 words. The first level of hash calculation is performed for each of these blocks individually. The results of these n hash values are then grouped as blocks of 32 and hashed again, and so on. As illustrated in Figure 4.1, a packet of 1500 bytes is segmented into multiple 32-word blocks. Then, the first level of hash calculation is performed based on each of these blocks. The hash result of each block is a single 32-bit word, so only one more round of hashing is needed to produce the final hash value of the packet.

The collision probability is determined by the product of the depth of the tree hashing l and the parameter ε of the universal hash function. For example, MMH has $\varepsilon = 1.5 \times 2^{-30}$; if we do a 2-level hashing ($l = 2$), the collision probability is 3×2^{-30}. The linear increase of collision probability with the depth of the tree can be avoided by using different hash families on each level of hashing. A good example can be found in the construction of the UMAC(2000) [63], where three different hash families are used.

Since the collision probability may be too high for some applications, a scheme to lower

the probability is to hash the same message again with another set of independent keys. By using this strategy, the probability will be $\frac{1}{p^2}$ instead of the original $\frac{1}{p}$. However, the cost is almost twofold on the amount of computation, the keys used, and the size of the hash values.

Fortunately, the issue regarding the doubling of the key size can be relaxed by using the Toeplitz matrix construction. That is, instead of using two different sets of keys $K =< k_0, k_1, \cdots, k_{n-1} >$ and $K' =< k'_0, k'_1, \cdots, k'_{n-1} >$, the second key set K' can be derived from the first key set of K by skewing one position and adding an extra element k_n added. Thus, the key set K' is $< k_1, k_2, \cdots, k_n >$. We are going to discuss more of the issues on the extra cost in Section 4.5.

4.3 Implementation

In this section, we present a brief overview of the implementation of MMH with multiple tags over the stream programming model. The stream programming model is composed of two levels of hierarchy: the stream and kernel. At the stream level, data is organized into streams and sent to the clusters at the kernel level for hash computation.

4.3.1 The Stream Level

We assume that the maximum packet size is 1536 bytes. Since MMH is designed with a fixed-size key of 1024 bits, a 2-level hash tree is needed for a packet of 1536 bytes.

The system consists of three main kernels: *level-1s*, *level-1* and *level-2*. For packets larger than 128 bytes, the packet streams will be directed to the *level-1* kernel where first level hash will be calculated. The output of these hash values, named *L1-Hash* streams, will be stored in the Stream Register File (SRF) and then consumed by the *level-2* kernel. The *level-2* kernel calculates the second level hash values similar to that in the *level-1* kernel. Then, the hash values are XORed with the One-Time-Pad (OTP) stream to yield the final tags. The operation for packet size greater than 128 bytes is shown in Figure 4.2.

For packets of 128 bytes and less, there is no need to do the two-level hashing. Therefore, only the *level-1s* kernel will be invoked to generate the *TAG* stream. Since the length of the packet is known ahead of the packet being processed by the kernel, the microcontroller can direct the packet stream and invoke the kernels accordingly.

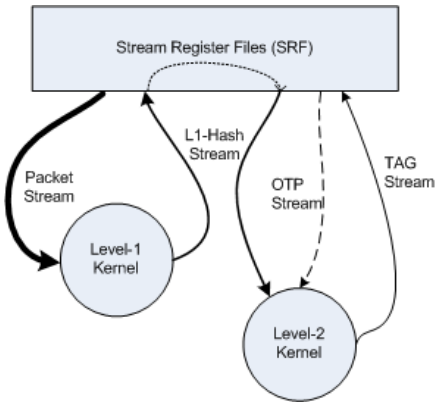

Figure 4.2: The stream level diagram of the MMH system.

4.3.2 The Kernel Level

The major computation[2] of the hash values is done in each cluster at the kernel level. Packets are organized as a stream of records and distributed sequentially across the clusters. Depending on the number of authentication tags desired, clusters are organized as groups. The records in the same group are further broadcast to groups of clusters through the inter-cluster network. Then, each group of clusters calculates hash values based on the same records. The same end can be achieved by skewing records of the packet stream at the stream level and resending them to the clusters. Since the inter-cluster communication network has higher bandwidth and lower latency than that of SRF, the record distribution is performed in the kernel level. The original collision probability ($\frac{1}{p}$) of MMH can be improved to $\frac{1}{p^4}$ by hashing the message four times, generating four tags. Therefore, the goals of designing the kernel are targeted at parallelizing and maximizing the efficiency of the computation for four duplicated authentication tags.

As an example shown in Figure 4.3, the clusters are evenly divided into four groups (G0, G1, G2 and G3). Given a packet stream of eight records, the pair of records within each group are sent to the other groups through the inter-cluster communications. Therefore, the hash values are calculated four times in parallel based on the same message with four independent sets of keys.

By using the Toeplitz scheme, the same set of keys is stored in the scratchpad of each cluster. Extra computations are then needed for clusters in different groups to identify the skewed-

[2]The Imagine supports two 32-bit multiplication by using two 32-bit registers holding the 64-bit result. However, the 64-bit addition is not available. Due to this restriction, we further instrument our code to double the cycles of the 32-bit add instruction to approximate the cost of 64-bit add operation.

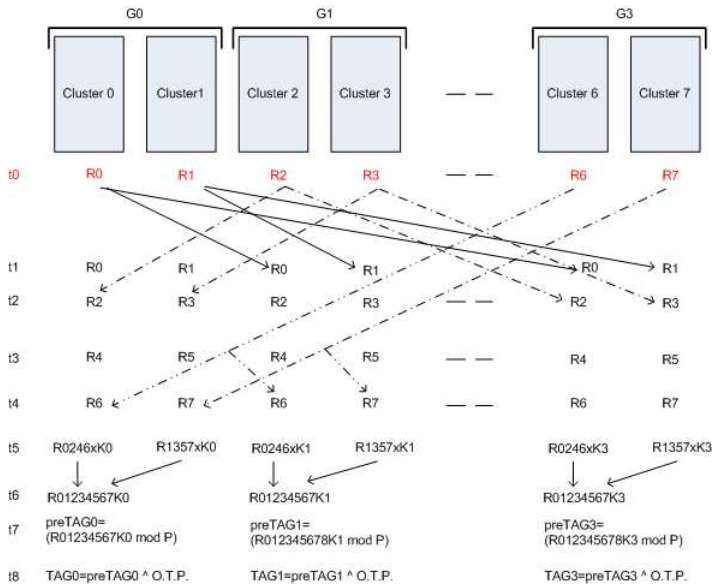

Figure 4.3: The Kernel in operation. A stream of eight records are distributed across each of the clusters of four groups. Pairs of the records are further broadcast to the other group for calculating four different set of authentication tags. The symbol of R0246xK0 denotes the sum of products for the records (0,2,4,6) and the first set of the keys (k0).

index to access the key.

Basically, the kernels share the same structure of hashing the message. The differences between the first and second level kernel are the extra XOR operation for the one-time pad, and the smaller size of the record in the *L1-Hash* stream.

4.4 Experimental Results

The packet stream consists of records of four 32-bit words. Due to the restriction of the simulation tools, the packet stream is further padded to be a multiple of eight records (for 8-cluster configuration). The IScd VLIW scheduling results are presented in Figure 4.4. The un-optimized result is shown on the left-hand side where the steps of modulo reduction of the prime $2^{32} + 15$ can be seen at the bottom. By using software pipelining, the IScd compiler can further optimize the result shown on the right-hand side of Figure 4.4. The cycle count for the second basic block is reduced by 60% and the average kernel utilization for adders is increased from 35% to 85%.

The throughput of MMH producing a 128-bit pre-tag is shown in Figure 4.5 with different

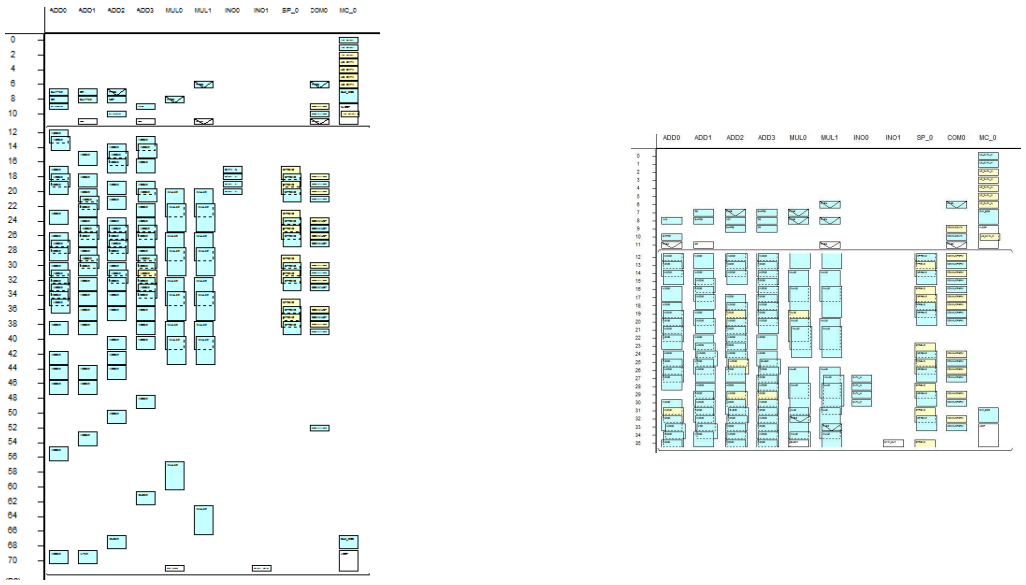

Figure 4.4: The VLIW scheduling results by IScd for the first kernel. The optimized result by using the software pipelining technique is shown in the right. For the second basic block, the total cycle is reduced by 60% and the average kernel utilization for adders is increased from 35% to 85%.

packet sizes. The pre-tag represents the hash value before XORing with the one-time pad. The best performance of the MMH is 7.14 Gbps with packet size of 1536 bytes. The worst case is for the packet size of 128 bytes where the throughput is only 2.23 Gbps.

The average throughput increases as the size of the packet increases. This is due to the fixed cost associated with each run of the calculation. The cost associated with the loop within the kernel, known as the short stream effect [92, 30], is a dominating factor for small packets.

Generally, the prologue and epilogue blocks are established before and after the main loop for setting up constants and initializing variables. Shown on the right-hand side of the Figure 4.4, the main loop (second basic block) only takes 22 cycles for processing 128 bytes of data due to the lightweight computation of the universal hash function. Therefore, as the stream passing the loop gets smaller, prologue block takes a significant portion of the kernel run time. That is, the average cost is higher.

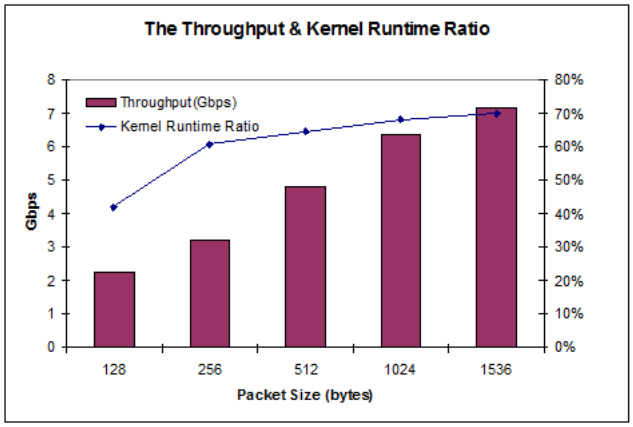

Figure 4.5: The throughput of MMH and the corresponding kernel runtime ratio with collision probability of 2^{-120} with different packet sizes. The system clock is 500 MHz, with 4-adder, 2-multiplier and 8-cluster configuration.

4.5 Discussion

4.5.1 Small Messages

Several universal hash functions are designed by utilizing special instructions provided by contemporary processor architectures to achieve fast processing speed. Examples such as the use of fast multiply-and-add instructions and SIMD extensions [7, 36] are very common. Furthermore, more speedup can be achieved by relaxing the algorithmic property of the hash function. As discussed in Section 4.2.1 and 4.2.2, techniques include ignoring the carry bits of the multiplication and summation processes, or even removing the prime modular operation.

A relatively long message usually requires several levels of hashing. Therefore, the optimization philosophy of the universal hash functions is to hash the long message very quickly at the first level by relaxing its algorithmic property. The relaxed property, on the other hand, can be compensated by using stronger functions at the other levels. The cost of using the stronger function at the other level can also be justified and minimized since the message has been greatly condensed.

The use of the NH [7] family in UMAC is a good example. Due to its weaker property of not being *strongly universal*, the authentication tag generation is different from the Carter and Wegman construct shown in Equation (4.1). The major difference is that a pseudo random function (PRF) is introduced and applied over the hash values to produce the authentication tag. For example, in a typical implementation of UMAC, the message is divided into segments of 1024 words. The

NH hashing results of each segment are concatenated together along with nonce and the encoded length value of the original message. Then, a pseudo random function (PRF), HMAC-SHA1, is used to hash the concatenated value, producing a 160-bit authentication tag. The whole operation is shown as follows:

$$TAG = HMAC_{SHA1}((NH(Msg_1)||NH(Msg_2)||\cdots NH(Msg_n)||Len)\,||Nonce) \quad (4.8)$$

The optimization schemes at the algorithmic level are very successful, especially for larger size messages. However, the benefits are limited for smaller messages.

It's a known fact that there are a large number of small-sized packets in the Internet[3]. Thus, due to the costly operation of SHA1 for small packets, the improved UMAC(2000) algorithm [63] discards the use of pseudo-random function shown in Equation (4.8). Instead, UMAC(2000) relies on the IPHash, a strongly universal, inner-product hash in the form of $\sum m_i k_i \bmod p$ and adopts the original Carter and Wegman MAC construct, using a one-time pad.

4.5.2 The One-Time Pad

Shown in Equation (4.1), the one-time pad is used in the Carter and Wegman construct to XOR the hashing result and produce the authentication tag. Therefore, in the system view point, as the size of the packet decreases, the cost of producing the one-time pad becomes more significant.

An obvious drawback of the one-time pad is its key distribution and management [75]. Therefore, the use of one-time pad is approximated by the pseudo one-time pad based on some pseudo random functions [10]. Given the same function and seed, both of the communicating parties can create the exact same sequence of pseudo random numbers.

The simple random number generator in the form of $x_n = (a \cdot x_{n-1} + b) \bmod p$ can not be used due to its weak property: the sequence can be easily predicted even without knowing the parameters used [94]. In practice, a stream cipher is used with its property of being computationally secure [75]. A block cipher is another choice, such as AES in the implementation of one-time pad for UMAC(2000) [61].

Sometimes, the generation of one-time pad is dependent on the incoming packets and can't be done ahead of time. For example, a different nonce may be used to generate the one-time pad with each message [61]. In order to synchronize the use of nonce between the sender

[3]Please see Figure 2.18 for detail.

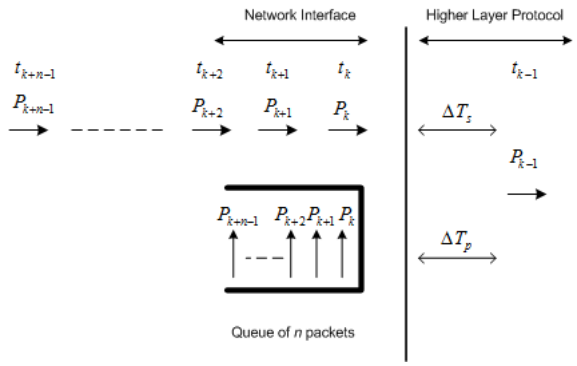

Figure 4.6: The queue up of n packets for parallel processing of the one-time pad.

and receiver, nonce is sent along with the message. This implies that the one-time pad has to be produced on line and the cost has to be taken into consideration.

Given the fastest performance on a 32-bit uniprocessor, the AES calculation on a 16-byte block in ECB mode is 232 cycles [72]. For an architecture configuration of A4M2C8 (4 adders, 2 multipliers and 8 clusters), the worst case AES calculation on *eight* 16-byte blocks in ECB mode is 77 cycles.[4] Therefore, the cost can be amortized by creating the one-time pads for multiple packets in parallel on the SIMD architecture. Instead of calculating the one-time pad upon receiving a packet at a time, a batch calculation of eight one-time pads can be done efficiently by processing eight packets from the queue.

Some protocols (e.g., IPSec) use time-out parameters to constraint the time period between two receiving packets, so the number of packets in the queue shown in Figure 4.6 has to be kept within a limit. In other words, the time difference of ΔT_p between the last packet P_{k-1} in the last batch and the first packet P_k of the current batch has to be within the time-out limit.

4.5.3 The Multi-SIMD Operation

Given an authentication tag of n bits, the attacker can break the MAC with a probability of $\frac{1}{2^n}$ [61]. That is, the probability of forging a message with a correct authentication tag without knowing the key is dependent on the length of the tag. While these hash functions are capable of generating a 32-bit hashing result at high speed, the high collision probability may not be suitable for

[4] Please see the simulation results in Chapter 2.

applications demanding high system security. Therefore, hashing the message multiple times using independent key sets is needed. For example, since the length of 96 bits is the default authenticator length as specified in IP authentication header [51], a triple-hashing scheme is needed such that three 32-bit hash values are concatenated to form 96 bits.

According to the performance results of MMH (200MHz Pentium-Pro), the best case throughput (message in cache) for generating the 32-bit and 64-bit output are 1080 and 500 Mbps respectively [44]. As the throughput decreases approximately linearly, we estimate the throughput of producing a 96 and 128-bit results are roughly in the range of 375 and 250 Mbps, respectively. On the other hand, a speculated throughput of 300Mbps is estimated on a 200 MHz Pentium-Pro processor [44, 9]. It is clear that the speed advantage for MMH diminishes as the number of tags required increases on a uniprocessor implementation.

The SIMD architecture is the best candidate to create multiple tags in parallel for the universal-hash-and-encrypt paradigm. The hash functions are easy to be implemented in parallel achieving high speed of generating multiple tags. In addition, due to the *Multi-SIMD* operation, the cluster utilization is better regardless of the packet size.

As the architecture is designed for handling larger data streams efficiently, handling a small-sized stream between kernels has higher average cost simply because of the inefficiency of resource utilization. A good example is the *L1-Hash stream* passing from the first kernel to the second by computing only ***one*** authentication tag. Given a 256-byte packet, the *L1-Hash stream* consists only two 32-bit words useful for the second level hashing, whereas the bandwidth is capable of transferring eight 32-bit words of *L1-Hash stream* from a 1024-byte packet in the same number of cycles.

Take the same example stated above but computing ***four*** authentication pre-tags: given a 256-byte packet, the *L1-Hash stream* now consists of eight 32-bit words useful for the second level hashing.

As shown in Figure 4.3, two clusters are arranged as a group. Each group processes the same packet by using the same hash function but different keys. We denote this process as *Multi-SIMD*, since it's slightly different from the pure SIMD operation. The efficient architecture support of such operation can easily be exploited by algorithms of this kind. Another good example is the Bloom filter [8] implementation (Chapter 5), where the multiple functions used to hash the same data in each cluster may originated from the same universal hash family, differing only in parameters.

4.6 Conclusions

For years, the focus has been targeted on the optimization of the universal hash functions of $h_k(M)$. As discussed in previous sections, the evolution from MMH^*, MMH^*_{32}, MMH_{32}, NMH^* to NH is a good example of this trend where multi-Gigabit throughput can be achieved in software implementation.

As more new families of universal hash functions are proposed [36, 62], little has been addressed with regard to the performance degradation due to the extra computation needed for more tags. As the simulation results suggest, the hashing speed is decreased approximately linearly with increasing the number of the tags. As the number of tag grows, the speed advantage is lost compared to the conventional MAC schemes.

The SIMD parallel architecture provides a simple yet elegant solution for this increasing demand for extra tags on the new universal-hash-and-encrypt paradigm. The best performance of the MMH is 7.14 Gbps with packet size of 1536 bytes and 2.23 Gbps for size of 128 bytes and less. The cost of one-time pad generation can also be greatly amortized by computing in parallel.

The parallel stream architecture not only provides a good accommodation for such hash based computation, the benefit of being programmable is also obvious: the developers can freely adopt the best and newest algorithm that fit the system requirements at minimum or no cost at all.

Chapter 5

Deep Packet Inspection

5.1 Introduction

The content matching process, i.e, hash computation over an entire packet, fits well into the stream programming model with an abundance of producer-consumer locality: portions of the the hash values computed and stored in the stream register file (SRF) are used for calculating a new set of hash values recursively.

One of a good example of such applications is the Network Intrusion Detection System (NIDS). The NIDS is designed to identify network attacks, based on anomaly traffic profiles or known patterns, called signatures, in packet payloads. One of the most challenging issues that these systems are facing is the speed of the emerging high-speed networks. The demanding costs make intrusion detection and analysis over such high-speed traffic in real-time very difficult [83].

Snort [91] is one of the most popular open-source NIDS programs. It uses a collection of matching rules to identify potential malicious packets. Some of these rules involve only fields in the packet header, but most require sophisticated string matching against the payload bytes. Most general-purpose systems are barely able to process minimum-size packets on an 100Mbps network [107]. The content matching engine for the network intrusion detection system is considered to be the critical part due to its computation complexity. For Snort, the String-Matching process takes 31% of total processing time on a uniprocessor. Furthermore, the processing time could go up to 80% for web intensive traffic [2]. Thus, most high-speed NIDS's rely on specialized hardware for

content matching. For example, Clark et al. [18] use an FPGA-based array of matchers, because the network processor (Intel IXP1200) does not have sufficient computing power to perform the task at wire speed. Such solutions are difficult to scale, in terms of the number of matching rules, and are inherently inflexible, because they are built with a single algorithm for a single purpose. Many sophisticated pattern match algorithms have been recently proposed. Of particular interest are probabilistic methods based on Bloom filters [8]. The Bloom filter is an algorithm that is used to probabilistically test membership in a large set. Content matching is essentially a *query* process. Given a set of signatures or contents seen in the attacking traffic, Bloom filter can quickly inspect the incoming packet and simultaneously match a large set of patterns for the suspicious target. Similar to the mechanism of a filter, the majority of packets that do not contain malicious signatures can be quickly bypassed, while those that do can be identified immediately for further processing. Such a system has been implemented in hardware [33], achieving high performance.

In this chapter, we present a design space exploration of content matching based on the Bloom filter using the programmable Imagine Stream processor [50]. In the following sections, the mathematical background of Bloom filter is first presented. Then, the system design and an implementation in the stream programming model are discussed. We also demonstrate the flexibility and performance of the stream architecture supporting the realization of the universal class of hash functions for the Bloom filter. A detail discussions of implementation tradeoffs and simulation results are provided at the end of this chapter.

5.1.1 The Bloom Filter and Pattern Matching

The Bloom filter [8] is a single-bit memory array. A set can be succinctly represented by this unique data structure, on which membership queries can be performed efficiently at the cost of rare false positive matches.

The whole operation consists of two phases - membership insertion and query. Initially, the value of the bit array is set to "0". At the membership insertion phase, a member x_i is hashed by k independent hash functions. The k different hashing results, ranging from 0 to $m-1$, will each address a bit in the array and set its value to "1". If there are n members in the set, and the size of the filter is m bits, the filter is said to have a bits-per-entry ratio of $\frac{m}{n}$. For the query phase, the same hashing operation is performed again on the member $x_j$being tested. The k different bit values addressed by the different hashing results will be read from the memory array. If all of the bits are "1", then x_j is said to be a member of the set. This is an approximate algorithm, because

the hash functions are not unique. Therefore, there is a probability of being false positive; that is, an item will be identified as a set member, even though it is not.

The pattern matching process is similar to the membership query. Given a byte-string of a specific length, the query result indicates if there is a match to any malicious patterns in the Bloom filter. That is essentially multi-pattern matching for patterns with the same length at the same time.

The theory of Bloom filter is based on the ideal property of hashing functions. Assuming the hashing results are uniformly distributed over the memory, then after the insertion phase the probability that a specific bit is still "0" is $p = (1 - \frac{1}{m})^{kn} \approx (e^{-\frac{kn}{m}})$. Therefore, the probability of false positive error can be modeled by the following equation:

$$P_{fpe} = (1 - p)^k \approx (1 - e^{-\frac{kn}{m}})^k. \tag{5.1}$$

The false positive error probability, shown in Figure 5.1, is dependent on the construction parameters k, m and n. Increasing the number of hash functions decreases the false error rate, up to a point. The same effect applies for the bits-per-entry ratio of $\frac{m}{n}$. When designing the Bloom filter, several interesting issues arise due to tradeoffs among the performance metrics, i.e., the number of hash functions, the size of the memory and the number of entries in the filter. The number of hash functions is limited by the available computing resources and memory bandwidth. Increasing the number of hash functions decreases the error rate to a certain degree. It has to go along with the size-per-entry ratio in order to yield good quality of false positive error rate. The higher the ratio, the lower the error rate. For a given bit-per-entry ratio of $\frac{m}{n}$, the false error probability P_{fpe} is minimized for $k = (\frac{m}{n} \ln 2)$ hash functions.

5.2 The Implementation of Bloom Filter on Imagine

There are two factors affecting the performance of software based implementation of Bloom filter: the speed and accuracy of the query phase. In other words, the performance means how quickly a computation can be done to generate the index and look it up through the memory array, and the possibility of being false positive if the outcome shows a match.

The design of Bloom filter is based on a family of independent hash functions where keys are transformed and distributed uniformly into a range specified by the size of the memory array. Therefore, the choice of hash functions and an efficient way to do the hashing over the entire packet dominate the speed of the computation; the quality of the hash influences the rate

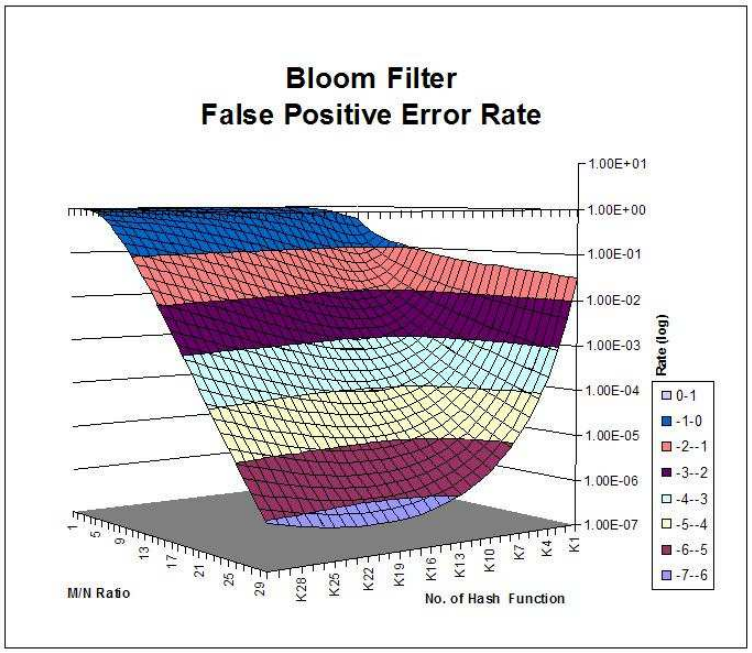

Figure 5.1: The false positive error rate of Bloom filter

of false positive errors. Another performance bottleneck for Bloom filter implementation is the table lookup. Given k hash functions, k lookups have to be performed for each query operation. Thus, memory bandwidth may become a critical issue for parallel lookups in a real-time system. Moreover, conventional cache architectures are unlikely to work well if the Bloom filter is larger than the cache, since the lookups to the memory array are purely random.

In the following sections, we present a design that relies on the unique memory structure of the stream processor, where these lookups can be performed in parallel efficiently. Moreover, the flexibility of supporting various hash function implementations and the efficient way to compute the hash values in the streaming model are discussed.

5.2.1 System Design

Sequential lookups can be avoided by incorporating banks of multi-port memory. For example, assume there are k banks of single-read-port memory and each holds $\frac{m}{k}$ bits of the the original Bloom filter contents. Given a key and k hash functions, each of the k hash values is used

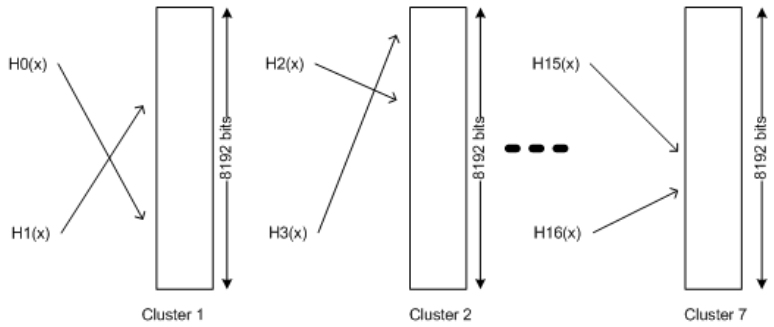

Figure 5.2: The scratchpads of eight clusters served as multi-segment memory for Bloom filter.

to index the individual bank simultaneously. Therefore, a k-parallel lookup can be done in one cycle. The false positive error of this scheme is slightly higher [78] than that in Equation (5.1). The probability that a specific bit is still "0" after the insertion phase is $(1 - \frac{k}{m})^n$ instead of the original $(1 - \frac{1}{m})^{kn}$.

The Imagine Processor consists of 8 clusters and each has a 256-entry, 32-bit word scratch-pad. These scratchpads, as illustrated in Figure 5.2, are good candidates serving as multi-segment memories which provide 65536 bits in total of storage for the Bloom filter. The hash calculations are distributed into each cluster. As shown in Figure 5.2, given $k = 16$, each cluster calculates two hash values in a 8-cluster configuration. The hash values are used locally in each cluster to address the scratchpad memory. The operation is conducted in a SIMD fashion since a family of hash functions share the same structure with only differences in parameters or tables. The logical AND operation is performed on the lookup results from all the clusters through inter-cluster communication. The single-bit result after the AND operation therefore represents the matching outcome. In order to minimize inter-cluster communication, the bit values of a sequence of 32 queries are accumulated by shifting through a 32-bit word. Thus, 32 query results can be obtained by doing a tree-based logical AND operation through inter-cluster communication. Bits are further packed in the 32-bit words and grouped as an output stream for other kernels or host processor to verify and locate the suspicious pattern within the packet.

Consider a filter with 2000 patterns for an 8-cluster configuration. Each cluster has 8K bits of scratchpad, for a total of 64K bits and a $\frac{m}{n}$ ratio of 32. The theoretical false positive error rate (i.e., false positive errors per query) can be as low as 10^{-7} by using 16 hash functions. For a

16-cluster configuration, the $\frac{m}{n}$ ratio becomes 64, and the false positive error rate is down to the level of 10^{-11}, while the number of patterns and hash functions are still the same. Moreover, the performance is increased as well, since only one scratchpad read is needed in each cluster for the query.

There are several benefits of holding the Bloom filter in scratchpad memory at each cluster. The most obvious one is the distributed, low latency accesses for the queries. Since it's relatively low-cost to scale the number of clusters, rather than the ALUs in each cluster [53], the expanding of clusters means potential increases on both the size of the Bloom filter and the number of hash functions. The Merrimac streaming supercomputer [26] is an example where each node consists of 16 clusters. Therefore, given a fixed amount of patterns, the false positive error rate decreases significantly with the increase of the clusters while the kernel run time remains the same.

5.2.2 The Hash Functions

The implementation of Bloom filer is based on a set of independent hash functions. Thus, the choice of hash functions dominates the performance of the system. A good example is the false positive error rate shown in Equation (5.1) where the derivation is based on the perfect property of the hash functions. That is, keys are assumed to be uniformly distributed. Then, keys are hashed and randomly distributed into each slot of the memory array.

However, in the real world scenario, the byte distribution within packets is not truly random. Dependencies among packets are commonly found which yield higher false positive errors. The universal class of hash functions [14], widely used in many applications [98, 99, 97], are suitable for resolving this issue. This is because of its randomized property: by randomly selecting hash functions from the family, the average performance can be guaranteed independent of the input keys. The detail descriptions on the universal hash functions as well as the schemes on hashing strings are provided in Chapter 3.

5.2.3 Programming Model

The stream programming model is composed of two levels of hierarchy: the stream and kernel. At the stream level, data is organized into streams and sent to the clusters at the kernel level for major computation. Packets are organized as stream of records and distributed sequentially across the clusters. Each record will be broadcast to all the clusters through the inter-cluster com-

munication so that all the clusters calculate hash values based on the same record. In other words, k different hash values are computed from a family of k hash functions based on the same record (the key) at all clusters in a SIMD fashion.

For a given packet $P = (p_0, p_1, p_3, ..., p_{l-1})$ of l bytes and a sliding window of m bytes, ${}^mC_j$ denotes the byte string qualified by the window of m bytes at the position of j over the packet string P. For example, if $m = 5$ and $j = 0$, ${}^5C_0$ represents the byte string of $(p_0, p_1, p_2, p_3, p_4)$.

In the query phase, given patterns with the same length of m bytes and a packet of length l, where $l > m$, a total of $K \times (l - m + 1)$ queries need to be performed. This is simply because there are K hash functions and a window of m bytes has to slide through the packet. For example, in the first iteration $(j = 0)$, K hash values of $h_k({}^mC_0)$ are used as the indexes to address the memory array. If all locations of the memory array addressed by those K indexes contain "1", then it's highly possible that the packet contains the target pattern of m bytes at the first position. For the case of processing patterns with n different lengths, ranging from m to $m + n - 1$ bytes consecutively, the total number of hash values that need to be computed is $\simeq \frac{K}{2} \times n \times l$.

The system consists of two kernels which are shown in Figure 5.3. The design is based on the *appending process,* which is suited for processing multiple patterns of contiguous lengths. Appending is a simple technique to compute the hash value efficiently. Instead of calculating the hash value of a m-byte string ${}^mC_j$ starting from the first byte, the hash value of $h({}^mC_j)$ can be derived from the value of $h({}^{m-1}C_j)$.

Given a set of patterns with lengths contiguously specified in a fixed range, e.g., $(m,\ m+1,\ m+2,\ m+3)$, the first kernel calculates the preceding hashes of specified lengths over the incoming packet. There are 4 sets of hashes: $h({}^mC_j)$ where $j = 0, 1, 2, 3$; $h({}^{m+1}C_j)$ where $j = 0, 1, 2$; $h({}^{m+2}C_j)$ where $j = 0, 1$; and $h({}^{m+3}C_j)$ where $j = 0$. These hash values will be used as indexes to lookup the Bloom filter in the query phase. In the mean time, the initial hash values of $h({}^{m-1}C_4)$, $h({}^mC_3)$, $h({}^{m+1}C_2)$, and $h({}^{m+2}C_1)$ will be grouped as an output stream S_0 for the base of the appending process. Along with the incoming packet stream, these hash values are streamed into the second kernel, where a recursive operation takes place. As discussed previously, hash stream S_1 of $h({}^mC_4)$, $h({}^{m+1}C_3)$, $h({}^{m+2}C_2)$ and $h({}^{m+3}C_1)$ can be derived from the stream S_0 of $h({}^{m-1}C_4)$, $h({}^mC_3)$, $h({}^{m+1}C_2)$ and $h({}^{m+2}C_1)$. Therefore, the newly generated hash values will be treated as a data stream and stored in the stream register file temporarily. Then, the same kernel will consume the data stream. The same process will be repeated until the end of the packet. That is, the kernel 2 scans through the rest of the packet and calculates the hashes recursively.

Due to the excess amount of computation, the kernel only processes up to four different

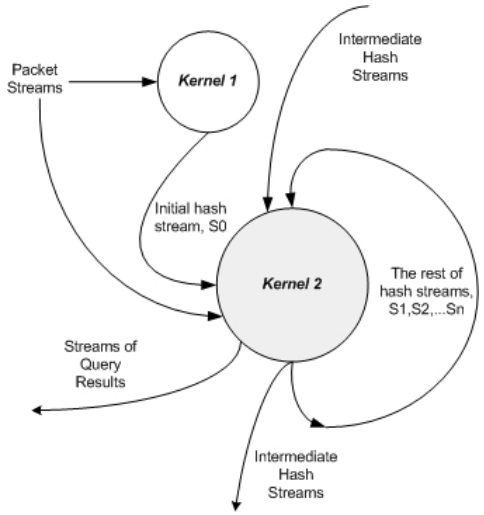

Figure 5.3: The kernel block diagram.

patterns in consecutive lengths. Since the use of patterns with size ranging from 1 to 4 bytes are more effective to associate with other *options* in Snort's *header* and *body* rules for the matching process, we deliberately skip those patterns to avoid high matches without loosing generality in this chapter. A pipeline of n processors shown in Figure 5.4 is capable of handling up to $4n$ different lengths of patterns. For those patterns with lengths greater than the capacity, a scheme called "*reduced-pattern length*" is adopted. This scheme uses the "*cutoff length*" instead of the original one for the matching process. Therefore, for a large number of patterns of different lengths, the large number of hash computations can be bounded at the cost of increasing false positive errors. The detail of this scheme is presented in Section 5.3.1.

The performance of the pipeline is limited by the stage which consumes the longest processing time. The cycle time for the first stage of the pipeline, i.e. on the top of Figure 5.4, is slightly longer than the rest of the three stages since an extra set of $base$-hash values must be computed: $h(\,^{m-1}C_j)$ denotes the extra $base$-hash value, which must be calculated at the first stage of the pipeline. The output stream of *intermediate hash stream* contains hash values of $h(\,^{m+3}C_j)$, where $j = (0, 1.., (l - m + 1))$. For example, at the second stage of the pipeline, the input *intermediate hash stream* to Kernel 2 can be sourced by the output *intermediate hash stream* from the first stage. Therefore, only the first stage has to compute the extra values. For each stage, the computation within the first kernel is only dependent on the input packets. The second kernel depends

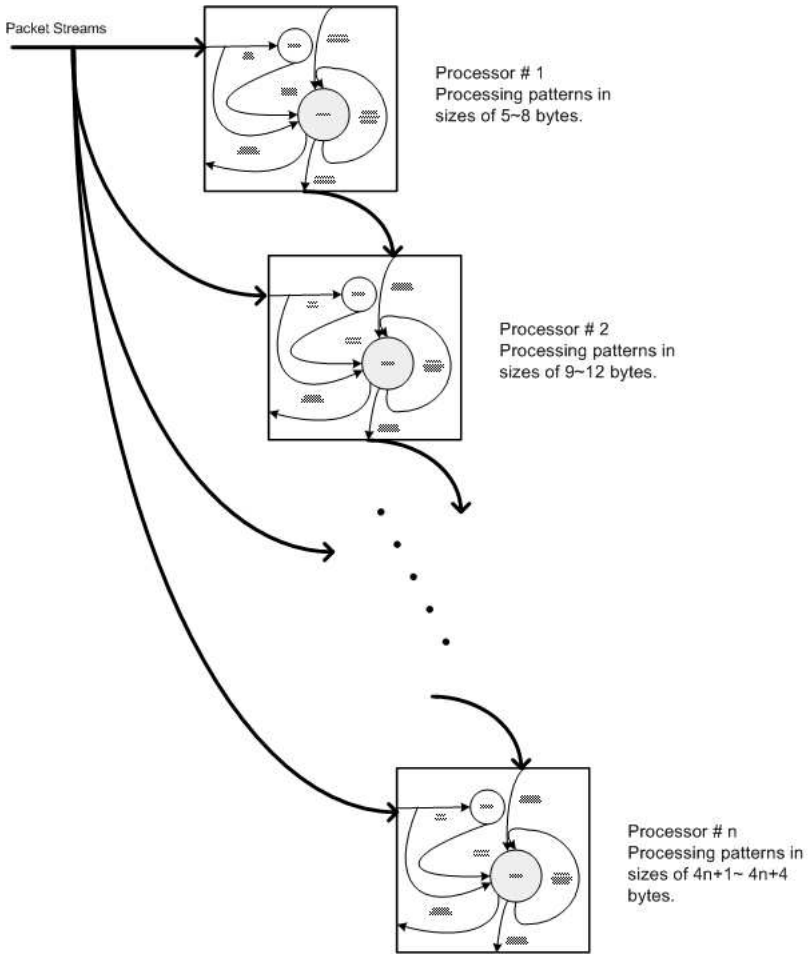

Figure 5.4: The pipelined flow architecture with n processors. Each processor processes patterns in a fixed range of continuous length. The depth of processors pipeline can be extended dependent on the number of pattern length distribution.

on the *initial hash stream* from the first kernel, the *input packet stream* and the *intermediate hash stream* from the previous stage. Since the second kernel consumes more time than the first one, we focus on the second kernel for further discussion and analysis.

5.3 Experiments and Results

The performance metrics in which we are most interested are (1) false positive error rate and (2) processing speed. In other words, by implementing the approximate filtering scheme over

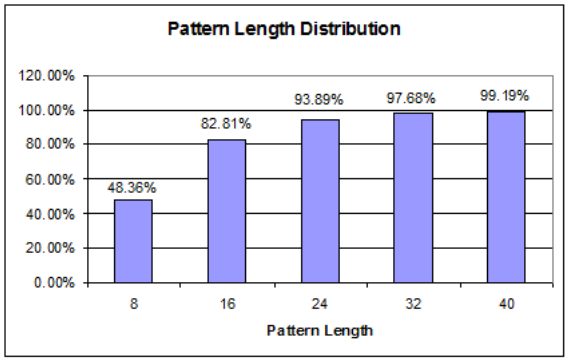

Figure 5.5: The accumulated length distribution of Snort rule contents, where 93% of the total patterns has the length less and equal to 24 bytes.

the traffic stream, how high are the accuracy and throughput? We describe the methodology of the pattern-matching experiment based on Snort distribution [91] and DEFCON9 [31] packet traces. A detail discussion on the heuristic named "*reduced pattern length*" is presented and the performance for both false positive error rate and processing speed are shown later in this section.

5.3.1 Reduced Pattern Length

In this experiment, only the pattern signatures (known as "*content*" in the Snort rules) are extracted and used as the golden patterns. In Snort, short patterns are usually used along with other *options* such as the *depth* and *offset* to minimize the false matches. Due to the ineffectiveness of the short patterns which cause many matches within a packet, the patterns with 4 bytes and less are skipped in the experiments.

There are approximately 2160+ rules which contain 2700+ distinct patterns in the Snort distribution. The accumulated pattern length distribution is shown in Figure 5.5. Among those patterns, 97% have length less than 32 bytes, and the longest one contains 122 bytes. Only 3% of the patterns have lengths sparsely distributed across the range from 33 to 122 bytes. To scan only for those pattern lengths, the efficiency gained from the iterative operation model described in the previous section may be very poor due to the SIMD architecture. On the other hand, the computational load is too high to scan all the patterns consecutively from length of 33 to 122 bytes.

A scheme named *reduced pattern length* is adopted for scanning the packet for suspicious patterns. For a pattern with length greater than a specified *cutoff length* l, the length l is used instead

of the original pattern length. This is based on the observation that a large enough *cutoff* could include the entire pattern for the majority.

Several heuristics can be applied to select a portion of l bytes (the *cutoff length*) from the original pattern. A *central-weighted* scheme shown in Figure 5.6 can be used to select the central portion of the string quite effectively for patterns listed in Table 5.1. For example, given a *cutoff length* of 9 bytes and a 30-byte string of "***filename=\"ICQ_GREETINGS.EXE\"***", the pattern offset is 10. Therefore, the new cutoff pattern selected from the central portion of the original string is "***"ICQ_GREE***". Note that, a scheme of selecting the first 9 bytes of the original pattern would not be able to distinguish pairs of patterns that share the same prefix in Table 5.1.

Table 5.1: The example of patterns.

filename=\"CHESTBURST.EXE\"
filename=\"ICQ_GREETINGS.EXE\"
Content-type\: video/x-ms-asf
Content-type\: audio/x-mpegurl
/cfdocs/exampleapp/email/application.cfm
/cfdocs/exampleapp/publish/admin/application.cfm

Although false matches may occur due to the scheme, chances are low and only a fraction of the patterns actually contribute to this type of errors as we select longer *cutoff length*. For example, the chance to find a substring of "***ame=\"ICQ_GREETINGS.***" in the packet which does not contain the exact pattern of " ***filename=\"ICQ_GREETINGS.EXE\"***" is low.

Simulation results based on several DEFCON traces indicate the effectiveness of certain *cutoff lengths*. Figure 5.7 shows the number of extra packets that need to be further inspected due to false matches, using different *cutoff lengths*. For both the *shift-add-xor* and tabulation schemes,

```
if (length>CutoffPatternLength) {
  PatternOffset=((int)floor((length-CutoffPatternLength)/2));
}
else {
  PatternOffset=0;
}
```

Figure 5.6: The pseudo code of the central-weighted scheme. Along with the *CutoffPatternLength*, the *PatternOffset* is used as an offset from the first byte to select a new pattern from the original one.

there are less than 100 false-match packets once the *cutoff length* is greater than 16 bytes. The *exact-match* curve shows the false-match packets based on an exact matching algorithm (Boyer-Moore). We highlight the area under the *exact-match* curve since the false matches are due to the reduced pattern length. As the *cutoff length* increases beyond 24 bytes, there are no such false match packets. Therefore, the false matches for the *shift-add-xor* and tabulation schemes outside the shaded area are due to the false positive error rate of the Bloom filter. The simulation is based on the DEFCON9 eth3.dump trace [31] which contains 1,691,267 packets. The true packet hit count based on the original patterns are 832,484. Therefore, the overhead of inspecting ~100 additional packets is very small. Although a dynamic method of analyzing the hamming distance on each of the pattern larger than the *cutoff length* can be applied, the simple *central-weighted* heuristic is used for all the simulations. The immediate benefits for this tradeoffs is tremendous since the computation complexity is reduced to $O(n)$ for a large number of patterns in different lengths. Moreover, the efficiency of SIMD operation can be maintained and the iterative appending process can be applied.

As expected, both the *shift-add-xor* and tabulation schemes demonstrate good performance with almost the same false positive error rate of 5.6×10^{-9}. The total number of hash functions is 16 and the *cutoff length* is 16 bytes. Therefore, the $\frac{m}{n}$ ratio is 41, which renders the expected false positive error rate of 1.43×10^{-8}.

We deliberately set the configuration of the third scheme, marked as "Conv." in Figure 5.7 to contrast the performance. The third implementation is based on the string-to-integer conversion scheme shown in Section 5.2.2. The false positive error rate is much higher since the distribution is not uniform due to the *base* is in the power of two. Moreover the conversion itself is not ideal: it maps many different strings into the same number [1]

5.3.2 The Kernel and Stream Level Performance Results

The kernels are implemented based on three different types of universal hash functions. Those are the *shift-add-xor* shown in Table 3.1, the tabulation method and the H_1 with radix conversion based on Mersenne prime in Section 5.2.2. The simulation and kernel scheduling results are based on the system configuration of eight clusters, and each has a single-read-port scratchpad of 512 words. The system consists of eight hash functions with clock frequency of 500 MHz. The main processing loop is located in the second kernel. For each iteration, hash values are calculated

[1] The base number is 512, byte of 0x0 is mapped to 0x100, and two prime numbers, $2^{31} - 1$ and $2^{19} - 1$, are used. A quick fix to lower the probability is to use different prime numbers in the hash family. In other words, the chance of having the relation $h({}^{m}C_1) = h({}^{m}C_0) + cpq$, is lower than $h({}^{m}C_1) = h({}^{m}C_0) + cq$.

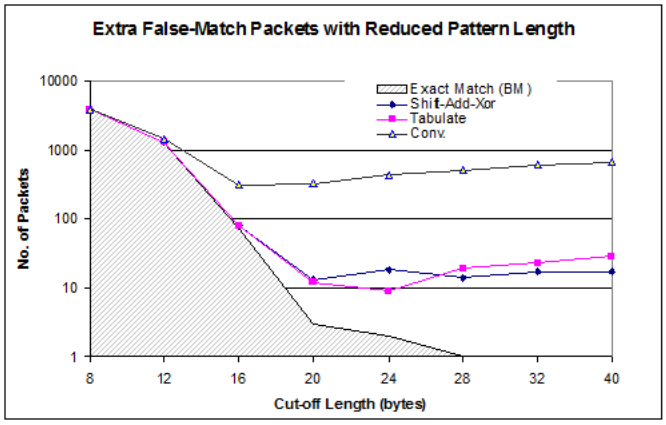

Figure 5.7: The extra packets need to be inspected with different *cutoff lengths*. The simulation is based on the DEFCON9 Eth3.dump trace, which contains 1,691,267 packets.

based on the same records of the incoming packet stream.

Shown in Figure 5.8, the cycle counts are almost reduced in half with the adders doubled from 4 to 8. As these results suggest, there is abundant data parallelism within the kernel. The tabulation scheme, on the other hand, reveals itself with lower cycle counts while having limited ALU resources. By comparing to the *shift-add-xor* scheme, the cost of the multiplication and Mersenne prime operations can be seen in Figure 5.8. As expected, the tabulation method has less parallelism as we increase the number of ALUs.

The cycle-based, stream level simulation for *shift-add-xor* scheme is shown in Figure 5.9, using the 8-adder configuration. The throughput for 1500-byte packets is approximately 400 Mbps, while performance of the shortest one barely tops line of 150 Mbps. The lower throughput for smaller packets is mostly due to the fixed operation cost of the first kernel and the short-stream effect [50, 95]. According to the kernel scheduling results, the tabulation scheme has shorter cycle counts than that of the *shift-add-xor*. However, the stream level simulation of 1500-byte packets does not show any performance gains, due to SRF stalls in the second kernel.

5.4 Design Exploration and Analysis

The whole system contains two major portions, i.e. the calculation of the hash values and the queries of Bloom filter. Thus, there are several ways to construct the systems. The first option is to calculate the hashes by the ALU and hold the Bloom filter in scratchpad at each cluster. The

second is to alleviate the computation of hashes within the cluster through table-lookup by using Index SRF [48] accesses while hosting the Bloom filter in scratchpad. The third is to accommodate the Bloom filter in SRF while hash values are computed by the ALUs at each cluster.

The realization of universal class of hashing functions can be roughly categorized into two styles, i.e. the arithmetic and tabulation. The arithmetic style, in general, requires lots of ALU resources. Therefore, for a system without enough computing resources, table lookup is popularly favored to obtain the hash value quickly for better system performance [117].

The Imagine stream processor provides a good infrastructure for implementation of both styles. At the cluster level, the functional units are designed for handling compute-intensive tasks. On the other hand, the scratchpad memory, which provides low latency data access, is a good candidate for holding small tables. Sometimes, depending on the hash function used, the size of the scratchpad may not be enough. Stream Register File (SRF), which is the next closest storage location to the processing unit can be used instead. The SRF is composed of banks of SRAM array and is optimized for delivering sequential chunks of data interleaved among the banks. The recent proposed indexed SRF [48] access extension provides the capability for non-sequential accesses, which typically require random indexing from each cluster.

5.4.1 Table Lookup for Hashing

An implementation of table lookup for both hashing and Bloom filter query at different memory hierarchy levels utilizes most of the memory bandwidth provided by the Imagine architecture. The long latency due to the SRF index read can be overlapped by the computations as well as the scratchpad access. A class of universal hash functions based on the tabulate method shown in section 5.2.2 is implemented. A stream S_m is assigned to hold a family of tables $a_t[m][x_i]$ where $t = \{1, 2, ..., k\}$, and k denotes the number of hash functions desired. The parameter m represents a specific length of a target pattern. For a given byte x_i, there are 256 different values. Therefore, a stream will hold $256 \times k$ entries, with each entry capable of holding 32 bits of hashing space. Since a kernel is processing a limited set of pattern lengths, for example $m, m+1, m+2, m+3$, only four streams $S_m, S_{m+1}, S_{m+2}, S_{m+3}$ are needed.

The realization is based on the "*in-lane access*" scheme; therefore, each row of the table $a_t[m][x_i]$, $x_i = \{0, 1, ..., 255\}$ has to be arranged in an interleaved form instead of a linear one along the stream. In other words, the x_ith row of a table $a_t[m][x_i]$ will be located in the position of $cid() + N \times x_i$ in the stream S_m, where N represents the total number of cluster in the processor

and $cid()$ denotes the *id* of the cluster currently accessing the table. For example, if the second cluster is accessing row 3 of the table in an 8-cluster system, the stream index of that location is $1 + 8 \times 2 = 17$. The scheduling result of the tabulation scheme is shown in Figure 5.10.

The cycle counts from the scheduling result is expected to be reduced compared to that of the *shift-add-xor* since less computation is needed. However, the reduction is limited due to the extra indexing overhead. For example, given a byte value, each cluster has to calculate the new stream index of $cid() + N \times x_i$ for *in-lane* reads of the hash value. The stream level simulation shows that the SRF bandwidth utilization increased 4.6 times, while the LRF bandwidth decreased only 9% compared to that of the *shift-add-xor* implementation. Although the index transformation can be done by the cluster itself, an extra *hardware assist* can be a great help to offload the computation. Without the extra index calculation overhead, almost 21% of the cycles can be further eliminated in Figure 5.10. Simulation results indicate a nearly 15% improvement in throughput for 1500-byte packets.

5.4.2 Bloom Filter Query

The Bloom filter is a bit-oriented data structure which can be easily packed and stored in the scratchpad memory at each cluster. Therefore, in the query phase, a *bit-extraction* procedure of masking and shifting is needed to locate the bit in order to identify the value. For example, given a 13-bit hash value, the upper 8 bits are used to address the scratchpad while the lower 5 bits are used to pinpoint the bit position within the 32-bit word read from the scratchpad. The *bit-extraction* alone takes almost 40% of the operations in the second basic block of kernel 2. This is a good example showing the inefficiency of processing the bit-oriented data structure in a word-oriented architecture. A hardware assist implementation which provides a dedicated resources may effectively resolve the overheads due to the *bit-extraction* process. Performance gain and speedup are commonly seen with similar approaches by incorporating special hardware assists [95] or special instructions like field extraction, byte-wise and boolean operations [93, 42].

As we optimize the scheduling results, i.e., offload the computation cycles by hardware assists, the scheduling result shown in Figure 5.10 reveals the possible critical path due to sequential read of the scratchpad memory. The current design of the scratchpad memory has only one read port, which sometimes leads to the bottleneck for the table-lookup intensive applications [95, 66]. A significant speedup is expected with the implementation of additional read-port and the hardware assists discussed.

5.4.3 Bloom filter in SRF and Main Memory

The counting Bloom filter [37] is introduced to embrace the membership deletion capability. This particular construction and its variants are widely used in many network applications [11], including network traffic measurement, distributed web caching and queue management. The implementation is based on an array of m-bit counters instead of a bit-wise memory array. Thus, the size of the implementation increases m-fold. The stream register file is a natural place to hold such data structure since it is the next closest storage location to the ALUs. Moreover, the access latency may be well overlapped by the computation of the hash values.

The design of multi-stage filters for traffic measurement [35] is one good example of such an implementation. For each stage with 1000 buckets, assuming each bucket has four bytes, the total storage requirement for an 8-stage (cluster) configuration is only 32,000 bytes, which can be easily fit into the SRF. Assuming 800Mbps link with 100,000 flows, this configuration is able to identify the flows above 1% of the link during a one second measurement interval with the error probability of 2.3×10^{-8} [35]. We have not yet implemented such application, but we are planning to do this as future work.

Another option is to hold the data structure in the main memory. This option only makes sense if the memory access latency is comparable to that of the computation. By using software pipeline techniques at the stream level, the latency can be effectively hidden. Since the index stream mechanism is optimized for sequential access, the randomized table lookup may introduce long delay. Moreover, the large amount of lookups may throttle the limited memory bandwidth, causing poor system performance. Our cycle-based simulation does show poor performance for this scheme, with a throughput under 90Mbps for 1500-byte packets.

5.5 Conclusions

This chapter explores the implementation of Bloom filter for content matching on the stream architecture. The calculation of hash values is transformed into stream processing, expressing producer and consumer locality and achieving efficient utilization of the unique memory hierarchy. By arranging multiple processors in a pipelined fashion, the system is capable of processing patterns extracted from the rules of the Snort distribution and achieving a throughput of 400Mbps for 1500-byte packets. Since the core of Bloom filter design is based on a family of independent hash functions, we demonstrate the flexibility and performance of the stream architecture to support

the realization of such constructs for the Bloom filter. In addition, the efficient implementation of universal class of hash functions may benefit many applications, such as message authentication codes (MACs) [82] and streaming data processing [117].

We explore the feasibility and flexibility of supporting Bloom filter in the stream architecture with some possible modifications suggested for better performance. The unique scheme of Bloom filter plays an important role in emerging network applications [11]. Its variants can be good vehicles to perform event counting and classification. For example, instead of using the single-bit array, the counter-based memory array is used for statistics collection on the traffic with the same attribute e.g, source/destination IP address, source/destination port address and protocol number. Moreover, the same concept can be applied to worm and virus signature detection [113].

To continue our investigation into stream-based network processors, we take this as the base and look forward to exploring the similar data structure for traffic analysis in the next chapter.

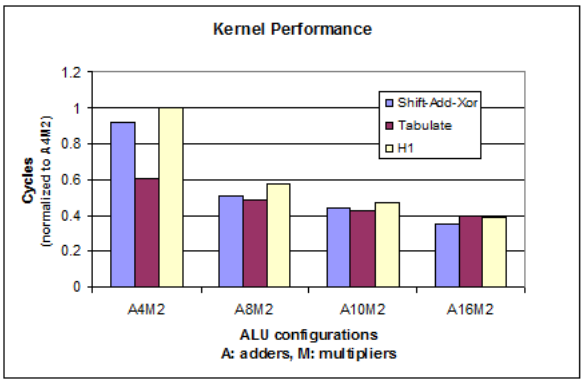

Figure 5.8: The kernel scheduling results for different type of hash functions.

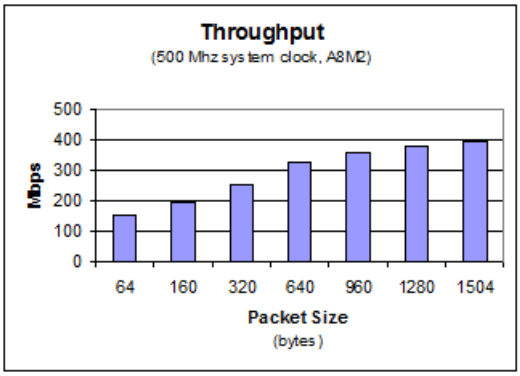

Figure 5.9: The Stream level simulation results for different sizes of packet. The implementation is based on the *shift-add-xor* scheme.

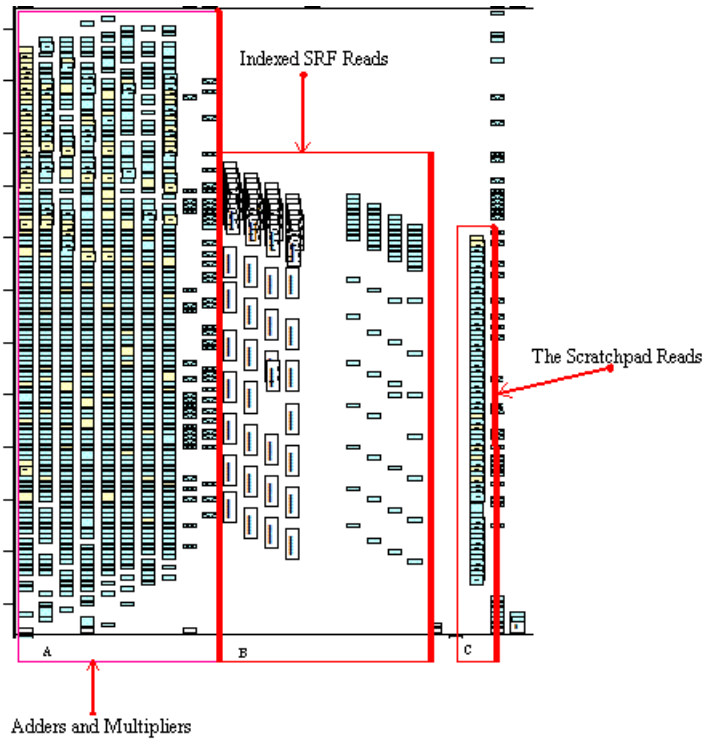

Figure 5.10: The second basic block of the IScd scheduler result for indexed SRF accesses.

Chapter 6

Streaming Data Processing

6.1 Introduction

The accurate estimation of Internet traffic statistics serves as the basis for infrastructure planning, network provisioning, capacity forecasting and accounting [127, 65, 35]. Anomaly detection on worm distribution and prevention of distributed denial of service (DDoS) attacks are also based on the same information. Therefore, traffic analysis and measurement have been the important tasks for the proper operation of IP networks [38]. As network bandwidth grows exponentially, the scaling of monitoring and measuring capabilities for collecting accurate statistics becomes a critical issue [35]. For example, given a bandwidth of 10 Gbps and a minimum-sized IP packet (40 bytes), the time to process each packet is 32 nsec. Moreover, there is only 8 nsec available for processing such a packet if the bandwidth increases up to OC-768.

The streaming data processing model is well fitted to measuring and monitoring heterogeneous and dynamic phenomena. Internet traffic, too, can be naturally regarded as a data stream since packet data arrives rapidly as a series of elements. The challenge we are facing is to process a potentially unlimited amount of data in a limited time and space. In addition, each element or record of the data stream might have only one chance to be examined. Therefore, these issues have recently drawn a lot of joint efforts [77, 40, 29, 100] from research communities of database, networking, architecture and theoretical computer science. Research is conducted in regard to the fundamental algorithmic models, programming languages and hardware and software support of

data stream management systems. More details are provided in the comprehensive surveys and tutorial [4, 41, 80].

With limited storage, computation power, and processing time, achieving the goal of *one pass processing* over a massive volume of data is extremely challenging. As a continuing research effort, the study of IP traffic analysis in the domain of data stream model over the Imagine stream architecture is presented in this chapter.

The backgrounds of the streaming data model will be introduced first with the emphasis on the important statistical attributes of the data stream. Although there are several other techniques for data reduction and synopsis construction such as sampling and wavelet [80, 4, 41], we focus on the sketch scheme, due to its wide applicabilities to various networking applications. The general introduction on Count-Min sketch [19] and K-ary sketch [60] are provided first. We transform the sketch processing into stream operations provided by the Imagine programming model. The focus is to explore the applicability of the specialized data structure and operation over the tiered memory hierarchies: Main Memory, Stream Register File (SRF) and Local Scratchpads.

A pipelined architecture over three Imagine processors is presented for the sketch-based change detection. We demonstrate the pipelined operation and analyze the system bottleneck throughput. The simulation shows that the processor is capable of handling the sketch update at 10 Gbps for the minimum-sized IP traffic.

6.2 The Background

A data stream $\phi = (a_1, a_2, ...)$ is a massive sequence of elements. Each element, $a_t = (k_t, u_t)$ consists of a *key*, $k_t \in \{0, 1, \ldots n-1\}$ and an *update*, u_t. There are two different models based on the property of the *update*. It's named *cash register* model if $u_t \geq 0$, and *turnstile* model for $u_t \in \mathbb{N}$. In IP traffic analysis, we are interested in enumerative (e.g. the number of packets in a flow) or cumulative values (e.g. the total number of bytes in a flow). Since these values are always positive, the *cash register* model applies. The key k_t in the data stream model can be used to represent the traffic flow. A *flow* is usually defined as a stream of packets with some common attributes. For example, it can be packets having the same pair of *source* and *destination* IP address. Another popular flow consists of the same 5-tuple attributes: the source and destination IP address, the source and destination port number and the protocol number.

The statistical measurement of data stream ϕ can be reflected by the *frequency moments*.

The z^{th} *frequency moment* F_z of data stream ϕ is defined as follows.

$$F_z = \sum_{k=0}^{n-1} f_k^z \tag{6.1}$$

The f_k represents the number of occurrences of key[1] k in the data stream of ϕ. Therefore, the number of *distinct values* in the data stream ϕ can be expressed as F_0. The *length* of the data stream ϕ is F_1, the first frequency moment of ϕ. The second frequency moment F_2, also known as the *repeat rate* or *Gini's index homogeneity*, is of particular interest and is widely used in the database community [1]. For the higher frequency moment, where $z \geq 2$, it indicates the "*degree of skewness*" of a data stream [1].

The L_2 norm, another important statistic attribute for many data stream applications, has a close relation to the *second frequency moment* F_2. For an update model like *cash register* and *turnstile*, the L_2 norm of the data stream ϕ is shown as follows:

$$L_2 = (\sum_t |u_t|^2)^{\frac{1}{2}} \tag{6.2}$$

Computing the exact answer of these attributes is very difficult because of the *space* constraint. For example, a naive implementation for counting the exact number of *distinct values* in the data stream ϕ has to use a counter for each element in the key space. Because the key space can be very large, this is not practical. For example, the 124-bit 5-tuple IP key space would require 2^{124}counters.

Therefore, research efforts have been focused on approximation approaches to estimate these attributes for applications like error estimation [47], data partitioning based on the degree of skewness [32] and abnormality detection for IP traffic [60].

6.3 The Sketch-based Algorithms

Sketch [15, 1, 80] is a powerful yet compact data structure capable of synopsizing substantial numbers of data elements without keeping its stateful information. The probabilistic property of this structure also provides a mathematical guarantee for accurate estimation on various attributes of the data streams.

[1]For convenience purpose, we discard the time subscript t for referring to the current state of the element over the data stream [80].

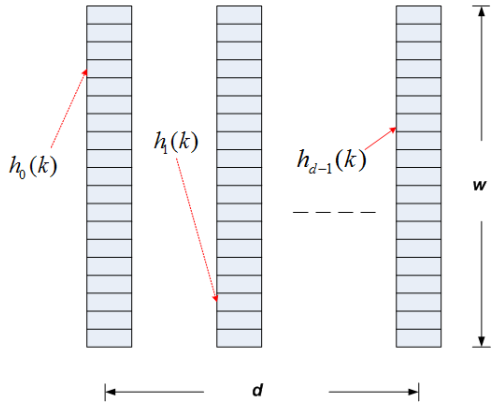

Figure 6.1: The two-dimensional ($w \times d$) array of counters. (w rows and d columns)

In general, the sketch algorithms rely on the probabilistic property of the universal class of hash functions (described in Chapter 3). Upon the arrival of each element in the data stream, the key is hashed and the update is applied for a counter determined by the hash value. A *sketch* data structure contains the final values of these counters in a specific time interval. Due to its linearity, we can combine several sketches together for query processing. A query is the estimation of a specific attribute on the data stream based on the sketch collected. Each algorithm has its own way of making the estimation. A detailed analysis and comparison of various sketch algorithms can be found in Cormode's survey [19].

Generally, there are two phases in the operation of sketch algorithm: the *update* and *query*. We highlight the update phase and then, illustrate two sketch algorithms in the following sections due to their excellent properties.

6.3.1 Count-Min Sketch

The sketch utilizes a two-dimensional array of counters $\mathbb{C}[i][j]$, where $0 \leq i < d, 0 \leq j < w$. As shown in Figure 6.1, d represents the number of arrays and w is the number of entries within each array. The two-dimensional array of counters is indexed by a set of independent hash functions $H = \{h_i,\ 0 \leq i < d\}$. Each hash function h_i maps a key $k \in \{0, 1, \cdots, n-1\}$ into the hashing space of $\{0, 1, \cdots, w-1\}$.

Initially, all the counters in the 2-D array are set to zero. As each element $a = (k, u)$ of the data stream arrives, the key k will be hashed by the set of d hash functions. Then, the value of u

will be updated into the array of counters indexed by the d hash outcomes, as illustrated in Equation 6.3.

$$\mathbb{C}[i][h_i(k)] = \mathbb{C}[i][h_i(k)] + u,\ 0 \leq i < d \tag{6.3}$$

The Count-Min sketch is capable of providing approximate estimation of a query for which the accuracy is guaranteed within an error factor of ε at a probability of δ. The values of d and w are determined by the parameters ε and δ: $d = \lceil ln\frac{1}{\delta} \rceil$ and $w = \lceil \frac{e}{\varepsilon} \rceil$, where e denotes the base of the natural logarithm function. The two-dimensional array of counters is indexed by a set of *2-universal* hash functions.

The Count-Min sketch can provide the (ε, δ) approximation of a point query over a data stream. By given a key k, the estimated point query is shown in Equation 6.4.

$$\hat{A_k} = min_i\{\mathbb{C}[i][h_i(k)]\}, 0 \leq i < d \tag{6.4}$$

The estimated value of $\hat{A_k}$ has the guarantee of $A_k \leq \hat{A_k}$, where A_k represents the total update value for the key k. The error is bounded (shown in Equation 6.5) with probability at least $1 - \delta$, where $\|a\|_1 = \sum_{k=0}^{n-1} |A_k|$.

$$\hat{A_k} \leq A_k + \varepsilon \|a\|_1 \tag{6.5}$$

An example of using $(\varepsilon, \delta) = (0.0001,\ 0.0005)$ yields an estimation result where there is a 99.95% chance that the error factor is within 0.01%. For this example, the data structure of the Count-Min sketch consists of 8 hash functions ($d = \lceil ln\frac{1}{0.0005} \rceil$) and 27183 counters ($w = \lceil ln\frac{e}{0.0001} \rceil$) in each array. Assuming a 32-bit counter, the total size of the data structure is $869,856$ bytes.

The Count-Min sketch can also provide the estimation for range query and inner product query. These queries are all related since the range query is essentially *"a sum of point queries*" and both point and range queries are "*specific inner product queries*" [19].

One of the important applications based on the point query is the finding of top-k items [15, 21]. For example, the estimated answer can be used to identify the top 10 flows of the IP traffic or top 100 IP addresses in terms of bandwidth consumption.

Another important contribution is the estimation of the L2 norm. For a skewed data ($z > 1$ in the Zipf's distribution model[2]), the estimated L2 norm is shown as follows.

[2]In the Zipf's distribution model, the frequency of the *k*-th most frequent item f_k is captured by the ratio of a scaling constant of C_z and the *k*'s power of parameter z. That is, $f_k = \frac{C_z}{k^z}$.

$$min_i\{(\sum_{j=0}^{w-1}\mathbb{C}[i][j]^2)^{\frac{1}{2}}\} \tag{6.6}$$

6.3.2 K-ary Sketch

K-ary Sketch is proposed by Krishnamurthy et al [60] based on the same update process illustrated in Equation 6.3. It utilizes the same two-dimensional ($w \times d$) array of counters as those in the Count-Min sketch. However, the counters have to be indexed by a set of *4-universal* hash functions.

The approximation of a point query over a data stream is different as well. By given a key k, the estimated point query is shown in Equation 6.7.

$$\hat{A}_k = median_{i\in[d]}\{\frac{\mathbb{C}[i][h_i(k)] - \mathbb{Z}(S)/w}{1-1/w}\}, 0 \leq i < d \tag{6.7}$$

where

$$\mathbb{Z}(S) = \sum_{j\in w}\mathbb{C}[0][j] \tag{6.8}$$

The variance of the above point query is bounded by $\frac{F_2}{w-1}$, where F_2 represents the second moment of the data stream.

The estimation of the second moment $\hat{F}_2$ of the data stream is defined as follows with bounded variance of $\frac{2F_2^2}{w-1}$.

$$\hat{F}_2 = median_{i\in[d]}\{\frac{w}{w-1}\sum_{j\in[w]}(\mathbb{C}[i][j])^2 - \frac{\mathbb{Z}(S)^2}{w-1}\} \tag{6.9}$$

6.4 Sketch Operation on Stream Architecture

6.4.1 The Stream Programming Model

The sketch's 2-D array data structure can be represented as a stream S in the stream programming model. The stream S consists of $d \times w$ records, where each record is a 32-bit integer. The i-th record of the stream S is located in the entry $\lfloor \frac{i}{d} \rfloor$ of the ($i \bmod d$)-th array, as illustrated in Equation 6.10.

$$S[i] = \mathbb{C}_t[i \bmod d][\lfloor \frac{i}{d} \rfloor],\ 0 \leq i \leq d \times w - 1 \tag{6.10}$$

For a d-cluster Imagine processor, each cluster can perform a read-modify-write operation over the w by d array for each *update* of the incoming data stream through *indexed SRF accesses* [48] in a SIMD fashion.

The sketch $S(t)$ is usually updated for a time interval $\triangle t$ started at time t. It can be stored back to the main memory while a new sketch $S(t + \triangle t)$ is being updated. Given a set of sketches (or streams $S(t), S(t + \triangle t), \cdots, S(t + (n-1) \triangle t)$ in stream level programming model), the processor can linearly combine these sketches very efficiently in a vector style operation. A generic combining process over a set of n sketches[3] is illustrated as follows.

$$COMBINE(S) = \sum_{s=0}^{n-1} S(t+s) = \sum_{t=0}^{n-1} \mathbb{C}_t[i][j],\ i \in d,\ j \in w \tag{6.11}$$

An estimation and query operation over the combined time interval can be performed based on the newly generated combined sketch. As shown in the Equation 6.6, 6.7 and 6.9, these operations are highly data parallel and fit very well in the SIMD processor architecture.

6.4.2 The Point Query

At the kernel level, the query can be simply accomplished by the *indexed SRF access* directly with the hashing values for each cluster in a SIMD fashion. The point query operation can also be performed efficiently by the *Indexed Stream* access at the stream level. Let's take a look at the following example on how a point query can be done by given a key k_1 and a stream S in the main memory.

There are d different indexes $h_0(k_1), h_1(k_1), \cdots, h_{d-1}(k_1)$ after key $k_1$is hashed by d hash functions. Then, an index stream S_{idx} can be formed by converting the above indexes according to Equation 6.10.

$$S_{idx} = (\,(h_0(k_1) \times d + 0),\ (h_1(k_1) \times d + 1),\ \cdots,\ (h_{d-1}(k_1) \times d + (d-1)\,) \tag{6.12}$$

The query stream S_{quy} can be derived by the following indexed stream operation.

[3] As the sketch operation is always based on the time interval $\triangle t$, we use $(t + i)$ for short instead of $(t + i \cdot \triangle t)$.

$$S_{quy} = S(\,0,\,(d \times w - 1),\,im\_acc\_index,\,S_{idx}\,) \tag{6.13}$$

Then, the query stream S_{quy} can be sent to the host processor or another kernel to pick up the minimum or median value.

6.5 Sketch-based Change Detection

The change detection of network traffic is referred as the monitoring of significant differences in traffic attributes over two observing intervals. The attributes of interest can be the number of packets, flows, or total bytes of the traffic.

Finding the significant changes in the network traffic is one of the key building block for several important applications. For example, it can be used for the billing and usage tracking by the Internet service providers. It can also be used for anomaly detection for network security purpose to prevent worms and DoS attacks.

Change detection is especially challenging at the network level because a massive amount of data has to be processed in real time at wire speed. Typically, sampling is a popular method applied to tackle the processing and storage cost incurred by the huge amount of traffic. Sketch is another interesting approach as it offers low space requirement and guaranteed accuracy.

The difference of traffic attribute, can be characterized in the following three types: *variational difference*, *relative difference* and *absolute difference*. The *relative difference* refers to the ratio of difference between two observing time interval. And, the *variational* means the large variance over multiple time periods [20].

In the following section, we take the sketch-based scheme [60] (based on K-ary Sketch) as an example to illustrate the efficiency of the stream architecture. The scheme is referred as the *absolute difference* as it looks for the large difference between two observing time intervals.

6.5.1 The Pipelined Architecture

The scheme [60] consists of three major modules: sketch, forecasting and detection. The three modules are transformed as kernels residing in three different stream processors connected by a communication network. The first kernel constantly updates the observed sketch S_o for a specific time interval $\triangle t$ as illustrated in Equation 6.3. The sketch S_o will be sent to the second processor

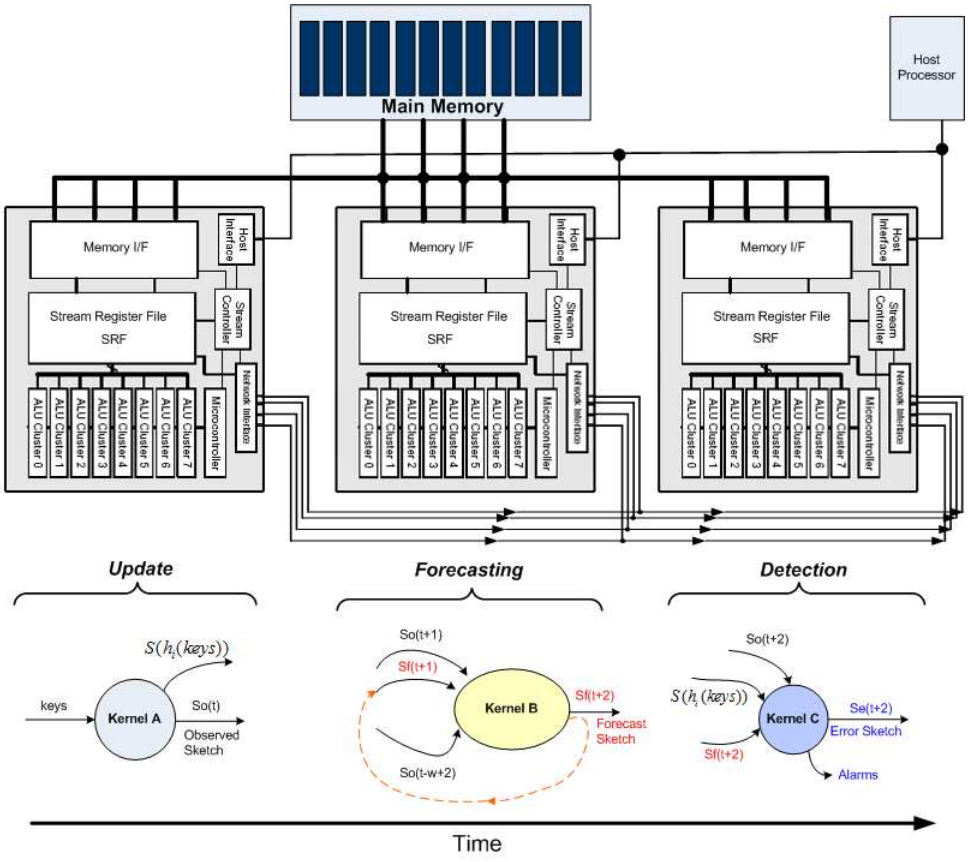

Figure 6.2: The architecture of sketch-based change detection over Imagine Stream Processor.

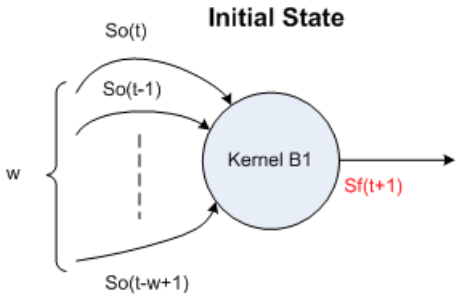

Figure 6.3: The kernel diagram for the *w*-moving average calculation at time $t+1$.

where sketch combine operation can be performed. In this kernel, it produces a series of forecast sketches S_f, which can be based on different forecast models as needed. The error sketch S_e is derived at the third processor by subtracting the observed sketch S_o with the forecast sketch S_f . The alarm is issued based on the threshold T_A. It is calculated based on the product of a predefined parameter T and the estimated second moment over the error sketch S_e. Given a key k, the alarm is raised if the point query of k over the error sketch S_e is larger than the threshold T_A.

Let's take the *moving average* as a forecast model for example to illustrate the stream operation. Given a data sequence of $D = \{d_i\}$, where $i \in \{0, 1, \cdots n-1\}$, the *W*-moving average MA_W of data sequence D is defined as follows.

$$MA_W = \frac{1}{W} \sum_{j=i}^{i+W-1} d_i \tag{6.14}$$

At this kernel, as shown in Figure 6.3, it computes the forecast sketch $S_f(t+1)$ (at time $t + 1 \cdot \triangle t$) based on the previous W observed sketches corresponding to the past W intervals.

$$S_f(t+1) = \frac{1}{W} \sum_{i=0}^{W-1} S_o(t-i) \tag{6.15}$$

Due to its linear property, the forecast sketch calculation can be computed incrementally and recursively. For example, at time $t+2$, the calculation of $S_f(t+2)$ is illustrated as follows.

$$S_f(t+2) = S_f(t+1) + \frac{1}{W}(S_o(t+1) - S_o(t-W+1)) \tag{6.16}$$

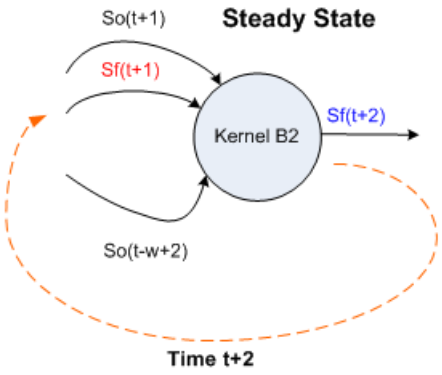

Figure 6.4: The kernel diagram for the *W*-moving average calculation at time $t + 2$.

The kernel does not need to calculate the forecast sketch based on the observed sketches in the past W interval. Since the SRF still holds the sketches of $S_f(t + 1)$ and $S_o(t - W + 1)$, the kernel only needs to take the latest observed sketch $S_o(t + 1)$ as input stream to speedup the computation.

The kernel diagram shown in Figure 6.4 demonstrates a classic producer-and-consumer locality captured by SRF in the stream processor.

The whole operation is arranged in a pipelined fashion. Figure 6.5 shows an example by using a window size of four update intervals $(W = 4)$. The first forecast sketch $S_f(t + 1)$ is generated shortly after the beginning of time $t + 1$. As the system finishes the update process at $t + 1$ where the observed sketch $S_o(t + 1)$ is available, the error sketch $S_e(t + 1)$ can be computed immediately at time $t + 2$. Thus, the alarms can be generated thereafter.

6.5.2 One-pass Processing

The query process of sketch based scheme has to rely on the original key. That is, the sketch data structure does not contain any information regarding the key itself (the IP source address in the example above). Therefore, the key streams have to be stored and then used for the query process. This may limit the scalability due to the extra cost for the storage. A way to avoid this is to use the current arriving key streams for the query process over the previous sketch data structure [60].

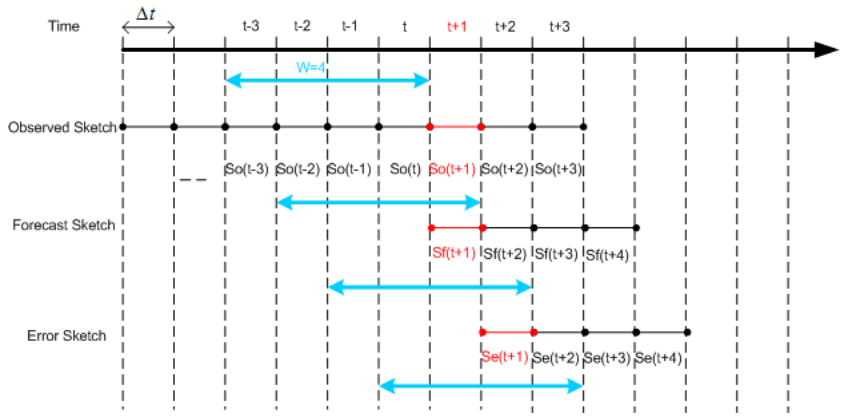

Figure 6.5: The pipelined operation of sketch processing.

For example, the alarm is created at the third processor by using the current key over the previous error sketch. As shown in Figure 6.5, the alarms can be generated at time $t + 2$ by using the current keys based on the error sketch $S_e(t + 1)$ created at time $t + 1$.

The drawback of using the current keys over the previous error sketch is that the system is likely to miss the detection or produce a false alarm, because the previous keys updated in the sketch are not showing up again in the current key stream.

As we decrease the update time interval $\triangle t$, the chance of missing the detection or producing the false error may be lower. Moreover, we may be able to boost the accuracy by having lower chance of collision rate while doing the sketch update. We will discuss this interesting phenomenon in Section 6.6.2.

6.6 Discussions

6.6.1 The System Bottleneck

The counter update process is usually set for a specific time interval $\triangle t$ in terms of minutes. It can be one or five minutes [5, 60] depending on the processing capability of a system. Thus, shown in Figure 6.2, the sketch processing time for deriving the forecast sketch S_f and error sketch S_e in kernel B and C is bounded by the observing interval as the observed sketch S_o is sent every $\triangle t$ time to the second and third processor.

The hash calculation for point query over the error sketch S_e in kernel C are based on the same set of universal hash functions used in the sketch update stage. There is no need to re-calculate the hash values again as those can be sent as a stream to kernel C from kernel A. Moreover, the point query for the alarm can be based on sampling [60] of the incoming key stream. For an interval of one and five minutes, the processing time is much relaxed compared to that at the first processor.

As the hashing and counter update (read-modify-write access) has to be quick enough for each incoming packet at line rate, the critical path of the change detection is in the stage of sketch update at the first processor.

The kernel of sketch update performs the hash calculation and updates the counters in the SRF by in-lane indexed stream access in a SIMD fashion for eight clusters (as described in Section 6.4.1). The kernel computes the hash value of a 32-bit key by hashing two 16-bit subwords in parallel[4]. Therefore, the abundant ILP can be exploited by increasing the number of ALUs in the cluster. Figure 6.6 shows the kernel performance of sketch update with different ALU configurations by using 2-universal and 4-universal hash functions.

For the simulation, each record of the key stream consists of a 32-bit IP source address and a packet length. The Imagine stream processor (3-adder and 2-multiplier configuration) is capable of processing 1,920 keys in total of 28,335 cycles. As each key has to be hashed by eight independent hash functions, there are total of 15,360 hash calculations (*2*-universal) and updates on the sketch counters . That is approximately, in average of 1.84 cycles per hash and update. It takes an average of 15 cycles to hash and update for a given key. The time for calculating the same number of hash values by using *4*-universal hash functions is 63,377 cycles. It takes an average of 33 cycles to hash and update for a given key.

For a system clock of 500 MHz, Imagine is capable of handling the update of minimum-sized IP traffic with the throughput of 10.8 Gbps (*2*-universal) and 4.8 Gbps (*4*-universal), respectively.

For the example shown in Figure 6.2, the observed sketch S_o is sent every $\triangle t$ time to the second and third processor. Thus, the sketch processing time in kernel B and C is bounded by the observing interval. For an interval of one and five minutes, it's much relaxed compared to that in the first processor. The interconnection network interface consists of eight stream buffers, each is capable of sustaining 2 GBps of bandwidth for a stream. Thus, the interface supports bandwidth of 16 GBps in total. However, the maximum memory throughput of 2 GBps may become the

[4]We use the hash functions described in Chapter 3.

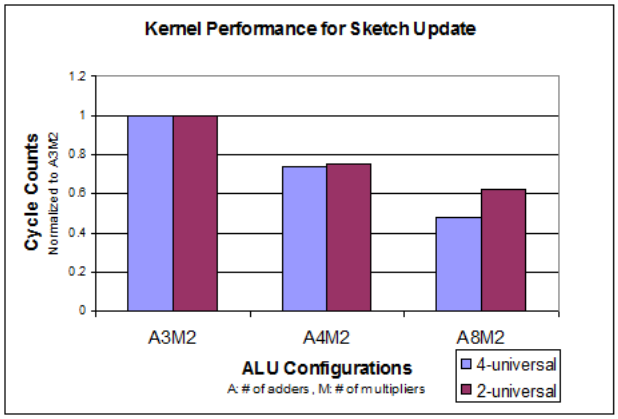

Figure 6.6: The kernel performance for sketch-update process on different ALU configurations.

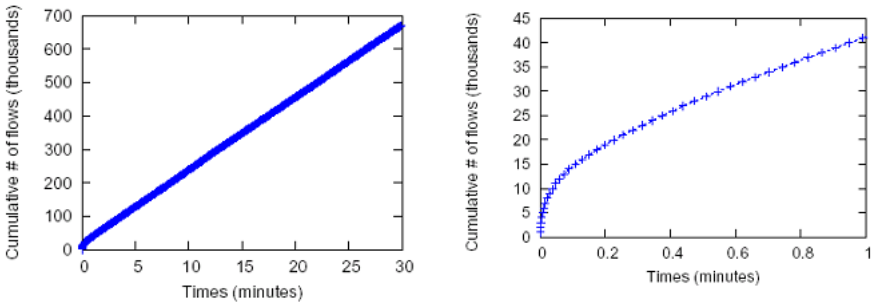

Figure 6.7: The cumulative number of flows during one 30 minutes interval (left) and 1 minute interval (right) for a OC-12 (622 Mbps) Internet backbone [5].

bottleneck of the sketch process as the system is moving data from the main memory to the SRF. It takes about 15,105 cycles to move two 32k-byte sketches from the main memory to the SRF. It takes about 8,201 cycles for to fetch one 32k-byte sketch.

6.6.2 The Estimation Accuracy

The error factor of the sketch is mostly governed by the number of counters w in the array. With limited space for holding the sketch S in the SRF, we may not be able to increase the number of counters w to a large value for better estimation accuracy for high volume traffic consisting of millions of flows.

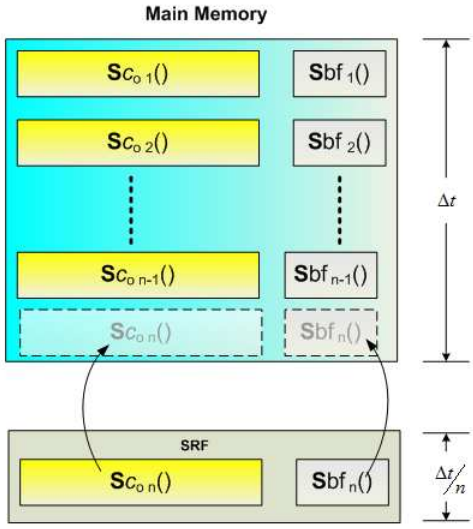

Figure 6.8: The operation of the time multiplexing scheme (TMS). At the end of the sub-sketch update period $\frac{\triangle t}{n}$, the newly observed sketch is sent to the main memory along with the Bloom filter sketch.

Generally, the cumulative number of flows scales linearly (shown in Figure 6.7) with the observed time interval [5]. Therefore, the expected number of collisions in each of the sketch array decreases as the measuring period $\triangle t$ reduced. In other words, fewer flows means fewer opportunities for collision.

This observation brings us an idea of maintaining n sub-sketches instead of one in a time period $\triangle t$. As the system performs the sketch update, the previous updated sub-sketch can be stored back to the main memory within the update time of $\frac{\triangle t}{n}$ without incurring extra latency. Figure 6.8 shows the update of n sub-sketches.

Based on the linearity, the final estimation of a point query (take Count-Min sketch as an example) with a key k, can be presented in Equation 6.17.

$$\hat{A}(k) = \sum_{t=0}^{n-1} min_i\{\mathbb{C}_t[i][h_i(k)]\},\ 0 \leq i < d \tag{6.17}$$

Denoted as the time multiplexed scheme (TMS), it can be further enhanced by having each key k associating a small tag indicating to which the sub-sketch $\mathbb{C}_t$ the key belongs to during the updating phase. Therefore, in the query phase, the system only makes the inquiry to the sub-sketch $\mathbb{C}_t$ containing the key, which yields more accurate estimation of point query $\hat{A}(k)$.

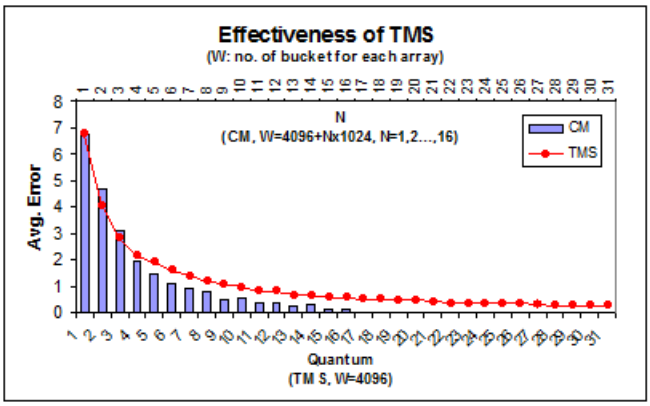

Figure 6.9: The effectiveness of TMS. PSC-1114637278-1.tsh, OC48c PoS link connecting the Pittsburgh Supercomputing Center [87]. The Quantum is denoted as the number of smaller update interval used within the original time period $\triangle t$.

In order to quantify the improvement on the accuracy for point query, we introduce the *Quantum* and the average error Err. The *Quantum* is denoted as the number of smaller update intervals used within the original time period $\triangle t$. The average error of Err is defined as follows.

$$Err = \sqrt{\frac{1}{n}\sum_{k=1}^{n}(\frac{\hat{A}(k) - A(k)}{A(k)})^2} \tag{6.18}$$

The simulation is based on a 90-second trace *PSC-1114637278-1.tsh* from an OC48c PoS link connecting the Pittsburgh Supercomputing Center [87]. Each flow, also defined as the key, is based on a distinct source address. And each point query of the key represents the total size of the flow. There are 5,167,489 packets consisting of 24,569 different source addresses.

By using *Quantum* of 3 and sketch size of w=4,096, the average error of Count-min point query is lower than the use of a single sketch of size w=7,168. Figure 6.9 shows the effectiveness of using this scheme.

6.6.3 The Tag Implementation

Given a key k, the inquiry to see if the key has been hashed and updated into the sub-sketch is a classic membership query problem. Therefore, it is a good fit to use the Bloom filter due to its compact data structure. The query results, however, are associated with a small probability of false positive error. That is, if the sketch does not contain the key, it may be revealed as the key

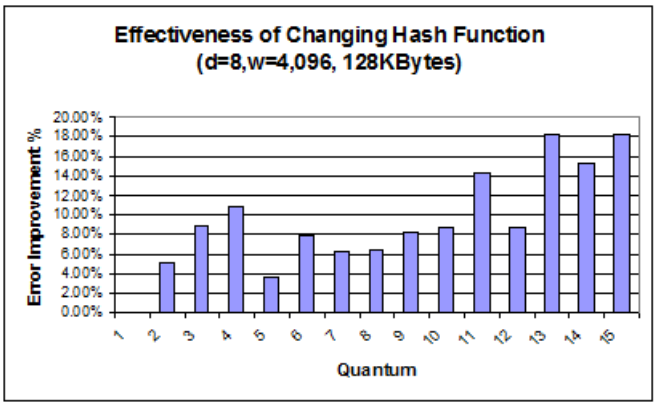

Figure 6.10: The improvement of the point query based on the change of parameters.

does exist. This low probability of false error is acceptable since the sketch algorithm itself is not precise, either.

The Bloom filter query accuracy is based on three parameters: the number of keys, the number of hash functions used and the m/n ratio[5]. Depending on the accuracy of the query, the size (in bits) of the Bloom filter data structure for each sub-sketch is shown as follows.

$$Size = \frac{Number_{key}}{Quantum} \times \frac{m}{n} \tag{6.19}$$

Let's take the simulation in the previous section for example. Assume that we construct 32 sub-sketches (quantum=32) for the update process based on the trace consisting of 24,569 flows. For each sub-sketch, the size of the Bloom filter is 960 bytes with the m/n ratio of 10. The false positive error rate is on the order of 10^{-3} by using the same 8 hash functions for the sketch updating.

We can easily compact the Bloom filter data structure as a stream $Sbf_t()$, which only consists of 240 32-bit records during the update phase.

In the point query phase, the Bloom filter query is proceeded first with the key k over the stream $Sbf_t()$. Then, the kernel can skip the query of the sub-sketch stream $S\mathbb{C}_t()$ if the result is false.

In addition, while multiplexing the sketches, the parameters (a and b in Equation 3.3) of the universal hash function can be changed randomly to achieve the *"mathematically predicted average performance"* of using universal hash functions [74]. The change of the parameters is mo-

[5]Please refer to Chapter 5 for more details of the Bloom filter operation.

tivated by Makowsky et al [74]: *"In order to be mathematically certain that the predicted average performance will be achieved, it is necessary to change hash functions periodically."* As the byte distribution within packets is not truly random, dependencies among packets are commonly found which yield the non-ideal behavior of the hash functions. Therefore, by randomly selecting hash functions from the family frequently, the average performance can be guaranteed, independent of the input keys. By comparing to the error factor of the Count-Min point query by using TMS, Figure 6.10 shows up to 18% improvement on the same scheme with the change of hash functions. The drawback of this approach is that we break the linear property within these sub-sketches.

6.7 Conclusions

The sketch algorithms have been widely used in many network management applications [80, 40, 41]. As many new algorithms [19, 60] and improvements [69, 46] have been proposed recently, we believe that an exploration on a programmable and efficient processor architecture is beneficial to applications based on the same algorithms.

In this Chapter, we introduce two sketch algorithms: the K-ary sketch and the Count-Min sketch. We illustrate the point query operation at the stream programming model with some possible improvements. These sketch operations are highly data parallel and suited for the SIMD paradigm. We also present the pipelined architecture for the application of sketch-based change detection [60] over the Imagine stream processor. As demonstrated, the sketch processing is accomplished in a recursive and incremented way, where the producer-and-consumer locality can be captured efficiently by the stream programming model.

Maintaining and updating the statistics counters for high-speed network is a challenging task. For example, upon the arrival of a minimum sized IP packet (40 bytes) in a OC-768 line (40 Gbps), a read-modified-write process has to be applied to a set of counters in 8 nsec. Assuming there are 8 counters and the size of each is 32 bits, the total memory bandwidth required is 4 GBps. With the maximum SRF bandwidth of 32 GBps, maintaining and updating the statistics counters can be effectively realized on Imagine stream processor with its computing resources and unique memory hierarchies.

As we look over the concurrent hardware architectures capable of supporting the high-speed statistics counter update [111, 96, 103], they share a common memory organization. That is, a set of faster but smaller statistics registers are implemented in SRAM and these register values are

periodically updated into a larger but slower set of counters in DRAM.

The sketch operation in Imagine processor can be modeled in a similar but at a coarse-grained fashion based on the stream programming model and the tiered memory hierarchy efficiently. In order to contrast the performance, we continue to explore the sketch update algorithm on a different processor architecture in the next chapter.

Chapter 7

Intel IXP Network Processor

As discussed in the previous chapter, the sketch algorithm can be utilized for many networking applications. We continue to explore the unique data structure on a different type of processor architecture for comparison.

In this chapter, we focus on the implementation of sketch update since it is regarded as the bottleneck of the sketch algorithm. First, we introduce Intel's Internet eXchange Architecture framework. Then, a brief hardware architecture of the IXP2800 is presented.

The implementation on the IXP2800 and the performance results are presented next. Based on those, we make a brief comparison with the results from the Imagine stream processor. The pros and cons of each approach are discussed at the end of this chapter.

7.1 Intel Internet eXchange Architecture

The Internet eXchange Architecture (IXA) is a popular network processor framework. Designers can build the systems with great programming flexibility using the network processor while still achieving high-performance packet processing at wire speed. Armed with several dedicated hardware assists, the processor architecture exploits the packet-level parallelism with multiple multi-threaded microengines.

As shown in Table 7.1, there are several network processor families designed for different performance requirements. The IXP1200 family is the first generation consists of six parallel mi-

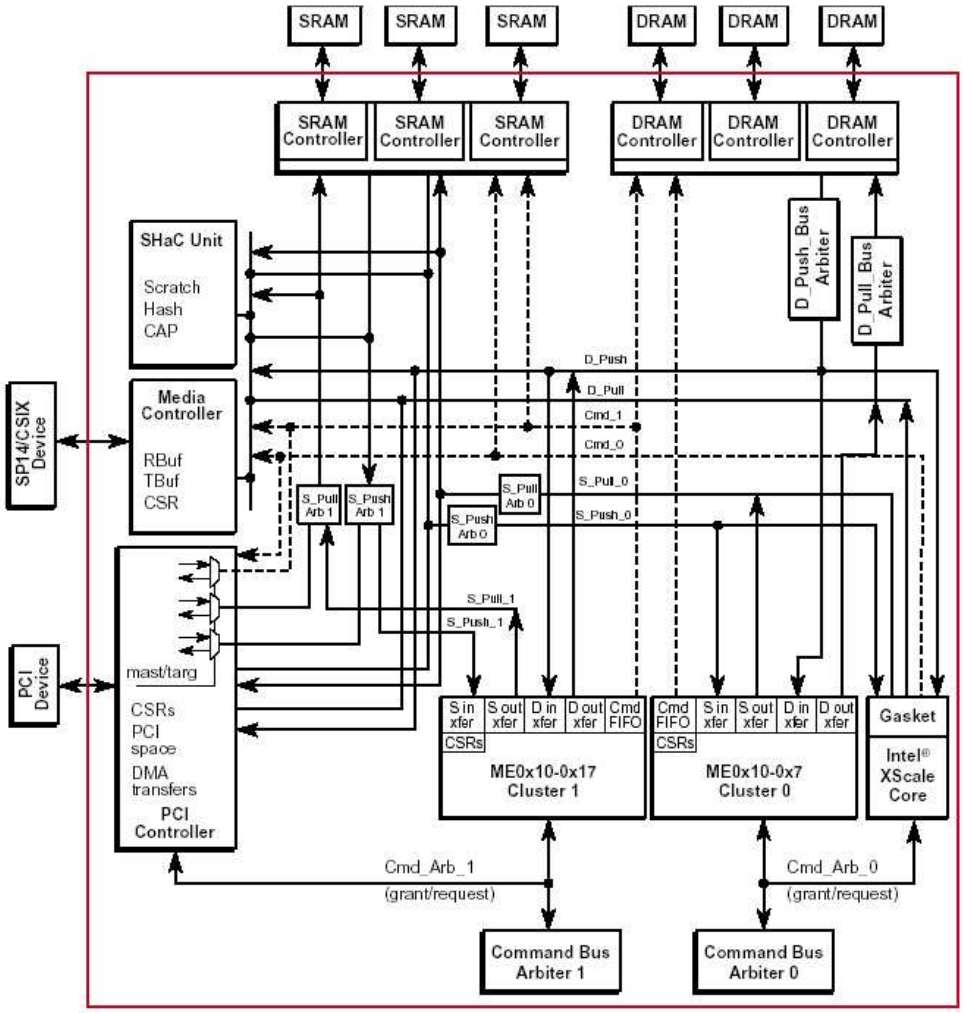

Figure 7.1: The Intel IXP2800 processor architecture diagram [22].

Table 7.1: Brief Comparison for IXP Network Processors.

	Max. Clock	Microengines	SRAM Bandwidth	XScale Core	Power
IXP12x0	232MHz	6	464Mbps	232MHz	5W
IXP2400	600MHz	8	2×16Gbps	700MHz	13W
IXP28x0	1400MHz	16	4×16Gbps	700MHz	21-26W

croengines. The IXP 2400 processor, with eight microengines operated at 600 MHz, is capable of processing tasks up to OC-12 line rate. The IXP2800 family is designed with 16 microengines for handling the traffic throughput up to OC-192.

7.1.1 The IXP2800 Architecture

The IXP2800's hardware architecture diagram is shown in Figure 7.1. The XScale core, running at 700 MHz, is designed to take care of the control plane tasks. Some typical examples are running the signaling stacks, exception packet handling and chip configurations. The receive and transmit buffer, located in the Media Controller, is responsible for holding the packet data in and out to the processing elements of the chip. The Media Controller is connected to the layer 2 devices and switching fabric through the System Packet Interface Level 4 (SPI4) and Common Switch Interface (CSIX) respectively.

As the packet comes in the receive buffer, it is then processed by the microengines arranged in two clusters. There are a total of 16 microengines running at a clock frequency of 1.4 GHz. The second-generation microengine (MEv2) features new instructions and hardware assists for processing packets at high speed. Each microengine has one execution unit, which is capable of supporting up to 8 different contexts (threads) simultaneously with little switching overhead.

As shown in Figure 7.2, each microengine has 8K bytes of instruction store and its own CRC unit to offload the intensive computation from the execution unit. The transfer registers hold the data transferred between the push/pull bus and the execution unit.

7.1.2 Memory Hierarchy

The memory hierarchy, consisting of local register files, local memory, scratchpad, SRAM and SDRAM, is capable of meeting the storage, bandwidth and latency requirements for various applications.

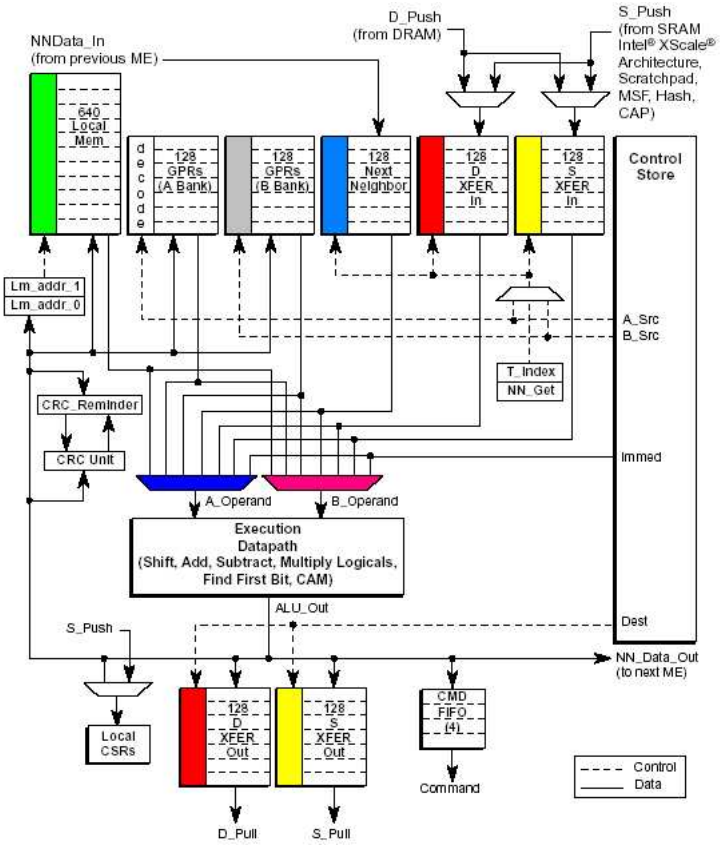

Figure 7.2: The micro architecture of the microengine [22].

There are two banks of general purpose register files and a local memory with the size of 2,560 bytes. The SHaC unit contains a 16K-byte scratchpad memory, which is commonly used for communication and data sharing among microengines. The scratchpad is capable of holding up to 16 ring buffer implementation. It also supports atomic get, put and subtract operations.

The SRAM controller has four channels. Each channel is capable of providing 2 GBps bandwidth running at 250 MHz. The frequently accessed data structure of the processing algorithm such as IPv4 forwarding table can be hold in the external SRAM. The SRAM controller also supports the atomic operations. It features the *queue array* implementation which can be easily configured as a ring or queue data structure.

The IXP2800 processor can support up to 2G bytes of DRAM with the bandwidth of 50Gbps. The packet payload is usually placed in the DRAM due to its large capacity.

7.1.3 The Programming Model

The programming model can be roughly divided into two layers of hierarchy: the *control* and *data plane*, as shown in Figure 7.3. The hardware abstraction library hides the lower level of complexity from the programmer with hardware specific drivers. The microblock library contains the specially optimized function calls written in Microengine C and Assembly languages. Programmers can reuse these functions for packet processing such as IPv4 forwarding, layer 2 bridging and filtering. The library in the control plane is based on the traditional C and C++ languages. The resource manager and core component libraries are designed for the Xscale core processing the management tasks such as call signaling, queue management and chip configurations.

Generally, the software design in the data plane can be organized into three major stages: the ingress, process and egress. In each of these stages, packet processing tasks such as receive, enqueue, dequeue and transmit can be done by using the microblocks provided in the library. A programmer has to decide a way of data communication and transfer methods among different processing entities. The processing tasks can be arranged in a pipelined fashion by using a series of microengines. Or, several microengines can execute the same tasks in parallel. The performance is highly depend on the proper partition and allocation of microengines for different tasks.

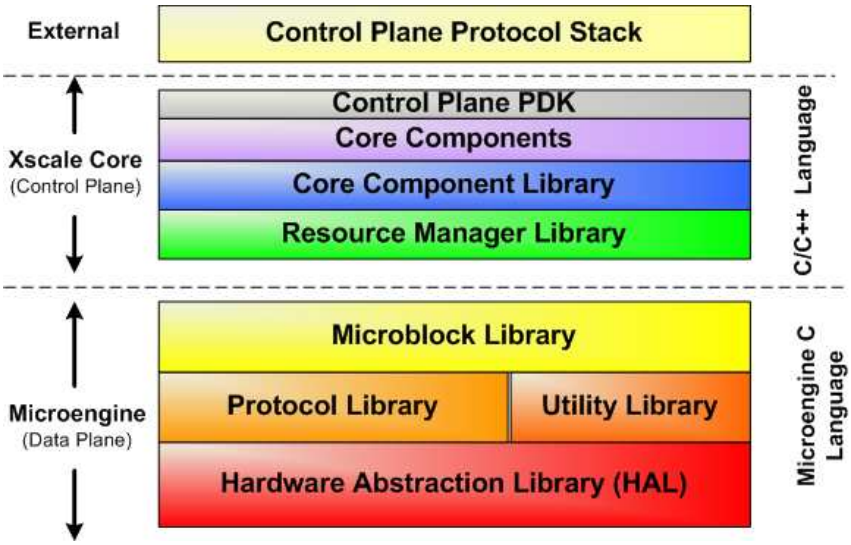

Figure 7.3: The software hierarchies of the IXA framework [13].

7.2 Implementation of Sketch Update

The sketch data structure is essentially a 2-D counter array. There are several possible locations within the processor to hold this counter array: Local Memory within each microengine, Scratchpad, external SRAM and DRAM.

The size of Local Memory within each microengine is 2,560 bytes. Therefore, it can only hold 640 entries of 32-bit counters. The size of the on-chip Scratchpad is 16K bytes and it is shared by all the microengines. For a $1K \times 8$ 32-bit counter array (as illustrated in Figure 6.1), it requires 32K bytes in total memory size. Thus, the choices of using Local Memory or Scratchpad are out of the scope. The only good choice is the SRAM since the access latency for the DRAM (300 clock cycles for IXP2800) is almost twice that of the SRAM (150 clock cycles).

Each thread of the microengine fetches a packet from the receive queue and calculates d different hashes based on the same attributes in the packet header. Thus, the implementation is different from that in the Imagine processor where d different hash calculations are distributed across the clusters. Figure 7.4 shows the differences of eight hash calculation over eight keys in IXP network processor and Imagine stream processor.

Upon the arrival of a packet, a thread will be assigned to pick up the key (say, IP source address for example) and calculate *eight* hash values as indexes to update the counters in SRAM.

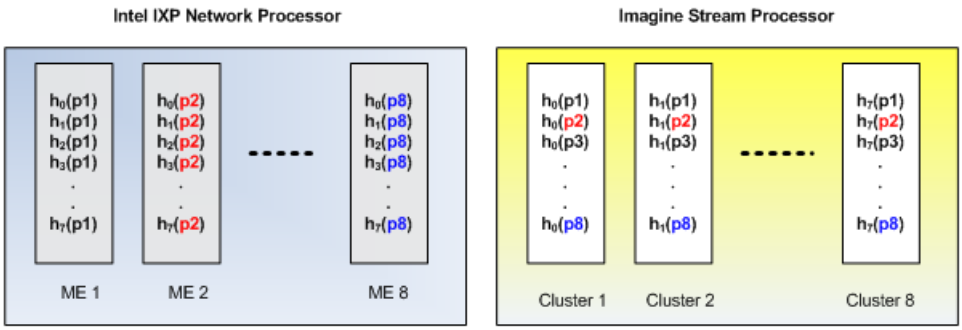

Figure 7.4: Each thread of the Microengine calculates eight different hashes based on the same key in IXP network processor. For Imagine processor, eight different hash calculations are distributed to each of the 8 clusters.

The updated value can be the size of the packet or simply a packet count, depending on the application. The next thread in the same microengine will be assigned to pick the next key from the packet queue as the first thread stalls. The IXP2800 is capable of processing 8 threads in each of its 16 microengines. Therefore, there are, in maximum, 128 in-flight threads updating the counters in the SRAM.

Maintaining the atomicity for each update is critical to the accuracy of the data structure. The processor provides several instructions to support the atomic operation. The atomic arithmetic operation "*test_and_add*" is used for its efficiency. Instead of issuing two commands: the SRAM read and SRAM write for each read-and-modify-write access, the microengine issues only one command that contains the address and increment to the SRAM controller [49].

7.3 Performance

The SRAM access latency is between 70 to 160 clock cycles [13]. The variance is mainly because of the arbitration latency and limited depth of command FIFO. We'll assume 120-cycle latency as typical.

The sketch update based on the *2*-universal hash function on a 32-bit key is a latency-bounded operation, because the hash calculation only takes 70 cycles and the access to the SRAM takes 120 cycles, as shown in Figure 7.5. However, it easily turns out to be computation-bounded by using the *4*-universal hash function. The hash calculation for a 32-bit key takes about 220 cycles as

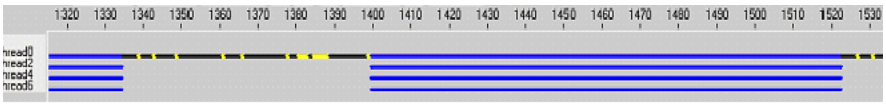

Figure 7.5: It takes about 70 cycles for a single thread to calculate the hashing of a 32-bit key. The thread becomes idle due to the memory access latency of 120 cycles.

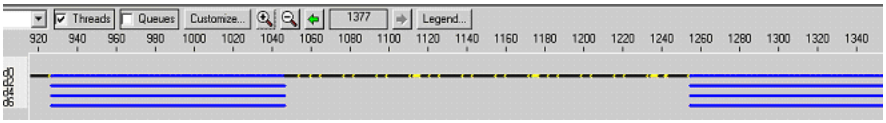

Figure 7.6: It takes about 220 cycles for calculating a 32-bit source address by using *4*-universal hash function.

illustrated in Figure 7.6. As we may have expected, the computation time is definitely longer for a key consisting of more than one tuple of attributes, such as both IP source and destination addresses.

For an average SRAM access latency of 120 cycles, it takes at least 1030 cycles[1] to hash a key and update counters in the SRAM for a single thread in one microengine. That is an average of 129 cycles per one hash and update. For a perfect scenario on eight microengines where each is capable of supporting eight threads, it takes an average of 2.01 cycles per one hash and update.

The simulation shows that, for 8 microengines (8 threads in each microengine), it takes about an average of 59.4 cycles per key. It's approximately 7.43 cycles for each hash and update as each key has to be hashed by eight independent hash functions. The performance can be increased to 33.57 cycles per key (an average of 4.2 cycles per each hash and update) by using 16 microengines. With 16 microengines, the processor is capable of hashing and updating twice the number of keys with the expense of extra cycles due to conflicts of resource sharing. The extra cycle counts are approximately 13% as shown in Figure 7.7, and those are because of increased stall and idle cycles in the microengines. The processing throughput for minimum-sized (40-byte) IP packets is 13.34 Gbps with system clock of 1.4 GHz.

The above simulation is based on hashing and updating 640 and 1280 keys for 8- and 16-microengine configuration respectively. The keys are pseudo-random 32-bit words stored locally and hashed by the 2-universal hash function. The sketch counters are distributed among 4 banks of SRAM. On average, the microengine achieves 72.9% utilization. The processing throughput for

[1]There are eight hash calculations and counter updates for a given key. The cycle count is based on a hash calculation of 70 cycles plus 8×120 cycles of memory latency. The remaining seven hash calculations are overlapped by the memory accesses.

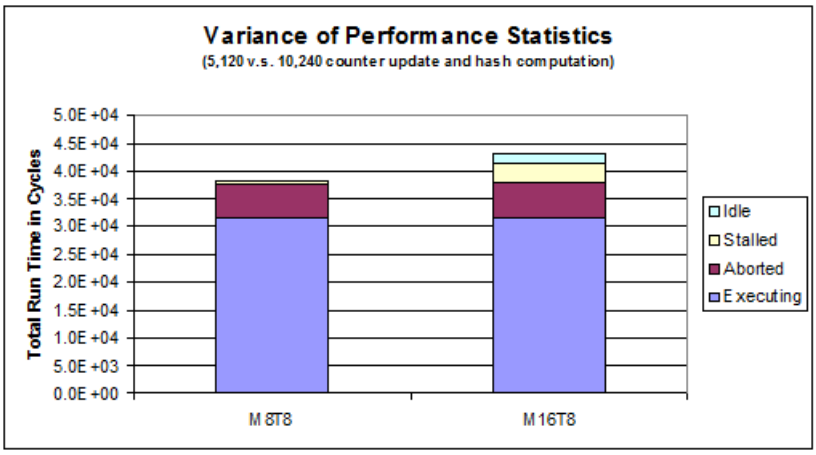

Figure 7.7: The variance of performance for different architecture configurations. The configuration of 16 microengines (M16T8) has more idle and stall cycles comparing to that of 8 (M8T8).

minimum-sized (40-byte) IP packets is 13.34 Gbps with system clock of 1.4 GHz.

7.4 A Brief Comparison with Imagine Stream Processor

The Imagine stream processor is capable of processing 15,360 hash calculations and updating the sketch counters in total of 28,335 cycles. That is approximately 1.84 cycles per hash and update. The simulation is based on the implementation of sketch counters over the Stream Register File (SRF). The kernel performs the *2*-universal hash calculation on a 32-bit key and updates the counters by in-lane indexed stream access in a SIMD fashion for 8 clusters.

Given the clock frequency of 500 MHz and 1400 MHz, the processing throughput for minimum-sized (40 bytes) IP packets is 10.86 and 13.34 Gbps for Imagine and IXP2800 respectively. That is about 3.68ns and 3ns for each hash calculation and counter update. The IXP2800 processor achieves approximately 22% higher throughput.

Another point of interest is the power consumption. Based on the simulation of NePsim [73], the estimated power consumption[2] for each microengine performing sketch update in IXP2800 is 0.95 watt. The power consumption is 15.2 watt for 16 microengines without counting the other function units and peripherals.

[2]Due to the limitation of the simulator, the estimation is based on the SRAM_READ and SRAM WRITE instructions. We are currently modifying the codes of NePsim [73] to support the atomic SRAM access. We put this as the future work for more accurate modeling of the power consumption for IXP2800.

Table 7.2: The power consumption and VLSI characteristics for Imagine [55, 54] and IXP2800 [23].

	Imagine	IXP2800
Clock	500MHz	1400MHz
Hardware Configuration	8 clusters	16 microengines
Die Size	16 mm $\times$ 16 mm	14.33 mm $\times$ 19.09 mm
Number of Transistors	21 million	82 million
Memory	SDRAM 167MHz	QDR SRAM 200MHz
Typical Power	2.89 W	21~26 W
Maximum Power	4 W	30 W
Technology	TI 0.15um CMOS	Intel 0.13um CMOS

The maximum power rating is provided based on a publicly available benchmark result [76] for reference purposes. By running the OC192 POS forwarding application with 100% throughput, the IXP2800 consumes about 30 Watts in the worst case without counting the power in the I/O interfaces [76]. The typical power consumption in the range between 21 and 26 watts is reported in the literatures [13, 76].

The typical power for Imagine running the sketch update simulation is 2.89 watts. This is based on the estimation by using the formulas [53] derived by Khailany et al. Compared to the sequential stream access, the use of indexed SRF access has approximately 4x increase of the energy consumption [48].

The maximum power of 4 Watts is based on the worst-case estimated power consumption over a range of applications [55]. The IXP2800 processor consumes almost 7 times more power than that of the Imagine in the maximum power rating as shown in Table 7.2.

7.5 Discussion

The two major steps for the sketch update operation are hash calculation and counter increment. That is, for a given key, there are series of arithmetic operations followed by memory accesses. For a relatively small-sized sketch implementation, the local memory (e.g., the L1 or L2 caches) is the best place to hold the data structure due to its short access latency. Therefore, for a scalar processor, better performance can be achieved by simply minimizing the processing cycle of the hash calculation.

There are several ways to shorten the hash calculation without compromising its quality.

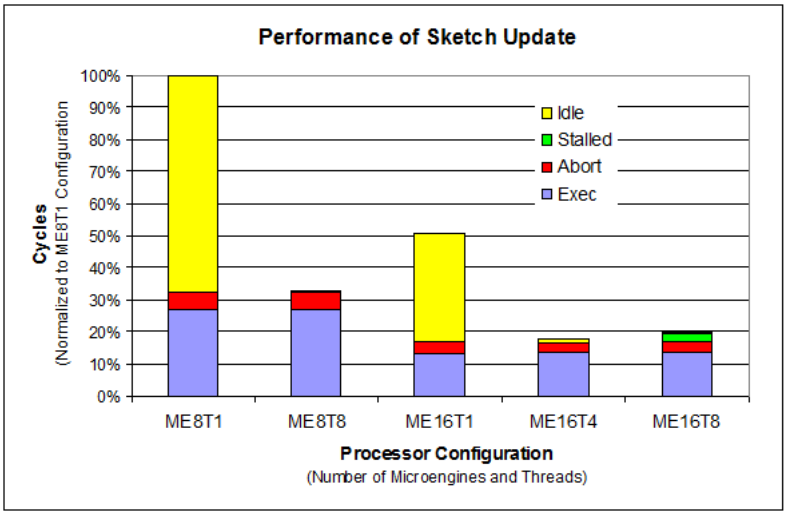

Figure 7.8: The performance of sketch update with the same number of keys on different processor configurations. The sketch update process consists of hash calculation (*2*-universal) and read-modify-write counter access.

The use of dedicated hardware assists and the algorithmic optimizations are commonly seen in the literature [116]. Among them, the tabulation method is a popular choice. The hash values are pre-computed for all possible input keys and stored in a table. Then, the lengthy calculation cycles can be replaced by a single table lookup. The tabulation based *4*-universal hashing proposed by Thorup and Zhang [117] is a good example of this time-and-space tradeoff.

However, the tabulation scheme may not be effective because of the limited space of the local memory. Different applications may require higher level of query accuracy where larger size of sketch is required. Thus, the data structure becomes too large to fit in the local memory along with other data structures. Once we put the sketch in the external memory, the cache can not effectively bridge the latency gap due to the randomized nature of the sketch access pattern.

The multi-threaded processor provides a way to hide the long memory access latency. The performance of sketch update with the same number of keys on different processor configurations is shown in Figure 7.8. For processor configured with eight microengines, the microengine idle cycles can be eliminated with eight threads. In this regard, minimizing the hash calculation cycle for higher throughput becomes less effective since the calculation can be overlapped by the memory access latency. The performance by using tabulation method may be even worse if the table can not fit in the local memory. As a result, the table lookups introduce more memory latencies which may

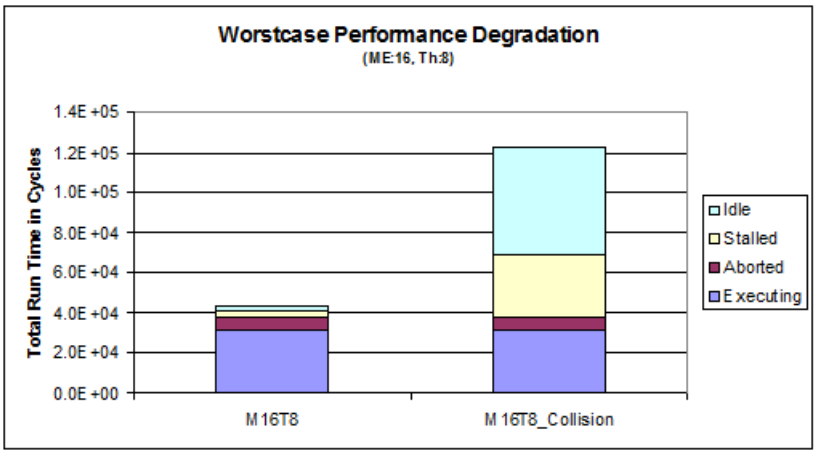

Figure 7.9: The worstcase performance degradation by hashing 1,280 keys of the same source IP address (M16T8_collision). The performance is based on the processor configuration of 8 threads in each of the 16 microengines. The performance with keys of pseudo-random source IP address is shown on the left (M16T8).

not be effectively overlapped.

The higher the thread count, the more effective latency overlapping can be achieved. However, more threads may also increase the possibility of contention. Since the counters are located in the shared memory space, the collision due to the atomic update on the same address may affect the performance. The simulation results shown in Figure 7.8 reveal the performance degradation as the thread count increases from four to eight in a 16-microengine configuration. The 11.4% processing cycle increase is mostly due to microengine stalls.

There are two possibilities where the collision might occur. First, two or more different keys are hashed to the same SRAM location. Second, it is due to the "packet train" effect where a burst of packets originated from the same traffic flow (same source and destination addresses). Figure 7.9 shows the performance degradation of a worst case scenario: processing the the sketch update with 1,280 keys of the same source IP address. The total runtime increases almost three-fold mainly due to the increase of idle and stalled cycles in the microengine.

The hot spot problem [43] can be solved by introducing buffer queues to aggregate the requests destined to the same memory address. However, this may be costly since the queue size has to be large to capture the traffic locality.

The implementation of sketch update in Imagine processor does not have such issues since each cluster in the processor updates its corresponding "lane" within the SRF. Furthermore, the in-

lane indexed SRF latency is only 4 cycles [48]. As long as the cycle for hash calculation is larger than the arbitration and round-trip access cycle, the atomicity and sequential access order can be maintained for each cluster.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis, we implement several security-centric networking applications based on the stream programming model. In the stream programming model, a stream is composed of a sequence of data elements or records. These streams are then applied for a set of complex operations organized as kernels at the clusters. The computation results are stored locally in LRF or SRF and consumed by another set of operations iteratively.

The applications presented include the AES encryption in parallel operation modes (ECB and OCB) with key agility, the MMH message authentication code (MACs), the Bloom filter based content inspection engine for intrusion detection and sketch-based algorithms for traffic analysis. The thesis explores the tradeoffs on different configurations of stream architecture and characterizes the processing throughput of these applications. Moreover, we explore the difference between Imagine and a MIMD architecture by implementing the sketch update application on the Intel IXP network processor.

In general, the stream processor provides a flexible and powerful computing infrastructure for these security-centric applications. The simulation results of these applications demonstrate up to multi-Gigabit-per-second throughput for system clock of 500 MHz. We would like to highlight some of the key architecture aspects which benefit the processing of packets with excellent performance.

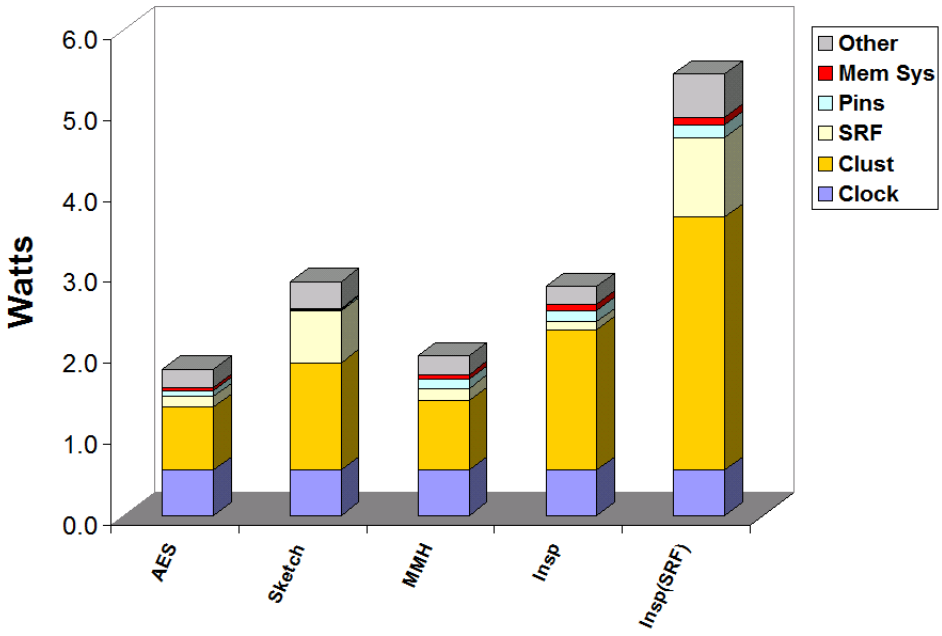

Figure 8.1: The spectrum of power consumption for applications discussed in the thesis. The cluster consumes most of the power because of the major computation performed. There are two types of implementation for the Bloom filter based packet inspection application. The content hashing in *Insp(SRF)* implementation is based on the tabulation method over the Stream Register File (SRF). The *indexed SRF* accesses cause the extra increase of the power consumption. The cost of chip power consumption is based on the VLSI model proposed by Khailany et al [53]. The power consumption for the AES does not include the extra dissipation due to the two-port scratchpad.

1. Hiding the memory access latency with computation:

   The DRAM access can be effectively overlapped by the computation. More than 95% of the AES encryption run time is taken by the kernel computation for a packet length larger than 16 blocks. Different from the way of measurement setup [3] where most of the data is in the level one cache, the simulation do include the data movement from the main memory to the processor itself. The performance can reach up to 32 cycles per block in a stream size of 96 blocks. One of the best performances published [72, 3] in non-feedback mode for a 32-bit architecture is 232 cycles per block (16-byte block).

2. Exploiting the stream locality computations in SIMD style:

   Explicit support for this computational model results in efficient, coordinated access to memory and effective exploitation of on-chip internal data movement. For many statistical and hash based algorithms in the networking applications, the computation is performed in a recursive and incremental fashion. Those operations do benefit from the architecture support where the producer-and-consumer locality can be captured efficiently without remote memory references. The operations of MMH authentication code and Bloom filter based content inspection engine are good examples. The calculation of hash values is transformed into stream processing, expressing producer and consumer locality and achieving efficient utilization of the unique memory hierarchy.

3. Exploiting the abundant parallelism:

   The applications presented in the thesis have abundant data and instruction level parallelism. They do benefit from the SIMD stream architecture and achieve substantial speedup. Moreover, without incurring extra computation cycles, the explicit SIMD architecture enhance the probability of accuracy: a unique performance aspect of the randomized algorithms other than the *time* and *space*.

4. Efficient vector style processing:

   Not only can the packet be modeled as a stream, a set of shared data structures can also be represented as a stream. Acting as a stream, the shared data structure can be processed in an efficient *vector* style operation. For example, the point query and linear combination of sketches are simply *vector gather* and *vector add* operations respectively.

5. Simplifying the memory access to shared data structure:

As shown in the sketch-based algorithm, the read-modify-write sketch update process is conducted by clusters based on the operation of in-lane *indexed SRF* access [48]. Different from the MIMD style implementation on Intel IXP processor, the SIMD operation simplifies the access to shared data structure without explicit synchronization and arbitration overhead. As a result, the system achieves high throughput and efficient utilization of maximum memory bandwidth.

6. High performance and power efficiency:

   The stream processor is capable of supporting the computation-intensive tasks with outstanding power efficiency. The estimated power consumption for these applications are shown in Figure 8.1.

7. Flexibility:

   The Imagine's two-level programming model (stream and kernel) provide the full flexibility for application implementation. As new algorithms and methodologies being proposed frequently, the benefit of being programmable with high computational capability is apparent: we can always utilize the latest and newest approaches to tackle the complex tasks while meeting the ever growing demand of throughput requirement with little or no cost at all.

8.1.1 The Tradeoffs

The applications can be categorized into two major groups: payload and header processing. For payload processing tasks, each packet is arranged as a stream of records. Each record of the stream can be a collection of bytes, words[1], or different data types. These records are arranged in the SRF sequentially as shown in Figure 8.3, and consumed by the clusters in a SIMD fashion.

Generally, the performance can be improved by exploring the data parallelism over the following two axes: the number of clusters and the number of ALUs within each cluster. As the record size can be increased for more data parallelism in the cluster, scaling up the number of ALUs is a straightforward way of exploiting the ILP and DLP within each cluster. However, scaling up the number of ALUs alone may not be effective as the operation may depend on the other function units.

[1] In the AES application, for example, each record is composed of four 32-bit words. We also denote the record as a *block* according to the AES specification.

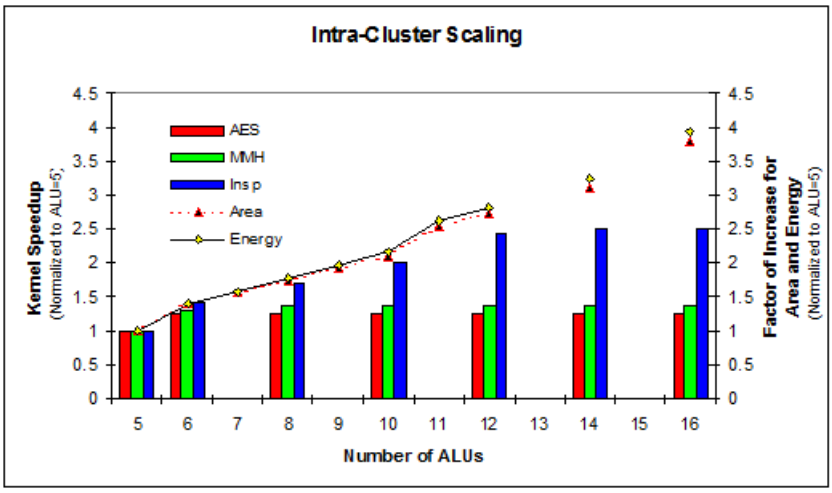

Figure 8.2: The Intra-cluster scaling. The cost of chip area and energy consumption is based on the VLSI model proposed by Khailany et al [53].

The lookup of the T-table over the scratchpad memory in the AES encryption application is an example of this kind. The limited bandwidth of single-port memory quickly becomes the bottleneck due to multiple read accesses. Moreover, the VLSI cost model [53] reveals that the average area and energy dissipation per ALU is not optimal as the number of ALUs scales beyond five.

The inter-cluster scaling provides another way of exploiting the DLP of these networking applications. As costs can be amortized among more clusters, the average area and energy dissipation per ALU remain approximately constant (3% increase) as the number of clusters increases from 8 to 32 [53]. Therefore, it makes the inter-cluster scaling more preferable than its counterpart.

Different from the traditional SIMD applications, the stream size of these network applications is not fixed due to the non-uniform distribution of variable packet sizes. Therefore, the SIMD processing efficiency is affected as the size of the stream varies. The efficiency of cluster is defined as the average ratio of the number of clusters processing valid data over the total number of clusters involved (the clusters working on valid data + the clusters working on null data). For a fixed number of clusters, the average efficiency goes down as the stream size decreases.

In Figure 8.3, as the size of the stream (in terms of records) has to be a multiple of the total number of clusters, only half of the clusters are doing the real work (processing records of R33, R34, R35, and R36) at the fifth iteration.

Figure 8.4 illustrates the AES kernel speedup, estimated efficiency and costs based on a

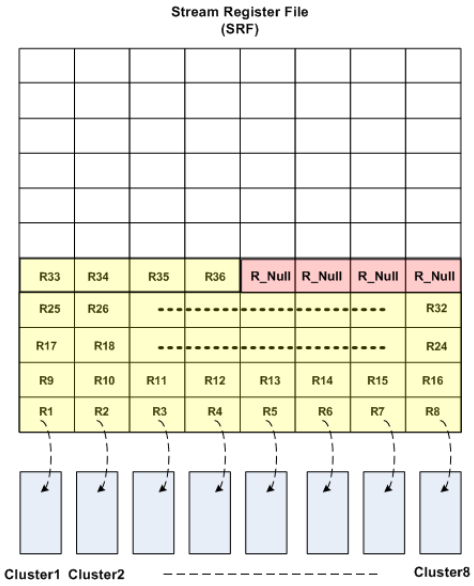

Figure 8.3: A stream of packet payload consists of 36 records and 4 extra null records are laid out in the SRF. The clusters are fetching 8 records at a time in a SIMD fashion.

real-world trace[2]. As the number of clusters increases from 8 to 32, the average cluster efficiency decreases from 79% to 50%. The energy dissipation and chip area increase linearly with the number of clusters, however, the speedup is limited due to a large portion of smaller-sized packets presented in the trace.

The processing of the small-sized packet (stream) in this architecture incurs another inefficiency denoted as the "*short-stream effect*" [102, 92]. The "*short-stream effect*" is mainly due to the fixed hardware cost and those associated with the setup and teardown of the data structure supporting the stream processing in the kernel. As the number of clusters is comparable to the stream size, the processing throughput is critically affected by this effect. These results suggest a limit of inter-cluster scaling for the payload processing applications.

The header processing tasks, to the contrary, do not incur such inefficiency since the size of packet header is fixed. Most of the networking systems organize these headers as a set of data structure stored in separate memory location. Thus, a certain amount of data (a larger stream size) consisting of several headers can be easily fetched and processed in a SIMD fashion independent of the packet length.

[2]AIX-1054837521-1.tsh, where 44.6% of the packets are less then 64 byte, and 31% of the packets are larger than 1000 bytes; total number of packets=13,812.

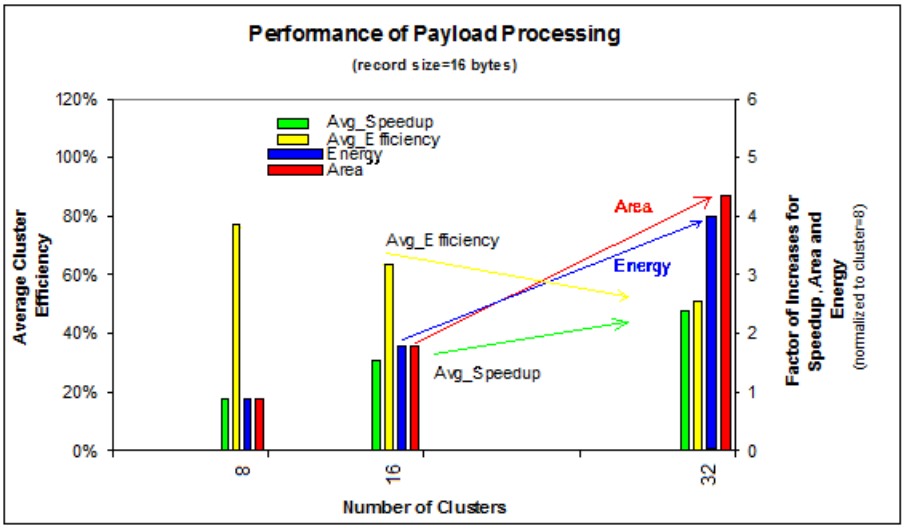

Figure 8.4: The estimated payload processing performance and cost with the number of clusters. The speedup is based on the kernel performance of AES encryption. The cost of chip area and energy consumption is based on the VLSI model proposed by Khailany et al [53].

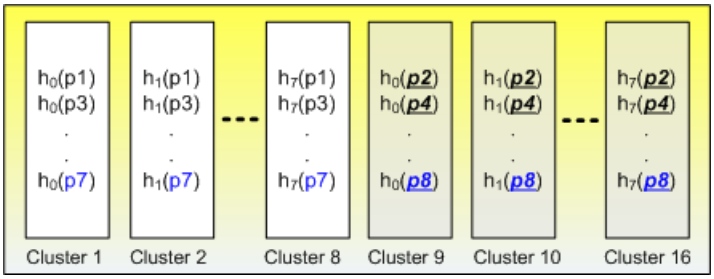

Figure 8.5: The sketch update process with 8 universal hash functions on 16 clusters.

The sketch update process shown in Figure 7.4 can be easily extended from 8 to 16 clusters. As illustrated in Figure 8.5, two headers (p1 and p2) are hashed by the first and second set of eight clusters simultaneously. The throughput can be scaled up linearly with the number of clusters.

The probability of accuracy is another important performance metric for the randomized algorithms such as sketch and Bloom filter. As part of the error probability depends on the number of hash functions used, the system may distribute these hash calculations in parallel to more clusters achieving lower error probability without extra processing cycles.

8.2 Future Work

We summarize the future work in three major parts: the simulation framework, software and hardware architecture.

The simulation of these applications are based on the assumption that the packet header and payload are located at the memory. It will be of great beneficial to construct the network interfaces at Media Access layer (MAC) and integrate with the Imagine processor simulation framework. The integrated simulation framework can facilitate the exploration of many network system designs. Example such as the high-speed, real-time intrusion detection system can be constructed based on a hybrid of content-based and statistical-based approaches on multiple stream processors. A full system simulation based on the real-world attacking traces can be performed and beneficial to the design of network processor architecture.

The handling of packet and data structure as a representation of stream provide a new paradigm of packet processing. Therefore, in the software level, we would like to explore new stream instructions to facilitate more efficient stream operations. The array of streams is one of the example which is not currently supported. Moreover, the implementation of additional kernel instructions and hardware assists for bit-level manipulation is crucial for the system performance.

Currently, the number of cluster is explicitly exposed to the programming model. Hence, the stream size has to be in multiple of cluster size. As the number of cluster changes, the source code has to be changed accordingly as well. As some of the applications may not fully utilize the clusters provided, the dynamic reconfiguration of number of clusters may help improving the efficiency and lowering power consumption. Adding another abstraction to hide the number of cluster from the programming model is of great beneficial.

Different from the traditional SIMD applications, however, packet processing over the

SIMD stream architecture exhibit issues such as control flow variation and load balancing due to the non-uniform distribution of variable packet sizes. The Multi-SIMD hybrid architecture: a group of SIMD clusters share the same microcontroller issuing the instructions while different group of these entities behave in MIMD mode, can be one of the solutions not only for the former issue but also for better performance and power efficiency. As a result, more applications are needed to fully characterize the new architecture for network processing.

Bibliography

[1] Noga Alon et al. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[2] S. Antonatos, K.G. Anagnostakis, et al. Performance analysis of content matching intrusion detection systems. In *Proc. of the International Symposium on Applications and the Internet*, 2004.

[3] Kazumaro Aoki and Helger Lipmaa. Fast implementations of AES candidates. In NIST, editor, *The Third Advanced Encryption Standard Candidate Conference, April 13–14, 2000, New York, NY, USA*, pages 106–122, Gaithersburg, MD, USA, 2000. National Institute for Standards and Technology.

[4] Brian Babcock et al. Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, New York, NY, USA, 2002. ACM Press.

[5] C. Barakat et al. Modeling Internet backbone traffic at the flow level. *IEEE Trans. on Signal Processing*, 51(8), August 2003.

[6] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Advances in Cryptology - Crypto '96*, pages 1–15, Berlin, 1996. Springer-Verlag. Lecture Notes in Computer Science Volume 1109.

[7] J. Black, S. Haler, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In *Proc. 19th International Advances in Cryptology Conference – CRYPTO '99*, pages 216–233, 1999.

[8] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[9] A. Bosselaers, R. Govaerts, and J. Vandewalle. Fast hashing on the Pentium. In Neal Koblitz, editor, *Advances in Cryptology - Crypto '96*, pages 298–312, Berlin, 1996. Springer-Verlag. Lecture Notes in Computer Science Volume 1109.

[10] G. Brassard. On computationally secure authentication tags requiring short secret shared keys. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology: Proceedings of Crypto '82*, pages 79–88, New York, USA, 1982. Plenum Publishing.

[11] Andrei Broder and Michael Mitzenmacher. Network applications of Bloom Filters: A survey. *Internet Math.*, 1(4):485–509, 2003.

[12] W. Bux et al. Technologies and building blocks for fast packet forwarding. In *IEEE Comm. Magazine*, volume 39, pages 70–77, Jan. 2001.

[13] Bill Carlson. *Intel Internet Exchange Architecture and Applications*. Intel Press, May 2003.

[14] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, April 1979.

[15] Charikar et al. Finding frequent items in data streams. *TCS: Theoretical Computer Science*, Vol.312:3–15, 2004.

[16] Jaewook Chung and Anwar Hasan. More generalized Mersenne numbers: (extended abstract). In *Selected Areas in Cryptography, 10th Annual International Workshop*, volume 3006 of *Lecture Notes in Computer Science*, pages 335–347. Springer, 2003.

[17] Craig Clapp. Instruction-level parallelism in AES candidates. In National Institute of Standards and Technology, editor, *Second AES Candidate Conference Proceedings, March 22–23, 1999, Rome, Italy*, page 16, Gaithersburg, MD, USA, March 1999. National Institute for Standards and Technology.

[18] Chris Clark, Wenke Lee, et al. A hardware platform for network intrusion detection and prevention. In *Workshop on Network Processors and Applications (NP3)*, pages 136–145, Madrid, Spain, Feb. 2004.

[19] G. Cormode and S. Muthukrishnan. Improved data stream summaries: The count-min sketch and its applications. Technical Report 2003-20, DIMACS, June 2003.

[20] G. Cormode and S. Muthukrishnan. What's new: Finding significant differences in network data streams. In *IEEE INFOCOM*, 2004.

[21] G. Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In *SIAM Inter. Conf. on Data Mining (SDM)*, 2005.

[22] Intel Corp. *Intel IXP2800 Network Processor Hardware Reference Manual*. November 2002.

[23] Intel Corp. Intel IXP2800 and IXP2850 Network Processors B1 Stepping Qualification Report, Sept. 2004.

[24] R. Crandall. Method and apparatus for public key exchange in a cryptographic system. U.S. Patent number 5159632, 1992.

[25] J. Daemen and V. Rijmen. AES proposal: Rijndael. NIST AES Proposal, June 1998.

[26] W. J. Dally, P. Hanrahan, et al. Merrimac: Supercomputing with streams. In *SC'03*, Phoenix, Arizona, November 2003.

[27] William Dally et al. The Imagine instruction set architecture. Technical Report Massachusetts Institute of Technology Stanford University, August 1997.

[28] William J. Dally. Packet processing with streams, Keynote Address. In *HPCA Workshop on Network Processors*, page 1, 2002.

[29] DaMoN'05. First international workshop on data management on new hardware, 2005 June.

[30] Abhishek Das, Peter Mattson, et al. Imagine programming system user's guide, November 2003.

[31] DEFCON9. The largest underground hacking event in the world, http://www.defcon.org/, 2001.

[32] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. Technical Report TR 1098, Computer Sciences Department, University of Wisconsin–Madison, July 1992.

[33] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel Bloom filters. In *Hot Interconnects*, Aug. 2003.

[34] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A strengthened version of RIPEMD. In Dieter Grollman, editor, *Fast Software Encryption: Third International Workshop*, volume 1039 of *Lecture Notes in Computer Science*, pages 71–82, Cambridge, UK, 21–23 February 1996. Springer-Verlag.

[35] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.*, 21(3):270–313, 2003.

[36] M. Etzel, S. Patel, and Z. Ramzan. Square hash: Fast message authentication via optimized unviersal hash functions. In *Proc. 19th International Advances in Cryptology Conference – CRYPTO '99*, pages 234–251, 1999.

[37] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[38] Anja Feldmann et al. Deriving traffic demands for operational IP networks: methodology and experience. *IEEE/ACM Trans. Netw.*, 9(3):265–280, 2001.

[39] Robert Friend. Making the Gigabit IPsec VPN Architecture Secure. *IEEE Computer Magazine*, June 2004.

[40] Minos Garofalakis. Analyzing massive data streams: Past, present, and future. 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, Invited Talk, 2003 June.

[41] Minos Garofalakis et al. Querying and mining data streams: you only get one look, A Tutorial. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 635–635, New York, NY, USA, 2002. ACM Press.

[42] H.R. Ghasemi, H. Mohammadi, et al. Augmenting general purpose processors for network processing. In *IEEE International Conference on Field-Programmable Technology (FPT), 2003*, pages 416–419, Dec. 2003.

[43] Allan Gottlieb et al. The NYU Ultracomputer: designing a MIMD, shared-memory parallel machine. In *ISCA '98: 25 years of the International Symposia on Computer Aarchitecture (selected papers)*, pages 239–254, New York, NY, USA, 1998. ACM Press.

[44] Shai Halevi and Hugo Krawczyk. MMH: Software message authentication in the Gbit/second rates. In Eli Biham, editor, *Fast Software Encryption: 4th International Workshop*, volume 1267 of *Lecture Notes in Computer Science*, pages 172–189, Haifa, Israel, 20–22 January 1997. Springer-Verlag.

[45] Hifn. Multi-Gbps multi-function security gateway system design. White Paper, http://www.hifn.com, March 2003.

[46] Hongsong Li; Houkuan Huang. New estimation methods of count-min sketch. RIDE Workshop of Stream Data Mining and Applications RIDE-SDMA'05, 2005 April.

[47] Yannis E. Ioannidis and Viswanath Poosala. Balancing histogram optimality and practicality for query result size estimation. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 233–244, New York, NY, USA, 1995. ACM Press.

[48] Nuwan Jayasena, Mattan Erez, et al. Stream register files with indexed access. In *10th International Symposium on High Performance Computer Architecture*, Madrid, Spain, February 2004.

[49] Erik J. Johnson and Aaron R. Kunze. *IXP2400/2800 Programming The Complete Microengine Coding Guide*. Intel Press, April 2003.

[50] Ujval Kapasi et al. The Imagine stream processor. In *IEEE International Conference on Computer Design*, pages 282–288, Sep. 2002.

[51] S. Kent and R. Atkinson. IP Authentication Header. RFC 2402 (Proposed Standard), November 1998.

[52] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401 (Proposed Standard), November 1998. Updated by RFC 3168.

[53] B. Khailany, W. Dally, et al. Exploring the VLSI scalability of stream processors. In *International Conference on High Performance Computer Architecture,HPCA*, 2003.

[54] B. Khailany et al. VLSI design and verification of the Imagine processor. In *IEEE International Conference on Computer Design:VLSI in Computers & Processors (ICCD02)*, pages 289–294, Washington - Brussels - Tokyo, September 2002. IEEE.

[55] Brucek Khailany et al. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, 2001.

[56] T. Kohno et al. The CWC-AES dual-use mode. Internet Draft, 2003. Crypto Forum Research Group.

[57] H. Krawczyk. LFSR-based hashing and authentication. In Yvo Desmedt, editor, *Advances in Cryptology - Crypto '94*, pages 129–139, Berlin, 1994. Springer-Verlag. Lecture Notes in Computer Science Volume 839.

[58] H. Krawczyk. LFSR-based hashing and authentication. *Lecture Notes in Computer Science*, 839:129–139, 1994.

[59] H. Krawczyk. New hash functions for message authentication. In Louis C. Guillou and Jean-Jacques Quisquater, editors, *Advances in Cryptology - EuroCrypt '95*, pages 301–310, Berlin, 1995. Springer-Verlag. Lecture Notes in Computer Science Volume 921.

[60] Balachander Krishnamurthy et al. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 2003 ACM SIGCOMM conference on Internet measurement (IMC-03)*, pages 234–247, New York, October 27–29 2003. ACM Press.

[61] T. Krovetz. UMAC: Message Authentication Code using Universal Hashing. CFRG Working Group Internet Draft, October 2004.

[62] Ted Krovetz and Phillip Rogaway. Fast universal hashing with small keys and no preprocessing: The PolyR construction. *Lecture Notes in Computer Science*, 2015:73–87, 2001.

[63] Theodore D. Krovetz. *Software-Optimized Universal Hashing and Message Authentication*. PhD thesis, Dept. of Computer Science, University of California Davis, Sept. 2000.

[64] M. Krueger and R. Haagens. Small Computer Systems Interface protocol over the Internet (iSCSI) Requirements and Design Considerations. RFC 3347 (Proposed Standard), July 2002.

[65] Abhishek Kumar, Minho Sung, Jun (Jim) Xu, and Jia Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 177–188, New York, NY, USA, 2004. ACM Press.

[66] Yu-Kuen Lai and Gregory T. Byrd. AES packet encryption on a SIMD stream processor. In *Embedded Cryptographic Hardware: Methodologies & Architectures*, pages 615–624. Nova Science Publishers, 2004.

[67] Yu-Kuen Lai and Gregory T. Byrd. Stream architecture for high speed packet inspection. In *Proc. of Advanced Networking and Communications Hardware Workshop (ANCHOR 2005) in conjunction with The 32nd Annual International Symposium on Computer Architecture (ISCA 2005)*, Madison, Wisconsin, June 2005.

[68] Yu-Kuen Lai and Gregory T. Byrd. Stream-based implementation of hash functions for multi-gigabit message authentication code, submitted for review. May 2006.

[69] Gene Moo Lee et al. Improving sketch reconstruction accuracy using linear least squares method. In *2005 Internet Measurement Conference (IMC 2005)*, Oct. 2005.

[70] H. Lipmaa. IDEA: A cipher for multimedia architectures? *Lecture Notes in Computer Science*, 1556:248–261, 1999.

[71] H. Lipmaa. Fast software implementations of SC2000. In *ISW: International Workshop on Information Security, LNCS*, 2002.

[72] Helger Lipmaa. Survey of Rijndael Implementations. http://www.tcs.hut.fi/~helger/aes/nfb.html.

[73] Yan Luo et al. NePSim: A network processor simulator with a power evaluation framework. *IEEE Micro*, 24(5):34–44, 2004.

[74] George Makowsky, J. Lawrence Carter, and Mark N. Wegman. Analysis of a universal class of hash functions. In *Mathematical Foundations of Computer Science 1978*, volume 64 of *LNCS*, pages 345–354, Zakopane, Poland, 4–8 September 1978. Springer-Verlag.

[75] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, editors. *Handbook of Applied Cryptography*. CRC Press, Sept. 1996.

[76] David Meng et al. IXP2800 Intel Network Processor IP Forwarding Benchmark full disclosure report for OC192-pos, Oct. 2003.

[77] MPDS'03. Workshop on management and processing of data streams in conjunction with ACM SIGMOD/PODS, 2003 June.

[78] James K. Mullin. A second look at Bloom filters. *Commun. ACM*, 26(8):570–571, 1983.

[79] James K. Mullin. A caution on universal classes of hash functions. *Information Processing Letters*, 37(5):247–256, March 1991.

[80] S. Muthukrishnan. Data streams: algorithms and applications. In *SODA '03: Proceedings of the 14th annual ACM-SIAM symposium on Discrete algorithms*, pages 413–413, 2003.

[81] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, and Edward Roback. Report on the development of the Advanced Encryption Standard (AES). Technical report, Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology (NIST), Technology Administration, U.S. Department of Commerce, October 2000.

[82] W. Nevelsteen and B. Preneel. Software performance of universal hash functions. *Lecture Notes in CS*, 1592:24–41, 1999.

[83] Peng Ning and Sushil Jajodia. Intrusion detection techniques. In Hossein Bidgoli, editor, *The Internet Encyclopedia*. John Wiley & Sons, Dec. 2003.

[84] NIST. Proposed modes of operation. http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/.

[85] NIST. DRAFT recommendation for block cipher modes of operation - methods and techniques. NIST SP800-38A, National Institute for Standards and Technology, December 2001.

[86] NIST. NIST brief comments on recent cryptanalytic attacks on SHA-1. http://csrc.nist.gov/news-highlights/NIST-Brief-Comments-on-SHA1-attack.pdf, Feb 2005.

[87] NLANR. http://pma.nlanr.net/index.html.

[88] NLANR. http://pma.nlanr.net/pma/sites/aix.html.

[89] NLANR. http://pma.nlanr.net/traces/traces/mdata/aix/plen/20030605.

[90] NLANR. http://pma.nlanr.net/traces/traces/mdata/aix/plen/20030605/1054837521-1.plen.

[91] M Norton and D Roelker. Snort 2.0: Detection revised, http://www.sourcefire.com/technology/whitepapers, 2002.

[92] John Douglas Owens. *Computer Graphics on a Stream Architecture*. PhD thesis, Stanford University, November 2002.

[93] P. Paulin, F. Karim, and P. Bromley. Network processors: a perspective on market requirements, processor architectures and embedded s/w tools. In *DATE '01:Conference on Design, Automation and Test in Europe*, pages 420–429. IEEE Press, 2001.

[94] J. B. Plumstead. Inferring a sequence produced by a linear congruence (abstract). In David Chaum, Ronald L. Rivest, , and Alan T. Sherman, editors, *Advances in Cryptology: Proceedings of Crypto '82*, pages 317–320, New York, USA, 1982. Plenum Publishing.

[95] Jathin S. Rai, Yu-Kuen Lai, and Gregory T. Byrd. Packet processing on a SIMD stream processor. In *Workshop on Network Processors and Applications (NP3) in conjunction with the 10th International Symposium on High-Performance Computer Architecture (HPCA10)*, pages 146–157, Madrid, Spain, Feb. 2004.

[96] Sriram Ramabhadran and George Varghese. Efficient implementation of a statistics counter architecture. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS-03)*, volume 31, 1 of *Performance Evaluation Review*, pages 261–271, New York, June 11–14 2003. ACM Press.

[97] M. V. Ramakrishna. Hashing in practice: analysis of hashing and universal hashing. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 191–199. ACM Press, 1988.

[98] M. V. Ramakrishna. Practical performance of Bloom filters and parallel free-text searching. *Commun. ACM*, 32(10):1237–1239, 1989.

[99] M. V. Ramakrishna and Justin Zobel. Performance in practice of string hashing functions. In *Intl. Conf. on Database Systems for Advanced Applications*, pages 215–223, Melbourne, Australia, April 1997.

[100] RIDE'05. RIDE workshop of stream data mining and applications RIDE-SDMA'05, 2005 April.

[101] S. Rixner, W. J. Dally, et al. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 3–13, 1998.

[102] Scott Rixner. *Stream Processor Architecture*. Springer, 2001.

[103] Michael Roeder and Bill Lin. Maintaining exact statistics counters with a multi-level counter memory. In *IEEE Global Communications Conference (GLOBECOM'04)*, Nov. 2004.

[104] P. Rogaway. Bucket hashing and its application to fast message authentication. In Don Coppersmith, editor, *Advances in Cryptology - Crypto '95*, pages 29–42, Berlin, 1995. Springer-Verlag. Lecture Notes in Computer Science Volume 963.

[105] Phillip Rogaway. E-mail conversation, Aug. 2003.

[106] Phillip Rogaway, Mihir Bellare, and John Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Trans. Inf. Syst. Secur.*, 6(3):365–403, 2003.

[107] Lambert Schaelicke, Thomas Slabach, Branden Moore, and Curt Freeland. Characterizing the performance of network intrusion detection sensors. In *6th International Symposium on Recent Advances in Intrusion Detection (RAID 2003)*, Lecture Notes in Computer Science, Berlin–Heidelberg–New York, September 2003. Springer-Verlag.

[108] Bruce Schneier, Doug Whiting, and Steve Bellovin. AES key agility issues in high-speed IPsec implementations, http://www.research.att.com/ smb/papers/aes-keyagile.ps, May 17 2000.

[109] B. Serebrin et al. A stream processor development platform. In *IEEE International Conference on Computer Design:VLSI in Computers & Processors*, pages 303–309, Washington - Brussels - Tokyo, September 2002. IEEE.

[110] Madhusudanan Seshadri and Mikko Lipasti. A case for vector network processors. In *Network Processor Conference West*, San Jose, CA, Oct 2002.

[111] Devavrat Shah, Sundar Iyer, Balaji Prabhakar, and Nick McKeown. Maintaining statistics counters in router line cards. *IEEE Micro*, 2002.

[112] V. Shoup. On fast and provably secure message authentication based on universal hashing. In Neal Koblitz, editor, *Advances in Cryptology - Crypto '96*, pages 313–328, Berlin, 1996. Springer-Verlag. Lecture Notes in Computer Science Volume 1109.

[113] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *OSDI*, pages 45–60, 2004.

[114] D. R. Stinson. Universal hashing and authentication codes. In Joan Feigenbaum, editor, *Advances in Cryptology - Crypto '91*, pages 74–85, Berlin, 1991. Springer-Verlag. Lecture Notes in Computer Science Volume 576.

[115] D. R. Stinson. On the connections between universal hashing, combinatorial designs and error-correcting codes. In *ECCCTR: Electronic Colloquium on Computational Complexity, technical reports*, 1995.

[116] Mikkel Thorup. Even strongly universal hashing is pretty fast. In *Proc. of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 496–497, San Francisco, California, 2000.

[117] Mikkel Thorup and Yin Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. In *Proc. of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 615–624, 2004.

[118] David Wagner, Helger Lipmaa, and Phillip Rogaway. Comments to NIST concerning AES modes of operations: CTR-mode encryption, September 27 2000.

[119] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, 2004.

[120] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Collision search attacks on SHA1. Technical report, Shandong University, Shandong, China, 2005.

[121] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In *Eurocrypt 2005*, May 2005.

[122] Mark N. Wegman and J. Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, June 1981.

[123] A. Wigderson. Lectures on the fusion method and derandomization. Technical Report SOCS-95.2, School of Computer Science, McGill University, 1995.

[124] Andrew Wolfe and John P. Shen. A variable instruction stream extension to the VLIW architecture. *ACM SIGPLAN Notices*, 26(4):2–14, April 1991.

[125] Haiyong Xie et al. Architectural analysis of cryptographic applications for network processors. In *Workshop on Network Processors*, pages 42–52, Cambridge,Massachusetts, Feb. 2002.

[126] Bo Yang, Ramesh Karri, and David A. McGrew. Divide-and-concatenate: an architecture level optimization technique for universal hash functions. In *Proceedings of the 41st Annual conference on Design Automation (DAC-04)*, pages 614–617, New York, June 7–11 2004. ACM Press.

[127] Qi (George) Zhao, Abhishek Kumar, Jia Wang, and Jun (Jim) Xu. Data streaming algorithms for accurate and efficient measurement of traffic and flow matrices. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 350–361, New York, NY, USA, 2005. ACM Press.