Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2013, Article ID 681894, 33 pages
http://dx.doi.org/10.1155/2013/681894

Research Article

Hardware Accelerators Targeting a Novel Group Based Packet Classification Algorithm

O. Ahmed, S. Areibi, and G. Grewal

School of Engineering and Computer Science, University of Guelph, Guelph, ON, Canada N1G 2W1

Correspondence should be addressed to S. Areibi; sareibi@uoguelph.ca

Received 10 December 2012; Revised 26 February 2013; Accepted 26 February 2013

Academic Editor: Michael Hübner

Packet classification is a ubiquitous and key building block for many critical network devices. However, it remains as one of the main bottlenecks faced when designing fast network devices. In this paper, we propose a novel Group Based Search packet classification Algorithm (GBSA) that is scalable, fast, and efficient. GBSA consumes an average of 0.4 Megabytes of memory for a 10 k rule set. The worst-case classification time per packet is 2 microseconds, and the preprocessing speed is 3 M rules/second based on a Xeon processor operating at 3.4 GHz. When compared with other state-of-the-art classification techniques, the results showed that GBSA outperforms the competition with respect to speed, memory usage, and processing time. Moreover, GBSA is amenable to implementation in hardware. Three different hardware implementations are also presented in this paper including an Application Specific Instruction Set Processor (ASIP) implementation, and two pure Register-Transfer Level (RTL) implementations based on Impulse-C and Handel-C flows, respectively. Speedups achieved with these hardware accelerators ranged from 9x to 18x compared with a pure software implementation running on a Xeon processor.

1. Introduction

Packet classification is the process whereby packets are categorized into classes in any network device. A packet P is said to match a particular rule R if all the fields of the header of P satisfy the regular expression of R. Table 1 gives an example of a twelve-rule classifier. Each rule contains five fields. The source and destination IP address are 32 bits each, while the source and destination port address are 16 bits each. An 8-bit address represents the protocol. Both IP and protocol are in starting "base/mask" format, while the ports are in range (starting : ending) format. The IP address should be converted to a range format by using the mask field. For example, an IP address "0.83.4.0/22" can be converted to a range by using the mask of 22 bits or 255.255.252.0 to produce the low part of the range (0.83.4.0). The high part of the range can be generated using the following formula: High = Low OR $2^{32\text{-Mask}}$. Thus, the IP 0.83.4.0/22 can be converted to 0.83.4.0 as the low part and 0.83.7.255 as the high part. When the mask equals 32, the IP is represented in exact format, which translates to a match of the IP. All fields which are in prefix (or exact format) in Table 1 can be easily converted to a range of high and low fields using the above formula. It is important to remember that when the IP address is converted to a range format, the length of the IP source and IP destination in Table 1 will change from 32 to 64 bits.

Almost every packet in a network is classified at one or more stages. For example, elements such as layer-2 (switches) and layer-3 (routers), as well as special-purpose classifiers such as firewalls and load balancers, classify a packet as they forward it from the end host to the web server. A router classifies the packet to determine where it should be forwarded, and it also determines the *Quality of Service (QoS)* it should receive. In addition to the process of classifying packets, the router attempts to balance the load among other servers that will eventually process these packets (load balancer). A firewall classifies the packets based on its security policies to decide whether or not to drop the packet, based on the set of rules contained in the database.

1.1. Challenges and Limitations. There are three main challenges faced when performing classification in today's networks [1]: configuration hardness, inflexibility, and

Table 1: A twelve-rule classifier.

No.	IP (64 bits)		Port (32 bits)		
	Source (32 bits)	Destination (32 bits)	Source (16 bits)	Destination (16 bits)	Protocol (8 bits)
1	240.118.164.224/28	250.13.215.160/28	88 : 88	53 : 53	0x01/0xFF
2	209.237.201.208/28	159.93.124.80/28	123 : 123	123 : 123	0x11/0xFF
3	209.67.92.32/28	159.102.36.48/28	69 : 69	21 : 21	0x06/0xFF
4	63.99.78.32/28	55.186.163.16/28	750 : 750	22 : 22	0x00/0x00
5	240.178.169.176/28	250.222.86.16/28	123 : 123	22 : 22	0x2f/0xFF
6	209.238.118.160/28	159.199.212.192/28	88 : 88	80 : 80	0x11/0xFF
7	63.20.27.0/28	55.103.209.48/28	162 : 162	123 : 123	0x11/0xFF
8	0.0.0.0/0	125.176.0.0/13	0 : 65535	0 : 65535	0x06/0xFF
9	0.0.0.0/0	125.0.0.0/8	0 : 65535	0 : 65535	0x00/0x00
10	33.0.0.0/8	0.0.0.0/0	0 : 65535	0 : 65535	0x06/0xFF
11	33.0.0.0/8	0.0.0.0/0	0 : 65535	0 : 65535	0x06/0xFF
12	69.0.206.0/23	0.0.0.0/0	0 : 65535	0 : 65535	0x00/0x00

inefficiency. These limitations result from the following characteristics of packet classification.

(i) Complexity of classification operations: each rule could contain any of the following five formats: prefix, exact, range, greater than, and less than. A combination of different formats tends to increase the complexity of algorithms in the literature [2].

(ii) Semantic gap between entities on the packet path: different entities on a packet's path, including the source and destination, have different amounts of semantic and local context related to a particular classification application.

(iii) Resource mismatch between entities on the packet path: different entities on a packet's path have different amounts of processing resources (e.g., CPU, memory) to utilize when classifying each packet.

1.2. Performance Metrics for Classification Algorithms. The following are important metrics that can be used to evaluate and compare the performance of packet classification algorithms.

(i) *Classification speed:* faster links require faster classification. For example, links running at 40 Gbps can bring 125 million packets per second (assuming minimum sized 40 byte TCP/IP packets).

(ii) *Low storage requirements:* lower storage requirements enable the deployment of fast memory technologies, such as SRAM, which can be used as an on-chip cache by both software algorithms and hardware architectures.

(iii) *Fast updates:* as the rule set changes, data structures need to be updated on the fly. If the update process is time consuming, the number of lost packets will increase accordingly.

(iv) *Scalability in the number of header fields:* the fields involved in packet classification vary based on the application that is being targeted.

(v) *Flexibility in specification:* a classification algorithm should support general rules, including prefixes and operators (range, less than, greater than, equal to, etc.), in addition to wild-cards.

1.3. Contributions. The main contributions of this work can be summarized as follows.

(1) First, we introduce a novel and efficient algorithm which has high scalability, is amenable to hardware implementation [3], requires low memory usage, and can support a high speed wire link. We explain in detail the software implementation and data structures used to help researchers reproduce the work.

(2) Based on standard benchmarks used by many researchers, results obtained indicate that the GBSA is one of the fastest existing algorithm that can accommodate a large rule set *of size 10 k or more* (which could be a candidate rule set for the next generation of core routers).

(3) We propose further enhancements to the GBSA, making it more accessible for a variety of applications through hardware implementation. Three hardware accelerators using different implementation techniques are presented in this paper.

The remainder of this paper is organized as follows. Section 2 gives an overview of the main published algorithms in the field of packet classification. In Section 3, the algorithm and the main steps to implement it are presented, with all stages from preprocessing to classification. Comparisons with other published algorithms are then presented in Section 4. Enhancements to the performance of the GBSA algorithm are realized using an Application Specific Instruction Processor (ASIP) introduced in Section 5. Pure RTL implementations based on Electronic System Level design using Handel-C and Impulse-C are discussed in Section 6. A detailed comparison of all hardware implementations of GBSA is provided in

Section 7. Finally, conclusions and future work are presented in Section 8.

2. Previous Work

Since the packet classification problem is inherently complex from a theoretical standpoint [2], previously published algorithms have not been able to fully satisfy the need of network devices. The authors in [4] categorize existing packet classification algorithms into four main classes (exhaustive search, decision tree, decomposition, and Tuple Space based methods), based on the algorithm's approach to classifying packets.

2.1. Software Based Approaches. *Brute force/exhaustive search techniques* examine all entries in the filter set sequentially similar to Ternary Content Addressable Memory (TCAM) [2] approaches. Brute force techniques perform well in terms of memory usage, do not require any preprocessing, and can be updated incrementally with ease. However, they require $O(N)$ memory access per lookup, where N is the number of rules in the rule set. For even modest sized filter sets, a linear search becomes prohibitively slow.

Decision Tree based approaches construct a decision tree from the filters and traverse the decision-tree using packet fields, such as HiCuts [5] and HyperCuts [6] which are very close in performance in terms of classification [7]. Decision trees are geared towards multiple-field classification. Each of the decision tree leaves contains a rule or subset of the rule set. Classification is performed by constructing a search key from the packet header fields and then using this key to traverse the tree. The main disadvantage of this method is the high memory requirement and long preprocessing time.

Decomposition/Divide-and-Conquer Techniques tend to decompose the multiple-field search into instances of single-field searches, performing independent searches on each packet field before combining the results, like RFC [8], ABV [9], Parallel BV [10], and PCIU [11]. Another viable approach that deals with the complexity of packet classification is the decomposition of multiple-field into several instances of a single-field search problem before combining the results at the end. This method provides a high-speed classification time, yet it requires more preprocessing time and memory usage, which makes it unsuitable for systems that require frequent updating of the rule set.

Tuple Space based methods partition the filter set according to the number of specified bits in the filter and then probe the partitions or a subset of the partitions using simple exact match searches, such as Tuple Space [12], Conflict-Free Rectangle search [13]. The Tuple Space approach attempts to quickly narrow the scope of a multiple-field search by partitioning the filter set using tuples. It is motivated by the observation that the number of distinct tuples is much smaller than the number of filters in the filter set. This class of classification algorithms has the lowest memory usage, yet it requires a high preprocessing time, and its classification time could vary based on the nature of the rule set.

2.2. Hardware Based Approaches. It has been noted that to circumvent the shortcomings of software based approaches to packet classification, many novel techniques employ both hardware and software components. It is emphasized that pure software implementations typically suffer from three major drawbacks: a relatively poor performance in terms of speed (due to the number of memory access required), a lack of generalizability (in order to exploit certain features of a specific type of rule set), and a large need for preprocessing.

To achieve the flexibility of software at speeds normally associated with hardware, researchers frequently employ reconfigurable computing options using Field Programmable Gate Arrays (FPGAs). Although the flexibility and the potential for parallelism are definite incentives for FPGA based approaches, the limited amount of memory in state-of-the-art FPGA designs entails that large routing tables are not easily supported. Consequently, researchers frequently make use of TCAM when developing new packet classification algorithms. Although TCAM can be used to achieve high throughput, it does exhibit relatively poor performance with respect to area and power efficiency. Nevertheless, the authors in [14] were able to develop a scalable high throughput firewall, using an extension to the Distributed Crossproducting of Field Labels (DCFL) and a novel reconfigurable hardware implementation of Extended TCAM (ETCAM). An Xilinx Virtex 2 Pro FPGA was used for their implementation, and as the technique employed was based on a memory intensive approach, as opposed to the logic intensive one, on-the-fly updating remained feasible. A throughput of 50 million packets per second (MPPS) was achieved for a rule set of 128 entries, with the authors predicting that the throughput could be increased to 24 Gbps if the designs were to be implemented on Virtex-5 FPGAs. In their development of a range reencoding scheme that fits in TCAM, the authors of [15] proposed that the entire classifier be reencoded (as opposed to previous approaches that elect not to reencode the decision component of the classifier). The approach in [15] significantly outperforms previous reencoding techniques, achieving at least five times greater space reduction (in terms of TCAM space) for an encoded classifier and at least three times greater space reduction for a re-encoded classifier and its transformers. Another interesting range encoding scheme to decrease TCAM usage was proposed in [16], with ClassBench being used to evaluate the proposed scheme. The encoder proposed in [16] used between 12% and 33% of the TCAM space needed in DRIPE or SRGE and between 56% and 86% of the TCAM space needed in PPC, for classifiers of up to 10 k rules.

Several other works on increasing the storage efficiency of rule sets and reducing power consumption have also been investigated, with pure Register-Transfer Level (RTL) hardware approaches proposed by many researchers [17, 18]. Although the dual port IP Lookup (DuPI) SRAM based architecture proposed in [17] maintains packet input order and supports in-place nonblocking route updates and a routing table of up to 228 K prefixes (using a single Virtex-4), the architecture is only suitable for single-dimension classification tasks. The authors of [18], on the other hand, proposed a five-dimension packet classification flow, based

Table 2: Benchmarks: rule sets and traces used.

Benchmark size	ACL		FW		IPC	
	Rule	Trace	Rule	Trace	Rule	Trace
0.1 k	98	1000	92	920	99	990
1 k	916	9380	791	8050	938	9380
5 k	4415	45600	4653	46700	4460	44790
10 k	9603	97000	9311	93250	9037	90640

on a memory-efficient decomposition classification algorithm, which uses multilevel Bloom Filters to combine the search results from all fields. Bloom Filters, having recently grown in popularity, were also applied in the approaches described in [18, 19]. The interesting architecture proposed in [19] used a memory-efficient FPGA based classification engine entitled Dual Stage Bloom Filter Classification Engine (2sBFCE) and was able to support 4 K rules in 178 K bytes memories. However, the design takes 26 clock cycles on average to classify a packet, resulting in a relatively lower average throughput of 1.875 Gbps. The hierarchical based packet classification algorithm described in [18] also made use of a Bloom filter (for the source prefix field), and the approach resulted in a better average and worst-case performance in terms of the search and memory requirements.

Several novel packet classification algorithms mapped onto FPGAs have been published in recent years including [20–23]. In [20], several accelerators based on hardware/software codesign and Handel-C were proposed. The hardware accelerators proposed achieved different speedups over a traditional general purpose processor. The authors of [21] proposed a multifield packet classification pipelined architecture called Set Pruning Multi-bit Trie (SPMT). The proposed architecture was mapped onto an Xilinx Virtex-5 FPGA device and achieved a throughput of 100 Gbps with dual port memory. In [22] the authors presented a novel classification technique that processes packets at line rate and targets NetFPGA boards. However, there is no report of preprocessing time nor evaluation of the architecture on any known benchmarks. An interesting work by [23] presented a novel decision tree based linear algorithm that targets the recently proposed OpenFlow that classifies packets using up to 12 tuple packet header fields. The authors managed to exploit parallelism and proposed a multipipeline architecture that is capable of sustaining 40 Gbps throughput. However, the authors evaluated their architecture using only the ACL benchmark from ClassBench. The author in [24] presents a low power architecture for a high-speed packet classifier which can meet OC-768 line rate. The architecture consists of an adaptive clocking unit which dynamically changes the clock speed of an energy efficient packet classifier to match fluctuations in traffic on a router line card.

3. A Group Based Search Approach (GBSA)

The most rudimentary solution to any classification problem is simply to search through all entries in the set. The complexity of the sequential search is $O(N)$, where N is the number of rules in the rule set. Therefore, classification time will increase linearly by increasing the number of rules. Linear search is still unsuitable for a database with thousands of rules. The increase in computation time can be mitigated by reducing N. The number of rules can be reduced by decomposing the rule set into groups and, accordingly, performing a parallel search. This approach tends to increase the speed by a factor of F, yet it increases the resources needed by a similar factor. An alternative approach to reducing the search size is to *cluster* the rule set. Accordingly, the search will then be performed on a smaller subset. Hence, performance is improved without necessarily increasing the computational resources required.

3.1. Main Idea. The main idea of the GBSA is to reduce the number of rules existing in the classifier using a clustering approach. This will be explained in more detail using the rule sets in ClassBench [25]. These rule sets (benchmarks) are used by many researchers to evaluate packet classification algorithms, since their characteristics are similar to actual rule sets used in backbone routers [7].

3.1.1. Benchmarks. Table 2 introduces three main benchmarks along with their testing packets (traces). The ClassBench [25] performed a battery of analysis on 12 real filter sets provided by Internet Service Providers (ISPs), network equipment vendors, and researchers working in the field.

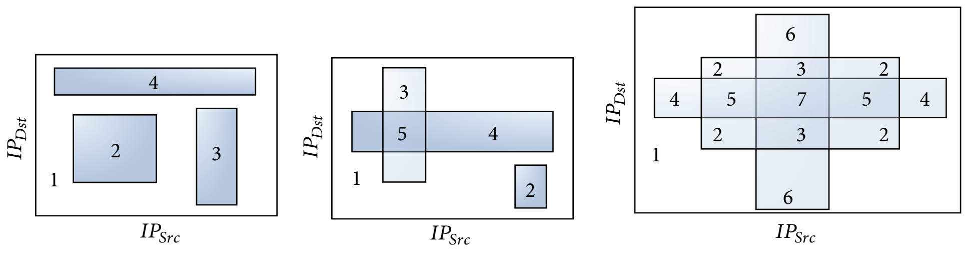
The filter sets utilize one of the following formats.

(i) Access Control List (ACL): standard format for security, VPN, and NAT filters for firewalls and routers (enterprise, edge, and backbone).

(ii) Firewall (FW): proprietary format for specifying security filters for firewalls.

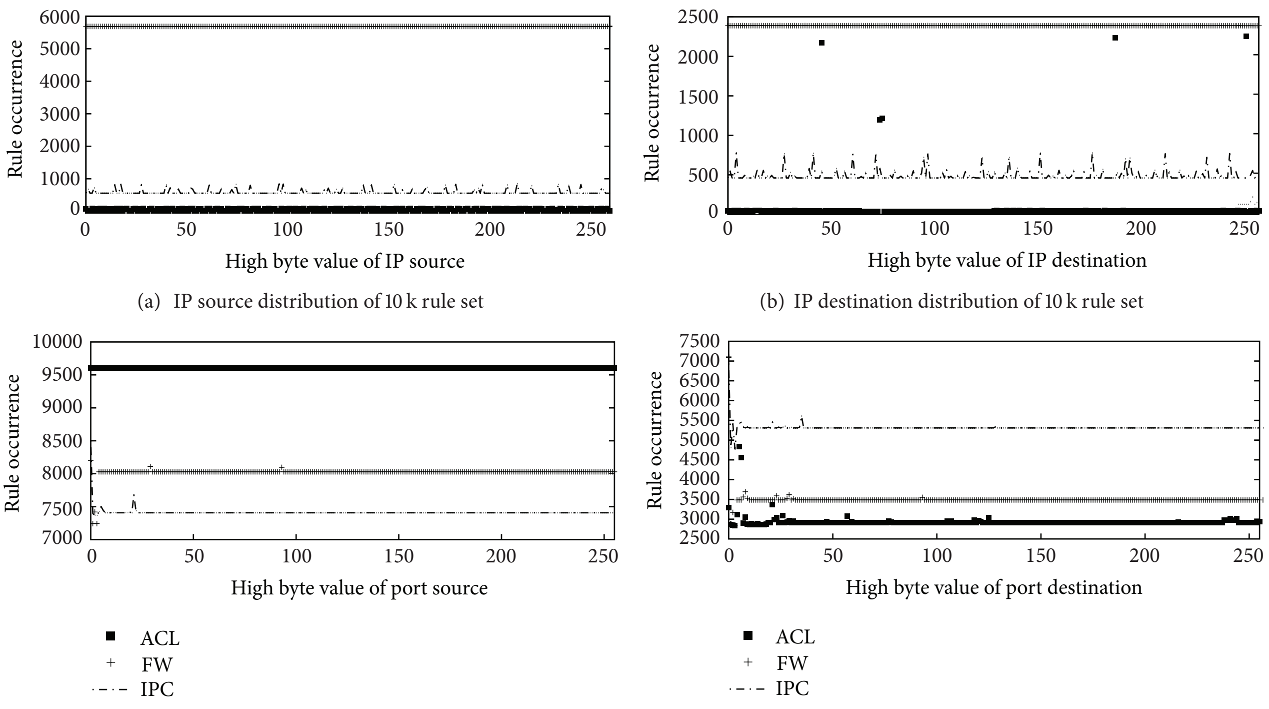
(iii) IP Chain (IPC): decision tree format for security, VPN, and NAT filters for software-based systems.

3.1.2. Rule Overlap. The rule set can be described as a two-dimensional surface that employs a few of the packet fields. The high byte of the source IP address is used to point to the x-axis, while the high byte of the destination IP address is used to point to the y-axis. Thus, a two-dimensional surface (256×256) can be plotted as an alternative representation of the three benchmarks, as demonstrated by Figure 1. It is clear from Figure 1 that the number of overlapping/nonoverlapping regions varies based on the nature of the rules and their corresponding overlap. Each region can contain one or more rules. The number of regions tends to increase as more overlap occurs among the rule set, which causes more challenges to the classification algorithm



(a) Four regions (ACL) (b) Five regions (IPC) (c) Seven regions (FW)

FIGURE 1: Overlap and regions of benchmarks.



(a) IP source distribution of 10 k rule set (b) IP destination distribution of 10 k rule set

(c) Port source distribution of 10 k rule set (d) Port destination distribution of 10 k rule set

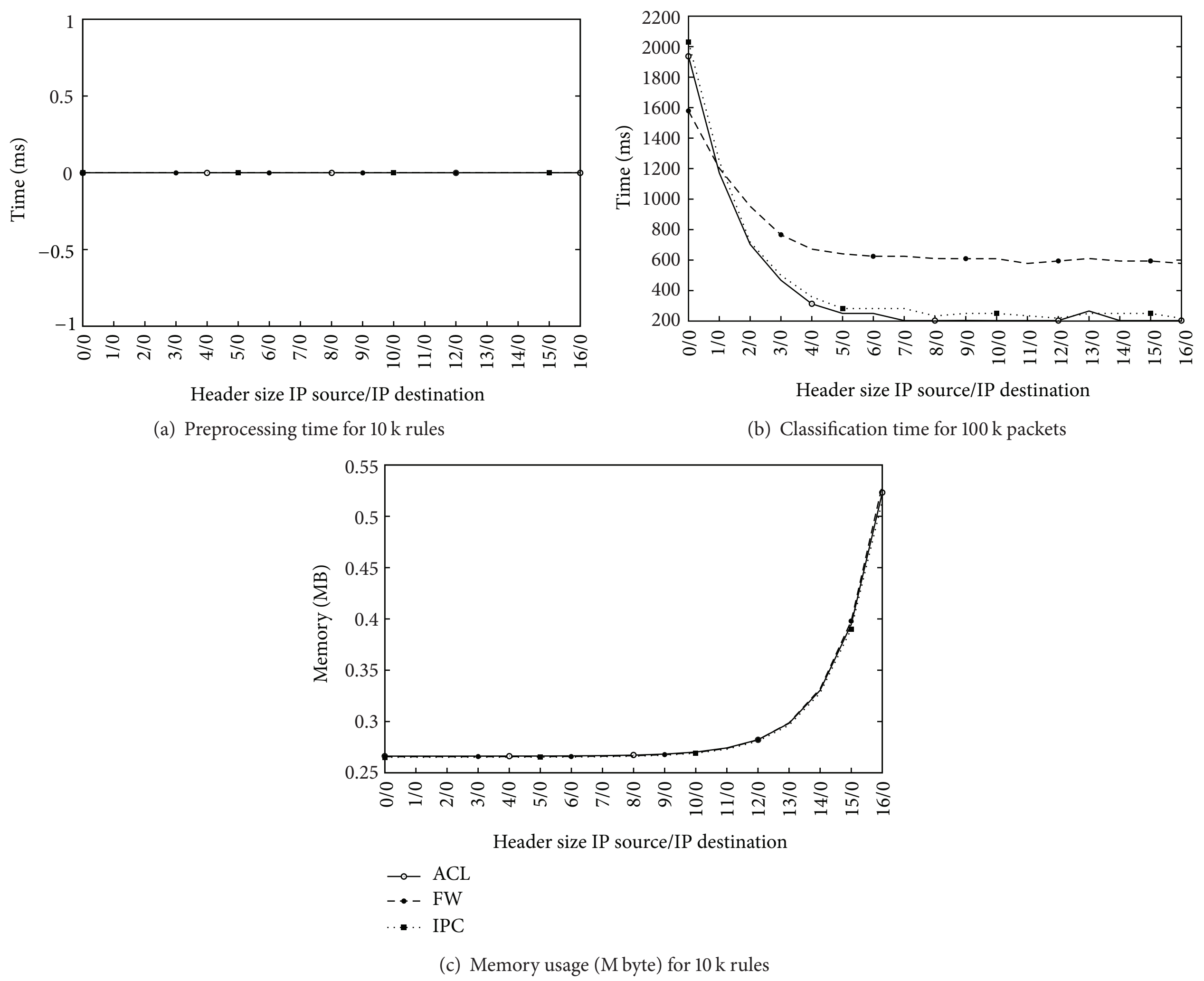
FIGURE 2: Distribution of rules using the first byte of each dimension (10 k rule set).

in terms of preprocessing and classification. As illustrated in Figure 1, the FW benchmarks in Table 2 are the most challenging rule sets due to the high number of overlapping regions. On the other hand, the ACL benchmark has no overlapping regions and therefore can be handled easier by classification algorithms.

3.1.3. Rule Distribution. One of the main objectives of GBSA is to distribute rules based on the first few bits of each dimension in the rule set. If the rules are distributed in a uniform manner throughout the group, this should reduce classification time. The uniform distribution will attempt to force each group or cluster formed to have an equal number of rules. A graphic representation for the first byte of each dimension should identify an appropriate rule distribution among the five dimensions. The distributions of the 10 k rule set for all benchmarks (using the first byte of each dimension) are shown in Figure 2. By observing Figure 2 carefully, we find that the highest IP byte (byte [3]) for both source and destination has the lowest average rule occurrence (distribution) for the majority of benchmarks compared to the average rule occurrence (distribution) using the source and destination port.

Despite the fact that the average occurrence in any of these distributions is much lower than the original rule set size (10 k), they are not practical for high-speed classification, due to the cluster size which needs to be searched. Combining two or more dimensions tends to reduce the number of rules per group to an even greater extent. Therefore, it would be beneficial to combine the two sets of bits of IP addresses from the rule set in the manner of a clustering methodology.

3.1.4. Grouping and Clustering. By combining the prefix of both source and destination IP addresses, the rules tend to distribute more uniformly and the average size should decrease dramatically. The highest significant bytes of the



(a) Preprocessing time for 10 k rules

(b) Classification time for 100 k packets

(c) Memory usage (M byte) for 10 k rules

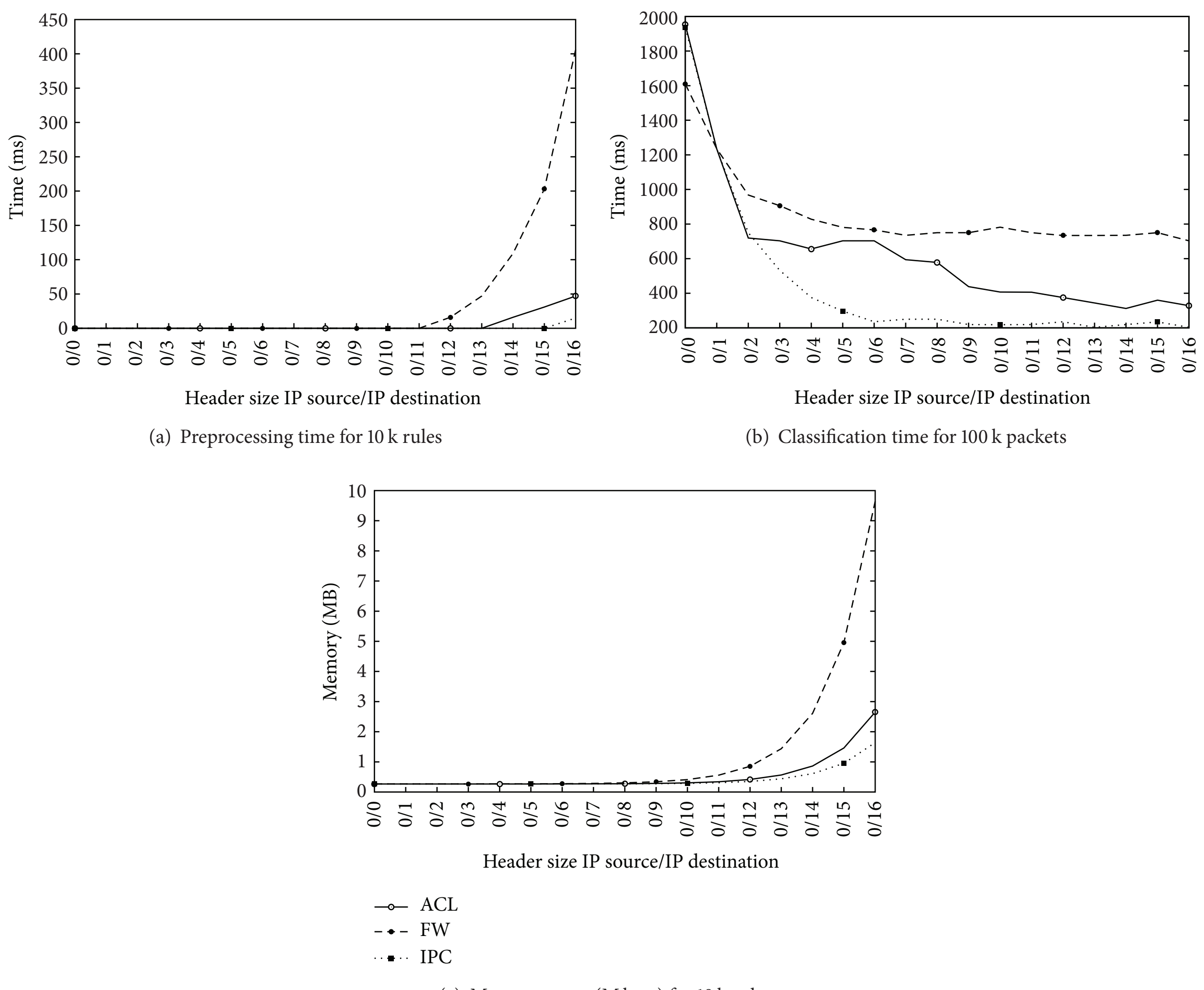
FIGURE 3: Effect of varying IP source size on preprocessing, classification, and memory.

source/destination IP address can be combined to form a 16-bit group address (cluster index). A lookup table of size 2^{16} (64 K) will then be used to point to all groups of rule sets. The maximum number of groups resulting from this operation should be less than the maximum possible size (i.e., 64 k). By examining the rule sets of all benchmarks carefully, we can conclude that the wild-card rules are the main contributors to the increase of number of rules per group. Therefore, this clustering methodology can be further improved in terms of preprocessing, memory usage, and classification by excluding all of the wild-cards in both IP source and destination. Two other lookup tables of size 2^8 (256 bytes) are used to point to the wild-card rules for both IP source and destination.

In general, the GBSA's main goal is to utilize the starting bits from both IP source and IP destination to generate cluster of rules. The GBSA can accommodate any number of bits from source/destination IP to group and cluster rules.

To ensure that the idea of combining IP source and IP destination to form groups is valid, we tested our idea by using either the IP source or IP destination to form the groups of the GBSA. Figure 3 depicts the result for the 10 k rule set for the three benchmarks when only the IP source prefix was used. In this experiment, we clustered the rules using only IP source with the prefix varying from 0 up to 16. It is clear that the classification time remains high for all sizes of the IP source prefix for the FW benchmark. It is also clear that after a prefix of size six, the classification time decreases for all other benchmarks. On the other hand, the memory increases sharply after size twelve.

Figure 4 depicts the result for the 10 k rule set for the three benchmarks when only the IP destination prefix was used. In this experiment, we cluster the rule using the IP destination instead of the IP source with the same prefix range (from 0 up to 16). It is evident that the classification time remains high for all sizes of the IP destination prefix for both FW and ACL benchmarks. It is also clear that after a prefix of size two the classification time does not improve for FW. On the other hand, the memory usage increases sharply after size of three for the FW. It is obvious from Figures 3 and 4 that using a single dimension does not reduce the classification time involved. Therefore, combining two or more prefixes of the rule dimensions should lead to better



(a) Preprocessing time for 10 k rules

(b) Classification time for 100 k packets

(c) Memory usage (M byte) for 10 k rules

Figure 4: Effect of varying IP destination size on preprocessing, classification, and memory.

solutions. From Figure 2, the IP source and IP destination are the best candidates for the GBSA algorithm proposed, as they have the lowest average distribution among the four dimensions. Based on a prefix of size 8, the three tables sizes will be as follows:

(i) $2^{8\text{(from the Source prefix)} + 8\text{(from the Destination prefix)}} = 2^{16}$ = 64 K Byte;

(ii) $2^{8\text{(from the Source prefix)}} = 2^8 = 256$ Byte;

(iii) $2^{8\text{(from the Destination prefix)}} = 2^8 = 256$ Byte.

The prefixes of both IP source and IP destination will be combined to form a 16-bit address for a lookup table of size 2^{16}. The maximum number of groups resulting from this operation should be less than the maximum possible size (i.e., 64 KB). For the 8-bit prefixes for both IP addresses, we use a 64 KB memory as a lookup table of group's pointers (i.e., index of all rules belonging to it). Each group will contain all the rules that have the same values of the high byte for both source and destination IP addresses. The number of rules belonging to each group will be condensed as compared to the original rule set size.

3.2. The Preprocessing Phase. Preprocessing in GBSA is divided into two main steps:

(i) generating groups for each of the three lookup tables,

(ii) removing redundancy within these groups.

(1) Group Generation. The group generation step is demonstrated by the pseudocode of Algorithm 1. The rule set is divided into three main groups:

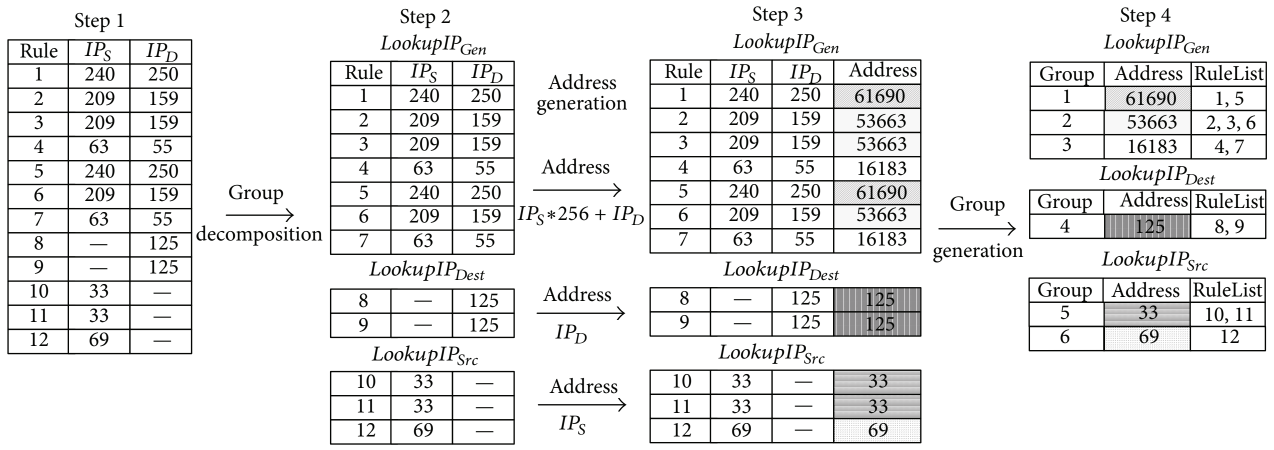
(i) source IP with wild-card rules,

(ii) destination IP with wild-card rules,

(iii) remaining rules.

```
For all rules in the rule-set
    IF SourceIP is a WildCard
        Append rule to $LookupIP_{Dst}$ Table
    Else IF DestIP is a WildCard
        Append rule to the $LookupIP_{Src}$ Table
    Else
        Append rule to $LookupIP_{Gen}$ Table
```

Algorithm 1: Preprocessing: group generation.



Step 1

Rule	IP_S	IP_D
1	240	250
2	209	159
3	209	159
4	63	55
5	240	250
6	209	159
7	63	55
8	—	125
9	—	125
10	33	—
11	33	—
12	69	—

Step 2

$LookupIP_{Gen}$

Rule	IP_S	IP_D
1	240	250
2	209	159
3	209	159
4	63	55
5	240	250
6	209	159
7	63	55

$LookupIP_{Dest}$

8	—	125
9	—	125

$LookupIP_{Src}$

10	33	—
11	33	—
12	69	—

Step 3

$LookupIP_{Gen}$

Rule	IP_S	IP_D	Address
1	240	250	61690
2	209	159	53663
3	209	159	53663
4	63	55	16183
5	240	250	61690
6	209	159	53663
7	63	55	16183

$LookupIP_{Dest}$

8	—	125	125
9	—	125	125

$LookupIP_{Src}$

10	33	—	33
11	33	—	33
12	69	—	69

Step 4

$LookupIP_{Gen}$

Group	Address	RuleList
1	61690	1, 5
2	53663	2, 3, 6
3	16183	4, 7

$LookupIP_{Dest}$

Group	Address	RuleList
4	125	8, 9

$LookupIP_{Src}$

Group	Address	RuleList
5	33	10, 11
6	69	12

Figure 5: The GBSA preprocessing steps using the rule set in Table 1.

(2) Redundancy Removal. The redundancy removal step performs further preprocessing on the three lookup tables by assigning unique IDs to each cluster generated.

(3) Preprocessing Example. In this example we explain the preprocessing phase in more detail. The example demonstrates the capability of the preprocessing phase to cluster rules that have similar attributes, based on contents of Table 1 and the steps shown in Figure 5.

Step 1. For each rule listed in Table 1, create a table entry consisting of the most significant byte of IP source and the most significant byte of IP destination as shown in Figure 5. Rules 1 to 7 have both fields present, yet rules 8 and 9 have a wild-card in the IP source (IP_{src}) field. On the other hand, rules 10 to 12 have a wild-card in the IP destination (IP_{dst}) field. A wild-card is shown as a "-" in Step 1 of Figure 5.

Step 2. Partition the table created in Step 1 to three separate tables, namely, $LookupIP_{Gen}$, $LookupIP_{Dest}$, and $LookupIP_{Src}$. The partitioning of these table depends on the values of both IP_{src} and IP_{dst} columns in Step 1.

Step 3. For each table entry in Step 2, if the rule belongs to the $LookupIP_{Gen}$, the address field generated is based on the following formula: $Address = IP_{src} * 256 + IP_{dst}$; otherwise the existing IP_{src} or IP_{dst} fields are copied to the address field.

Step 4. All rules which have similar address are combined into a single group or cluster. For example, Group (cluster) 1 contains rules 1 and 5, as both of these rules have the same $LookupIP_{Gen}$ address, namely, 61690.

After the completion of Step 4, six groups (clusters) are formed from the original twelve rules. Figure 6 shows the memories (lookup tables) required to store the generated groups representing the rule set.

Each memory consists of two fields: *ADR* (the starting address of the rule list of the group) and *Size* (the number of rules in the group). For example location 16183 in the $LookupIP_{Gen}$ memory contains $ADR = 5$ and $Size = 2$ which represent Group 3 (cluster). Group 3 (cluster) (Step 4, third row) of Figure 5 consists of two rules (namely, 4 and 7). In the RuleList memory, location 5 contains an entry 4 (i.e., rule 4) followed by rule 7 in location 6.

3.3. Classification. Following the preprocessing phase, the classification phase proceeds with the following steps.

(i) The high bytes of the IP source and destination addresses of the incoming packets are first extracted as shown in Figure 7.

(ii) Searching within $lookupIP_{Gen}$ table:

  (a) the most significant bytes of the IP source and destination addresses are concatenated to form a 16-bit address,

  (b) the 16-bit address formed is then used to access rules in the 64 KB $lookupIP_{Gen}$ table,

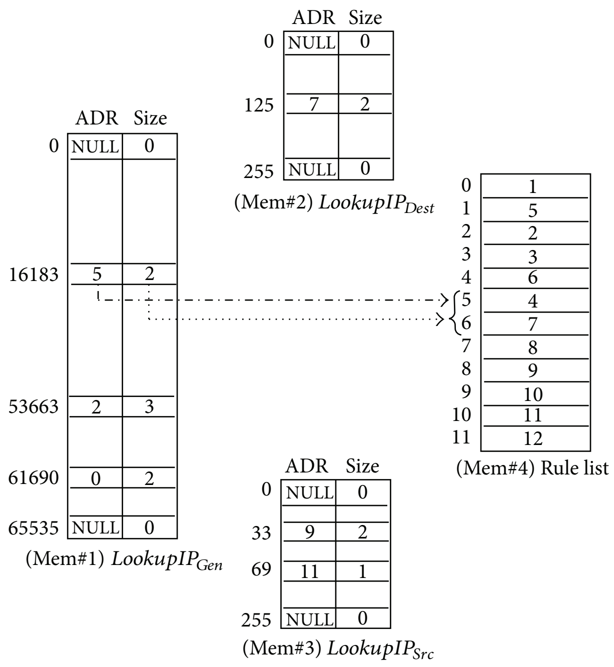


FIGURE 6: The GBSA memory structure following the preprocessing stage.

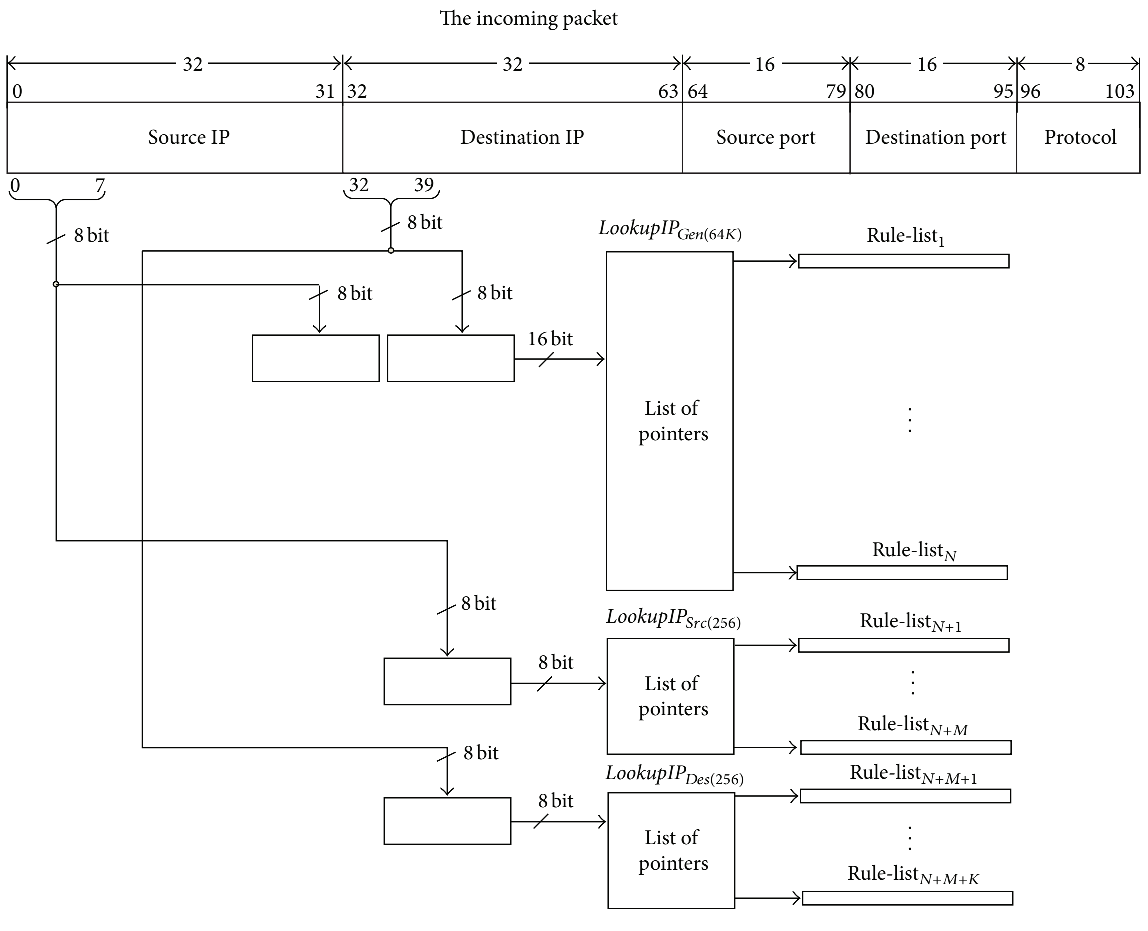


FIGURE 7: GBSA classification phase.

Table 3: Classification example: applying a 5-packet set.

Incoming packet	IP_{Src} (32 bits)	IP_{Dst} (32 bits)	$Port_{Src}$ (16 bits)	$Port_{Dst}$ (16 bits)	Protocol (8 bits)	Packet belongs to Rule ID
a	240.118.164.31	250.13.215.168	88	53	1	1
b	209.238.118.166	159.199.212.198	88	80	17	6
c	63.99.78.40	55.186.163.23	750	22	0	4
d	100.251.52.13	125.10.251.18	80	80	17	9
e	69.0.207.9	68.103.209.50	162	123	17	12

(c) a group ID will be identified. For example, when the IP_{Scr} = 63 and the IP_{Des} = 55 of the incoming packet, then a group ID = 16183 is calculated. This ID is used as an address to the *lookupIP*$_{Gen}$ (Mem 1) whose contents are a group of size 2 and a starting address of 5,

(d) if the group ID is not empty, a sequential search is performed within the group to match the incoming packet with an existing rule in the group. In this example the incoming packet will be matched with rule 4,

(e) if a match exists, the winning rule is generated and the search is terminated. Otherwise, a search is initiated in either *lookupIP*$_{Src}$ (Mem 2) or *lookupIP*$_{Dst}$ (Mem 3).

(iii) Searching within *lookupIP*$_{Src}$ table:

(a) the most significant byte of the IP source address of the incoming packet is used as a pointer to *lookupIP*$_{Src}$,

(b) a group ID will be identified,

(c) if the group ID is not empty, a sequential search is performed within the group to match the incoming packet with an existing rule in the rule set. For example, when IP_{Scr} = 63, the value will be null (which indicates that there is no group to point to),

(d) if a match exists, the winning rule is generated and the search is terminated. Otherwise, a search is initiated in *lookupIP*$_{Dst}$ table,

(iv) Searching within *lookupIP*$_{Dst}$ table:

(a) the most significant byte of the IP destination address of the incoming packet is used as a pointer to *lookupIP*$_{Dst}$,

(b) a group ID will be identified,

(c) if the group ID is not empty, a sequential search is performed within the group to match the incoming packet with an existing rule in the rule set,

(d) if a match exists, the winning rule is generated. There will be, no match for the incoming packet.

3.3.1. Classification Example. The classification phase is demonstrated using Figure 5 along with Table 3, which lists five possible incoming packets. The high bytes of both packets IP source and IP destination are used as addresses for the three lookup tables. For example, the first trace (packet **"a"**) has the following addresses:

(i) IP_{src} address in the *LookupIP*$_{Src}$ memory = 240,

(ii) IP_{dst} address in the *LookupIP*$_{Dst}$ memory = 250,

(iii) *LookupIP*$_{Gen}$ = 61690 the address is calculated based on the following formula: ((*LookupIP*$_{Src}$ $*$ 256) + *LookupIP*$_{Dst}$).

The result of *LookupIP*$_{Src}$ and *LookupIP*$_{Dst}$ will produce a NULL, yet *LookupIP*$_{Gen}$ produces rule 1. The same steps are applied to the remaining incoming packets, producing the appropriate Rule ID, as shown in Table 3.

4. Experimental Results

The evaluation of the GBSA algorithm was based on the benchmarks introduced in Table 2. The seeds and program in ClassBench [25] were used to generate the rule sets.

4.1. RFC, HiCut, Tuple, and PCIU: A Comparison. Several packet classification algorithms exist in the literature, and it would be prohibitive to compare the proposed GBSA technique to all available algorithms. Accordingly, we have chosen the most efficient packet classification algorithms [7] to be compared with our proposed approach. Figure 8 shows a comparison between RFC [8], HiCut [5], Tuple [12], PCIU [11], and GBSA [26]. It is clear from the results obtained that GBSA outperforms all algorithms in terms of memory usage, preprocessing, and classification time. The comparison was mainly based on the ACL benchmark since the number of distinct overlapping regions in both FW and IPC benchmarks are quite high, which leads to prohibitive CPU time for the preprocessing stage of the RFC algorithm. The overall performance of Tuple search using the FW benchmark deteriorates and its memory consumption increases dramatically. The GBSA's performance, on the other hand, was not affected by the complexity of the FW benchmark. Figure 8 clearly shows that the GBSA has an advantage over the other algorithms. Figure 9 shows a comparison between both GBSA and PCIU [11] on the three benchmarks with different sizes. It is clear that the GBSA algorithm outperforms the PCIU on all benchmarks.

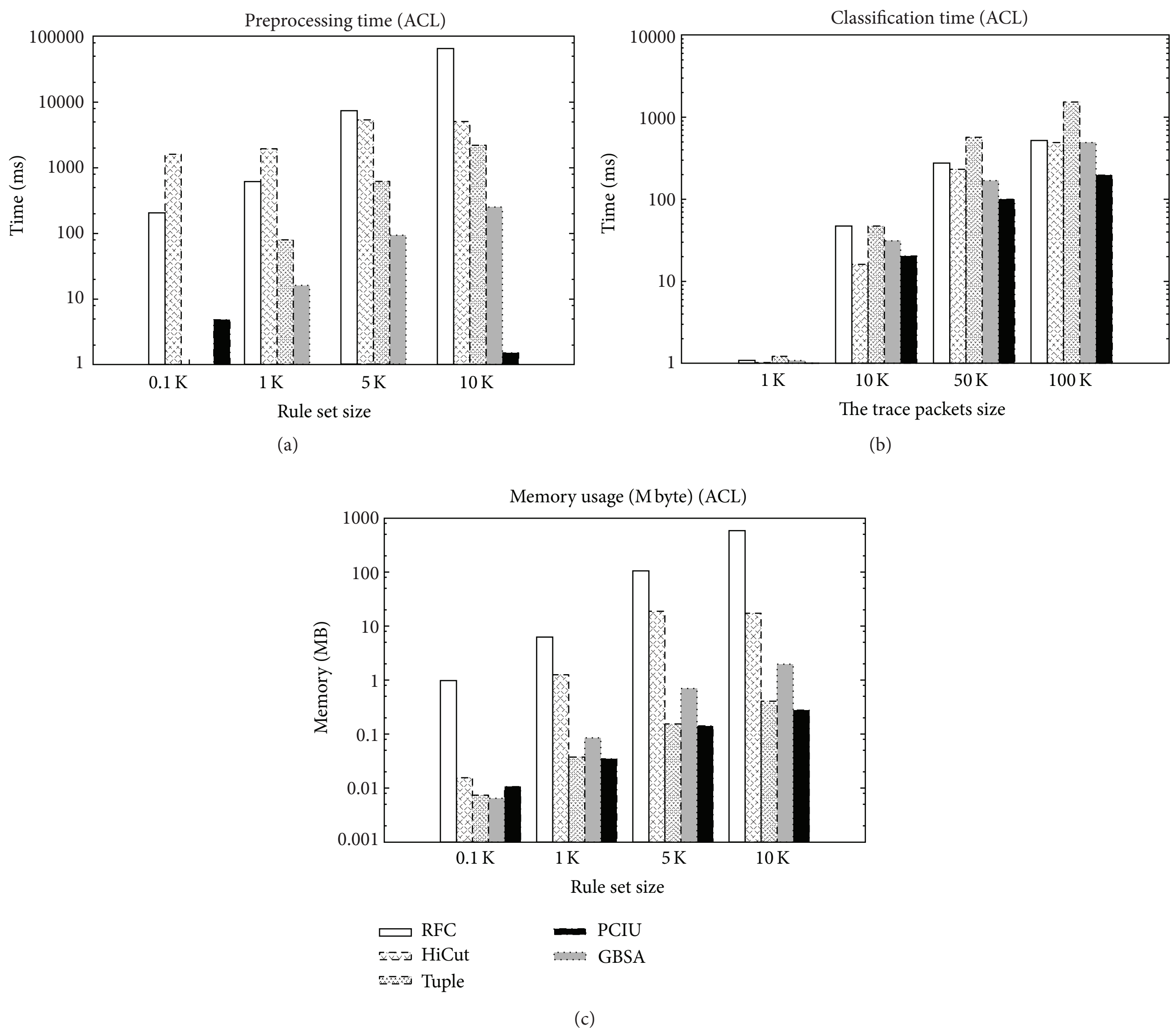


Figure 8: A comparison between RFC, HiCut, Tuple, PCIU, and GBSA.

4.2. Analysis of Results. The GBSA algorithm outperforms all the previously published packet classification algorithms due to the following properties.

(1) The GBSA avoids using complex data structures similar to those utilized in decision tree based algorithms (which necessitates traversal of the tree). Accordingly, no comparison is required at each node to make a decision about the next nodes address (these comparison operations and the number of times memory accessed tend to increase the classification time). Also, the number of nodes and their pointers tends to consume a huge amount of memory. Also, when the rule set has too many overlap regions, the size of memory and the preprocessing time increase sharply, as demonstrated previously by Figure 1 for the IPC and FW benchmarks.

(2) The GBSA does not require intensive preprocessing time similar to that required by the RFC. The high preprocessing time in RFC can easily take several days even when a powerful processor is used. Again, this problem tends to occur when a rule set has a huge overlap region.

(3) The GBSA main classification operation is "comparison" within a small group due to clustering performed in the preprocessing phase. Using a simple combination of the IP source and IP destination as an address for a lookup table makes the classification and preprocessing time faster than other methods published in the literature.

Simply put, the performance of GBSA is attributed to the simplicity and efficiency of the clustering method used. Unlike other algorithms that require sophisticated data structures,

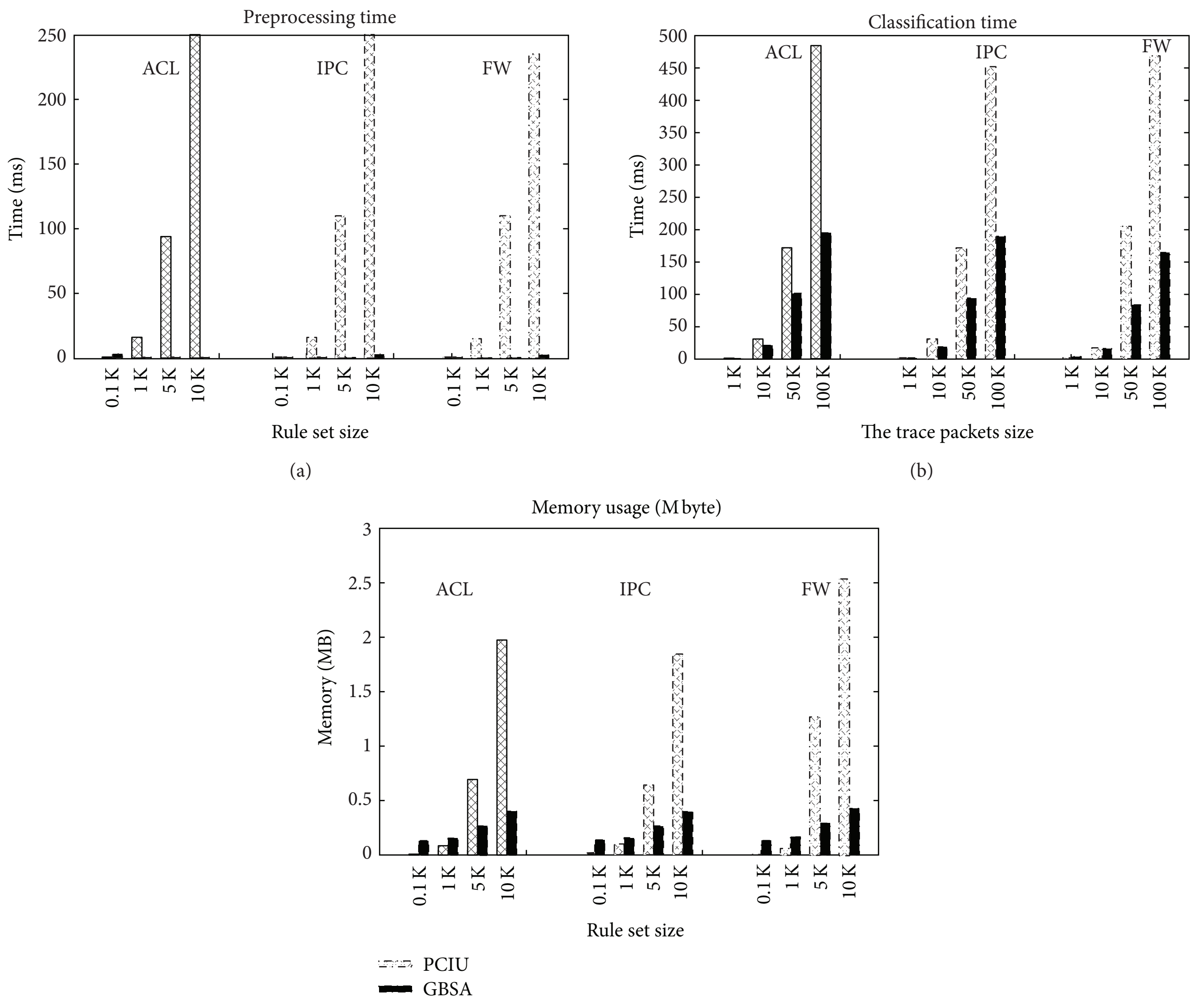


Figure 9: A comparison between PCIU and GBSA based on three benchmarks.

decision trees, and long preprocessing time, GBSA attempted to distribute the rule sets more uniformly, and thus the size of the latter decreased dramatically.

4.3. Improving Performance. While a pure software implementation can generate powerful results on a server, an embedded solution may be more desirable for some applications and clients. Embedded, customized hardware solutions are typically much more efficient in terms of speed, cost, and size as compared to solutions on general purpose processor systems. The next few sections of this paper cover three of such translations of the GBSA into the realm of embedded solutions using FPGA technology. In Section 5, we discuss an ASIP solution that attempts to develop a specialized data path to improve the speed and power of the GBSA algorithm. In Section 6, we discuss two solutions that employ a pure RTL approach with an ESL design methodology using Handel-C and Impulse-C. These sections are complete with full disclosure of implementation tools, techniques, strategies for optimization, results, and comparison.

5. Application Specific Instruction Processor (ASIP)

One of the key advantages of an ASIP implementation [27] is the substantial performance to development time ratio that also allows for greater flexibility in case updates or other upgrades become necessary in the future. While a pure RTL implementation would be much more effective and efficient at satisfying constraints in terms of power and design speed, it would normally take a longer time to develop. Customized processors (with specialized coprocessors within the data path), on the other hand, are efficient and easy to develop. These customized processors can achieve greater performance enhancements via a dedicated, specialized instruction set in a shorter development time frame as compared to

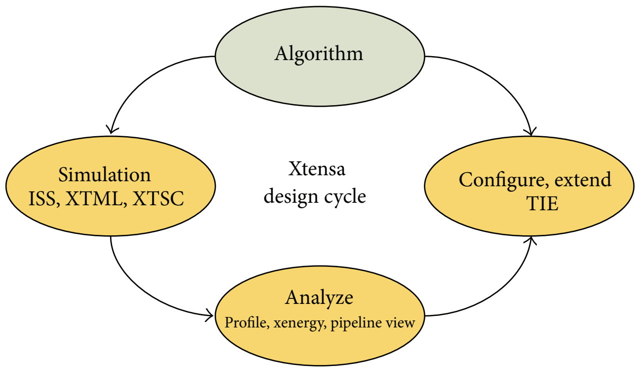


FIGURE 10: Tensilica's Eclipse based Xtensa Xplorer IDE tools.

pure RTL implementations. Furthermore, designers can often effectively fix the problematic bottlenecks of software by tweaking the processor configuration/specification. These design details include instruction set, processor pipeline depth, processor instruction/data bus width, cache memory size, register array size, and variable/constant register. As underlying tools mature with time and demand, the ASIP approach will become an increasingly attractive option for many products.

5.1. CAD Tools Used. The key development tools used in this work were based on the Xtensa Software Developers Toolkit [27]. The Xtensa Development Environment V4.0.4.2 was used to design, test, simulate, and enhance the new customized processor and its related software. Xtensa provides a set of tools that if used together facilitate an efficient, user-friendly, comprehensive, and integrated dataplane processor design and optimization environment that interfaces with both sides of the spectrum very well (both software and hardware).

The Tensilica Eclipse based Xtensa Xplorer IDE can generate a full custom processor system, as shown in Figure 10. The tools offer several means for optimizing the design. The first method is based on an Automated Processor Development (APD) approach, which profiles the application and identifies its weaknesses while generating a new configuration with improved performance. The second method assists the designer in defining a new data-path function (or function) and modifies the Xtensa dataplane processor (DPU) configuration. The tools provide a Simple Verilog-like Tensilica Instruction Extension (TIE) language for processor customization, whereby the designer can enforce the new instruction(s). The tools provided by the Xtensa Software Development framework offer a fully integrated flow for software compilation, profiling, simulation, debugging, and so forth. This allows a designer to test and modify software intended to run on the DPU. It incorporates powerful tools that are especially useful for ASIP development, such as the ability to profile software and report the gathered timing data in various graphs and tables.

5.2. Various Implementations. The following are the basic steps involved in designing a baseline processor along with optimized versions that improve the performance over the nonoptimized baseline implementation:

(i) profile the entire application,

(ii) identify bottlenecks of the algorithm,

(iii) combine existing instructions and create new functions using TIE language,

(iv) repeat, until the desired speed is achieved.

5.2.1. Baseline Implementation. The baseline implementation in this work will refer to a basic nonoptimized processor capable of executing the GBSA algorithm. The main goal of this baseline implementation is to allow the designer to compare and understand the effect of different optimization steps performed on performance. Before any optimization could be applied, the GBSA algorithm (coded in C++) had to be imported to Xtensa IDE and profiled using the ConnX D2 Reference Core (XRC-D2SA) processor. As the tools support open and read/write from/to file in the PC using the standard C-library, both the rule set and trace files were read directly from the host computer. With the environment for testing final setup, the next step was to run and profile the application. The results of the baseline implementation were extracted from the profiler, which reported the total number of clock cycles for the application. Also, the system generated the processor clock frequency depending on the design and the original processors architecture. The reported total time was calculated by multiplying the clock period by the total number of clock cycles.

5.2.2. Optimized 128-Bit Engine. As described earlier, the GBSA packet classification algorithm is based on two main blocks, preprocessing and classification, which both need to be optimized. The primary design philosophy dictates that small, simple, and frequently used operations (instructions) should be aggregated and moved to hardware. A typical new instruction is described using the TIE language and then compiled. Moreover the TIE compiler also updates the compiler tool chain (XCC compiler, assembler, debugger, and profiler) and the simulation models (instruction set simulator and the XTMP, XTSC, and pin-level XTSC system modeling environment) to fully utilize the new proposed instructions. The first obvious step to enhance the XRC-D2SA processor is to increase the data bus bandwidth from 32 bits to 128 bits. This modification could enable the processor to run both Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) classes of parallel computational models.

(i) Preprocessing engine: based on the discussion in Section 1 and contents of Table 1, when the IP address is converted to a range format, the length of the IP_{src} and IP_{dst} changes from 32 to 64 bits each. Therefore, the total length of the IP_{src} and IP_{dst} will be 128 bits. The example introduced in Section 3.2 (that explains the preprocessing stage within GBSA) clearly indicates that the original rule set (Step 1 in Figure 5) is decomposed into three partitions (Step 2

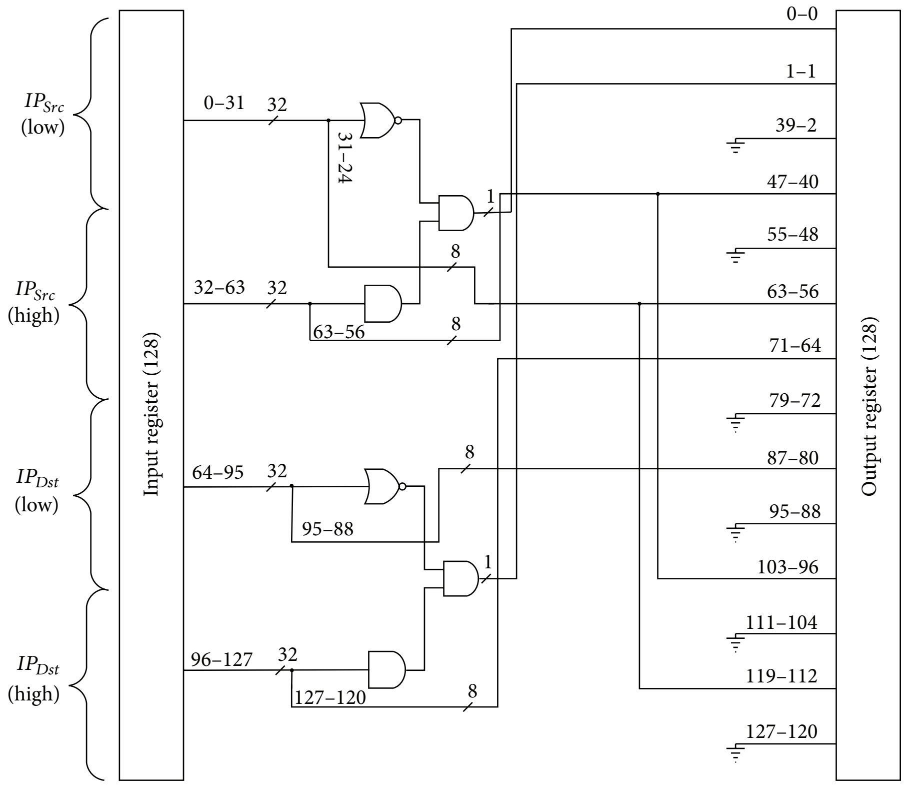


FIGURE 11: Preprocessing: partitioning rules based on their IP_{src}/IP_{dst}.

in Figure 5) based on the presence or absence of a wild-card.

The architecture proposed in Figure 11 performs the decomposition of the rule set into three partitions and also generates the necessary addresses (Step 3 in Figure 5). This module accordingly processes the high/low IP addresses (source/destination) of 128 bits and generates the following information:

(a) the first two bits generated are necessary to discriminate among the three produced partitions (*LookupIP*$_{Dst}$, *LookupIP*$_{Src}$, and *LookupIP*$_{Gen}$). The ASIP processor will utilize this information to access the information within the tables,

(b) the corresponding addresses of the original rules in the three newly created tables as demonstrated by Step 3 in Figure 5 will also be generated. These addresses are used to point to the contents of the memories (Mem 1, Mem 2, and Mem 3 in Figure 6).

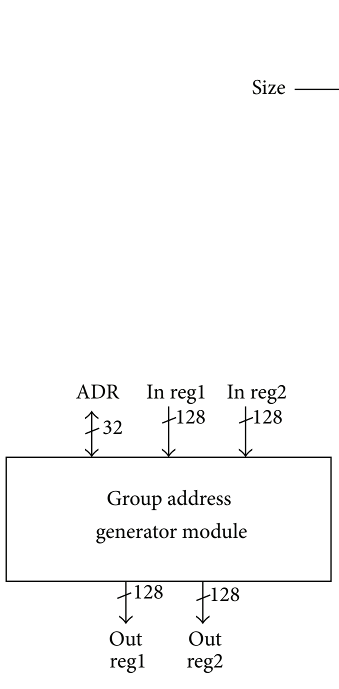
At this point, only the total number of rules (refered to as "Size") belonging to a cluster is generated by the architecture introduced earlier in Figure 11. The next step involves generating the starting address of each cluster within the RuleList (Mem 4 in Figure 6). Figure 12 demonstrates this process. The proposed architecture in Figure 12 attempts to process eight locations simultaneously. This operation emulates memory allocation of space in Mem 4.

(ii) Classification engine: next, we will describe the overall architecture of the ASIP classification engine. Our description is based on the classification example introduced earlier in Figure 7.

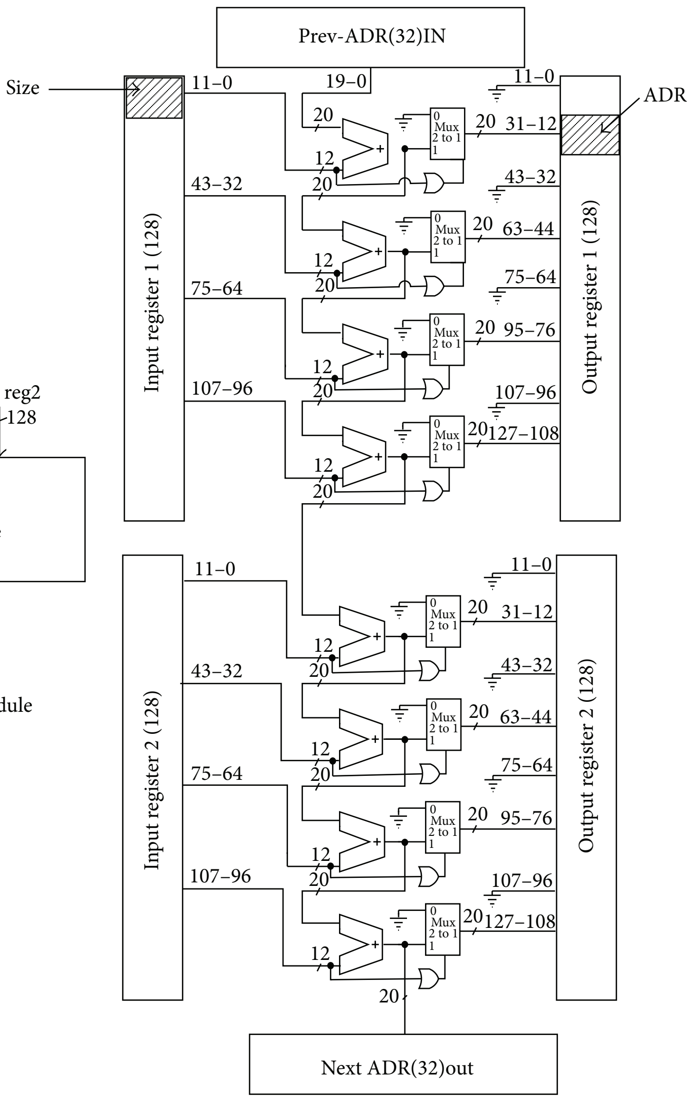
Step 1. Since the incoming packet is used frequently until a best-match rule is found, a better design strategy is to store the incoming packet in a temporary register file. This will make it more accessible to all instructions avoiding any extra cycles to fetch the packet.

Step 2. Figure 13 shows the second step in classification for the incoming packet. As the high byte for both IP source and destination will be used as an address to point to the three lookup tables, the custom design in Figure 13 uses the stored packet to generate an address for the corresponding table and reads the contents of the table. Three instances of this function are required for each lookup table (*LookupIP*$_{Gen}$, *LookupIP*$_{Src}$, and *LookupIP*$_{Dst}$). Each instance will be responsible for generating the "ADR" and "Size" fields that access the rules from Mem 4.

Step 3. Once a cluster has been identified correctly (as described earlier in the example of Figure 7 in Section 3.3), an efficient linear search within the cluster should be initiated. The search for the best-match rule inside the cluster involves comparing the five dimensions of the "incoming packet"



(a) Block of the module



(b) Internal architecture

FIGURE 12: Group address generator (128-bit preprocessing engine).

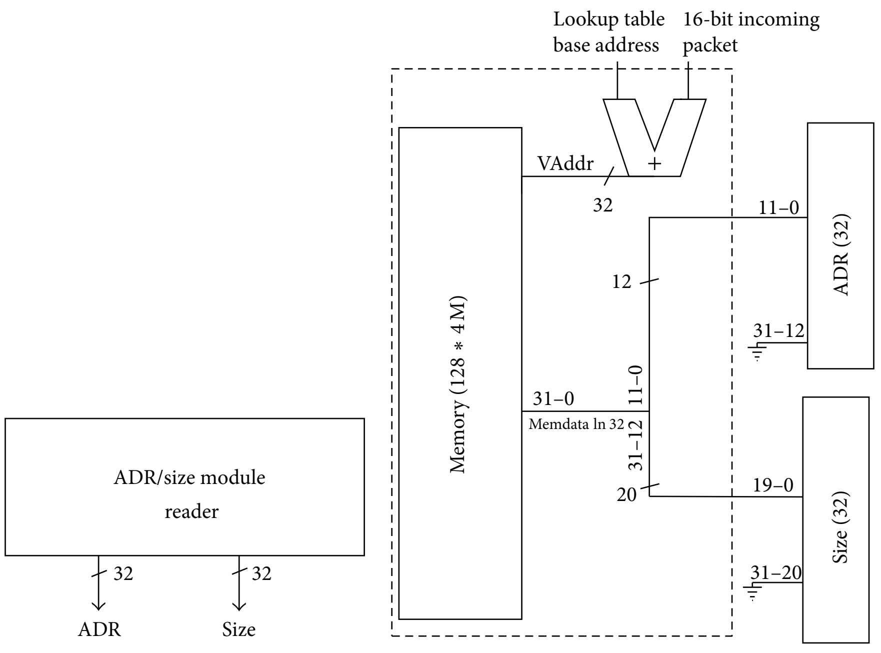
(which is already stored in a temporary register) with all rules belonging to the cluster. This process is computationally expensive. Therefore, a new instruction (along with an appropriate module) is developed to simultaneously match the two rules with the stored packet. Figure 14 illustrates how a match between the incoming packet with existing rules in the database will be performed. If a match exists, either "Result bit1" or "Result bit0" will be set. Figure 14(b) shows the I/O registers for the matching function.

5.2.3. Optimized 256-Bit Engine

(i) *Preprocessing Engine.* Since classification is considered to be the most important phase within packet classification, we concentrated our effort in improving the classification phase. Accordingly, the original optimized 128-bit preprocessing engine described earlier will be used in a 256-bit engine, as described in the following.

(ii) *Classification Engine.* The 128-bit classification engine described earlier (Figures 13 and 14) requires four clock cycles to perform memory reads and another four clock cycles to write to the internal registers. This overhead can be further reduced by increasing memory read bandwidth or increasing the data bus width. Recall that the classification is basically a matching operation between an incoming packet and rules in the database.

(a) As each rule needs 208 bits (IP_{src} = 64 bits, IP_{dst} = 64 bits, $Port_{src}$ = 32 bits, $Port_{dst}$ = 32 bits, and Protocol = 16 bits), extending the width of the bus from 128 to 256 bits (Figure 15) should



(a) Block of the module (b) Internal architecture

Figure 13: Group "address/size" access (128-bit classification engine).

enable the architecture to read an entire rule from the memory.

(b) The speed of the matching module could be further improved by forcing it to read one of the two rules directly from memory instead of reading the rule and then sending it to the matching module.

(c) Moreover, the number of matches per iteration of the loop can be increased to four instead of two as shown previously in the 128-bit classification engine. This enhancement can be accomplished by adding a second matching module. Figure 16 illustrates the block diagram of the entire matching step for four rules.

 (1) The first module compares and saves the result in a flag register, while the second module compares, tests the previous result from the flag register, and then generates the final result.

 (2) Figure 15 illustrates the internal architecture of the matching module. The matching function performs a direct memory read for one rule, while accessing a second rule from a temporary register. The first matching function generated two results, namely, "Flag1" and "Flag2". The second matching function will also match the incoming packet with another two rules. The four flags will be fed to a priority encoder to generate the winning rule as shown in Figure 16.

(d) Yet another improvement in the matching process can be achieved by simultaneously reading the addresses of four rules (belonging to a cluster) that will be matched to an incoming packet in each iteration. Figure 17 shows the internal design of the four addresses generated. These addresses are saved in four temporary registers that are accessible to any module inside the ASIP processor. Two of these addresses will be used directly by the two matching functions described earlier (Figure 16).

(e) The other two addresses are used in the new module (Figure 18) that is responsible for direct reading of rules from memory and storing the results in temporary register which is used by the matching function.

All the above enhancements, unfortunately, did not produce substantial speedup (as it will be shown shortly) compared to those based on the 128-bit engine, since the algorithm is highly memory dependent. The memory dependency introduced delays due to stalls in the pipeline.

5.3. Results. Figures 19 and 20 showcase the final results of the ASIP processor implementation of the GBSA before and after optimization for the 128-bit/256-bit engines. It is evident from Figure 19 that preprocessing received improvement in performance (3.5x speedup) after adding the new functions as new instructions to the newly designed processor. Also, expanding the data bus helps to increase the preprocessing speed.

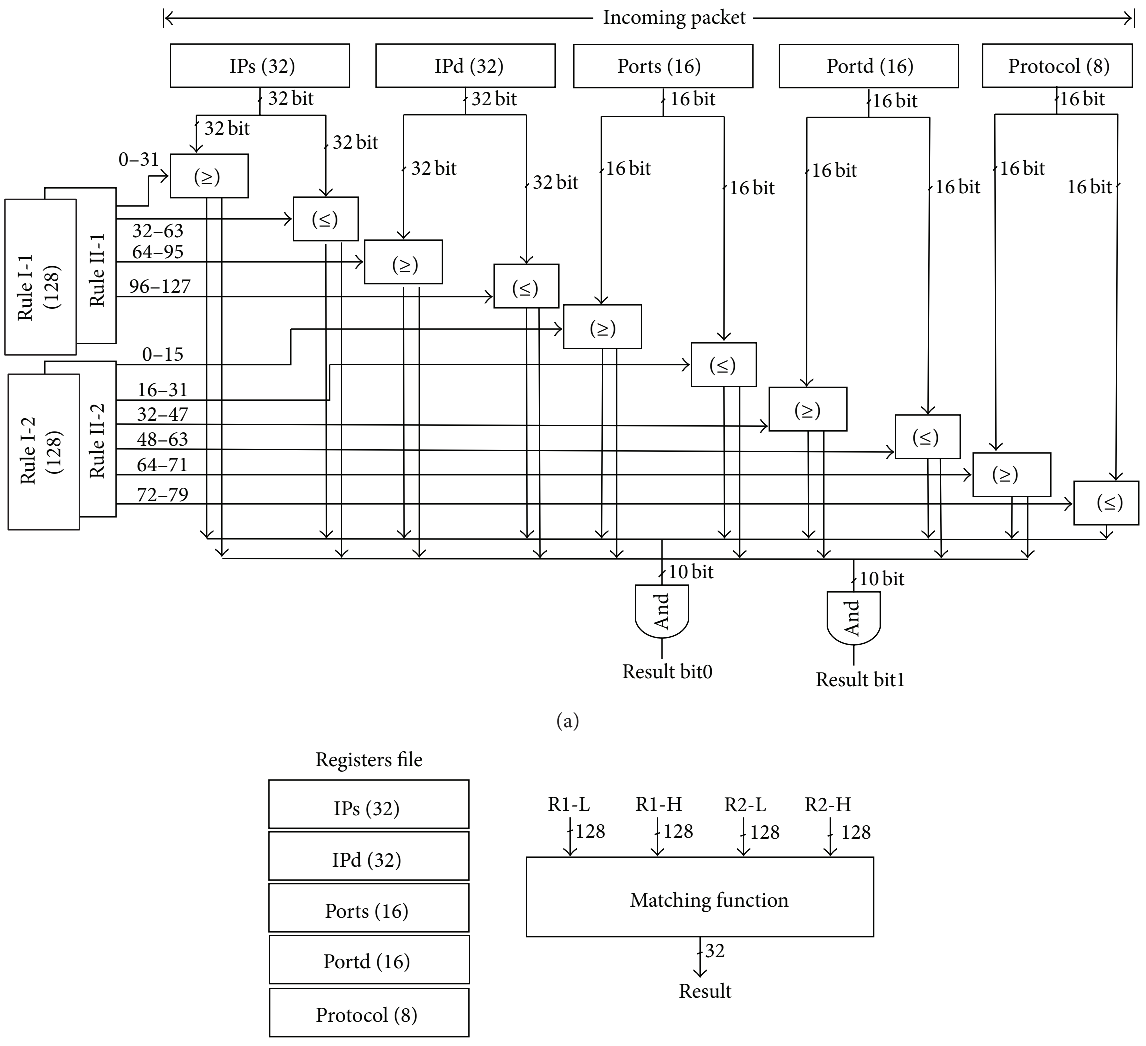


Figure 14: (a) Matching function of a packet simultaneously against two rules (128-bit classification engine) and (b) the external I/O of the matching function (128-bit classification engine).

The classification speed achieved using the optimized custom ASIP processors is on average 5.3x over the baseline. This performance boost can be entirely attributed to the following: (a) a new matching function, (b) the direct memory read, and (c) temporary values saved inside the register file acting like a local cache.

6. RTL Hardware Implementation Based on ESL

The design of a pure RTL based hardware accelerator using an Electronic System Level (ESL) language is a different kind of approach of hardware and software design philosophies. An ESL is typically a high-level language with many similarities to software based languages, such as C, in terms of syntax, program structure, flow of execution, and design methodology. The difference from such software languages comes in the form of constructs that are tailored to hardware development design, such as the ability to write code that is executed in parallel. This makes it convenient to translate a software application into its RTL equivalent without having to start the design from scratch. The higher level of abstraction afforded by ESLs allows designers to develop an RTL implementation faster than using pure VHDL or Verilog. However, the performance of the resulting hardware is often less than that which can be achieved using VHDL or Verilog where the designer has (almost) complete control over the architecture of the synthesized circuit. In addition, most C programmers should be able to create effective ESL hardware designs without much additional training at all. Instead of taking a long time to master VHDL and Verilog, one can take advantage of the long-standing and widespread foundation of C.

6.1. Handel-C Implementation (Tools). A hardware accelerator of the GBSA algorithm was implemented in RTL

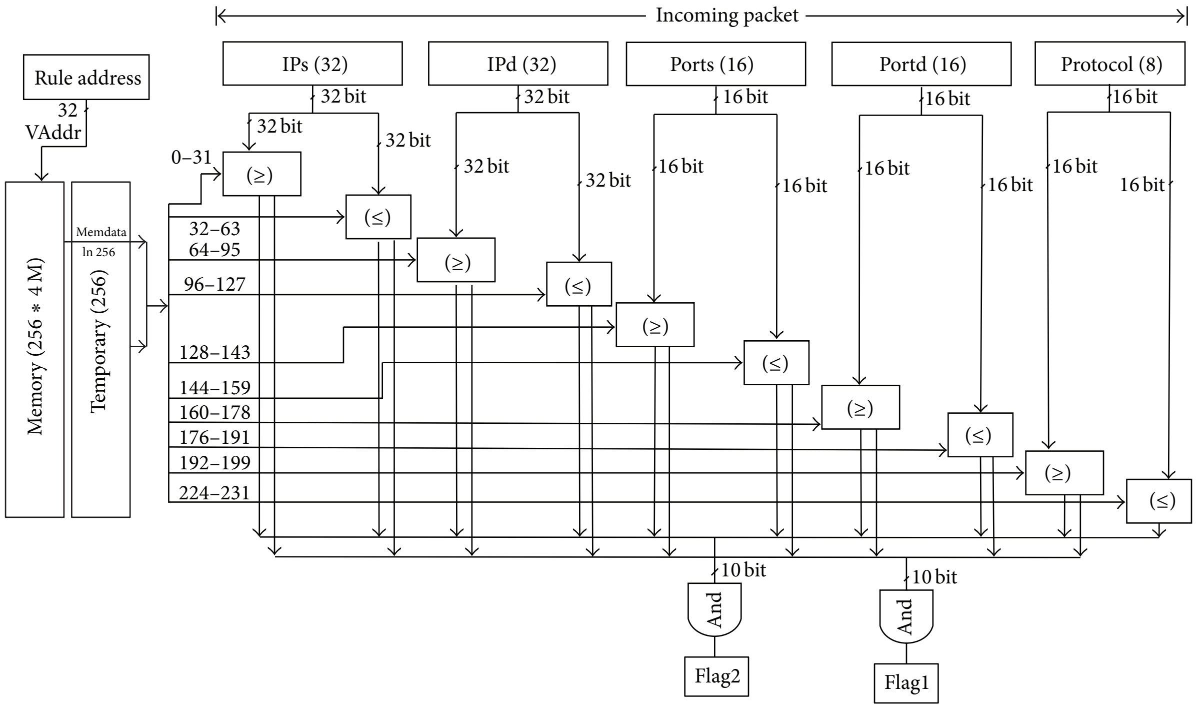


Figure 15: The internal architecture of the matching function (256-bit classification engine).

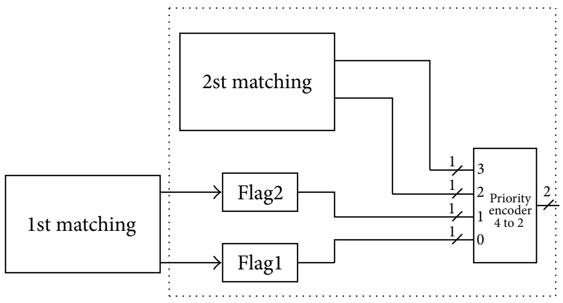


Figure 16: The block diagram of the overall matching step.

using Mentor Graphics Handel-C [29]. The Mentor Graphics Development Kit Design Suite v5.2.6266.11181 facilitated a thorough debugging toolset, the creation and management of several output configurations, the ability to build simulation executables, hardware-mappable EDIFs, or VHDL/Verilog files from the Handel-C source files, file I/O during simulation, and the ability to include custom-built scripts for specific target platforms. When building an EDIF, Handel-C will also produce log files that display timing, size, and resources used for the design. In our experiments, Xilinx ISE v12 was used to map the generated HDL files from Handel-C to Virtex-6 (xc6vlx760) FPGA chip. The synthesis and place and route reports were both used to identify the critical-path delay or the maximum running frequency.

6.1.1. Techniques and Strategies. Handel-C provides a file I/O interface during simulation, which is used to feed test bench files. For this implementation, preprocessing of all test benches was performed by a general purpose processor (GPP), well in advance, using a pre-processor application written in C. Accordingly, the main focus of the RTL design was solely targeting classification. Handel-C also provides unique memory structures in RAMs and ROMs. These modules have fast access times, but the downside is that multiple branches within the designed classification engine cannot access RAM concurrently. A side effect of this is that as parallelism is exploited (and there is quite a bit of this in a highly optimized RTL design), and special care must be taken to ensure that only a single branch is accessing RAMs.

Handel-C debugging mode allows for examining the number of cycles required by each statement during simulation. This is a good metric, when coupled with the critical-path delay for the design, for gaining an overall sense of the timing. Both the critical-path delay and the cycle count also serve as metrics for improvement when it comes to optimization in any Handel-C design.

6.1.2. Handel-C Optimization. Optimization in Handel-C is quite different from an ASIP based approach. The technique of extracting hardware from blocks of code serves no purpose for the designer since Handel-C implementations produce RTL code (VHDL). Instead, the key method of optimization is to exploit the following properties from a software design: a single statement takes a clock cycle; the most complicated statement dictates the critical-path delay and, therefore, the clock frequency; statements can be executed either sequentially or in parallel branches; and special memory structures can be used for fast memory access.

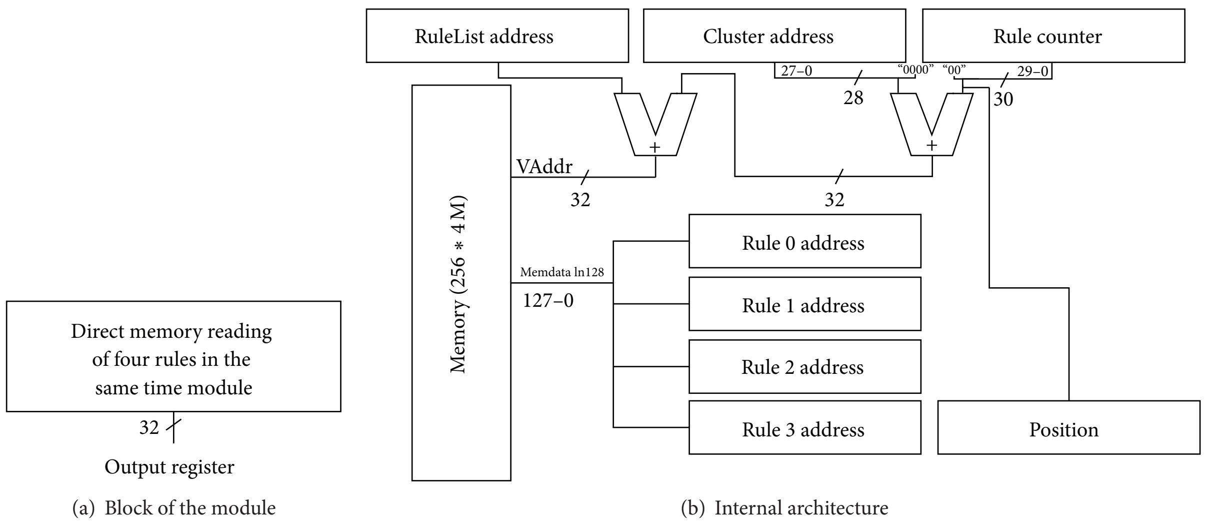


Figure 17: Simultaneous direct memory read for four rule addresses.

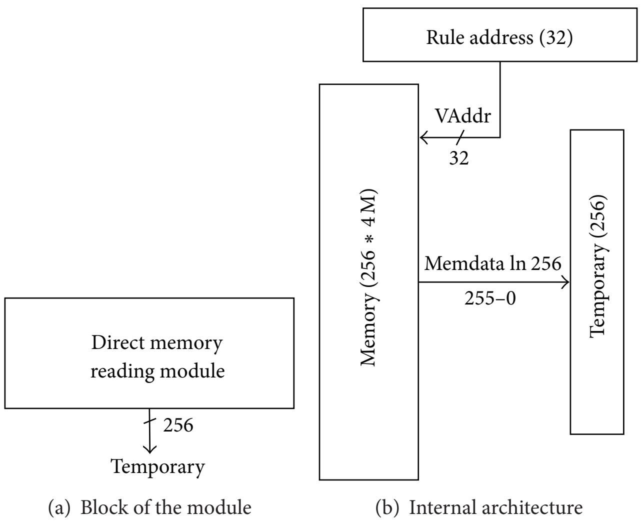


Figure 18: A direct memory read for an entire rule.

The following steps will explain the different optimization phases taken when mapping the GBSA algorithm into hardware.

(1) *Baseline Design (Original).* The GBSA original code was imported to the Handel-C development environment. The code is adapted for RTL design. Memories were defined instead of arrays, a general clock was declared for the entire system, all dynamic variables were replaced by a static declaration, and channels were introduced for external data exchange. The remaining C-code was kept unchanged with minor modifications. The Baseline Design will be used as a baseline reference model that can be compared to optimized versions of the algorithm.

(2) *Fine Grain Optimization (FGO).* The first step taken to improve the original Handel-C implementation of the GBSA was to replace all "For" loops within the design to "do while" loops. "For" loops take a single cycle to initialize the counter variable, a cycle for each sequential statement within the loop body, and a cycle to increment the counter variable. "Do while loops" are much more efficient in terms of cycles consumed since one can place the counter increment within the loop body and run it concurrently with the other statements. Algorithm 2 illustrates the code conversion of the "For" to the "Do While" loops using the Handel-C Par statement. The "For" loop in the code example of Algorithm 2 consumes 21 clock cycles, yet the same code, when implemented in "Do while" loops, consumes 11 clock cycles. The second important FGO step is to combine several lines of code together and parallelize them. Algorithm 3 shows the code conversion from sequential to parallel

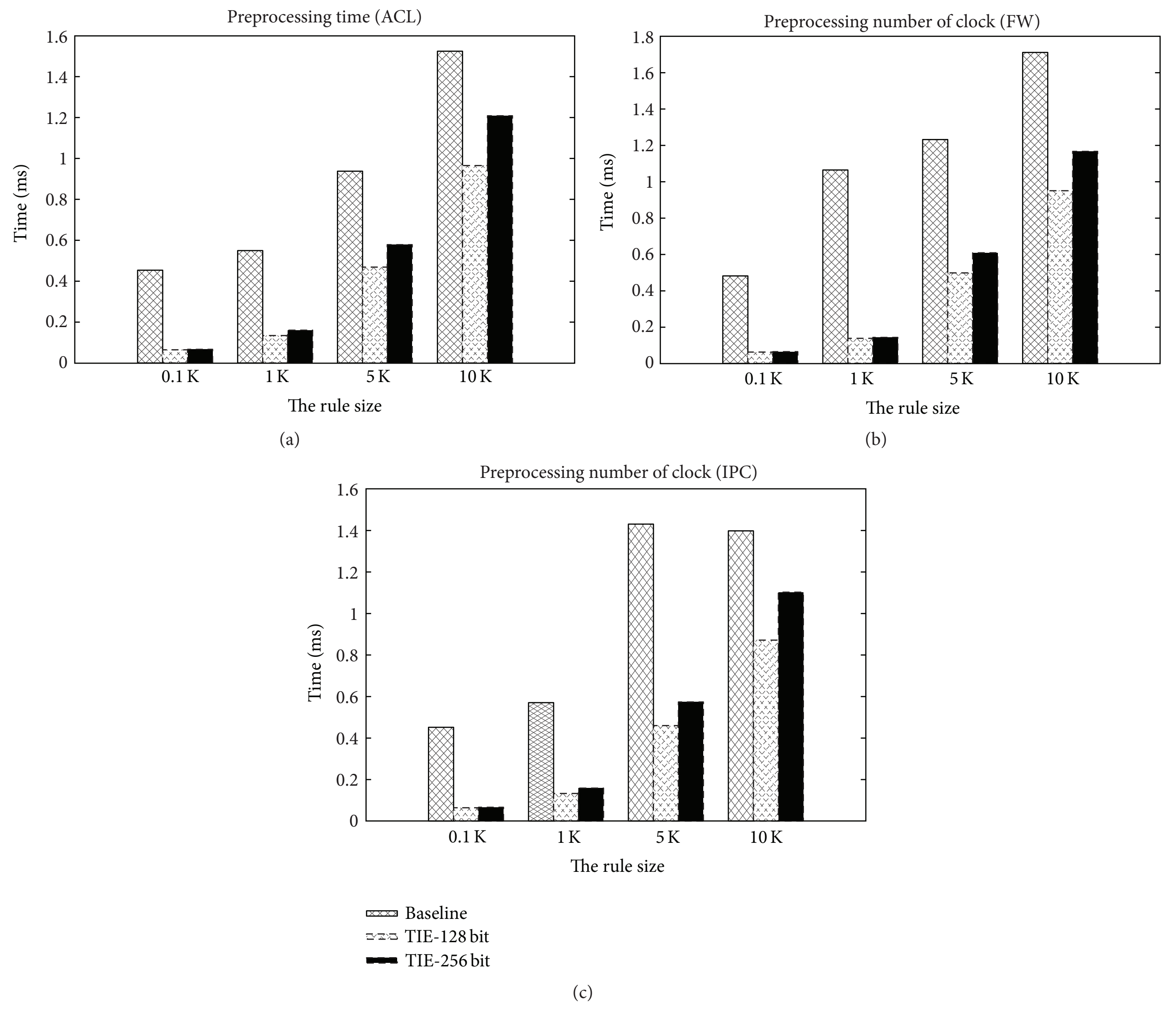


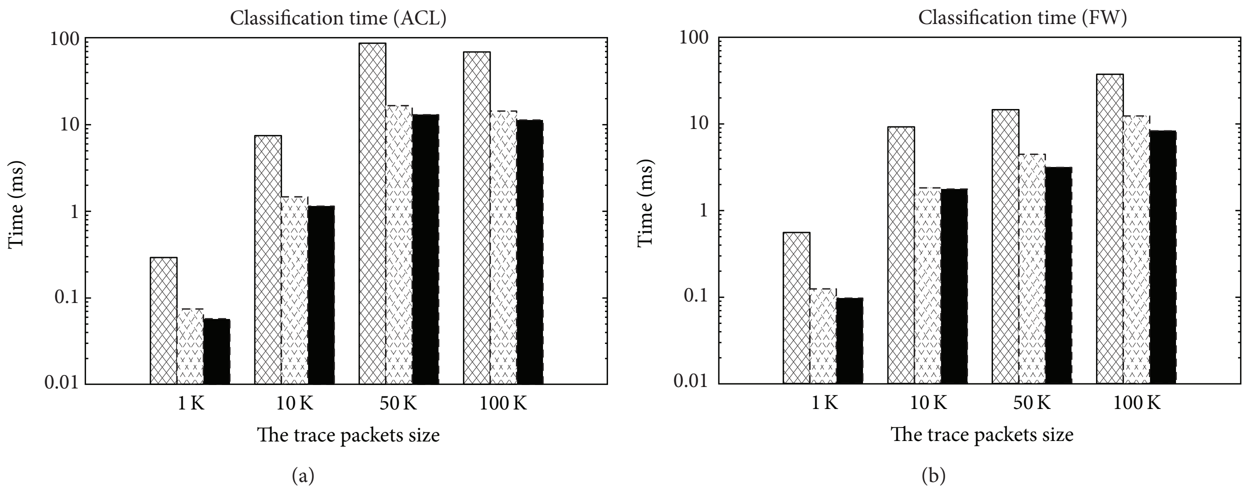
Figure 19: GBSA's ASIP preprocessing implementation: timing result.

statements. The original sequential code requires five clock cycles, yet the parallel code needs only a single clock cycle.

(3) *Coarse Grain Optimization (CGO).* One of the key methods of improving performance of the GBSA algorithm is to divide the classification phase into several stages and pipeline the trace packets through them. This is a rather simple task, as long as blocks that are required to be sequential, such as "For" loops, are kept intact. The key construct required to build an efficient, effective pipeline scheme is the channel. In Handel-C, channels are utilized to transfer data between parallel branches, and this is required to ensure that data processed in one stage of a pipeline is ready for use in the next stage. Channels also provide a unique type of synchronization: if a stage in the GBSA is waiting to read a value from the end of a channel, it will block its execution until the data is received. This not only makes the pipeline well synchronized, but it also improves efficiency in terms of the number of cycles used to complete classification. Figure 21 illustrates a timing diagram for the pipelined implementation of the GBSA. Along the y-axis there exists a pipeline stage. Stage 3 in particular is divided into three substages due to the following reasons:

(i) the GBSA algorithm searches in three different tables, namely, $LookupIP_{Gen}$, $LookupIP_{Dest}$, and $LookupIP_{Src}$, sequentially,

(ii) the complexity of the internals merits the division and the substages need to be executed sequentially to produce accurate results.



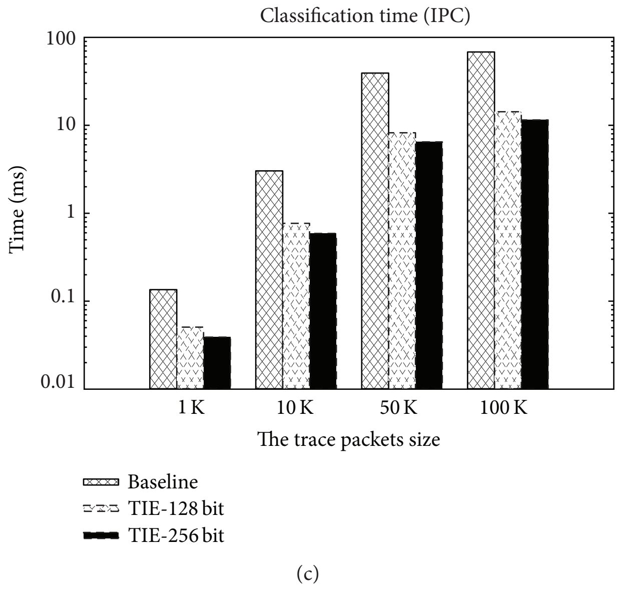


Figure 20: GBSA's ASIP classification implementation: timing result.

```
   /∗ For Code∗/                 /∗ Do While Code∗/
for(i=0;i < 10;i++)              i=0;
      {                          do{
      MyRam[i]=i;                   par{
      }                                MyRam[i]=i;
                                       i++; }
                                  } while(i < 10);
```

Algorithm 2: A FOR to Do-While conversion code.

The "Coarse Grain Optimization" stage is a combination of the "Fine Grain" and "Pipelining" techniques described above.

(4) *Parallel Coarse Grain Optimization (PCGO).* The final strategy employed to improve the GBSA was to divide the memory space of the "*LookupIP$_{Gen}$*" into odd and even banks and then split the algorithm itself into a four-parallel-pipelined searching engine, as illustrated in Figure 22. Accordingly, the GBSA's preprocessing stage was altered to generate four input rule files, and the RAM was also partitioned into four segments. While one would expect this to maintain the same memory size as the normal pipelined model,

```
  /∗ Ungroup Code∗/                          /∗ Group Par Code∗/
                                           par{
Trace_IPs = packet\\ 72;                     Trace_IPs = packet\\ 72;
Trace_IPd = (packet\\40)< −32;               Trace_IPd = (packet\\40)< −32;
Port_Src = (packet\\24)< −16;                Port_Src = (packet\\24)< −16;
Port_Des = (packet\\8) < −16;                Port_Des =(packet\\8) < −16;
Protocol = packet< −8                        Protocol = packet< −8
                                           }
```

Algorithm 3: A group code parallelism.

additional resources had to be used in order to account for additional channels, counter variables, internal variables for each pipeline, and interconnecting signals for each pipeline. The design diverges into four pipelines once a trace packet has been read in. Each pipeline runs the packet through and attempts to classify it. The pipelines meet at a common stage, and at each cycle the algorithm checks if a match has been found in any of the four memory spaces. A priority scheme allows only a single pipeline to write a result at any given time. The IP source/IP destination odd search engine has the highest priority to terminate the search and declare a match found; on the other hand, the IP destination search engine has the lowest priority and cannot terminate the search until all other search engines complete their task. As memory is further divided, the total amount of resources must be increased for accuracy, but the gains in terms of speed far outweigh the additional resources used. In the next subsection, a more detailed explanation of the specific trade-offs and trends is presented.

6.1.3. Results. Figures 21 and 22 illustrate the execution models for both the basic pipelined implementation, "Coarse Grain Optimization (CGO)" of the GBSA, and the pipelined implementation with a divided memory space that is, the "Parallel Coarse Grain Optimization (PCGO)." It is clear from the architectures that both models have a best-case time of 1 cycle per packet (which is constant). The "Coarse Grain" pipelined version has a worst-case time of the sum of the maximum group size in the three tables cycles, while "Parallel Coarse Grain" has a worst-case time of the maximum group size in one of the three tables cycles. Both designs are incapable of processing one packet per cycle with large rule sets, but continued memory division should lower the worst-case time in future endeavors. Figure 23 shows the result of all four implementations of GBSA. Since the Pipeline itself has unbalanced stages, pipelining alone did not generate a substantial boost in speed. The largest contributors to speedup were the "Fine Grain" and the "Parallel Coarse Grain" approaches. Converting all "For" loops into "While" loops and grouping all the independent steps to one group "Fine Grain" resulted in an average acceleration of 1.52x for the 10 K rule set size. The "Coarse Grain," on the other hand, contributed only an additional 1.05x speedup over the Fine Grain version and, overall, a 1.6x speedup from the baseline Handel-C implementation. On the other hand, the "Parallel Coarse Grain" approach achieves a 1.67x acceleration over the "Coarse Grain" revision and, overall, a 2.6x speedup over the baseline Handel-C implementation.

Resource usage in terms of equivalent NAND gates, Flip Flops, and memory are presented in Table 4.

It is clear from Table 4 that the "Parallel Coarse Grain" implementation with four classifiers consumes almost 1.03 times more NAND gates than the "Coarse Grain" architecture. The same number of memory bits is used by the two designs, and an additional 1.11 times more Flip Flops were required for the "Parallel Coarse Grain" approach. It is also interesting to note that the increase in resource consumption of Flip Flops (comparing Baseline to Fine-Grained implementation) tends to be larger than the increase in the number of NAND gates used for both implementations. The main difference between the Baseline implementation and the Fine-Grained implementation is that the latter required the saving of some temporary results to speedup the classification phase, and therefore it is quite natural that the increase was more dramatic in the number of Flip Flops versus the logic used.

6.2. A Hardware Accelerator Using Impulse-C. Impulse-C is yet another ESL flow that supports the development of highly parallel, mixed hardware/software algorithms and applications. It extends ANSI C using C-compatible predefined library functions with support for communicating process parallel programming models. These extensions of the C language are minimal with respect to new data types and predefined function calls; this allows multiple parallel program segments to be described, interconnected, and synchronized [30].

6.2.1. Tools and Equipment. The Impulse CoDeveloper Application Manager Xilinx Edition Version 3.70.a.10 was used to implement the GBSA algorithm. The Impulse CoDeveloper is an advanced software tool that enables high-performance applications on FPGA based programmable platforms. In addition to the ability to convert a C based algorithm to HDL, CoDeveloper Application Manager also provides the CoValidator tools for generating all necessary test bench files. These testing files can be run directly under the ModelSim hardware simulation environment. The CoValidator provides

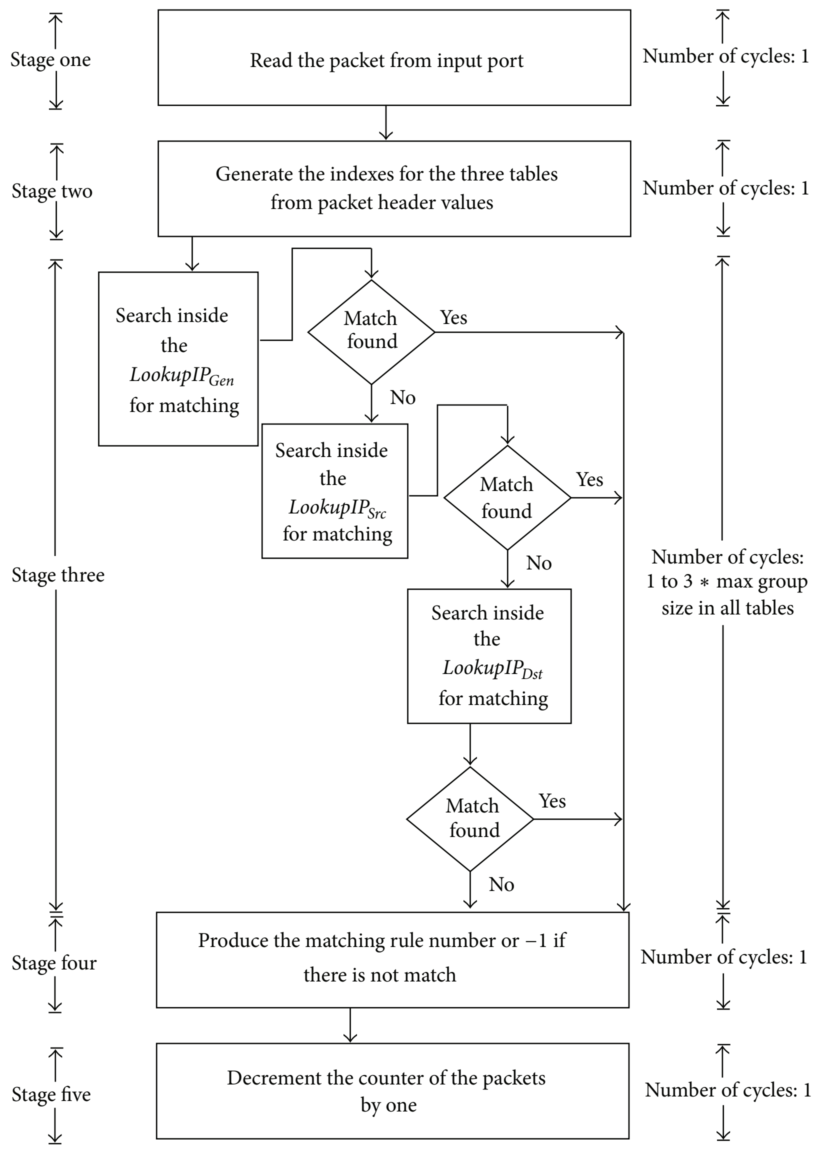

Stage one
Read the packet from input port
Number of cycles: 1
Stage two
Generate the indexes for the three tables from packet header values
Number of cycles: 1
Search inside the $LookupIP_{Gen}$ for matching
Match found
Yes
No
Search inside the $LookupIP_{Src}$ for matching
Match found
Yes
No
Stage three
Number of cycles: 1 to 3 ∗ max group size in all tables
Search inside the $LookupIP_{Dst}$ for matching
Match found
Yes
No
Stage four
Produce the matching rule number or −1 if there is not match
Number of cycles: 1
Stage five
Decrement the counter of the packets by one
Number of cycles: 1

Figure 21: Coarse Grain Optimization of the Handel-C (Pipelining).

simulation and HDL generation for design test and verification.

6.2.2. Techniques and Strategies. All of the test benchmark files were preprocessed on a regular PC in advance using a preprocessor application written in C. Only the classification part of the GBSA is considered and mapped to the FPGA, due to its high importance and frequency of use. (Mapping preprocessing as a hardware accelerator on-chip would waste resources, since it is a one-time task.) Figure 24 depicts the overall GBSA Impulse-C system organization.

The CoDeveloper tool is used to feed the system with the preprocessed rule set and test packets. The Impulse-C development tools generate all necessary files to synthesize and simulate the project, using ModelSim. ModelSim allows one to examine the required number of clock cycles to classify the test packets during simulation.

A quick method for examining design performance is to employ the Stage Master Explorer tool, a graphical tool used to analyze the GBSA Impulse-C implementation. It determines, on a process-by-process basis, how effectively the compiler parallelized the C language statements. An estimation of the critical-path delay for the design associated with C code and the number of clock cycles is used to get an overall judgment of the system timing and performance during the implementation phase. Both the critical-path delay and the cycle count play an important role in optimizing the Impulse-C design.

In our experiments, Xilinx ISE v12 was used to map the generated HDL files to the Virtex-6 (xc6vlx760) FPGA chip. The synthesis and place and route reports were both utilized to identify the critical-path delay. ModelSim SE 6.6 was also employed to simulate and accurately count the number of clock cycles needed for each benchmark.

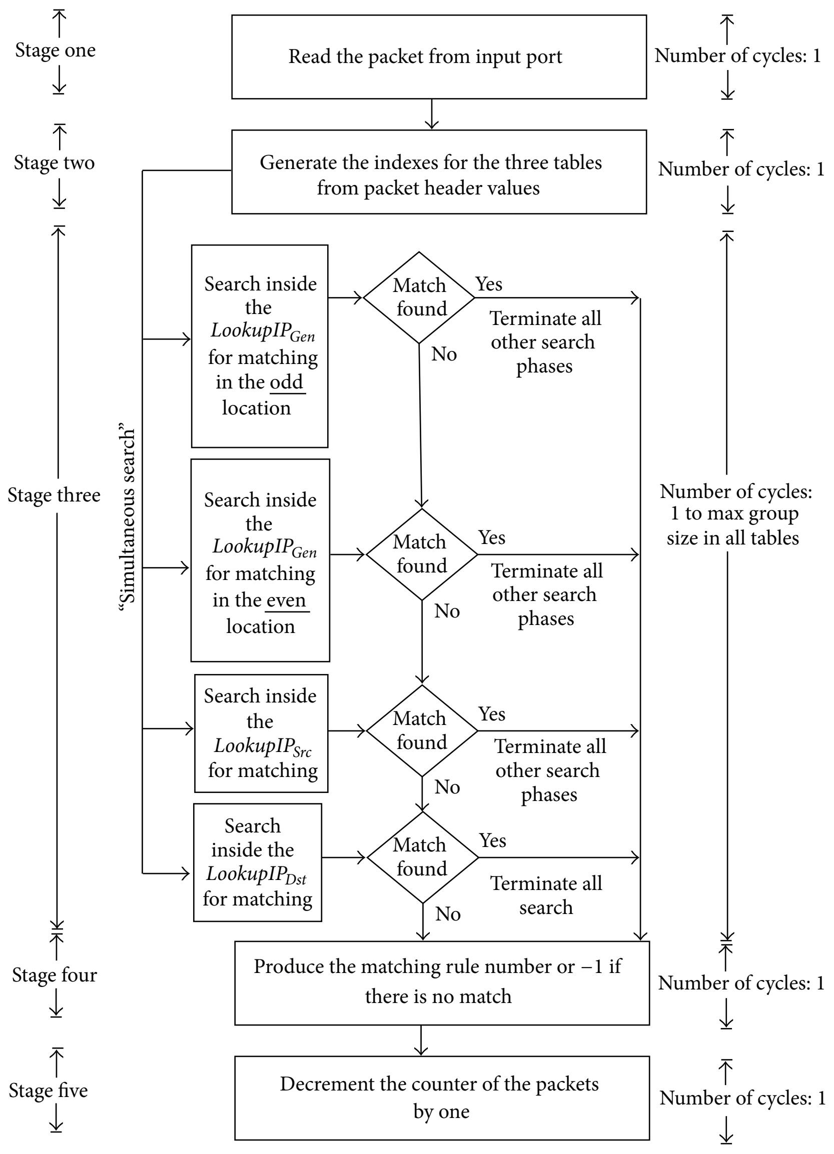

Stage one
Read the packet from input port
Number of cycles: 1
Stage two
Generate the indexes for the three tables from packet header values
Number of cycles: 1
"Simultaneous search"
Search inside the $LookupIP_{Gen}$ for matching in the odd location
Match found
Yes
No
Terminate all other search phases
Stage three
Search inside the $LookupIP_{Gen}$ for matching in the even location
Number of cycles: 1 to max group size in all tables
Search inside the $LookupIP_{Src}$ for matching
Search inside the $LookupIP_{Dst}$ for matching
Terminate all search
Stage four
Produce the matching rule number or −1 if there is no match
Number of cycles: 1
Stage five
Decrement the counter of the packets by one
Number of cycles: 1

Figure 22: Parallel Coarse Grain Optimization of the Handel-C implementation.

Figure 25 illustrates the main blocks of the GBSA Impulse-C implementation, which include IndexGen, Search and Match, and WinGen.

6.2.3. Impulse-C Optimization. Although both Impulse-C and Handel-C are Electronic System Level design based languages, their implementation, optimization, and objectives are quite different. Handel-C is more of a statement level optimization oriented language, while Impulse-C is more system and streaming level optimization oriented. Moreover, Impulse-C is not built on the assumption of a single clock per statement language, as it is the case with Handel-C, yet it deals with more block based optimization.

Impulse-C provides three main pragma(s) (PIPELINE, UNROLL, and FLATTEN) to improve the performance of the block of code and loops. Using the Stage Master tool in addition to the optimization pragma provides an efficient method to improve the target design. Also, Impulse-C designs take advantage of dual port RAM optimization, which enables the design to simultaneously access multiple locations, thus reducing the total number of clock cycles.

In general, optimization applied can be classified into (a) Fine Grain and (b) Coarse Grain, as described in the following.

(1) *Baseline Implementation (Original).* The original GBSA C-code was mapped to the CoDeveloper with the following minor modifications:

(a) adding the stream reading/writing from/to the buses,
(b) changing the variables to fixed sizes,
(c) collapsing all functions to one main function,
(d) converting all dynamic arrays to local static arrays.

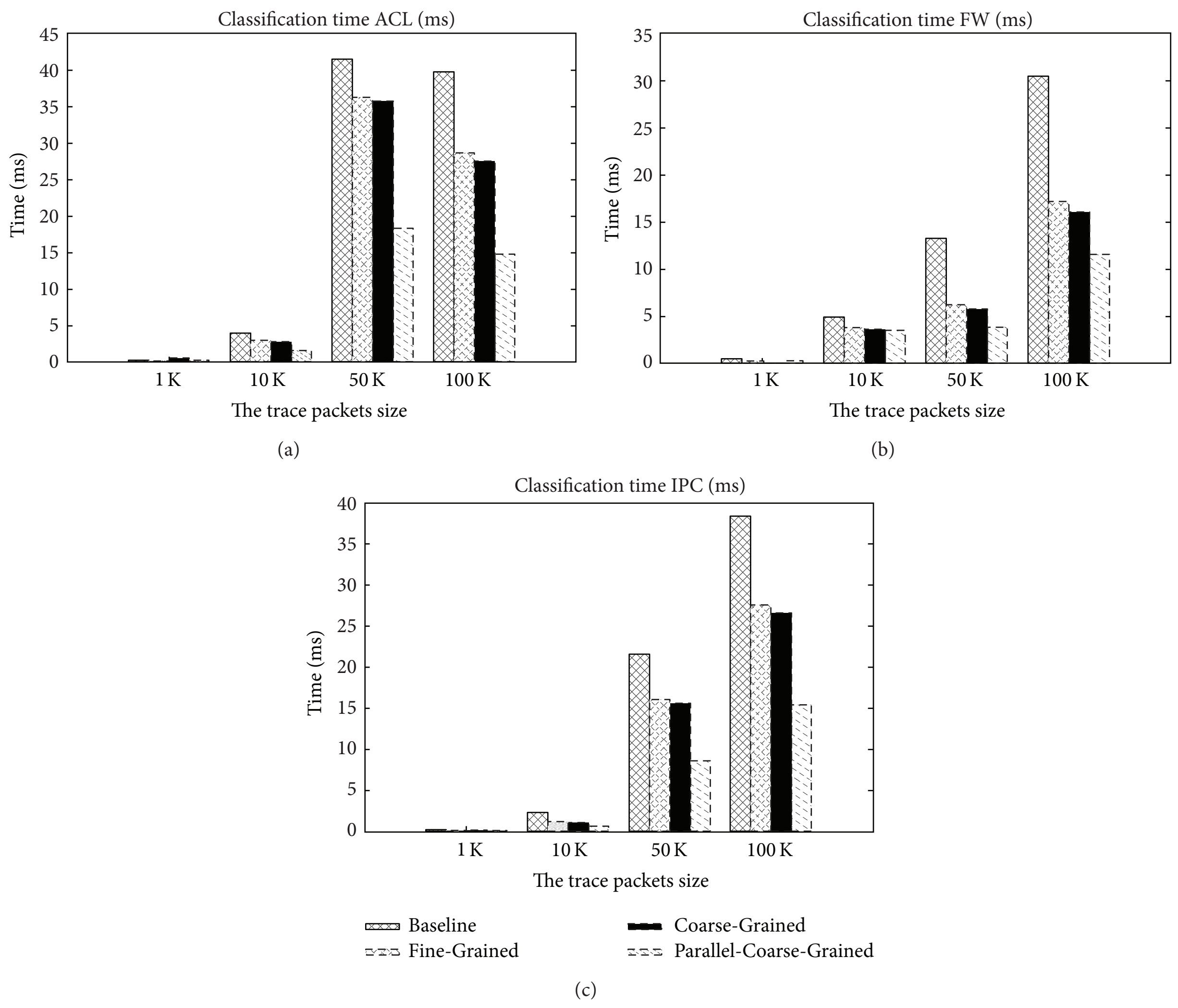


Figure 23: GBSA's Handel-C implementation: classification timing result.

Table 4: Frequency and device utilization of the GBSA based on Handel-C.

Specification	NANDs	FFs	Memory bits	Maximum frequency
Baseline	330657567	702	165313536	141.997 MHz
Fine-Grained	330665160	863	165313536	142.013 MHz
Coarse-Grained	330667259	1090	165313536	153.614 MHz
Parallel-Coarse-Grained	340667259	1207	165313536	157.914 MHz

The baseline implementation is used as a baseline reference model for the sake of comparison with optimized implementations.

(2) *Fine Grain Optimization (FGO).* The initial step to improve the original Impulse-C implementation of the GBSA was based on applying *PIPELINE* and *FLATTEN* pragmas, individually, to all the inner loops of the baseline implementation. This approach tends to convert the generated HDL block from sequential to either pipelined or parallel block. The selection between *PIPELINE* and *FLATTEN* pragmas can be performed via the Stage Master exploration tool. The main optimization techniques applied in FGO are as follows.

(i) FGO-A: the first optimization step involves converting the "For" loops to "Do While" loops, similar to Handel-C, with the exception of replacing the *par* statement with *FLATTEN* as depicted in Algorithm 4. Although the Impulse-C compiler converts different types of loops to "do while" loops when it generates the HDL file

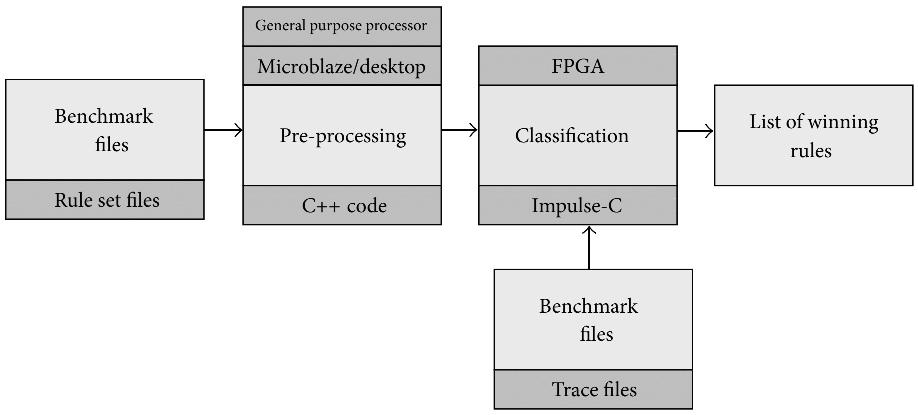


Figure 24: An overview of Impulse-C system (preprocessing and classification).

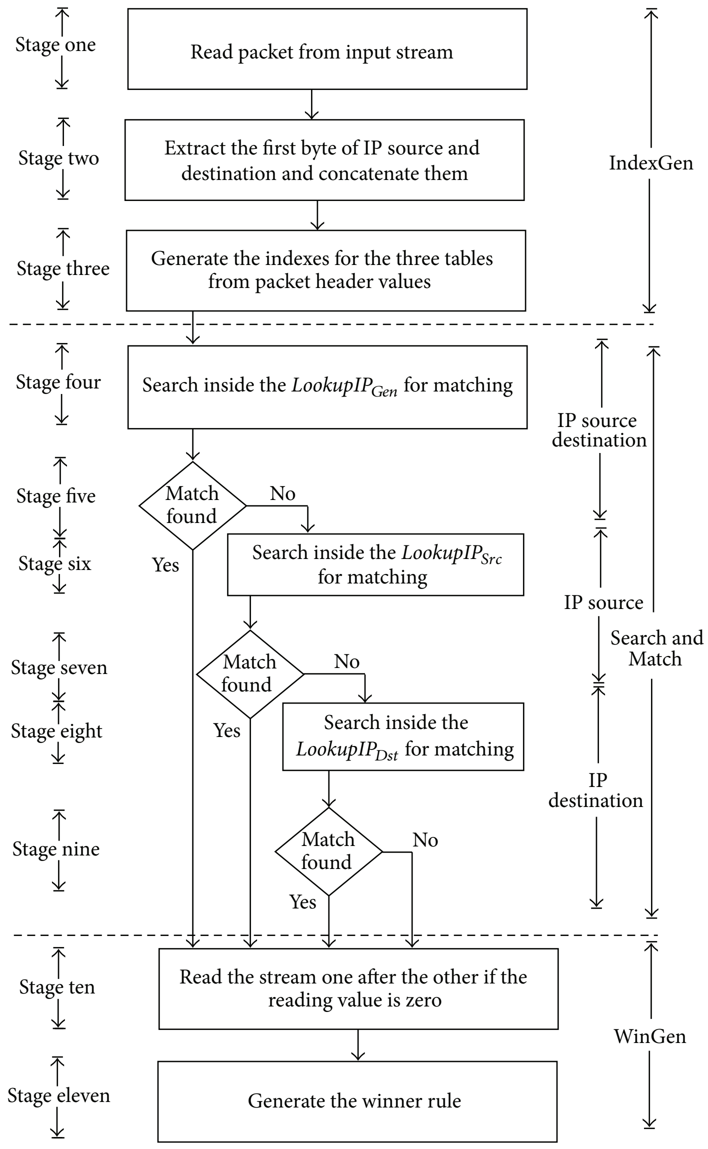

Stage one
Read packet from input stream
Stage two
Extract the first byte of IP source and destination and concatenate them
Stage three
Generate the indexes for the three tables from packet header values
IndexGen
Stage four
Search inside the $LookupIP_{Gen}$ for matching
IP source destination
Stage five
Match found
No
Yes
Stage six
Search inside the $LookupIP_{Src}$ for matching
IP source
Search and Match
Stage seven
Match found
No
Yes
Stage eight
Search inside the $LookupIP_{Dst}$ for matching
IP destination
Stage nine
Match found
No
Yes
Stage ten
Read the stream one after the other if the reading value is zero
WinGen
Stage eleven
Generate the winner rule

Figure 25: The GBSA Impulse-C classification stages.

```
                          /∗ Do While Code∗/
    /∗ For Code∗/         i=0;
for(i=0;i < 10;i++)       while(1){
{                             ♯ pragma CO FLATTEN
    MyRam[i]=i;                   MyRam[i]=i;
}                                 i++;
                                  if(i < 10)break;
                          }
```

Algorithm 4: FGO-A: a loop optimization using *FLATTEN* pragma.

```
i = 0;                                          i = 0;
Sum = 0;                                        Sum = 0;
while(i<=10)                                    while(i<=10)
{                                               {
    ♯ pragma CO FLATTEN                             ♯ pragma CO FLATTEN
    /∗ Adding one location from memory ∗/           /∗ Adding two locations from memory ∗/
    Sum = Sum + MyRam[i];                           Sum = Sum + MyRam[i]+ MyRam[i | 1];
    i = i + 1;                                      i = i + 2;
}                                               }
```

Algorithm 5: FGO-B: dual port RAM code optimization.

code, adding the *FLATTEN* pragma tends to improve the loop execution time.

(ii) FGO-B: the second optimization step takes advantage of the distributed dual port RAM so that multiple locations can be accessed simultaneously. Algorithm 5 illustrates the usage of dual port RAM in the code. Applying the dual port RAM optimization techniques tends to reduce the needed clock cycles sharply; however, it tends to increase the critical-path.

(iii) FGO-C: a group of instructions that run sequentially can be aggregated together where no dependency or partial results are required. Adding curly brackets to the set of instructions can combine them. Also, a *FLATTEN* pragma can be added to make the system generate hardware that runs in parallel.

(iv) FGO-D: the final optimization step performed is the adaptation of streams' width and depth which can have a huge influence on system performance. This effect is prominent when the stream is reading or writing inside the pipeline loops. In this case, the depth of the stream has to be at least one more than the pipeline stages number; otherwise the stream will become a system bottleneck. The stream width affects the reading of the date size per cycle, which subsequently affects the total number of clock cycles that is needed.

(3) *Coarse Grain Optimization (CGO).* The main goal of CGO is to divide the entire architecture into several modules that should operate simultaneously as seen in Figure 26. The system is divided into five independent blocks (processes) which operate in parallel and communicate through streams. The five submodules, as seen in Figure 26, are the IndexGen, SrcDest Search and Match, Source Search and Match, Destination Search and Match, and WinGen.

(i) The IndexGen subsystem generates the address of the three memories (as shown previously in Figure 6). The three addresses combine into a first stream (namely, ADR1) of width 48 bits. Also the three sizes of the groups combine together into a second stream (namely, Size1) of width 48 bits.

(ii) The SrcDest Search and Match reads the two generated streams (ADR1, Size1) and writes the remaining address and sizes to another two streams, namely, ADR2, Size2 of width 32 bits.

(iii) The Source Search and Match reads ADR2, Size2 streams and writes the remaining address and sizes to yet another two streams, namely ADR3 and Size3 of width 16 bits.

(iv) The Destination Search and Match reads ADR3, Size3 streams.

The designed Search and Match modules will operate simultaneously to generate the candidate winning rule number (if a match exists) via three streams

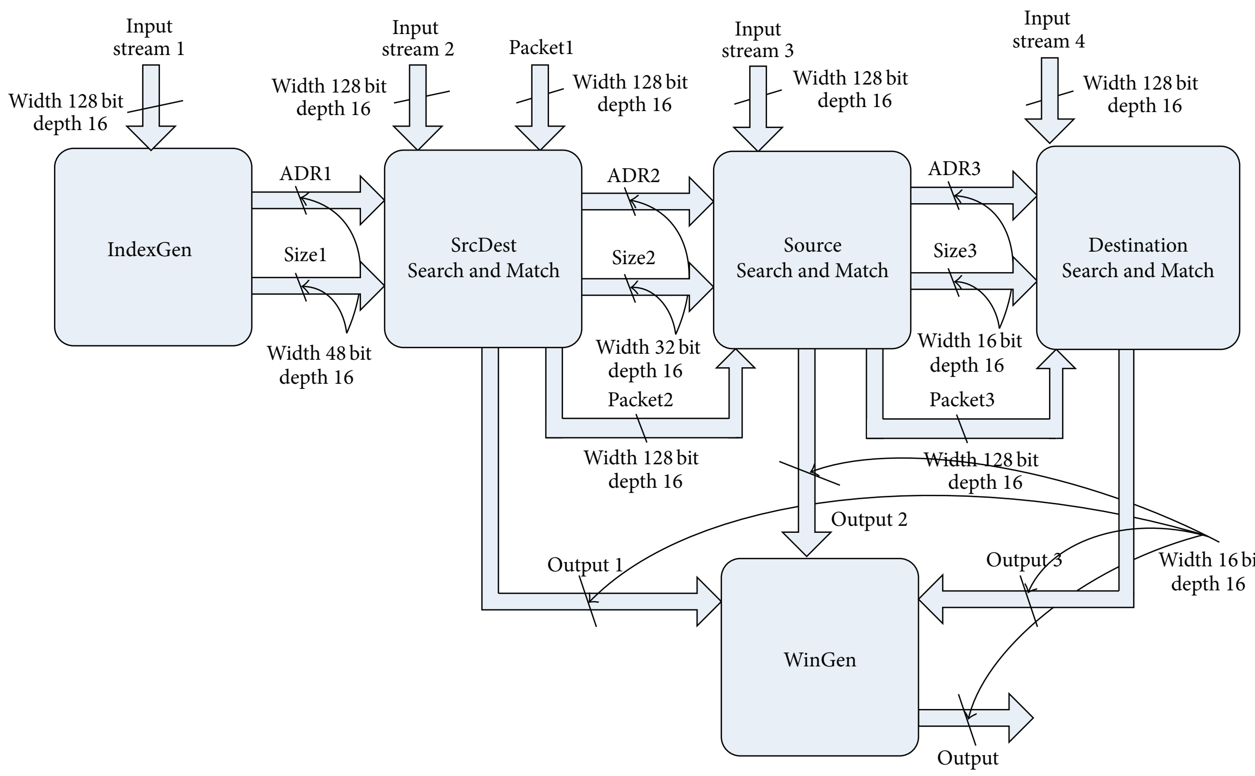


Figure 26: The GBSA Impulse-C CGO based implementation.

(output1, output2, and output3). The WinGen module will decide upon which rule will be the best match for the incoming packet.

The high number of streams used in the CGO system introduced extra delays in the system which led to deterioration in speedup achieved. The deficiency in the CGO proposed architecture can be attributed to the following reasons:

(i) the system cannot be partitioned into equally sized parts (unbalanced pipeline),

(ii) overhead delay of writing/reading to/from streams,

(iii) the search is performed within three separate tables sequentially.

The main difference between the Handel-C channels and Impulse-C stream is that the Handel-C can read/write to the channel without adding an extra clock cycle, yet Impulse-C needs one clock cycle to write to the stream and another clock cycle to read from it. Therefore, CGO is appropriate for Handel-C and inappropriate for Impulse-C.

6.2.4. Results. Figure 27 illustrates the total timing results for both the baseline and FGO implementations of the GBSA. The FGO achieved an average acceleration of 6x over the baseline implementation, although the latter was almost 7x higher than FGO in terms of clock cycles. Yet the increase in its critical-path delay was less than 1.15x. Table 5 lists the resources utilization of the baseline and FGO implementations. It is clear that FGO consumes almost twice as many resources as compared to the baseline. The sharp increase in resource consumption is attributed to the loop pipeline, flattened group of instructions, bus size increase, and usage of dual port memory.

7. Discussion and Analysis of Results

The most interesting parameters to consider when examining the effectiveness of any implementation of a packet classification algorithm are classification time, preprocessing time, and memory usage. Among the three, the one of least concern is preprocessing time since it is performed once within a larger continuous process. A shorter preprocessing time implies a shorter downtime for the system and, overall, a more versatile, resilient, and effective classification procedure.

7.1. Hardware Implementations. The pure software implementation on a desktop demonstrated excellent performance results but at the expense of an expensive general purpose processor with several dedicated ALUs and memory resources. A general purpose processor is generally not practical for anything but a server implementation, and in that respect is not directly comparable to the embedded alternatives. However, the pure software implementation is perhaps the most flexible in terms of debugging and modification.

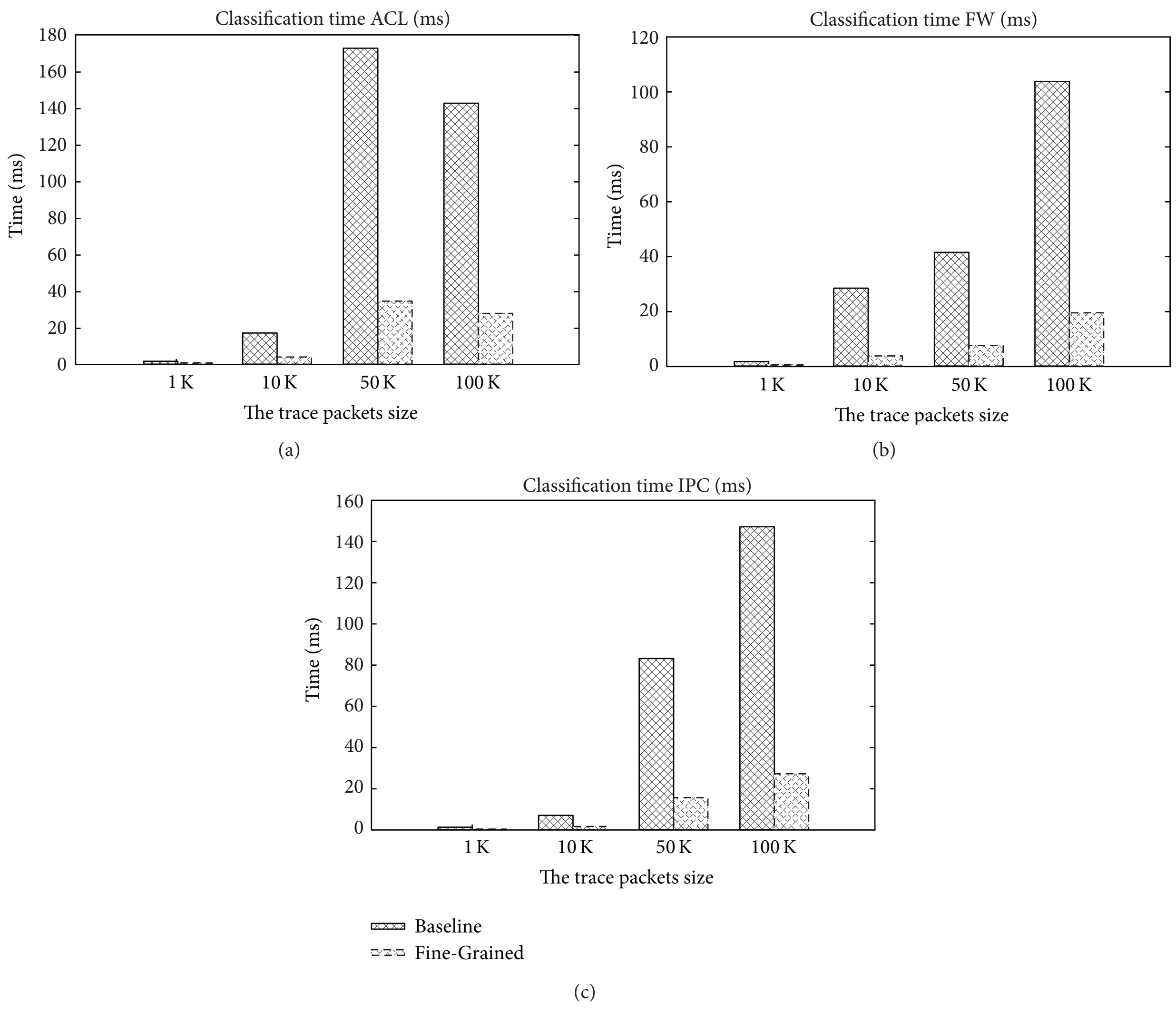


Figure 27: GBSA's Impulse-C implementation: classification timing result.

Table 5: Impulse-C design specification.

Specification	Slice Reg	Slice LUTs	LUTs (logic)	LUTs (memory)
Baseline	1176	1644	1598	46
Fine-Grained	3383	2878	2878	0
Specification	LUT-FF-Pairs	Unique-Control-Sets	Number-of-IOs	Clock-Period (ns)
Baseline	2233	30	73	4.587
Fine-Grained	5316	65	169	5.282

The alternative embedded implementations each have their own specific advantages. The ASIP implementation with a modified data path and instruction set utilizes an Application Specific Processor known as the ConnXD2 Reference Core, which allows such a design to have some degree of flexibility. Several applications, if a design is ambitious enough, could work in tandem in this structure using runtime configuration. Perhaps a client would prefer flexibility, for example, to run their GBSA algorithm, as well as a firewall in quick succession on the same system. This implementation can be easily mapped to an Application Specific Integrated Circuit (ASIC) to achieve faster speed as compared to the pure RTL version written in Handel-C and Impulse-C which are usually mapped to FPGAs.

Both RTL implementations can be mapped onto FPGAs, but the implementations would be the most cost-effective for translation into ASIC, especially considering that it is a single module. In addition, the RTL implementations are much more simple to pipeline, due to its lack of interaction with a software component.

Table 6: Performance achieved in terms of preprocessing (rule/sec).

Benchmark	Preprocessing (rule/sec)		
	Desktop	ASIP	Speedup (x)
ACL (10 K)	6,001,875.00	7,946,153.76	1.32
FW (10 K)	6,001,875.00	8,234,708.98	1.37
IPC (10 K)	3,201,000.00	8,694,129.12	2.72
Average	5,068,250.00	8,291,663.95	1.64

Table 7: Performance achieved in terms of classification (packet/sec).

Benchmark	Classification (packet/sec)			
	Desktop	Impulse-C	Handel-C	ASIP
ACL (10 K)	504,945	4,017,014	4,566,753	8,616,898
FW (10 K)	568,944	5,514,377	5,525,460	11,162,983
IPC (10 K)	475,550	3,843,463	4,036,393	8,023,022
Average	516,480	4,458,284	4,709,536	9,267,634

The key difference in design philosophy here is that optimizations in Handel-C occur as the result of changing the way in which one writes code, as opposed to ASIP where the designer finds hot spots in the form of bottlenecks. Both are forms of hardware/software codesign in general, but they each ultimately take different paths and have different advantages.

7.2. Overall Comparison. Tables 6 and 7 summarize the performance obtained via different implementations for preprocessing (in terms of rule/sec) and classification (in terms of packets/sec).

Table 8 along with Figure 28 presents a comparison of the GBSA algorithm running on different platforms. Based on the results obtained we can conclude the following.

(1) A desktop (32-bit WinXP running on Xeon 3.4 GHz with 2 G RAM) with the vast resources and power produces the slowest classification time.

(2) ASIP (894-923 MHz with 4 MB DRR): the ASIP system results in a substantial classification speedup as compared to all the other designs.

(3) Pure RTL using ESL:

   (i) Handel-C: the pure RTL nature of this design makes it the second fastest in terms of classification time. In addition, it also has more rooms for optimization, and this has been thoroughly shown through its various revisions. The pure RTL in terms of "Parallel Coarse Grain" implementation achieved on average 9.08x speedup over the desktop approach,

   (ii) Impulse-C: the means by which Impulse-C translates the C-code to HDL making it a suitable language for the GBSA. The Impulse-C FGO based implementation achieves a similar speed to that achieved by the PCGO using Handel-C. However, the nature of the sequential search of the GBSA had a negative effect on Coarse-Grain implementation.

7.3. Comparison with Existing Hardware Accelerators for Packet Classification. As mentioned in Section 2 several hardware accelerators have been developed in the past few years. However, it is important to remind the reader that it is difficult to compare our proposed hardware accelerators to those published in the literature for the following reasons.

(1) Most previous approaches tend to utilize Hardware Description Languages in the form of VHDL which leads to an unfair comparison with ESL based approaches such as Handel-C and Impulse-C.

(2) Hardware accelerators for packet classification in the literature are based on algorithms that are *different* from the GBSA algorithm. For example CAM based approaches can only accommodate up to 1 K rules whereas the GBSA can accommodate any size 1 K, 10 K, and 100 K.

(3) Results in the previous publications are based on benchmarks that are different from the benchmarks used by the authors. Unlike some packet classification algorithms, the GBSA can be used for all types of benchmarks (ACL, FW, and IPC) and even those that have high overlapping regions as explained in the paper.

However, for the sake of completeness we attempt to include a table that compares some of the previous work with our current proposed approach. Table 9 shows a comparison of the different GBSA based hardware implementations (ASIP, Handel-C, and Impulse-C) with some hardware accelerators published in the literature. It is very clear from the table that the TCAM [14] based implementation achieves the highest throughput. However, TCAM based approaches suffer from very high power dissipation and can only support a fewer number of rules. A fair comparison can be made between the GBSA ASIP implementation with [28] since both are implemented using the same platform. The GBSA implementations based on Handel-C and Impulse-C are comparable to [19] even though the latter was written in VHDL.

8. Conclusion and Future Work

In this paper, an efficient and scalable packet classification algorithm, GBSA, was implemented in both software and hardware and described in detail. The GBSA has excellent classification time, and preprocessing time, and its memory needs are moderate. The classification speed of GBSA is not affected by increasing the rule set and the number of wild-card rules in the classifier. The proposed algorithm was evaluated and compared to state-of-the-art techniques such as RFC, HiCut, Tuple, and PCIU using several benchmarks. Results obtained indicate that GBSA outperforms these algorithms in terms of speed, memory

Table 8: Classification time and speedup achieved.

Benchmark	Desktop	Impulse-C		Handel-C		ASIP	
	Time (ms)	Time (ms)	Speedup (x)	Time (ms)	Speedup (x)	Time (ms)	Speedup (x)
ACL (10 K)	192.1	24.15	7.96	21.24	9.04	11.26	17.07
FW (10 K)	163.9	16.91	8.08	16.88	8.49	8.35	16.87
IPC (10 K)	190.6	23.58	9.69	22.45	9.71	11.26	19.62
Average	182.2	21.55	**8.58**	20.19	**9.08**	10.3	**17.85**

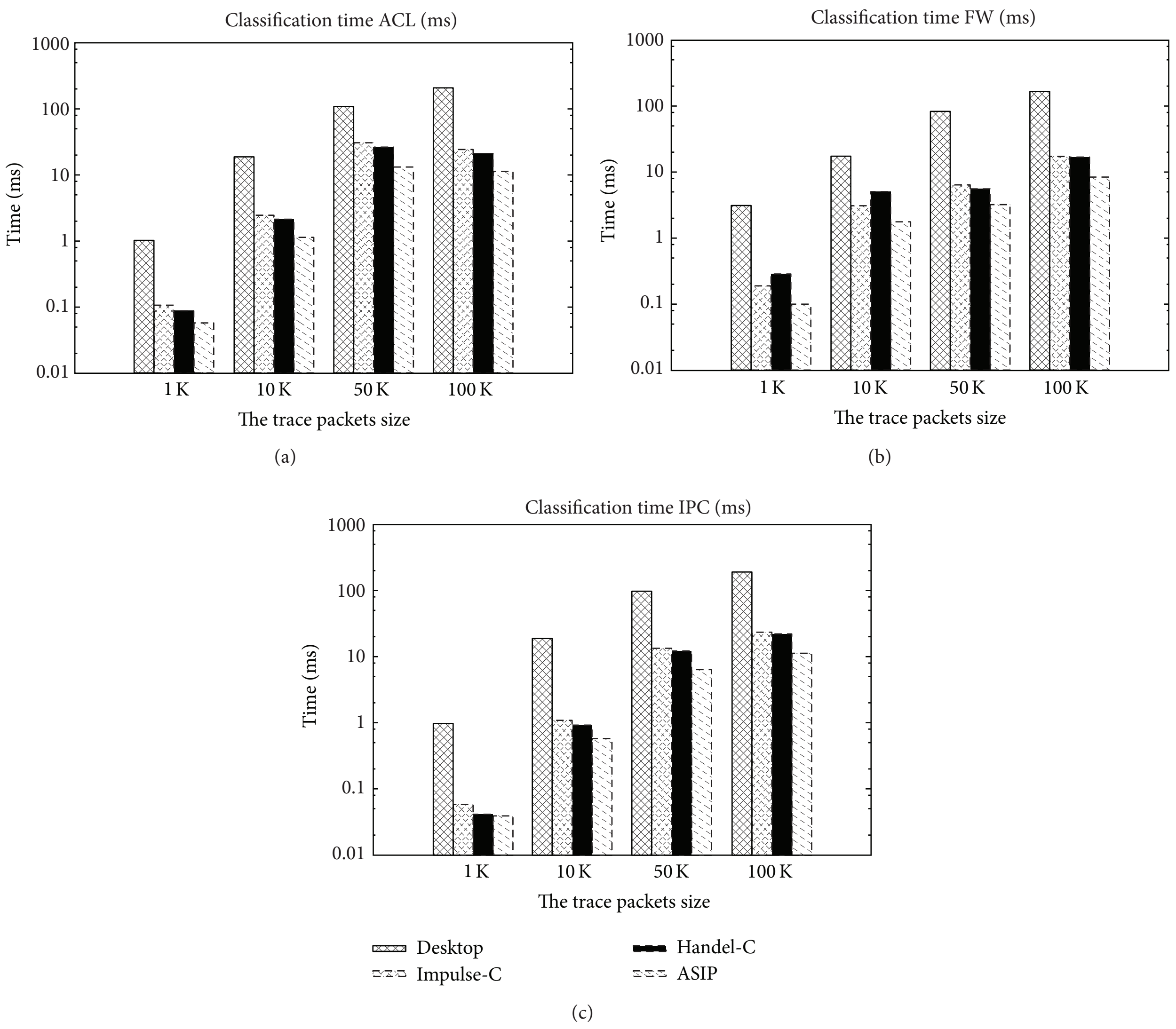


Figure 28: GBSA: overall comparison of the classification time.

usage, and preprocessing time. Furthermore, three hardware accelerators were presented in this paper, along with results analysis. Results obtained indicate that ESL based approaches using Handel-C and Impulse-C achieved on average 9x speedup over a pure software implementation running on a state-of-the-art Xeon processor. An ASIP based implementation on the other hand outperformed the RTL hardware accelerators and achieved a speedup of almost 18x. In our future work, we will attempt to further improve the performance of the GBSA by using dedicated hardware accelerators (VHDL) for the classification phase and using other ESL languages such as (AutoESL). Moreover, we will

Table 9: GBSA: comparison with hardware based packet classification accelerators.

Implementation	Language	Throughput	Platform
Xtensa [28]	TIE	2.1 Gbsp	ASIP
2sBFCE [19]	VHDL	1.87 Gbps	FPGA
DCFL [14]	VHDL	16 Gbps	TCAM
GBSA Xtensa	TIE	3 Gbps	ASIP
GBSA ESL I	Handel-C	1.5 Gbps	FPGA
GBSA ESL II	Impulse-C	1.4 Gbps	FPGA
HyperCuts [24]	VHDL	3.41 Gbps	FPGA

also attempt to verify the functionality of the GBSA with the IPv6.

References

[1] D. Joseph, *Packet classification as a fundamental network primitive [Ph.D. thesis]*, EECS Department, University of California, Berkeley, NC, USA, 2009.

[2] S. S. F. Baboescu and G. Varghese, "Packet classification for core routers: is there an alternative to CAMs?" in *Proceedings of the Conference of the IEEE Computer and Communications*, vol. 1, pp. 53–63, April 2003.

[3] O. Ahmed, K. Chattha, and S. Areibi, "A hardware/software co-design architecture for packet classification," in *Proceedings of the IEEE International Conference on Microelectronics*, pp. 96–99, Cairo, Egypt, December 2010.

[4] D. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Computing Surveys*, vol. 37, no. 3, pp. 238–275, 2005.

[5] P. Gupta and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," *IEEE Micro*, vol. 20, no. 1, pp. 34–41, 2000.

[6] G. V. S. Singh, F. Baboescu, and J. Wang, "Packet classification using multidimensional cutting," in *Proceedings of the Conference on Applications, Architectures and Protocols for Computer Communications*, pp. 213–224, ACM, New York, NY, USA, 2003.

[7] A. G. Priya and H. Lim, "Hierarchical packet classification using a Bloom filter and rule-priority tries," *Computer Communications*, vol. 33, no. 10, pp. 1215–1226, 2010.

[8] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 147–160, ACM, New York, NY, USA, 1999.

[9] F. Baboescu and G. Varghese, "Scalable packet classification," *IEEE/ACM Transactions on Networking*, vol. 13, no. 1, pp. 2–14, 2005.

[10] T. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 4, pp. 203–214, 1998.

[11] O. Ahmed, S. Areibi, and D. Fayek, "PCIU: an efficient packet classification algorithm with an incremental update capability," in *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS '10)*, pp. 81–88, Ottawa, Canada, July 2010.

[12] S. S. V. Srinivasan and G. Varghese, "Packet classification using tuple space search," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 135–146, ACM, New York, NY, USA, 1999.

[13] S. S. P. Warkhede and G. Varghese, "Fast packet classification for two-dimensional conflict-free filters," in *Proceedings of the Conference of the IEEE Computer and Communications Societies (INFOCOM '01)*, vol. 3, pp. 1434–1443, 2001.

[14] A. Ramamoorthy, G. S. Jedhe, and K. Varghese, "A scalable high throughput firewall in FPGA," in *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '08)*, pp. 43–52, April 2008.

[15] C. R. Meiners, A. X. Liu, and E. Torng, "Topological transformation approaches to TCAM-Based packet classification," *IEEE/ACM Transactions on Networking*, vol. 19, no. 1, pp. 237–250, 2011.

[16] Y.-K. Chang, C.-I. Lee, and C.-C. Su, "Multi-field range encoding for packet classification in TCAM," in *Proceedings of the IEEE INFOCOM*, pp. 196–200, April 2011.

[17] H. Le, W. Jiang, and V. K. Prasanna, "Scalable high-throughput sram-based architecture for ip-lookup using FPGA," in *Proceedings of the 2008 International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 137–142, September 2008.

[18] I. Papaefstathiou and V. Papaefstathiou, "Memory-efficient 5D packet classification at 40 Gbps," in *Proceedings of the 26th IEEE International Conference on Computer Communications (IEEE INFOCOM '07)*, pp. 1370–1378, May 2007.

[19] A. Nikitakis and I. Papaefstathiou, "A memory-efficient FPGA-based classification engine," in *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '08)*, pp. 53–62, April 2008.

[20] O. Ahmed, K. Chattha, S. Areibi, and B. Kelly, "PCIU: hardware implementation of an efficient packet classification algorithm with an incremental update capability," *International Journal of Reconfigurable Computing*, 2011.

[21] Y.-K. Chang, Y.-S. Lin, and C.-C. Su, "A high-speed and memory efficient pipeline architecture for packet classification," in *Proceedings of the 18th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM '10)*, pp. 215–218, May 2010.

[22] G. Antichi, A. Di Pietro, S. Giordano, G. Procissi, D. Ficara, and F. Vitucci, "On the use of compressed DFAs for packet classification," in *Proceedings of the 15th IEEE International Workshop on Computer Aided Modeling, Analysis and Design of Communication Links and Networks (CAMAD '10)*, pp. 21–25, December 2010.

[23] W. Jiang and V. K. Prasanna, "Scalable packet classification on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 9, pp. 1668–1680, 2011.

[24] Z. Liu, A. Kennedy, X. Wang, and B. Liu, "Low power architecture for high speed packet classification," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '08)*, pp. 131–140, ACM, New York, NY, USA, November 2008.

[25] D. Taylor and J. Turner, "Classbench: a packet classification benchmark," in *Proceedings of the IEEE INFOCOM*, pp. 2068–2079, 2005.

[26] O. Ahmed and S. Areibi, "GBSA: a group based search algorithm for packet classification," in *Proceedings of the IEEE International Workshop on Traffic Analysis and Classification (TRAC '11)*, pp. 1789–1794, Istanbul, Turkey, July 2011.

[27] "Xtensa instruction set architecture (ISA)," Technical Publications 95054, Tensilica Inc., Santa Clara, Calif, USA, 2012.

[28] Tensilica, "Xtensa processor extensions for fast ip packet classification," Tech. Rep. AN02-603-00, Tensilica Inc., Santa Clara, Calif, USA, 2002.

[29] RG, "Handel-C language reference manual," Technical Report, Celoxica, London, UK, 2005.

[30] D. Pellerin and S. Thibault, *Practical Fpga Programming in C*, Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2005.