Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 648483, 21 pages
doi:10.1155/2011/648483

Research Article

PCIU: Hardware Implementations of an Efficient Packet Classification Algorithm with an Incremental Update Capability

O. Ahmed, S. Areibi, K. Chattha, and B. Kelly

School of Engineering, University of Guelph, Guelph, ON, Canada N1G 2W1

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

Received 6 May 2011; Accepted 14 July 2011

Academic Editor: Patrick R. Schaumont

Packet classification plays a crucial role for a number of network services such as policy-based routing, firewalls, and traffic billing, to name a few. However, classification can be a bottleneck in the above-mentioned applications if not implemented properly and efficiently. In this paper, we propose PCIU, a novel classification algorithm, which improves upon previously published work. PCIU provides lower preprocessing time, lower memory consumption, ease of incremental rule update, and reasonable classification time compared to state-of-the-art algorithms. The proposed algorithm was evaluated and compared to RFC and HiCut using several benchmarks. Results obtained indicate that PCIU outperforms these algorithms in terms of speed, memory usage, incremental update capability, and preprocessing time. The algorithm, furthermore, was improved and made more accessible for a variety of applications through implementation in hardware. Two such implementations are detailed and discussed in this paper. The results indicate that a hardware/software codesign approach results in a slower, but easier to optimize and improve within time constraints, PCIU solution. A hardware accelerator based on an ESL approach using Handel-C, on the other hand, resulted in a 31x speed-up over a pure software implementation running on a state of the art Xeon processor.

1. Introduction

Packet classification is the process of matching an incoming packet to rules in the classifier, and accordingly identifying the type of action to be performed on the packet. The classifier, also known as a policy database, is a collection of rules or policies. Each rule specifies a class (flow) that the arriving packet may belong to based on some criteria in its header. An action is associated with each rule in the rule set. The packet header has F fields, which can be used in the classification process. Each rule has F components which identifies all possible combination of packet header that match the rule. Accordingly, a packet will belong to the rule if and only if all the fields in that packet belong to the corresponding field in the rule. Figure 1 depicts the general packet classification system.

There are a number of network services that require packet classification, such as routing, policy-based routing, rate limiting, access control in firewalls, virtual bandwidth location, load balancing, provision of differentiated qualities of service, and traffic billing. In each case, it is necessary to determine which flow an arriving packet belongs to, for example, whether to forward or filter (firewall), where to forward the packet (router), and the class of service it should receive (QoS), or how much should be charged for transporting it (traffic billing). The main bottleneck of the above applications is the classification stage. Therefore, packet classification is one of the most important issues to deal with in the design of network devices.

Most popular classifiers deal with a certain field in the packet header to define the flow. For example, it could depend on the values of source and destination IP addresses or particular transport port numbers. Otherwise, it could be simply defined by a destination prefix and range of port values. Sometimes, even the protocol type could be used to define a flow. Our work focuses mainly on the problem of identifying the class to which a packet belongs.

The main contribution of this paper can be summarized in introducing a novel and efficient algorithm [1] which has high scalability, hardware/software implementation capability [2], incremental update, reasonable memory usage, and supports a high-speed wire link. The main difference between [1] and this publication is introducing detailed

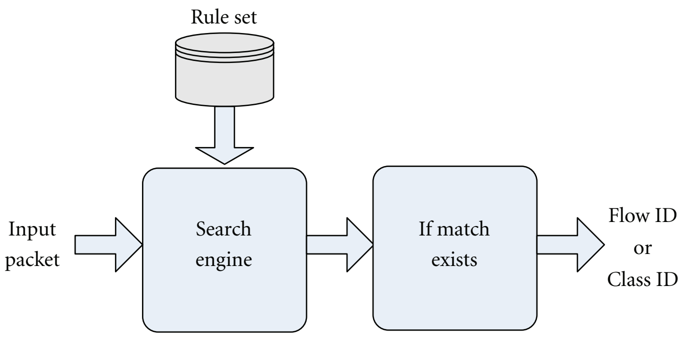


Figure 1: The general packet classification system.

explanation of the proposed technique, fine tuning of parameters, and hardware implementations of the packet classification algorithm. Two novel hardware accelerators are presented in this paper. The first is based on a hardware/software codesign approach while the second is based on a pure RTL implementation using Handel-C. In the hardware/software codesign approach an optimized hardware accelerator was designed and attached to a soft core.

To the best of our knowledge, the PCIU is the first proposed algorithm that can accommodate efficient incremental update for the rule set. Another contribution is introducing a hardware/software codesign and using an ESL approach coprocessor of the PCIU algorithm which achieves a speedup of 22x over a state-of-the-art general purpose processor.

The remainder of this paper is organized as follows. Section 2 provides an overview of the packet classification problem. Section 3 briefly lists the most important work published in the field of packet classification. In Section 4, the algorithm and the main constraints in implementing it will be presented. All stages from preprocessing, classification, and updating are also described in detail. Comparisons with other published algorithms are presented in Section 5. A hardware/software codesign implementation of PCIU along with a coprocessor implementation using Handel C are described in Section 6. Section 7 briefly compares various implementations of the PCIU algorithm with pure software implementation running on a general purpose processor. The paper concludes in Section 8 along with future directions.

2. Background

An example of a five-rule classifier is shown in Table 1. Each rule contains five fields. The first two fields represent the network layer address (source IP and destination IP), and the size of each is 32 bits. The next two fields of size 16 bits are the transport layer address (source and destination port). The last field is the packet protocol field. The IP address and the protocol are present in either prefix or exact format for matching, yet the port is in range format. The IP address parts of the rule are represented in the following format IP/mask.

An IP address such as 0.83.4.0/22 can be converted to its range equivalent by using the mask of 22 bit 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 a 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.

Table 2 shows the result of converting the five-rule classifier from different representation to the range format. An incoming packet belongs to a certain flow when all the fields of the packet are in the range of the flow's rule(s). In other words, each rule has F components (five in our example). The ith component of rule R, referred to as R[i], is a regular expression on the ith field of the packet header. A packet P is said to match a particular rule R if and only if for every ith field of the header P field is in the range of R[i].

A trivial software classification algorithm implementation can be described as accepting a packet, evaluating each rule sequentially, until a rule is found that matches all the fields in a packet's header. However, the search need not be terminated, since the arriving packet might match more than one rule, and the best match [3] has to be found. It is important to note that the best match implies the rule with the shortest range among all matched rules. In the case of multiple matches, the best match rule will be chosen. Therefore, the classification time is constant for all different combination of the arriving packets and depends on the number of rules. This method has the most efficient memory usage, because it needs simple preprocessing. Moreover, all different types of classifier updating are straightforward without the need to reboot the device. However, the main disadvantage is the poor scalability. On the other hand, a hardware implementation could employ a ternary CAM (content addressable memory) [4]. Ternary CAMs store words with three-valued digits: 0, 1 or X (wildcard). The rules are stored in the CAM array in the order of decreasing priority. Given a packet header to classify, the CAM performs a comparison against all of its entries in parallel, and a priority encoder selects the best matching rule [4]. While CAMs are efficient and flexible, they are currently suitable only for small-size classifiers, since they are too expensive and consume too much power for large classifiers. Furthermore, some operators are not directly supported and need

Table 1: A five-rule classifier.

No.	IP (64 bits)		Port (32 bits)		Protocol (8 bits)
	Source (32 bits)	Destination (32 bits)	Source (16 bits)	Destination (16 bits)	
1	0.0.0.0/0	0.0.0.0/0	0 : 65535	21 : 21	0/ff
2	0.83.1.0/24	0.0.4.6/32	0 : 65535	20 : 30	17/ff
3	0.83.4.0/22	0.0.0.0/0	0 : 65535	21 : 21	0/0
4	0.0.9.0/24	0.0.0.0/0	0 : 65535	0 : 65535	0/ff
5	0.83.0.77/32	0.0.4.6/32	0 : 65535	0 : 65535	17/ff

Table 2: A five-rules classifier in range format.

	IP (64 bits)				Port (32 bits)				Protocol (8 bits)	
	Source		Destination		Source		Destination			
No.	L	H	L	H	L	H	L	H	L	H
1	0.0.0.0	255.255.255.255	0.0.0.0	255.255.255.255	0	65535	21	21	0	0
2	0.83.1.0	0.83.1.255	0.0.4.6	0.0.4.6	0	65535	20	30	17	17
3	0.83.4.0	0.83.7.255	0.0.0.0	255.255.255.255	0	65535	21	21	0	255
4	0.0.9.0	0.0.9.255	0.0.0.0	255.255.255.255	0	65535	0	65535	0	0
5	0.83.0.77	0.83.0.77	0.0.4.6	0.0.4.6	0	65535	0	65535	17	17

preprocessing, so the memory array may be used inefficiently.

3. Related Work

According to [5], packet classification algorithms can be classified into four categories: (a) exhaustive search, (b) decision tree (c) decomposition and (d) tuple space, as shown in Figure 2 with four quadrants. The position of an algorithm within a quadrant is irrelevant. The performance of an algorithm does not rely on its position within any quadrant.

The researchers in [3] proposed a very interesting decomposition-based algorithm called recursive flow classification (RFC), which takes advantage of the considerable redundancy in real network filter sets. The RFC maps S bits in the packet header to T bits of precomputed classID ($T \ll S$) based on real filter rules. However, the results will not hold if the characteristics of the filter set change in the near future. The authors assumed that the number of distinct overlapping regions is way smaller than the worst case which is $O(n^F)$, where n is the number of rules with dimension F. In addition, since the scheme requires extensive precomputation even with the normal case, it is assumed the filter changes infrequently. This is nonpractical with session-based dynamic updates. The memory usage grows exponentially with the number of rules, as they tend to have a high number of distinct overlapping regions.

The original idea of using bit vector algorithm proposed in [6] was improved in [7] by introducing the aggregated bit-vector algorithm (ABV) using recursive bit aggregation and rule set rearrangement to reduce memory access. It is based on the assumption that the number of rules that a packet will match in a real filter database is inherently small. The scheme reduces the number of memory accesses by recursively generating an aggregation bit for every X bit in the original bit vector. The bit map values need to be examined only if the aggregation bit is set. The number of memory accesses can be further reduced by rearranging the filter rules such that multiple filters which match a specific packet are placed close to each other. As a result, multiple matching filters can be placed in the same aggregation group. However, as the wildcards in the filter rules increase, a further reduction in memory accesses is achieved. Also, the use of Grid of Tries has two drawbacks. Firstly, the worst case classification might increase memory usage dramatically. Secondly, the classification time can vary depending on the arriving packet value which changes the path inside the Grid of Tries.

The authors of HiCut [8] and HyperCut [9] tend to organize the rule of the classifier in a decision tree structure. Classification is performed by traversing the tree until a leaf node is identified, which stores a small number of rules. A linear search is then performed among these rules to find the match. HyperCut [9] minimizes the depth of the decision tree by splitting it based on multiple fields as opposed to the single field in HiCut [8]. Due to the amount of preprocessing required, neither algorithm supports incremental updates. In addition, both of these algorithms have a variable range of classification time per arriving packet.

By combining a rules set statistical usage and multiple decision trees based on ternary string representation, the authors in [10] built a modular packet classification scheme. The most frequently accessed rules are placed near the root of the search tree. Similar to HiCut, it also requires a linear search at the leaf nodes of the decision tree.

Several works on increasing the storage efficiency of rule sets and reducing power consumption have been published in [11]. Pure RTL hardware approaches have been proposed by many researches including [12–16]. A Dual port IP

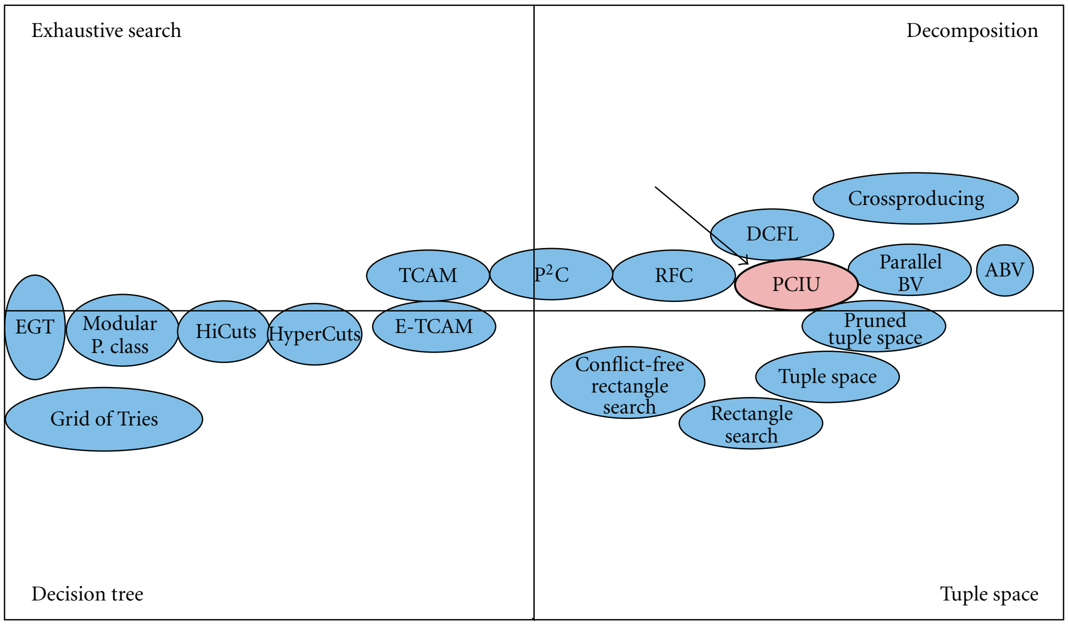


Figure 2: Classification of the PCIU algorithm.

Lookup (DuPI) SRAM-based architecture has been proposed by the authors in [12]. By using a single Virtex-4, DuPI can support a routing table of up to 228 K prefixes. Moreover, it maintains packet input order, and supports inplace non-blocking route updates. However, this architecture is suitable for single dimension classification.

A five dimension memory-efficient packet classification was proposed in [13]. It is based on a memory-efficient decomposition-based packet classification algorithm, which uses multilevel Bloom Filters to combine the search results from all fields. The authors used ClassBench [14] to evaluate their work with a small rule set less than 4 k. In [15], a scalable high throughput firewall using an FPGA was proposed. The Distributed Crossproducing of Field Labels (DCFL) is used on the firewall application and an improvement has been added to DCFL to be Extended (DCFLE). A Xilinx Virtex 2 Pro FPGA is used for the implementation, using a memory intensive approach, as opposed to the logic intensive one, so that on-the-fly update is feasible. A throughput of 50 MPPS was achieved for a rule set of 128 entries. They also predict the throughput can be 24 Gbps when the design is implemented on Virtex-5 FPGAs. Another interesting architecture has been proposed in [16]. The authors proposed a memory-efficient FPGA-based classification engine called *dual stage bloom filter classification engine (2sBFCE)*. This design can support 4 K rules in 178 K bytes memories. However, the design takes 26 clock cycles on average to classify a packet, resulting in low throughput of 1.875 Gbps on average.

The authors in [17] discuss various packet classification algorithms by grouping the algorithms into several categories depending on their approaches and characteristics. Most packet classification algorithms suffer from the following short comings. First, they are generally slow because of the number of memory accesses required. For example, the "binary search on length" method proposed by [18] requires on average 18 → 67 memory accesses for rule set of size 5000. Also, packet classification algorithms generally build on certain features of a special kind of rules set, which allows them to perform well on only that type of rule set. For example, RFC [3] assumes that the number of distinct overlapping regions are slightly low even with high number of rule set. Finally, almost all the proposed algorithms need considerable amount of pre-processing. Therefore, none of the above algorithms work well with incremental updates, which is a key feature in our proposed algorithm to further support session-based packet classification.

4. Proposed Algorithm

Despite the fact that the packet classification problem has been studied intensively by many researchers [4, 7, 8, 18], we believe that solutions offered are far from optimal. A near optimal classifier should exhibit the following features.

(1) It has to consume memory equal to or less than the original rule set size.

(2) It has to perform rule updating without the need of resetting or powering down the network device.

(3) Its classification speed has to be higher than the wire link speed.

(4) Its pre-processing time has to be reasonable and should not require too much processing effort.

Based on the taxonomy of packet classification algorithms introduced by [5], the PCIU is considered to be a decomposition or divide and conquer approach as shown in Figure 2.

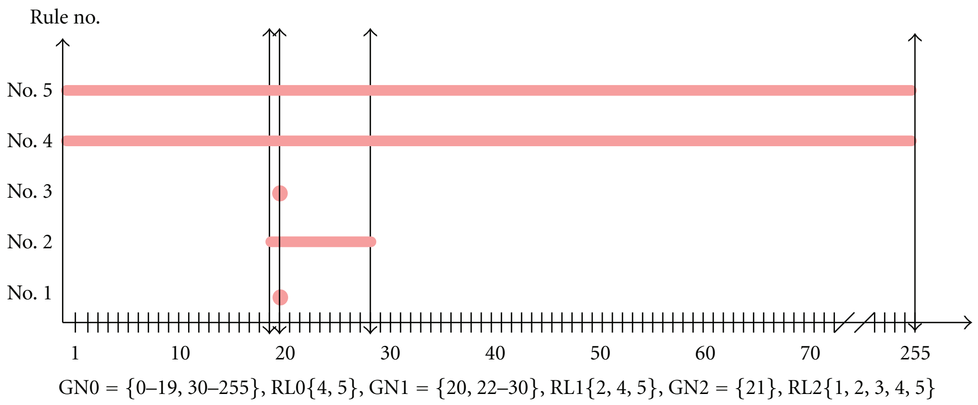


Figure 3: An Example of processing chunk no. 10 in the rule set of Table 1.

4.1. Specifications and Features. The main features of our proposed algorithm can be summarized as follows.

(i) The algorithm is fast enough to operate at 40 Gb/s or even higher. Moreover, it allows matching on arbitrary fields, including link layer, network layer, transport layer and “in some exceptional cases” the application layer headers without any effect on its performance.

(ii) The algorithm is suitable for implementation in both hardware and software.

(iii) The memory requirements of the algorithm are not prohibitively expensive.

(iv) The algorithm scales well in terms of both memory and speed with the size of the classifier.

(v) The algorithm is capable of updating the classifier without the need to reboot the device.

(vi) The algorithm has minimum pre-processing time that does not change dramatically by changing the rule set.

4.2. The Preprocessing Phase. The basic idea is based on the simple concept that there is a redundancy in the rule set in case it is chopped to several chunks. We propose to have a five-dimension packet classification problem similar to that in Table 1. A hierarchical approach is used where we decompose it into subproblems and then combine the result at the end of the classification process.

The main idea of the algorithm [1] can be summarized in the following steps.

Step 1. Convert the rule set from all the different representations to a range representation as shown in Table 2.

After completion of the first step, each of the five dimensions (or fields) in the rule set has upper and lower ends of its value.

Step 2. Divide each of the five dimensions to 8-bit chunks. Since the rule size is 104 bits, the total number of chunks is thirteen. The range value of each chunk$_i$ is filled in the range (0 to 255). For each chunk, a lookup table of size 2^8 is assigned.

Step 3. Generate a group of equivalent rules. It can be stated that point X in the lookup table belongs to group G if and only if X belongs to all the rules in G. In other words, two intervals are in the same group if exactly the same rules project onto them. As an example, consider chunk no. 10 (Destination port low byte) of the classifier in Table 1. The end points of the intervals (RL0 · RL4) and the constructed group sets (GN0·GN2) are shown in Figure 3. The *lookuptable* $_{10}$ for this chunk is filled with the corresponding group number (GN). Thus, in this example, $table_{10}(19) = 00_b$, $table_{10}(20) = 01_b$, $table_{10}(21) = 10_b$.

Step 4. Convert the groups to a binary vector where the bit location represents the rule ID, the value of which indicates whether it belongs to the group or not. The pseudocode of finding the groups of rules is shown in Figure 4. The preprocessing algorithm complexity is $\Theta(N)$, where N is the number of rules.

A bit vector (BV) of size equal to the rule set size, five bits in our example, is required for each group. This BV has one bit for each rule in the classifier. For example, GN_0 in Figure 3 will have the BV 11000_b indicating that the fourth and the fifth rules of the classifier in Table 1 belong to GN_0 in chunk no. 10. Note that the group table is physically stored in a separate table with address equal to its number.

4.3. Classification Phase. Figure 5 shows an example of the preprocessing and classification stages. Following the construction of the thirteen lookup tables with their corresponding BVs, the lookup tables are ready for the classification phase. The incoming packet header chopped to thirteen chunks of eight bits each is used as an address for its lookup table. Each lookup table points to a specific BV. As a result, thirteen BVs of size N will be obtained where N is the number of rules in the rule set. A simple “ANDing” operation of these vectors produces the winner rule for the arriving packet as demonstrated by Figure 5.

```
/* The Pre-processing Phase*/
1.  FOR each of the 13 Chunks do
2.    GN:=0
3.    FOR I:=0, I ≤ 255, I++
4.        VecPos:= 1
5.        BitVec:= 0
6.        FOR ID:=0, ID ≤ NumberRule-1, ID++
7.            IF I ≥ Rule[ID].LOW AND I ≤ Rule[ID].HIGH THEN
8.                BitVec:=BitVec OR VecPos
9.            END IF
10.             VecPos:= VecPos *2
11.         ENDFOR
12.          G:=0
13.         WHILE G≤GN AND BitVec ≠ Group[G] DO
14.             G++
15.         ENDWHILE
16.         IF G ≥ GN THEN
17.             Group[GN]:= BitVec
18.             GN:=GN+1
19.         ENDIF
20.     ENDFOR
21.   ENDFOR
```

Figure 4: PCIU preprocessing phase.

4.4. Incremental Update Phase

4.4.1. Adding New Rules. One of the main features of the PCIU algorithm is that it can accommodate efficient incremental updates for the rule set. However, one of the design constraints is that the system has a certain capacity that cannot be exceeded during updates. As long as the system limit is not exceeded new rules can be added incrementally with ease. A new rule can be added by the following steps. (i) Find the first empty location in the rule set and save it in this location. (ii) Chop it to 13 parts of 8 bits each and assign a part for each lookup table. (iii) Each part will represent a range inside the lookup table. (iv) From the high to the low value of each part, perform bitwise “ORing” of all the bit vector in this range with 2^{RuleID}, where RuleID is the first empty location of the rule in the rule set.

4.4.2. Deleting Rules. The deletion of any rule in the rule set can be accomplished by the following steps. (i) Mark the rule location in the rule set as empty. (ii) Chop the deleted rule to 13 parts, with high and low part for each. All the vectors in each range of these parts (form low to high), will be *Anded* with the complement of (2^{RuleID}), where RuleID is the deleted rule’s number.

4.5. Parameter Tuning. The PCIU algorithm can be reorganized to accommodate alternative chunk sizes. The initial PCIU [1] algorithm used 8-bit chunks. Varying the number of bits in each chunk results in changes in terms of memory usage, classification speed, and pre-processing speed. There are two possible variable versions of the PCIU: Uniform chunk size and nonuniform chunk size. The uniform chunk size variation is a lot simpler to implement algorithmically and ensures that all chunks are of the specified size and only deal with the data for a single rule, as accomplished with the standard 8-bit size. The nonuniform chunk size variation is different in that it attempts to fit as much data into the defined chunk size without wasting any space. This implies that multiple rules can have their bits strung together in separate chunks. Ultimately, this approach uses far less memory but is difficult to predict with standard algorithmic design techniques and therefore difficult to implement. Accordingly, only uniform chunk size parameter tuning will be discussed.

Figures 6 and 7 illustrate the effect of varying uniform chunk size in terms of pre-processing speed, classification speed, and memory usage. Of particular note is the data as displayed in Figure 6, because it demonstrates the effects in a complex-ruleset, high-volume environment.

It is apparent that in a uniform chunk size scheme, there is no optimum chunk size that satisfies all test parameters. However, there is an effective range. Depending on what a client wants, an optimum chunk size can be selected from 7 to 11. Moving from 7 to 11, there is a decrease in preprocessing and classification time and an increase in memory usage. Most notably, at chunk size of 11 the classification time greatly dips and the memory usage greatly spikes.

5. Experimental Results and Analysis

In this work, we used ClassBench [14] as the source for our rule sets. Table 3 shows the sizes of the rule sets and their corresponding trace files. The seeds and program used to generate these rule sets are based on ClassBench [14].

The ClassBench [14] performed a battery of analyses on 12 real filter sets provided by internet service providers (ISPs), a network equipment vendor, and other researchers working in the field. The filter sets utilize one of the following formats.

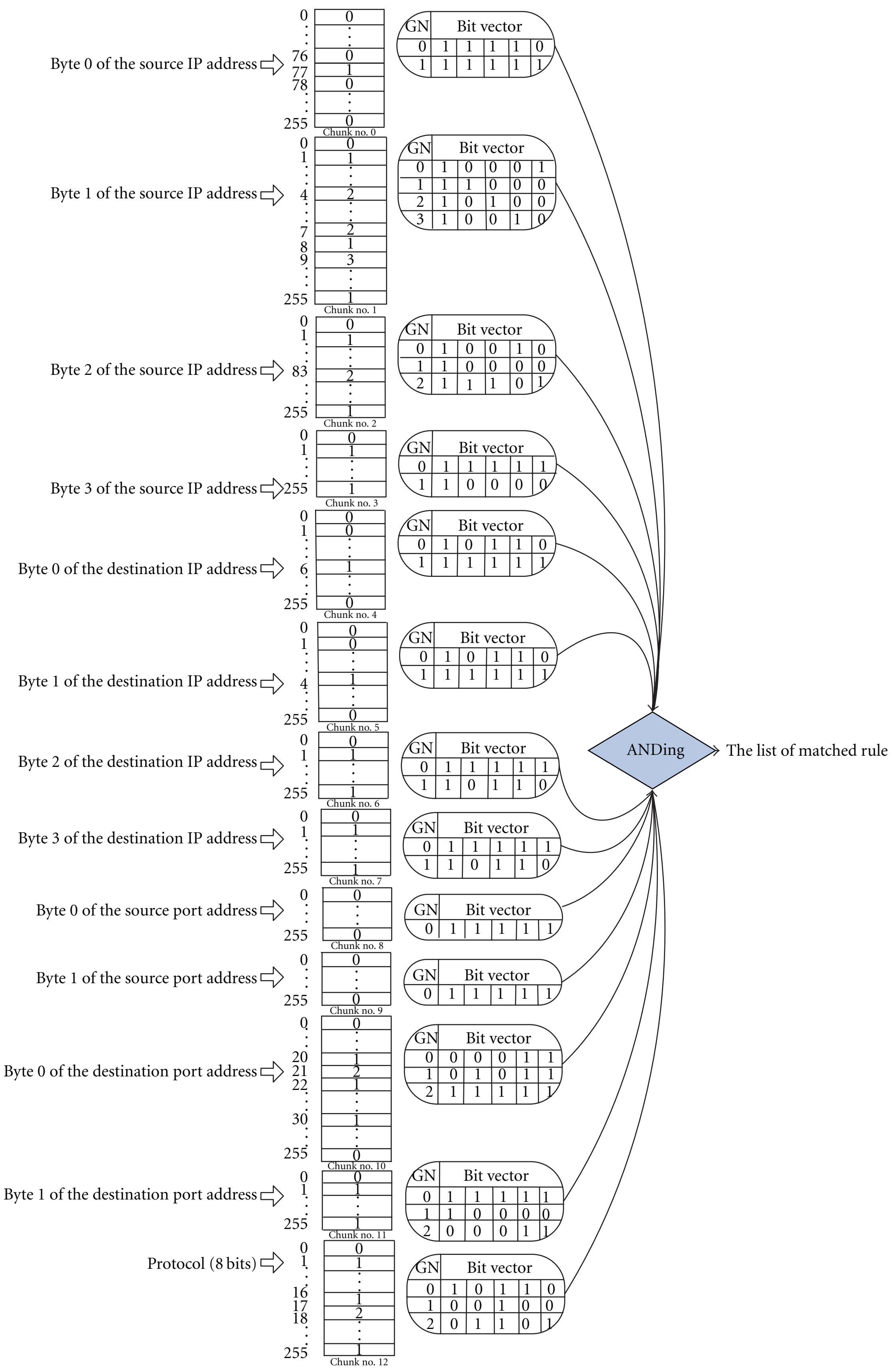

Byte 0 of the source IP address
Byte 1 of the source IP address
Byte 2 of the source IP address
Byte 3 of the source IP address
Byte 0 of the destination IP address
Byte 1 of the destination IP address
Byte 2 of the destination IP address
Byte 3 of the destination IP address
Byte 0 of the source port address
Byte 1 of the source port address
Byte 0 of the destination port address
Byte 1 of the destination port address
Protocol (8 bits)
Chunk no. 0
Chunk no. 1
Chunk no. 2
Chunk no. 3
Chunk no. 4
Chunk no. 5
Chunk no. 6
Chunk no. 7
Chunk no. 8
Chunk no. 9
Chunk no. 10
Chunk no. 11
Chunk no. 12
GN
Bit vector
ANDing
The list of matched rule

Figure 5: Example: PCIU flow using the rule set in Table 1.

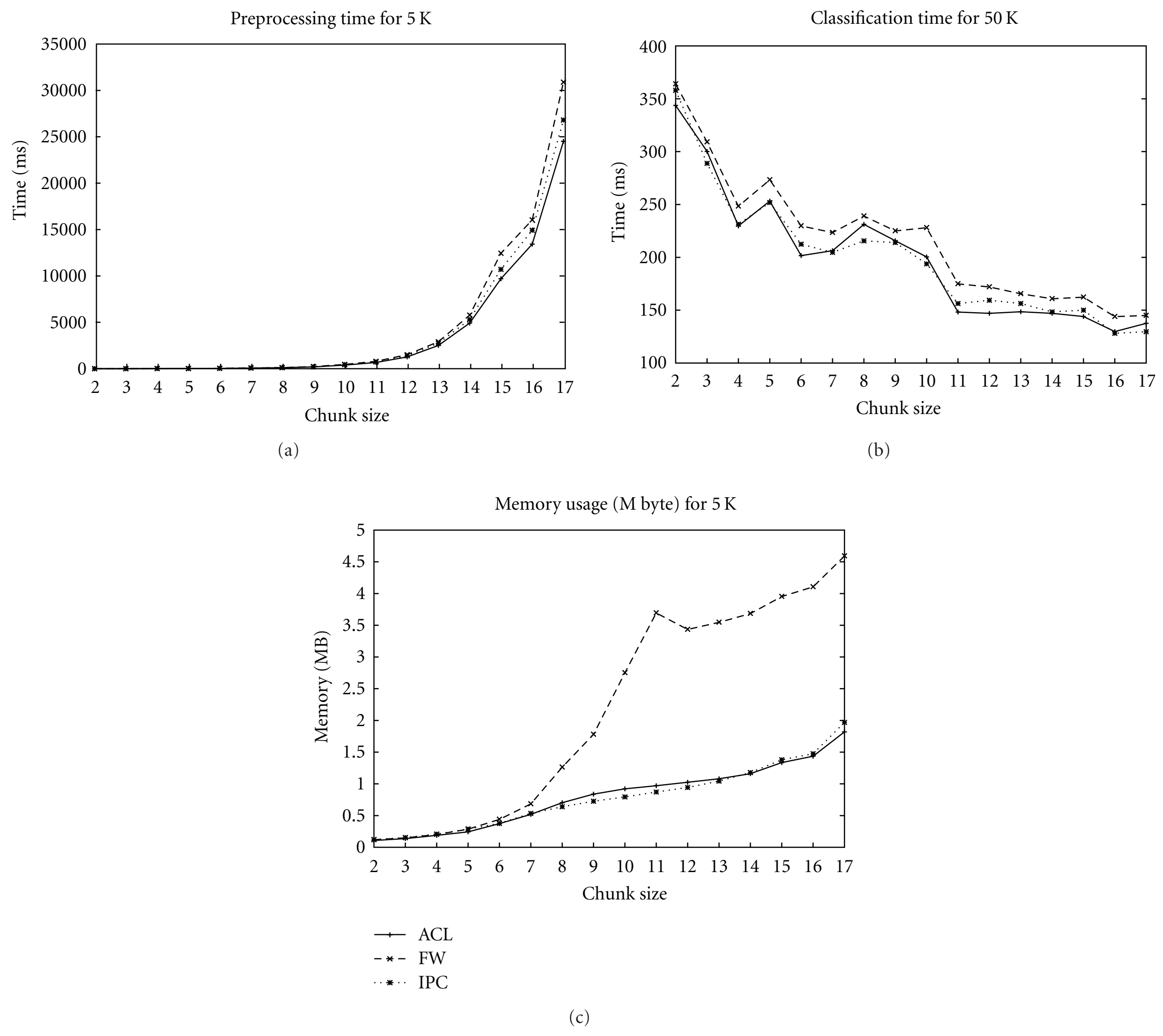


Figure 6: Uniform chunk size variation for PCIU with 5 K rule sets.

(1) *Access control list-(ACL-)* standard format for security, VPN, and NAT filters for firewalls and routers (enterprise, edge, and backbone).

(2) *Firewall-(FW-)* proprietary format for specifying security filters for firewalls.

(3) *IP chain-(IPC-)* decision tree format for security, VPN, and NAT filters for software-based systems.

The algorithms are evaluated using a desktop computer running a 32-bit installation of Windows XP with an Intel Xeon 3.4 GHz CPU. The results shown in Figure 8 are based on a single processor core. The following can be concluded.

(1) The maximum usage of memory in a 10 k rule set is 2.5 MB which is reasonable and could fit in any type of embedded network processor.

(2) The pre-processing is quite simple yet effective and could be performed by any RISC processor. The pre-processing speed is 39.2 K rule/sec which could be improved by using a hardware accelerator.

(3) The classification time is appropriate for most network applications. The results shows that the classification is 0.2 M packet/sec in the worst case

5.1. PCIU, RFC, and HiCut: A Comparison. A comparison between PCIU, RFC, and HiCut is shown in Figure 9 based on benchmarks introduced in Table 3. Based on [17], the performance of RFC and HiCut compared to other state-of-the-art algorithms is close and even better in terms of classification time. It is obvious that the PCIU has the lowest memory consumption and pre-processing time. We only used the ACL benchmark in this comparison, since

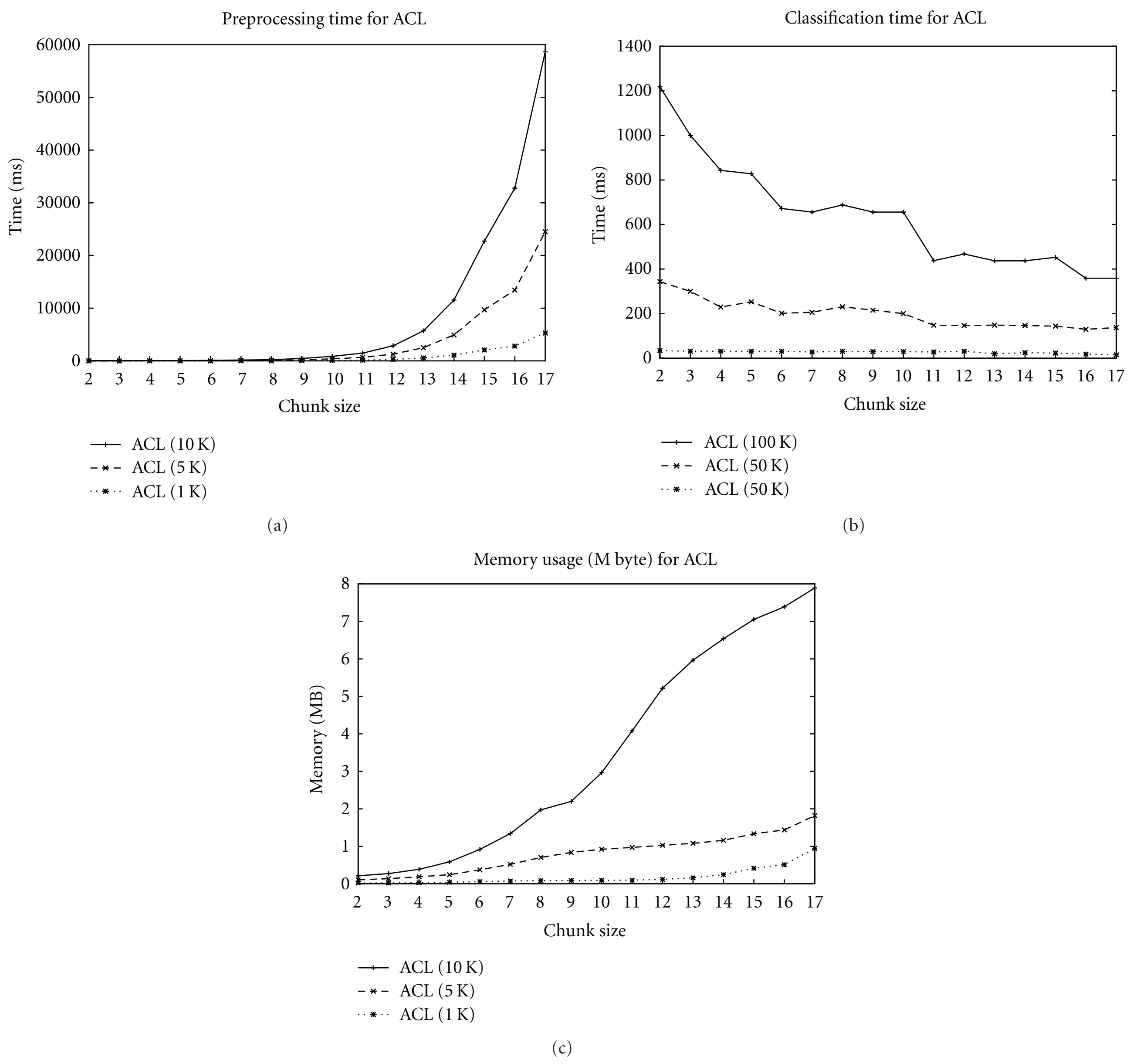


Figure 7: Uniform chunk size variation for PCIU based on the ACL benchmark.

the number of distinct overlapping regions in both FW and IPC are high. Therefore, the RFC's pre-processing time could easily consume several days for IPC and FW. The maximum memory which can be supported by windows XP is 2 GB, which is insufficient to support the BV algorithm [6] for the 10 k rule set. The PCIU algorithm outperforms all the previous published work due to the following properties.

(1) The PCIU algorithm does not utilize a decision tree that needs to be traversed. Accordingly, no comparison is required at each node when making a decision regarding the next node address. The comparison operations tend to increase classification time. In addition, the total number of nodes and its associated pointers tend to consume extra memory. Finally, when the rule set has too many overlapping regions, the size of memory and the pre-processing time increase sharply.

(2) PCIU does not require intensive pre-processing time like RFC. The RFC algorithm suffers from high pre-processing time which can easily consume several days even when a powerful processor is used. This drawback magnifies when a rule set tends to have a lot of region overlap.

(3) PCIU's main classification operation is the logical "ANDing" which is assumed to be a simple and minimally time consuming operation. Combining a look-up table and the bit vector make the PCIU efficient

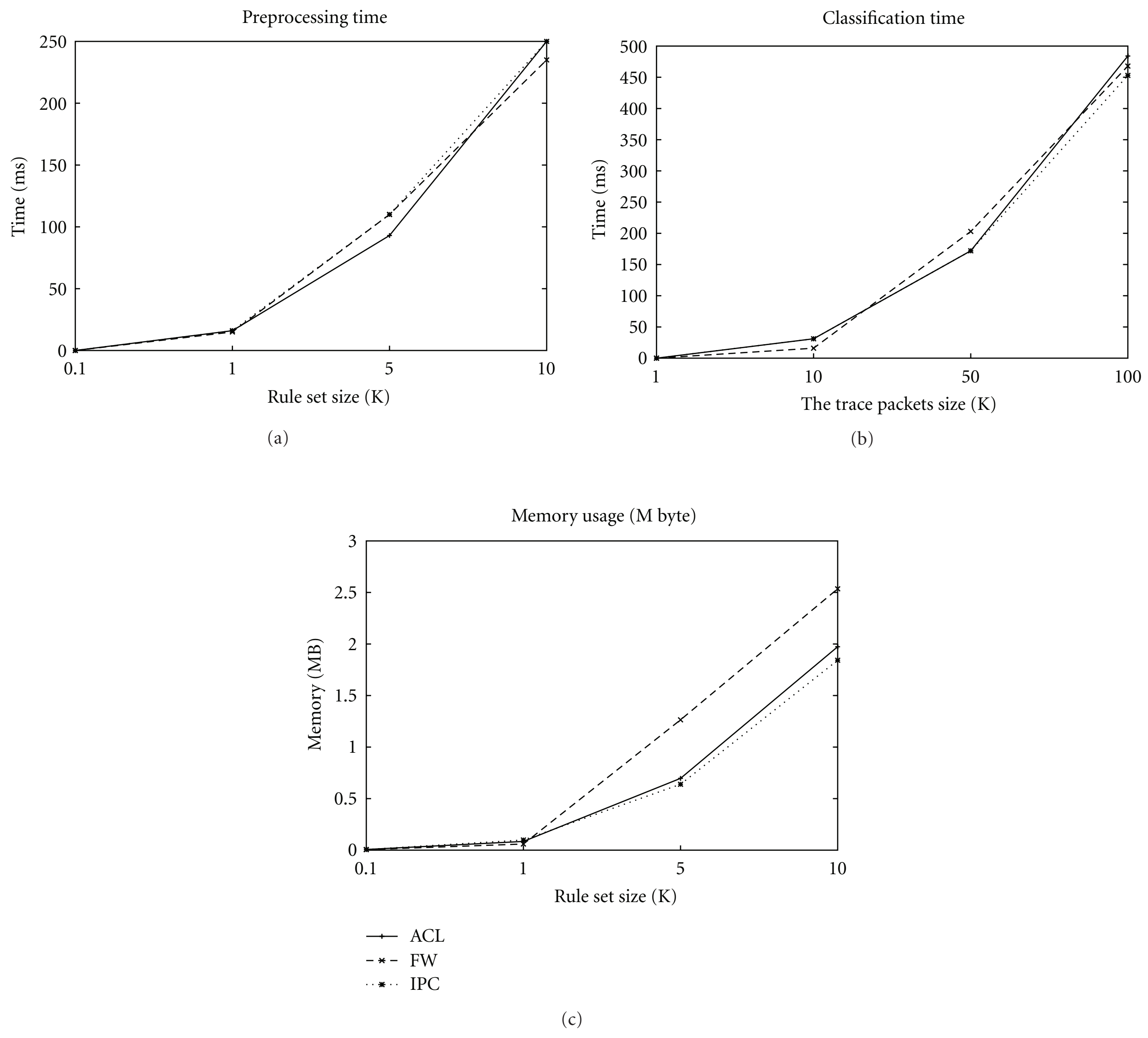


FIGURE 8: PCIU: Preprocessing, classification time, and memory usage.

for any kind of rule set. Moreover, the reduction in bit vectors tends to reduce the memory size.

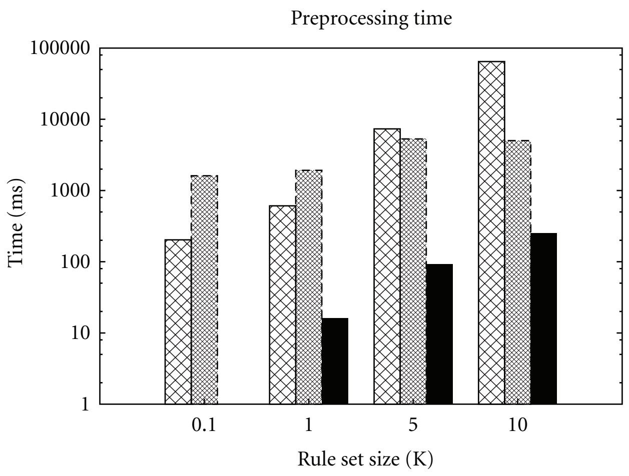
6. Techniques for Hardware Implementation

While a pure software approach can generate powerful results on a server machine, an embedded solution may be more desirable for some applications and clients. Embedded, specialized solutions are typically much more efficient in speed, cost, and size than solutions on general purpose processor systems. This paper covers two such translations for the PCIU into the realm of embedded solutions using reconfigurable technology. In the next few sections we discuss a solution developed using a hardware/software codesign methodology and another using a pure RTL approach with an ESL design methodology based on Handel-C.

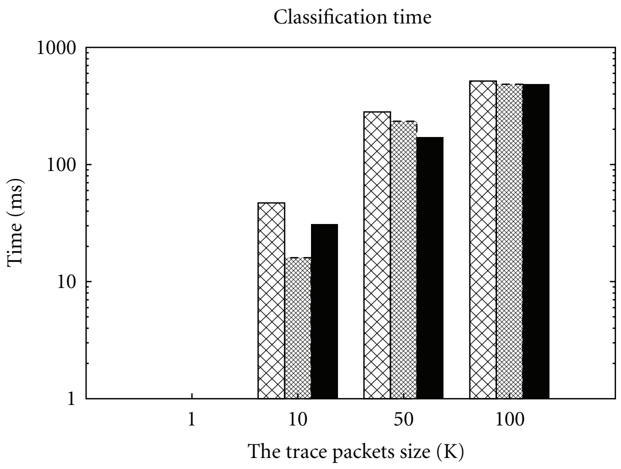
6.1. A Hardware/Software Codesign Approach. One of the key advantages of an HW/SW implementation is the incredible performance to development time ratio. While a pure RTL implementation would be much more effective in terms of satisfying constraints, it would generally take a long time to develop. Software, on the other hand, is very flexible and easy to develop and can be mapped onto any general purpose processor. Therefore, as long as an embedded system has a GPP, a software algorithm can be easily realized on the system. Furthermore, designers can often effectively tackle the bottlenecks of software by using dedicated coprocessors that are simple by design and highly modular, that is, reusable. For all of these reasons, and as the approach and underlying tools mature with time and demand, the HW/SW approach is one that is very feasible for most development groups.

Table 3: Benchmark rule sets and its traces.

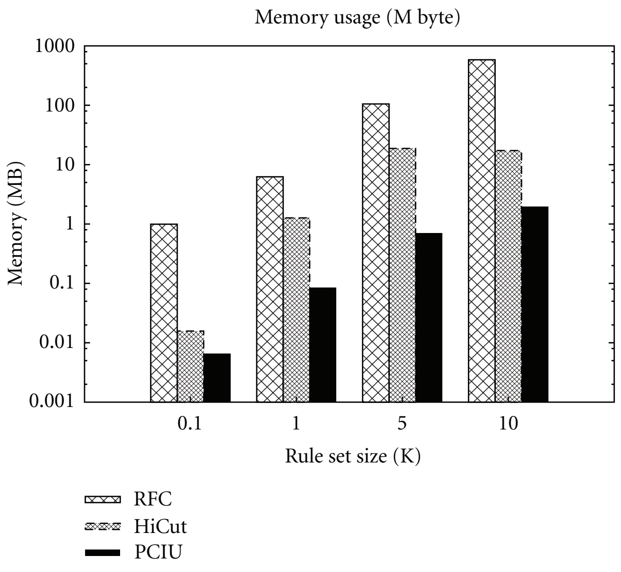
Benchmark	ACL		FW		IPC	
Size	Rule	Trace	Rule	Trace	Rule	Trace
0.1k	98	1000	92	920	99	990
1k	916	9380	791	8050	938	9380
5k	4415	45600	4653	46700	4460	44790
10k	9603	97000	9311	93250	9037	90640



(a)



(b)



(c)

Figure 9: A comparison between RFC, HiCut, and PCIU.

6.1.1. Tools and Equipment. The key development tools consisted of Xilinx ISE v11.4, Xilinx EDK v11.4, and Xilinx SDK v11.4. ISE was used to design and test dedicated coprocessors written in VHDL. EDK and SDK together facilitated an efficient and user-friendly co-design environment that interfaced both sides of the spectrum very well. EDK can generate a full processor system (either using Xilinx's Micro-Blaze core, as in the case of this specific implementation, or an integrated IBM PowerPC core) on a given FPGA using predefined "IP cores". EDK also allows designers to integrate custom cores and hardware accelerators into the aforementioned system.

In terms of the actual hardware, a Spartan3E XCS3S500E was used. The XS3S500E utilizes 10476 logic cells, 1164 CLBs, 4656 slices, 73 Kb of distributed RAM, 360 Kb of Block RAM, 20 dedicated multipliers, and 4 digital clock managers.

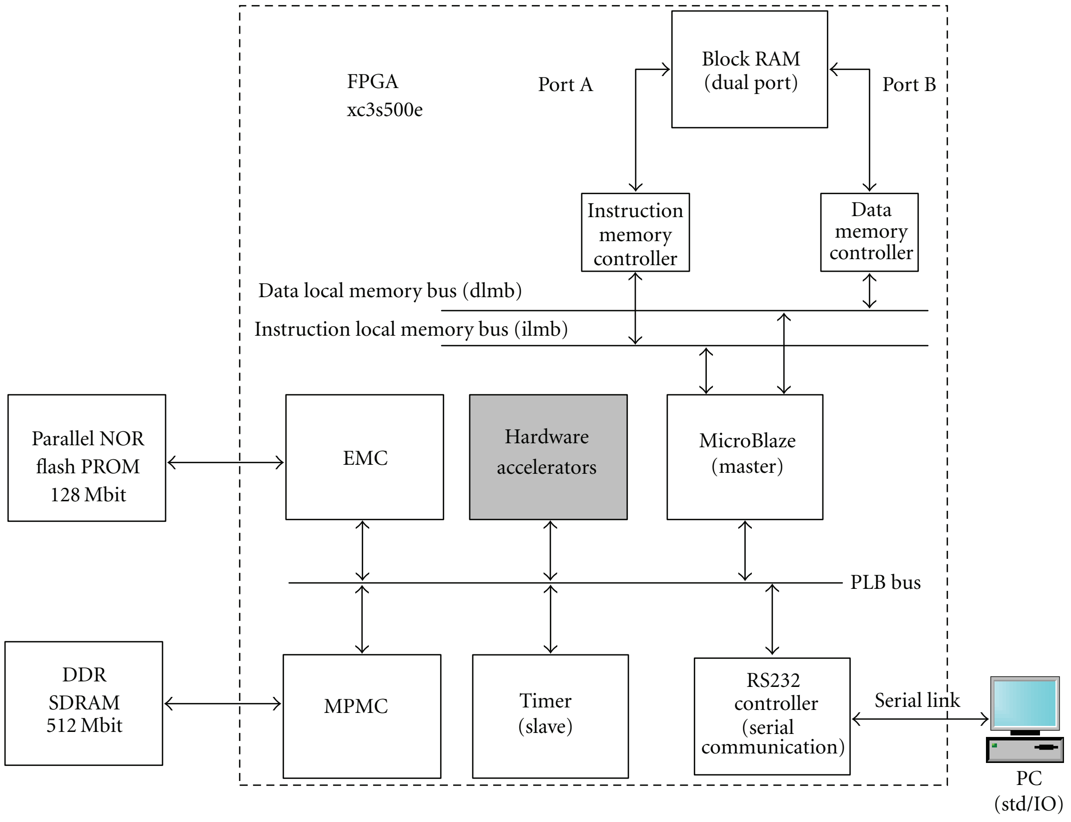


Figure 10: System configuration for Hw/Sw codesign implementation.

In addition to this, the implementation utilized 32 MB of external DDR and 16 MB of external Flash memory.

6.1.2. Techniques and Strategies. Before any optimization could be utilized, the C code of the PCIU algorithm was slightly modified as described below. The application was then executed on the MicroBlaze system that had been instantiated by the EDK. Implementing the I/O for the PCIU was performed by writing all of the necessary test bench files to Flash memory, configuring the hardware to support a Xilinx Memory File System, and modifying the software such that it read from these files. File I/O is handled by very low-level functions that resemble system calls in Unix. For output, a Hyperlink terminal collected data sent over the board's UART. In addition, less intensive functions were designed to replace standard C functions such as sscanf() and printf(). Figure 10 illustrates the MicroBlaze system with integration of RTL co-processors.

6.1.3. Optimizations. By partitioning functions into subfunctions and then further partitioning those subfunctions into smaller subfunctions, it is possible to pinpoint bottlenecks with some accuracy using the SDK's profiler. Table 4 shows a summary of the profiling performed for the PCIU algorithm.

Table 4: Profiling of the ACL10 K rule set (running on Microblaze).

Time	Function name	Functionality
27.40	match_list	Preprocessing
20.05	shift_comparison_less	Preprocessing
16.66	shift_comparison_outside	Preprocessing
12.41	list_output	Preprocessing
9.98	matching	Classification
6.72	shift_comparison_inside	Preprocessing
3.45	mfs_file_read	EEPROM Reading
1.56	rsscanf	RAM Reading
0.62	reader	EEPROM Reading
0.47	element_num	Preprocessing
0.21	preprocessing	Preprocessing
0.13	loadrule	Preprocessing
0.13	mapping	Classification
0.03	classification	Classification
	Summary	
84.05	Preprocessing	
10.14	Classification	
5.63	Others	

One of the main problems found in both classification and pre-processing was in the shift instruction. The MicroBlaze performs a single bit shift per clock cycle. For example, shifting a value by 24 bits left requires about 24 clock cycles. Because the nature of the program is divide and conquer, the "shift" and "AND" have been used intensively. In this

```
if (rising_edge(CLK)) then
      count:= conv_integer(shift);
      store1:= (To_bitvector(input1) srl count); -- srl (shift right operator)
      store2:= (To_bitvector(input2) srl count);
      actual1:=to_stdlogicvector(store1(24 to 31));
      actual2:=to_stdlogicvector(store2(24 to 31));
      if (actual1 < actual2) then
              if (((actual1 <= point) and (actual2 >= point))=true) then
                output <="00000000000000000000000000000001";
              else
                output <="00000000000000000000000000000000";
              end if;
      elsif (((actual1>= point) and (actual2<= point))=true) then
          output <="00000000000000000000000000000001";
      else
          output <="00000000000000000000000000000000";
      end if;
else
        null;
end if;
```

Algorithm 1

work we attempted to use the processor local bus (PLB) due to its efficiency in terms of cache fill, fixed/variable burst as indicated by Xilinx [19].

Classification. The main bottleneck for the classification phase of the algorithm was the ANDing of the calculated bit vectors. In software, a 13-bit AND operation (thirteen inputs, 1 bit each) was used BV_LEN times, where BV_LEN is the length of the bit vectors, for a single packet. A 13-bit ANDing co-processor was actually less efficient in this case because of the overheard required to write the input values to the module using the MicroBlaze. ANDing itself is an operation that the MicroBlaze performs well but all the same the majority of improvement for classification hinged on somehow improving this very task.

Implementing a priority AND module in hardware, however, leads back to the overhead of communication; sending 13 bits to the module would take more time than the ANDing operation itself and cancel out any benefits of translating the operation to hardware. The simpler solution was to exploit the Micro-Blaze's instruction set in such a way that it worked in the design's favor. The basic functionality of an AND operation can be defined by the following: if even a single 0 exists on the inputs, the result is 0. When ANDing a large number of bits together, one way to improve efficiency is to search for a 0 within the inputs before actual computation of the output takes place. This is normally not feasible because the operation itself would be much faster than any search routine, but at the same time, the complexity of the operation increases with more inputs. At the worst-case, this method, referred to as priority ANDing, takes much longer than blind logic ANDing. When looking at the average case, however, there stands to be a significant improvement if and only if the provided ALU for the GPP is a bottleneck for the operation. If one were to implement a software version of the priority AND on any general processor, the timing results would be the same or worse, because the general processor handles a single 13-bit operation and 12 2-bit operations in the same way. The MicroBlaze, on the other hand, seems to favor multiple small operations as opposed to a single large one. This is not only a side effect of the instruction set, but also of the fact that the MicroBlaze only has a single ALU to work with. The single ALU, then, serves to bottleneck complex operations such as a 13-bit AND. By having several sequential 2-bit ANDing operations, one can implement a software priority AND that outperforms the original implementation.

Essentially, by dividing a 13-bit AND into 12 2-bit sequential ANDs, with a comparator that checks if the output of any given stage is 0, a massive speed boost was achieved on the MicroBlaze. This same translation failed to yield anything on the PC, however.

Preprocessing. The key bottlenecks appear in the match_list function, which is responsible for populating the bit vectors in accordance to the set of all loaded rules and the currently tested chunk (of 13, and therefore, this function is called $13 \times 256 = 3328$ times per execution). This is illustrated in lines 7–13 of Figure 4. The first key area for improvement in this section is the comparison in lines 8-9 of Figure 4. Each set of comparisons is different for each chunk in that the tested point must fall within the low and high ranges of the relevant parameter of the tested rule. These low and high ranges are given a shift in order to account for the disparity between the chunk and the parameter. To simplify, recall that each rule is broken down into 13 8-bit chunks, but is also expressed in terms of parameters such as IP Source LOW and IP Source HIGH. The shift accounts for the size and positional differences between these two different schemes. The VHDL equivalent of this comparison nest was designed as in Algorithm 1.

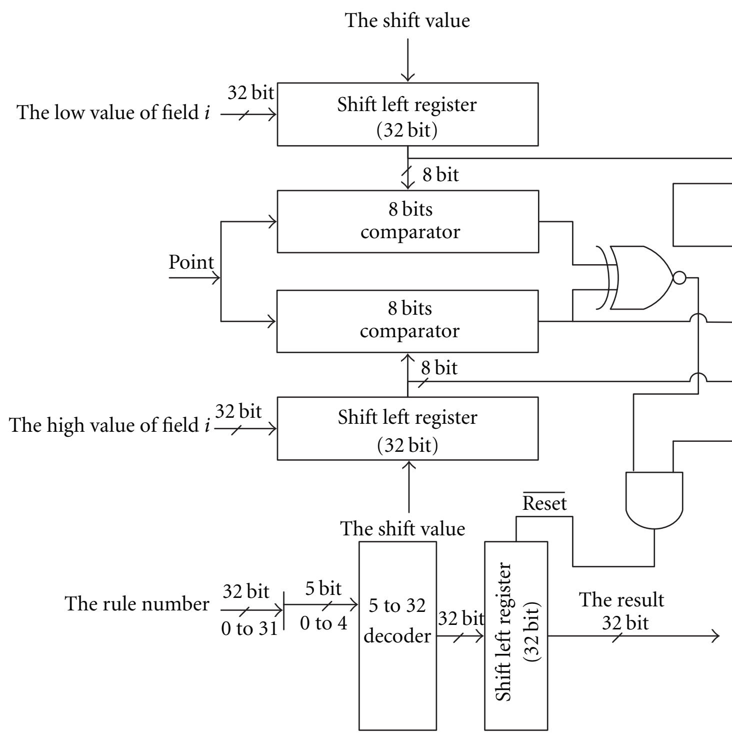


FIGURE 11: Pre processing *coprocessor*$_2$ block diagram.

One thing to note is how this hardware module interfaces with the software algorithm. The MicroBlaze is responsible for sending the LOW and HIGH ranges of the tested parameter, the shift value, and the point to be tested. This takes four sequential instructions. However, since the entire comparison function is taken care of inside the hardware module, the MicroBlaze must only read back a single value from the module. Figure 11 illustrates the RTL logic that composes this module which we refer to as *Coprocessor*$_2$. First, a comparator is used to determine if the low value of the tested field is actually less than the high value of the tested field. If true, the point is expected to fall between the low and high values when processed through a comparator. If false, the point is expected to "fall outside the range". That is, the point must be less than the labeled high field and greater than the labeled low field. The XNORs enforce this logic: they will only both transmit high if the above conditions are met. The output bits are then ANDed together and used to reset the Shift Left Register. Note that all I/O connections are between the MicroBlaze and the coprocessor.

The last area for improvement is the portion of match_list() that actually populates the bit vector. When a point satisfies the LOW and HIGH ranges of the tested rule's parameters, a value corresponding to the number of the rule OR'd with the bit vector is produced. In software, this value can easily be computed as 1 shifted to the left by the number of the rule modulus 32. That is, 1 shift_left (i%32). A much more efficient way in hardware is to use a 5-32 decoder. Decoders are especially efficient using the FPGA's resources and so this is an optimal solution. With the number of the rules fed into the module, the output is simply 1 shifted left by the input number. The entire operation requires a single write to the module and a single read from it, and the decoder design makes for an impressively fast coprocess.

6.1.4. Device Utilization and Average Speed-Up. It is obvious that the H/S implementation outperforms the pure software implementation in both preprocessing and classification. Results obtained indicate that the hardware co-processor attached to the soft processor was able to speed up the performance on average by 4.3x for preprocessing and 5.3x for classification. The MicroBlaze along with other peripherals occupied 3021 LUTs of the XC3S500E (32%). On the other hand, the complete system that includes the MicroBlaze with hardware accelerator occupied 6035 LUTs of the XC3S500E (64%).

6.1.5. Results. The final results of the HW/SW codesign implementation of the PCIU before and after adding of both coprocessors and MicroBlaze-specific optimization are shown in our paper [2]. It is clear that, while the software on the MicroBlaze exhibits a nonlinear relationship for classification with respect to ruleset complexity and packet volume, the optimized version of the software exhibits a linear relationship for classification. Not only that, but, the classification speed is faster by a factor of 5.3x. This performance boost can be entirely attributed to the reorganization of the AND in software in order to exploit the fact that the MicroBlaze only has access to a single ALU. Again, while large and complex operations (such as a 13-input AND) are bottle-necked by the lack of additional ALUs, a priority AND thrives in such an environment, because it only requires a 2-input operation at a time and the overall operation will terminate early if a 0 is computed.

It should also be noted that pre-processing received a substantial speed boost, about 4.3x speed-up, after adding the coprocessors. While the pre-processing time is still non-linear with respect to ruleset complexity, the trends are not nearly as extreme in terms of magnitude. Further optimizations may be able to make the pre-processing linear in the future.

6.2. A Hardware Coprocessor Based on ESL.

The design of a pure RTL system using an electronic system level (ESL) language is a different kind of partnership of hardware design and software design philosophies. An ESL is typically a high-level language with many similarities to software 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 very easy to translate a software application into its RTL equivalent without having to start the design from scratch. The higher level of abstraction also allows designers to much more easily and quickly develop an RTL solution than what would be possible in pure VHDL or Verilog. Admittedly, the efficiency of hardware generated by ESL is not as efficient as a VHDL or Verilog design, but the time savings tend to more than make up for this shortcoming.

6.2.1. Tools and Equipment.

The PCIU algorithm was implemented using Mentor Graphics Handel-C [20]. Handel-C is a high-level programming language which specifically targets low-level hardware and is commonly used in programming FPGAs. The Handel-C development suite is a rich subset of C with specific extensions that emphasize parallelism. Handel-C is unique, since it can be compiled to a number of design languages and then synthesized to the corresponding hardware which frees developers to concentrate on the design. The Mentor Graphics Development Kit Design Suite v5.2.6266.11181 facilitates: a thorough debugging tool-set; 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 build scripts for specific target platforms. When building an EDIF, Handel-C will also produce log files that display timing, size, and resource-use information for the design.

6.2.2. Techniques and Strategies.

Handel-C provides a file I/O interface during simulation, and so, testbench files were fed in and a output file was created with no calls to doubt. In this implementation, preprocessing of all test benches was performed by a PC well in advance using a preprocessor application written in C. This placed the focus of the RTL design solely on classification. Handel-C also provides unique memory structures in RAMs and ROMs. The access times for these structures are much quicker than variables, but the downside is that no single RAM can be accessed concurrently by multiple branches.

Handel-C's debugging mode allows one to calculate the number of cycles required by each statement during simulation. When coupled with the critical path delay of the design this offers an excellent means of calculating precise timing of the system. 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.

6.2.3. Optimizations.

Optimization in Handel-C is quite different from optimization in a co-design methodology. The approach of extracting blocks of code to hardware serves no purpose to the designer because a Handel-C implementation is already pure RTL. Instead, the key method of optimization is exploiting the fact that an RTL design has very different 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, the statements can be executed either sequentially or in parallel branches, one can "thread" an algorithm in a way no general processor system can, and special memory structures can be used for fast memory access.

(1) Fine Grain Optimization (FGO). The first step taken to improve on the original Handel-C implementation (baseline) of the PCIU was to replace all *for loops* within the design to *do-while loops*. *For loops* take a single cycle to initialize the test 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 because one can place the counter increment within the loop body and run it concurrently with the other statements. This effectively reduces the number of cycles consumed by the loop by almost half. Figure 12, illustrates the code conversion of the *FOR* to the *Do-while loops* by using the Handel C *par* statement. The for-loop in the code example of Figure 12 consumes 21 clock cycles, yet the same code implemented by do-while consumes 11 clocks cycles. The efficiency of this optimization method increases dramatically when nested loops are used. The second example in Figure 12 shows the code conversion for the nested loop from the for-loop to nested do-while where both the *par* & *seq* Handel C statements are used.

(2) Coarse Grain Optimization (CGO). One of the key methods of exploiting parallelism for the PCIU algorithm is to divide the classification phase into several parallel chunks and pipeline the trace packets through them. This is a rather simple task so 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 kind of synchronization: if a stage in the PCIU is waiting to read a value from the end of a channel, it will block its

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

/* Nested For Code*/
for(i=0;i < 10;i++)
{
for(j=0;j < 10;j++)
{
MyRam[i][j]=i*j;
}
}
	/* Nested Do While Code*/
	i=0;
	do{
	par{
	seq{
	j=0;
	do{
	par{
	MyRam[i][j]=i*j;
	j++;}
	} while(j < 10);
	}
	i++;}
	} while(i < 10);

Figure 12: A FOR to Do-While conversion code.

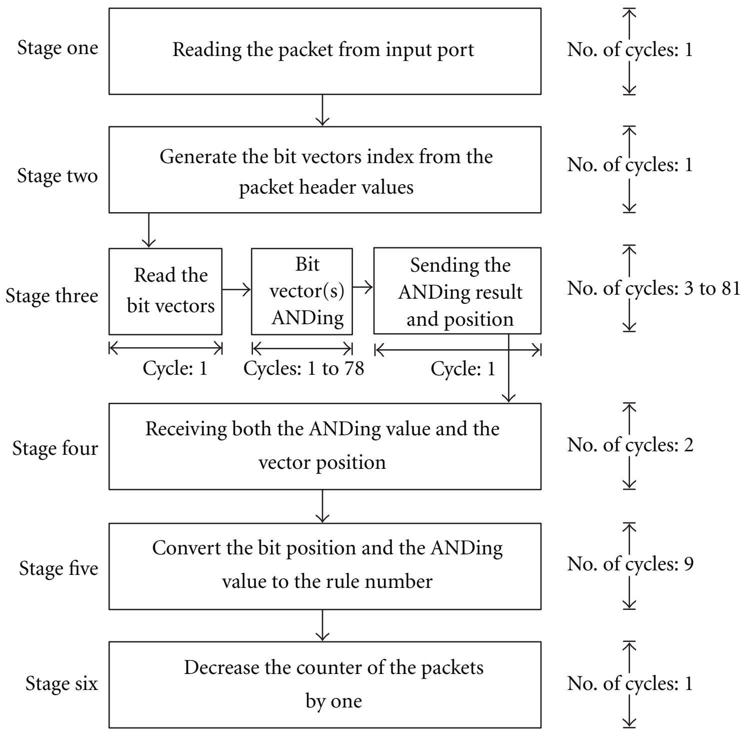

Stage one
Reading the packet from input port
No. of cycles: 1
Stage two
Generate the bit vectors index from the packet header values
No. of cycles: 1
Stage three
Read the bit vectors
Bit vector(s) ANDing
Sending the ANDing result and position
Cycle: 1
Cycles: 1 to 78
Cycle: 1
No. of cycles: 3 to 81
Stage four
Receiving both the ANDing value and the vector position
No. of cycles: 2
Stage five
Convert the bit position and the ANDing value to the rule number
No. of cycles: 9
Stage six
Decrease the counter of the packets by one
No. of cycles: 1

Figure 13: Flow of the coarse grain version of the Handel-C implementation.

execution until the data is received. This not only makes the pipeline well synchronized and organized, but also improves efficiency in terms of the number of cycles used to complete classification. Figure 13 illustrates a timing diagram for the pipelined implementation of the PCIU. Along the Y-axis, there are parallel pipeline stages. Stage 3 in particular is also divided into substages along the X-axis, because

(a) the complexity of the internals merits it,

(b) the substages needed to be executed sequentially in succession to produce accurate results.

This optimization stage is a combination of the *Fine grain* and pipe-lining technique described above.

(3) Parallel Coarse Grain Optimization (PCGO). The final strategy taken to improve the PCIU was to divide the memory space of the PCIU and split the algorithm itself into a series of parallel pipelines. Accordingly, the PCIU's preprocessing stage was altered to generate four input rule files instead of one, and the utilized RAM was also segmented into four sections. While one would expect this to maintain the same

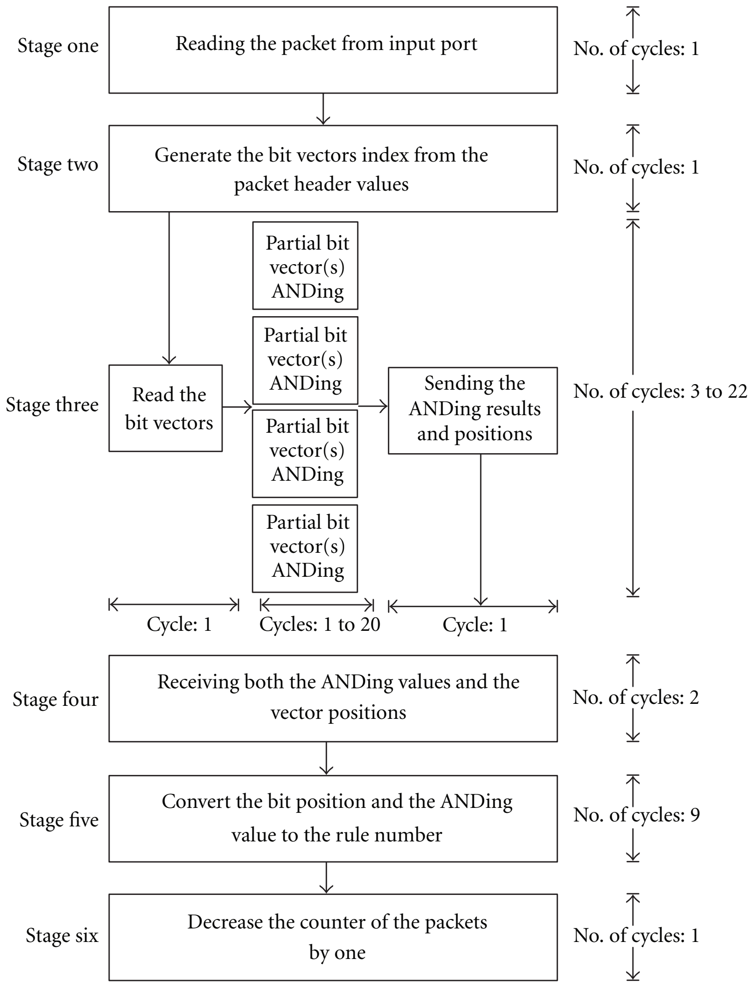

Stage one
Reading the packet from input port
No. of cycles: 1
Stage two
Generate the bit vectors index from the packet header values
No. of cycles: 1
Stage three
Read the bit vectors
Partial bit vector(s) ANDing
Partial bit vector(s) ANDing
Partial bit vector(s) ANDing
Partial bit vector(s) ANDing
Sending the ANDing results and positions
No. of cycles: 3 to 22
Cycle: 1
Cycles: 1 to 20
Cycle: 1
Stage four
Receiving both the ANDing values and the vector positions
No. of cycles: 2
Stage five
Convert the bit position and the ANDing value to the rule number
No. of cycles: 9
Stage six
Decrease the counter of the packets by one
No. of cycles: 1

FIGURE 14: Flow of the paralleled coarse grain version of the Handel-C implementation.

memory size as the normal pipelined model, 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 4 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. There is, of course, a priority scheme that dictates that only one pipeline is ever allowed to write a result to the file at any given time. As one further divides the memory, the total amount of resources must be increased for accuracy but the speed gains far outweigh the additional chip resource usage. In the next subsection, a detailed explanation of the specific tradeoffs and trends will be introduced.

6.2.4. Results. Figures 13 and 14 illustrate the execution models for both the basic pipelined implementation "coarse grain" of the PCIU and the pipelined implementation with a divide memory space "parallel coarse grain". It can be shown that both models have a best-case time of 9 cycles per packet (this is constant). The basic pipeline "coarse grain" has a worst-case time of 81 cycles while the divided memory-space approach "parallel coarse grain" has a worst-case time of 22 cycles. Both designs are incapable of processing a packet per cycle with large rule sets, but continued memory division may be able to lower the worst-case time in future endeavors. Figure 15 shows the result of all four implementations of PCIU (i.e., baseline version along with optimized versions). The resource usage in terms of equivalent NAND gates, flip-flops, and memory are presented in Figure 16.

It is clear from Figure 16 that the "parallel coarse grain" implementation with 4 classifiers consumes almost 3.9985 times more NAND gates than the "coarse grain". The same number of memory bits is used by the two designs and 1.93 times more flip-flops were required for the "parallel coarse grain" approach.

Also of note is the fact that because the pipeline itself has unbalanced loading on its stages, pipelining itself did not generate a substantial boost in speed. The largest contributors to speed-up were the fine grain and the parallel coarse grain approach. Converting all for-loops into while loops "Fine Grain" resulted in an average speed-up of 1.7x for the 10 K rule set. The coarse grain, on the other hand, only introduces an additional 1.12x speed-up over the Fine

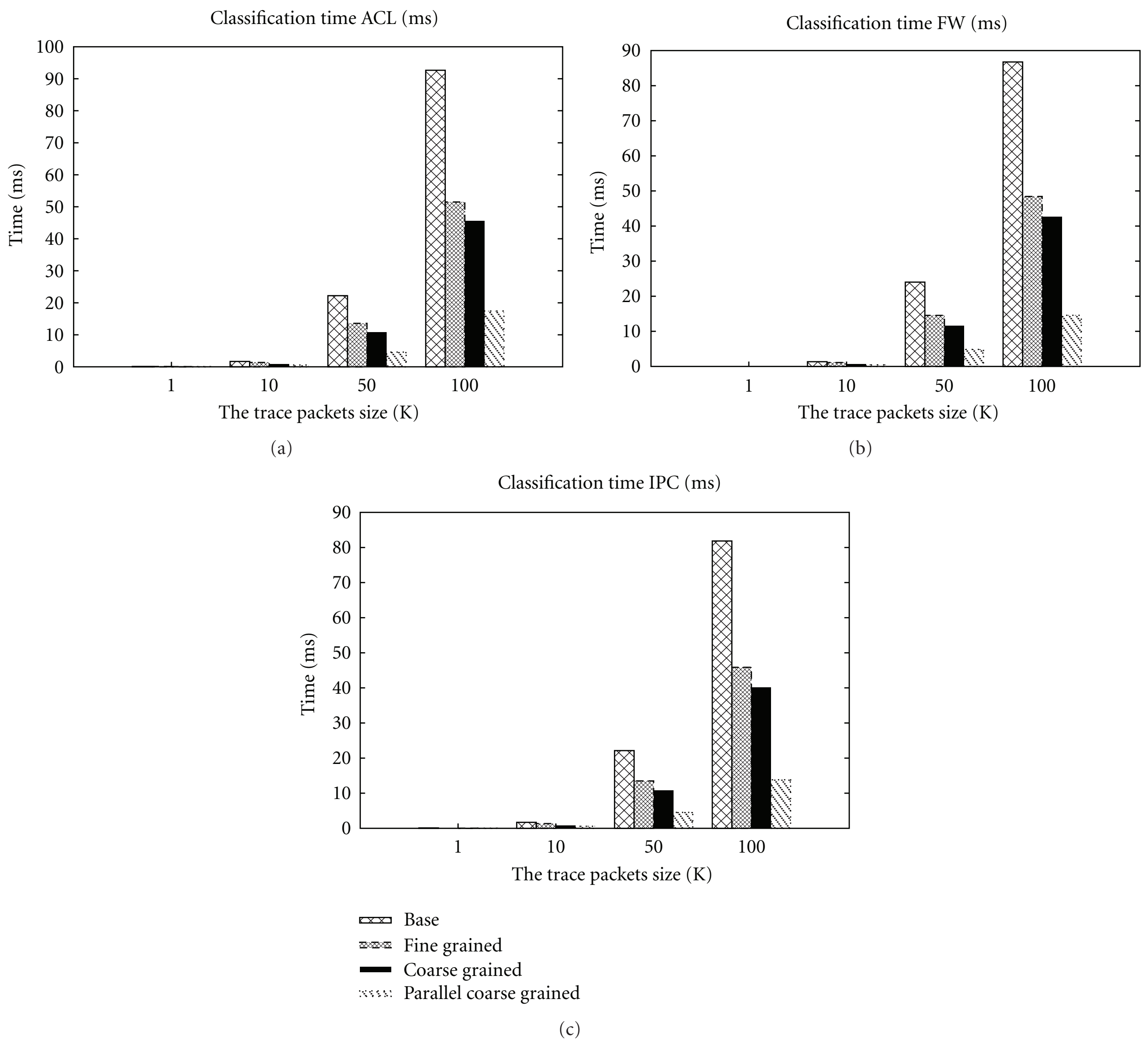


Figure 15: PCIU's Handel C implementation: timing result.

grain version and overall a 1.92x speed-up from the baseline Handel-C implementation. In contrast, the parallel coarse grain approach achieves a 2.32x speed-up over the coarse grain revision and overall a 4.44x speed-up over the baseline Handel-C implementation.

7. Discussion and Comparison of Results

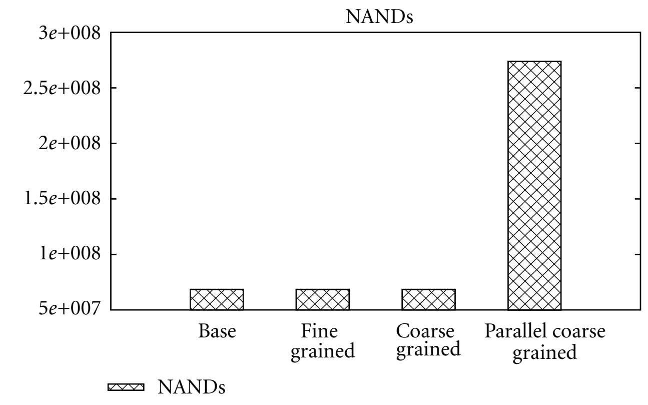
The key parameters of interest when examining the effectiveness of any implementation of PCIU are classification time, pre-processing time, and memory usage. Of the three, the one of least concern is preprocessing time, because it is a one-time operation for a larger, continuous process. However, improvement of preprocessing is still desired, and especially so in the case of the PCIU because of its incremental update capability. A smaller pre-processing time means shorter sdowntime for the system and overall a more versatile, resilient, and effective classification procedure.

Section 4.5 highlighted the effect of altering the chunk size on classification speed, pre-processing speed, and memory usage. It is clear that the PCIU algorithm has an effective range of 7 to 11. Depending on the resources available and the speed required, a designer may select the desired degree of tradeoff from this range. Chunk size 11 is of particular interest, because instead of a gradual change as exhibited along the rest of the range, there is a strong spike and a strong dip in memory usage and classification time respectively. It should be noted that all implementations mentioned in this paper utilized a chunk size of 8 bits.

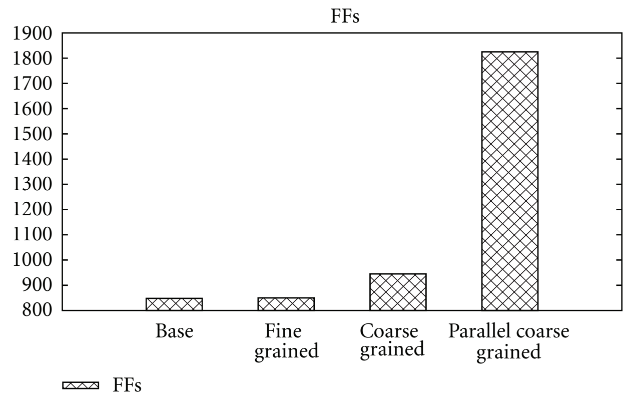
The pure software implementation on the PC has powerful results but also has a powerful general-purpose processor, several dedicated ALUs, and incredible memory resources to use. The hardware is generally not practical for anything but a server implementation, and in that respect it is not directly comparable to the embedded alternatives. However, the pure

Table 5: The performance achieved by all implementations in terms of classification/preprocessing.

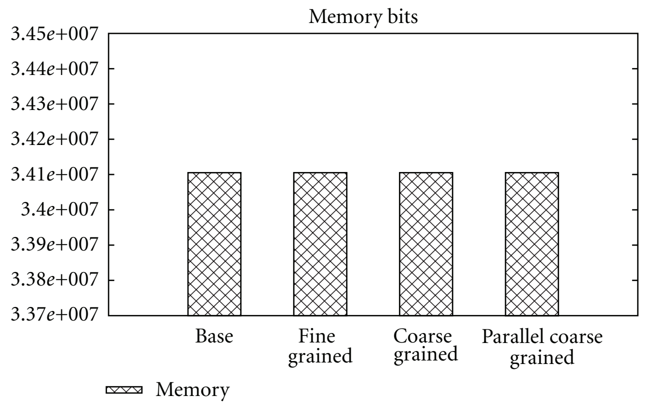
Benchmark	Classification (Packet/sec)				Preprocessing (Rule/sec)		
	MB soft	Hw/Sw	Desktop	Handel C	MB soft	Hw/Sw	Desktop
ACL (10 K)	408.23	3,787.58	200,413.22	5,565,677.97	5.92	18.41	38,412.00
FW (10 K)	453.67	4,127.66	207,264.96	6,403,975.82	5.68	18.40	40,863.83
IPC (10 K)	458.87	4,055.18	214,128.04	6,556,710.84	5.66	19.24	38,412.00
Average	440.26	3,990.14	207,268.74	6,175,454.88	5.76	18.68	39,229.28



(a)



(b)



(c)

Figure 16: Device utilization of four implementations of PCIU based on Handel C.

software implementation is perhaps the easiest to debug and modify of the three.

The two embedded implementations both have certain advantages. The HW/SW codesign implementation utilizes a general purpose softcore processor in the form of a MicroBlaze, and this allows such a design to have some degree of flexibility. Several software modules could work in tandem in this structure using run-time configuration. Perhaps a client would prefer flexibility, for example, to run their PCIU algorithm as well as a firewall in quick succession on the same system. This implementation, of course, is not nearly as fast as the pure RTL version written in Handel-C. Both systems require FPGAs but the Handel-C implementation would be the most cost effective to translate into ASIC especially considering that it is a single module. In addition, the Handel-C implementation is much more simple to pipeline due to its lack of interaction with a software component.

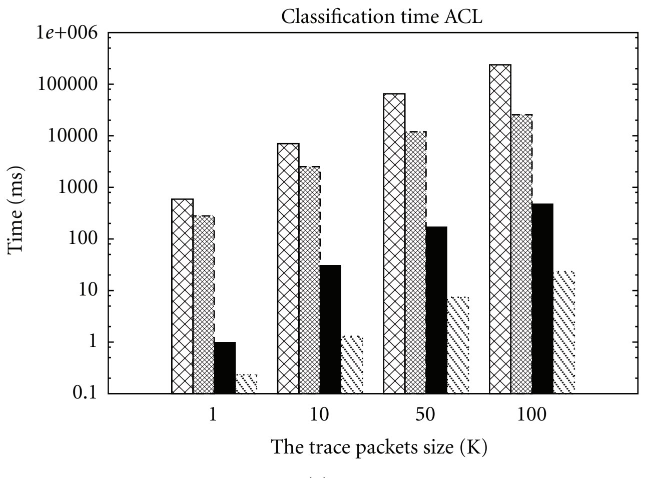
The key difference in design philosophy here is that optimizations in Handel-C come from changing the way in which one writes code as opposed to HW/SW codesign where the designer finds hot spots in the form of bottlenecks. Both are forms of hardware/software co-design in the long run, but they each ultimately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

Table 5 summarizes the performance obtained by different implementations for classification in terms of packets/sec and preprocessing in terms of rule/sec.

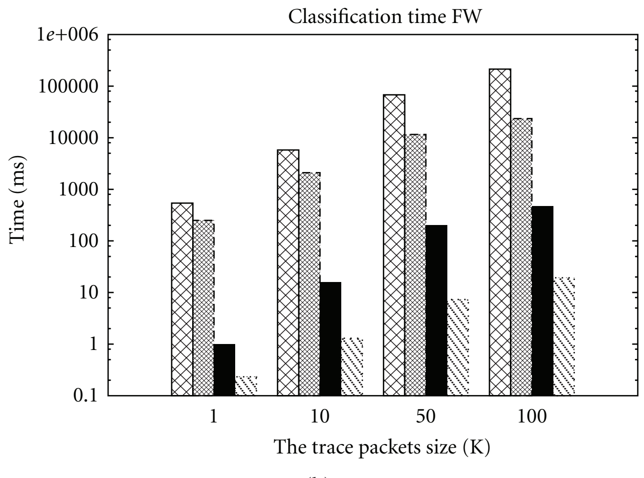
Figure 17 along with Table 6 present a comparison of the PCIU algorithm running on different platforms.

Table 6: Speedup achieved using Handel-C.

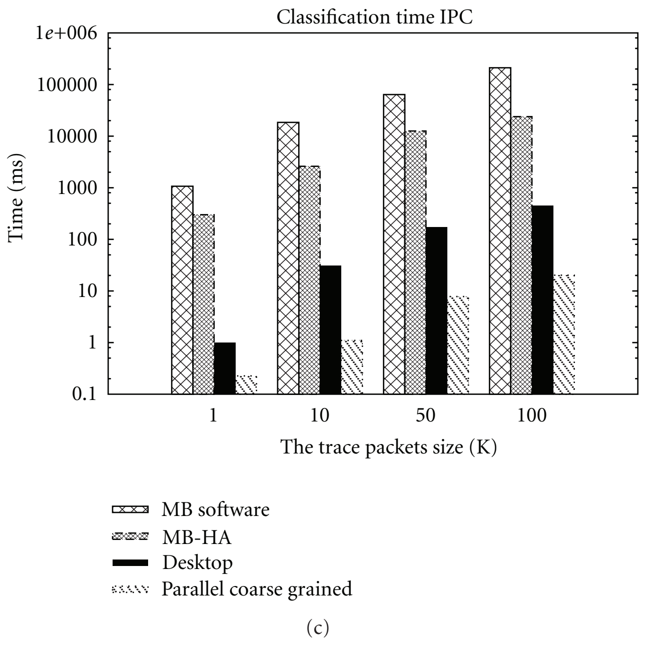
Benchmark	Time (ms)				Speed up of the Handel C (x)		
	MB soft	Hw/Sw	Desktop	Handel C	MB soft	Hw/Sw	Desktop
ACL (10 K)	237610	25610	484	17.42	13,633.62	1,469.45	27.77
FW (10 K)	213810	23500	468	14.56	14,683.48	1,613.87	32.14
IPC (10 K)	211390	23920	453	13.82	15,291.52	1,730.32	32.77
Average	220937	24343	468	15.27	14,467.59	1,594.05	30.65



(a)



(b)



(c)

Figure 17: A comparison of the classification time among "software on MB, H/S codesign, desktop, and pure RTL".

Results obtained indicate the following.

(1) PC. The vast resources and power of the PC allow it to generate the second fastest classification results of the group.

(2) MicroBlaze. The MicroBlaze is tailored to embedded system development and is itself a general purpose processor. This results in such a system being very similar in architecture to that of the PC but with far more limited resources. This system easily has the worst classification speed of all of the implementations.

(3) HW/SW Codesign. The hardware/software codesign system results in a massive classification speed-up over the

original MicroBlaze system. While it still under-performs with respect to the original PC implementation it is clear that this version is the one with the most potential for further optimization.

(4) Pure RTL Using Handel-C. The pure RTL nature of this design makes it the absolute fastest in terms of classification speed. In addition, it also has a good amount of room for optimization and this has been thoroughly shown through its various revisions. The pure RTL in terms of parallel coarse grained achieved an average of 1594.05x speed up over the MicroBlaze with hardware co-design. Moreover, it achieved an average of 31x speed-up over the desktop approach.

8. Conclusion and Future Work

PCIU is a novel packet classification algorithm with a unique incremental update capability. It has demonstrated powerful results and shown to be scalable. The incremental update capability allows it to change its rule set with minimal downtime and, therefore, continue classification at a steady rate while at the same time being very adaptive and versatile. The PCIU is also an algorithm that greatly benefits from hardware acceleration and RTL translation and achieves greater performance boosts. Most performance shortcomings with respect to other algorithms are nullified by incorporating dedicated hardware. The complete HW/SW codesign implementation was able to gain a 5.3x speed-up in classification and a 4.3x speed-up in preprocessing over its MicroBlaze only counterpart. The hardware implementation based on Handel-C was able to achieve a 4.4x speed-up after all optimizations over its original baseline counterpart. The Handel-C implementation also achieved on average a 31x speed-up over a pure software implementation running on a powerful general purpose processor. Our future work will target implementing more efficient hardware accelerators using Impulse C and comparing results with current Handel C implementation. Moreover, we intend to implement a pure RTL design based on VHDL and compare our current results in terms of the area, power consumption, and maximum clock frequency.

References

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

[2] O. A. K. Chattah 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.

[3] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '99)*, pp. 147–160, ACM, New York, NY, USA, 1999.

[4] Y. Chen and O. Oguntoyinbo, "Power efficient packet classification using cascaded bloom filter and off-the-shelf ternary CAM for WDM networks," *Computer Communications*, vol. 32, no. 2, pp. 349–356, 2009.

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

[6] T. V. 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.

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

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

[9] G. V. S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proceedings of the Conference on Applications, Architectures and Protocols for Computer Communications (SIGCOMM '03)*, pp. 213–224, ACM, New York, NY, USA, 2003.

[10] T. Y. C. Woo, "A modular approach to packet classification: algorithms and results," in *Proceedings of the 19th IEEE Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '00)*, vol. 3, pp. 1213–1222, Tel Aviv, Israel, March 2000.

[11] A. Kennedy, Z. Liu, X. Wang, and B. Liu, "New optimizer using particle swarm theory," in *Proceedings of the 18th International Conference on Computer Communications and Networks (ICCCN '09)*, August 2009.

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

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

[14] D. E. Taylor and J. S. Turner, "Classbench: a packet classification benchmark," in *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '05)*, pp. 2068–2079, March 2005.

[15] A. R. G. Jedhe, A. Ramamoorthy, 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, Palo Alto, Calif, USA, April 2008.

[16] 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, Palo Alto, Calif, USA, April 2008.

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

[18] H. Lim and J. H. Mun, "High-speed packet classification using binary search on length," in *Proceedings of the 3rd ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '07)*, pp. 137–144, ACM, New York, NY, USA, December 2007.

[19] Xilinx, "Xilinx corporation," 2010, http://www.xilinx.com/.

[20] RG, "Handel-C language reference manual," Tech. Rep., Celoxica, Europe, 2005.