Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2013, Article ID 130765, 23 pages
http://dx.doi.org/10.1155/2013/130765

Research Article

An Impulse-C Hardware Accelerator for Packet Classification Based on Fine/Coarse Grain Optimization

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

Faculty of Engineering and Computer Science, University of Guelph, Guelph, ON, Canada

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

Received 26 March 2013; Revised 10 June 2013; Accepted 10 July 2013

Academic Editor: Walter Stechele

Copyright © 2013 O. Ahmed et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Current software-based packet classification algorithms exhibit relatively poor performance, prompting many researchers to concentrate on novel frameworks and architectures that employ both hardware and software components. The Packet Classification with Incremental Update (PCIU) algorithm, Ahmed et al. (2010), is a novel and efficient packet classification algorithm with a unique incremental update capability that demonstrated excellent results and was shown to be scalable for many different tasks and clients. While a pure software implementation can generate powerful results on a server machine, an embedded solution may be more desirable for some applications and clients. Embedded, specialized hardware accelerator based solutions are typically much more efficient in speed, cost, and size than solutions that are implemented on general-purpose processor systems. This paper seeks to explore the design space of translating the PCIU algorithm into hardware by utilizing several optimization techniques, ranging from fine grain to coarse grain and parallel coarse grain approaches. The paper presents a detailed implementation of a hardware accelerator of the PCIU based on an Electronic System Level (ESL) approach. Results obtained indicate that the hardware accelerator achieves on average 27x speedup over a state-of-the-art Xeon processor.

1. Introduction

The task of packet classification entails the matching of an incoming packet with rules (established in an existing classifier) to determine the type of action that would be appropriate. Although this problem has been studied extensively, the fast emergence of new network applications, coupled with the rapid growth of the Internet, has introduced many new challenges, and the research community remains motivated to design novel and efficient packet classification solutions. Packet classification plays a crucial role for a number of network services, including, but not limited to, policy-based routing, traffic billing, and preventing unauthorized access using firewalls. Moreover, packet classification algorithms that will scale to large, multifield databases are becoming essential for a variety of applications, including load balancers, network security appliances, and quality of service filtering. Unfortunately, the current, software-based packet classification algorithms exhibit relatively poor performance, prompting many researchers to concentrate on novel frameworks and architectures that employ both hardware and software components. The continuous explosive growth of Internet traffic will ultimately require that future packet classification algorithms are implemented with a purely hardware approach.

With our recent work [1] on the packet classification problem, we proposed a novel algorithm, called "Packet Classification with an Incremental Update" (PCIU), that exhibited a substantial improvement over previous approaches. The PCIU provides lower preprocessing time, lower memory consumption, greater ease for incremental rule update, and faster classification time (when compared to other state-of-the-art algorithms), with the maximum memory required by PCIU for accommodating 10,000 rules requiring less than 2.5 MB for the worst case. In this paper, we attempt to give detailed explanation of the implementation of the PCIU algorithm that was previously published in [1] in order for the readers to reproduce the work. We are releasing the design and implementation publicly to assist those interested in implementing the PCIU algorithm along with benchmarks [2]. Furthermore, we propose enhancements to the PCIU [1] in this paper by making it more accessible, for a variety of applications, by way of a hardware implementation. Field Programmable Gate Arrays (FPGAs) are considered to be excellent platforms and candidates for mapping packet classification to hardware. FPGAs provide an excellent trade off between reprogrammability and performance compared with traditional Application Specific Integrated Circutis (ASICs). The term flexibility refers to the concept of reprogramming the FPGA as the algorithm is modified and updated. The performance on the other hand refers to the performance that can be achieved by exploiting parallelism at the bit level in addition to instruction and task level. FPGAs are considered to be a good fit for classification since the target is "embedded systems" which with typically to consume less power than current state-of-the-art general-purpose processors. In addition to reducing power consumption "Embedded systems", attempt to increase reliability and decrease operating cost. Embedded systems are specialized HW/SW computer systems, that are custom-designed to perform an often highly real-time constrained task in generally small form factor designs. An FPGA is also an excellent candidate for run time dynamic reconfiguration where only some parts of the algorithm can be present while others are swapped in and out as required. This enables any classification based algorithm to consume less power and also to fit into FPGAs with different sizes.

When mapping any algorithm onto a reconfigurable computing platform such as FPGAs, an important step involves using an appropriate language for design entry and hardware synthesis. VHDL and Verilog are two popular hardware description languages (HDL) used in both industry and academia. The main advantage of these languages is the efficient hardware produced via synthesis since they describe the hardware at the register-transfer level. However, designers consume quite a substantial amount of time dealing with structural details of the actual hardware. On the other hand, higher level languages or Electronic System Level (ESL) based languages such as Handel-C [3], Impulse-C [4], and Catapult-C [5] have started to gain popularity as an alternative to VHDL and Verilog for the purpose of hardware acceleration of software-based applications. One of the goals of these languages is to enable designers to focus their attention at higher level of abstraction; that is, on the algorithm to be mapped into hardware rather than the lower level details of the circuit to be built. Designers can start with automatic compilation and then focus their efforts on improving loops and constructs to further enhance the performance of the hardware accelerator. The designer's required effort is, therefore, reduced in many cases to simply restructuring the code or embedding simple compiler directives.

In this work we implement the PCIU algorithm [1] using Impulse-C [4]. Impulse-C is compatible with standard ANSI C, allowing standard C tools to be used for designing and debugging applications targeting FPGAs. The Impulse-C compiler accepts a subset of C and generates FPGA hardware in the form of hardware description language (HDL) files. Impulse-C allows embedded systems designers and software programmers to target FPGA devices for C-language application acceleration. One of the main disadvantages of Impulse-C is the loss of the fine-grained control over the resulting hardware. Therefore, designers are encouraged to manually restructure their code to optimize the resulting hardware. Typically, this is done by applying various transformations to the original source code. However, the sheer volume of these language-level transformations leads to a whole design space of potential solutions, all based on different optimizations. The main goal of this paper is (i) to improve the run-time performance of the PCIU packet classification algorithm by parallelizing the algorithm and eventually mapping it onto an FPGA, and (ii) to perform an empirical study to determine the overall effectiveness of different language-level transformations when using Impulse-C to implement the PCIU algorithm.

The main contributions of this paper can be clearly stated as follows.

(1) The majority of networking applications are beyond the capabilities of general-purpose processors since current networking trends are pushing towards complex protocols that provide additional and improved network services. The Impulse-C based implementation proposed in this work achieves substantial speedup (27x) over a pure software implementation running on a state-of-the-art Xeon processor.

(2) An extensive experimental analysis is performed in which all possible combinations of optimizations are considered. To the best of our knowledge, this is the first paper to propose such extensive exploration for fine-grained optimization of Impulse-C. The exploration performed can be easily extended to similar applications that utilize ESL based approaches.

(3) In addition to a full factorial experiment that will allow us to test interactions between different combinations of language-level transformation based on fine grain optimization (FGO), the authors seek to further improve performance via Coarse Grain (CGO) and Parallel Coarse Grain Optimization (PCGO) by exploiting both data parallelism and pipelining.

The remainder of this paper is organized into six sections. Section 2 provides an overview of the packet classification problem, along with necessary background. Section 3 provides a brief overview of the most significant work published in the field of packet classification. In Section 4, the PCIU algorithm [1] is described briefly along with the different stages of preprocessing, classification, and updating. An ESL based implementation of the PCIU algorithm, using Impulse-C, is described extensively in Section 5. Section 6 then provides a comparison between the Impulse-C implementation and the pure software version running on a state-of-the-art Xeon processor. Conclusions and future directions are finally presented in Section 7.

2. Background

The principle objective of a packet classification algorithm is to match the information contained in a packet's header to a set of rules to determine how the packet should be processed.

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)	
	Chunk# 0 : 1 : 2 : 3	Chunk# 4 : 5 : 6 : 7	Chunk# 8 : 9	Chunk# 10 : 11	Chunk# 12
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

A typical IP packet is usually classified according to the first five fields in the packet header, and Table 1 exemplifies a classifier that specifies five rules against which packet headers could be matched. The first and second fields are each 32 bits in length and correspond to the network layer addresses (the source IP and destination IP addresses, resp.). The third and fourth fields are each 16 bits in length and correspond to the transport layer addresses (the source and destination ports, resp.). The fifth and final field is 8 bits in length and corresponds to the protocol. It should be noted that although the transport layer address fields are specified as ranges, the network layer address and protocol fields are specified by way of a prefix. As a clarifying example, for a network layer address 0.83.4.0/22, let $A = 0.83.4.0$ and let M be a 22 bit mask equivalent to 255.255.252.0, the lower bound of the range is the result of a logical "AND" operation of A with M, and the upper bound of the range is the result of a logical "OR" operation of the lower bound with $2^{32\text{-Mask}}-1$. For the preceding example, the range specified by the 22 bit mask would extend from 0.83.4.0 to 0.83.7.255. When the mask is specified to be exactly 32 bits in length, a network layer address is specified precisely by the field. Contrarily, for shorter masks, the classifier can specify a large range of addresses and could conceivably be matched quite frequently. Given that the task of matching a packet header to a set of rules would clearly entail a complex, multidimensional search, support for large rule sets can be difficult to obtain with a purely software approach. This has led many researchers to explore reconfigurable computing approaches and, more specifically, FPGAs.

2.1. Reconfigurable Computing. Reconfigurable computing [6] is an innovative approach that attempts to cope with the inefficiency of conventional computing systems and has demonstrated significant potential for the acceleration of general-purpose computing. More specifically, application specific properties, such as parallelism, data granularity, and a regularity of computation can be exploited by reconfigurable computing approaches through the creation of pipelines, custom operators, varying bit widths (compared to the fixed width of general-purpose processors), and interconnection pathways. At the heart of reconfigurable computing is the SRAM-based FPGA, providing fine grained logic and interconnection elements that exhibit a function and structure that users can customize to suit the specific requirements of an application. FPGAs are designed to provide hardware designers with the near flexibility of software at almost ASIC hardware speeds. Flexibility in the context of FPGAs refers to the ability of designers to program and reprogram the device as the algorithm is modified over time. Using traditional HDLs such as VHDL or Verilog limits the flexibility of such devices since the compile time of such devices (place and route) takes a substantial amount of time compared to compiling software programs in traditional general-purpose processors. However, raising the abstraction level by using Electronic System Level (ESL) tends to reduce development times since verification of the designs can be achieved much faster.

Because of its capacity for reconfiguration and massive parallelism, FPGA technology has become an attractive option for implementing real-time network processing engines. State-of-the-art FPGA devices, such as the Xilinx Virtex-7, provide large amounts of on-chip dual-port memory and a high clock rate with configurable word widths. However, if an FPGA device does not contain enough Block RAM then designers have to resort to platforms with external memory to accommodate the architecture and this will affect the speed expected of the hardware accelerator due to communication overhead. FPGA logic devices can achieve high levels of performance if they are used to implement custom, algorithm-specific circuits to accelerate the overall execution speed of the algorithm. These systems remain flexible because the same custom circuitry for one algorithm can be reused as the custom circuitry for a completely separate (and different) algorithm.

2.2. Electronic System Level. Designing hardware accelerators based on the ESL [7] is considered a different form of partnership between the hardware 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 between such an implementation and a pure software implementation 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 HDL equivalent without the need to start the design process from scratch. The high level of abstraction also allows designers to develop HDL solutions more quickly and easily than what would be possible in a pure hardware description language such as VHDL or Verilog. Although the efficiency of the hardware that is generated by an ESL is generally less than that which can be achieved using VHDL or Verilog, decrease in development time will often justify the trade-off. Furthermore, most C programmers should be able to create effective ESL hardware designs with little additional training instead of investing the time to master VHDL and Verilog; the programmer can, instead, take advantage of the long-standing and widespread foundation of C.

3. Related Work

As stated previously, a considerable body of research has been invested in the development of packet-classification algorithms. The need for a standard approach to performance evaluation was addressed in [8], where the authors introduced a suite of benchmarking tools entitled ClassBench, that has since then been frequently employed by researchers for evaluating novel 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 accesses required), a lack of generalizability (in order to exploit certain features of a specific type of rule-set), and a large need for preprocessing. In addressing the first of these shortcomings, the authors in [9] proposed an algorithm that would employ binary search on prefix length, reducing the average number of memory accesses necessary to between 18 and 67 (for rule sets of about 5000 rules). In an alternative effort to improve performance (with respect to speed), the authors in [10] made the assumption that the number of distinct overlapping regions would be low even when the number of rules is high. Although the performance improvement was noticeable, the features exploited were specific to the networks being considered, limiting the generalizability of the approach.

To achieve the flexibility of software at speeds normally associated with hardware, researchers frequently employ reconfigurable computing options using 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 ternary content addressable memory (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 [11] 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). A 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 design were to be implemented on Virtex-5 FPGAs. In their development of a range reencoding scheme that fits in TCAM, the authors of [12] 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 [12] (i.e., the treatment of the reencoding as though it were a topological transformation process between hyperrectangles) significantly outperforms previous re-encoding 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 reencoded classifier and its transformers. Another interesting range encoding scheme to decrease TCAM usage was proposed in [13], with ClassBench being used to evaluate the proposed scheme. The encoder proposed in [13] 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 register transfer level (RTL) hardware approaches proposed by many researchers [14, 15]. Although the dual-port IP lookup (DuPI) SRAM-based architecture proposed in [14] 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 [15], on the other hand, proposed a five-dimension packet classification flow, based 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 [15, 16]. The interesting architecture proposed in [16] 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 [15] 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 targeting reconfigurable computing platforms (mapped on FPGAs) have been published in recent years [17–21]. In [17] 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. In [21], a novel algorithm (GBSA) is proposed. The GBSA was evaluated and compared to several state-of-the-art techniques including RFC, HiCut, Tuple, and PCIU. Results obtained indicate that the GBSA outperforms these algorithms in terms of speed, memory usage, and preprocessing time. The GBSA algorithm introduced in [21] was mapped into hardware using different ESL flows (Impulse-C and Handel-C). The resulting hardware accelerator based on the above mentioned ESL techniques achieved on average 9x speedup.

The authors of [18] proposed a multifield packet classification pipelined architecture called Set Pruning Multibit Trie (SPMT). The proposed architecture was mapped onto a Xilinx Virtex-5 FPGA device and achieved a throughput of 100 Gbps with dual-port memory. In [19] 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 [20] 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 to sustain 40 Gbps throughput. However, the authors evaluated their architecture using only the ACL benchmark from ClassBench.

From the previous discussion it is clear that there are several deficiencies in the current published algorithms. The proposed PCIU algorithm clearly distinguishes itself from other algorithms published in the past. The PCIU attempts to improve speed via clustering and also efficient hardware implementations. The PCIU also can accommodate efficient incremental updates to the rule set which is a key feature to support session based packet classification which is lacking in many published algorithms.

4. The PCIU Algorithm

A taxonomy for packet classification algorithms was introduced in [22] and specifies that these algorithms can be categorized according to whether or not they are based upon an exhaustive search, a decision tree, a decomposition approach, or a Tuple space. Since the PCIU was designed as a "divide-and-conquer" approach, our algorithm should be categorized as a decomposition based approach. The following subsections will briefly describe the three stages of the PCIU algorithm: preprocessing, classification, and incremental update, and will also describe our experimental design and benchmarking approaches.

4.1. The Preprocessing Phase. The PCIU algorithm [1] is based on the simple notion that there is a redundancy that exists in the rule set as it is divided into chunks. For a five-dimensional packet classification problem, similar to that which is depicted in Table 1, a hierarchical approach is used, with the main classification task being decomposed into subproblems and with the final results recombined at the end of the process. The principle behind the PCIU algorithm can be summarized in the following steps. The first two steps involve converting the rule set from different presentations to range presentations. The last two steps ((3) and (4)) are more related to the detailed example presented in the next subsection.

(1) The rule set is converted from the different possible representations to a range representation, and an upper bound and a lower bound on the range associated with each of the five dimensions (or fields) in the rule set are computed.

(2) Each of the five dimensions is divided into 8-bit chunks. Since the rule size for this example would be 104 bits, the total number of chunks would be 13. The range associated with each chunk (designated chunk$_i$) would be (0 to 255), and, for each chunk, a lookup table (designated table$_i$) of size 2^8 is assigned.

(3) A group of equivalent rules is then generated. The goal is to exploit the overlap of rules (see Table 1) and then generate clusters or groups of rules which share similar features.

(4) The groups are converted to a binary vector, where the bit locations correspond to the rule ID. This information can be used in the classification phase.

4.2. A Detailed Preprocessing Example. In this subsection, we present an example to clarify how preprocessing is performed in our approach by PCIU. The example is based on the information for chunk #10 found in Table 1. The overall procedure is illustrated in Figure 1. Note that since the focus is on a single, 1-byte chunk, the range used for representing the rules is limited to values between 0 and 255.

Step 1. For each rule listed in Table 1, create a table entry of the form (rule, range).

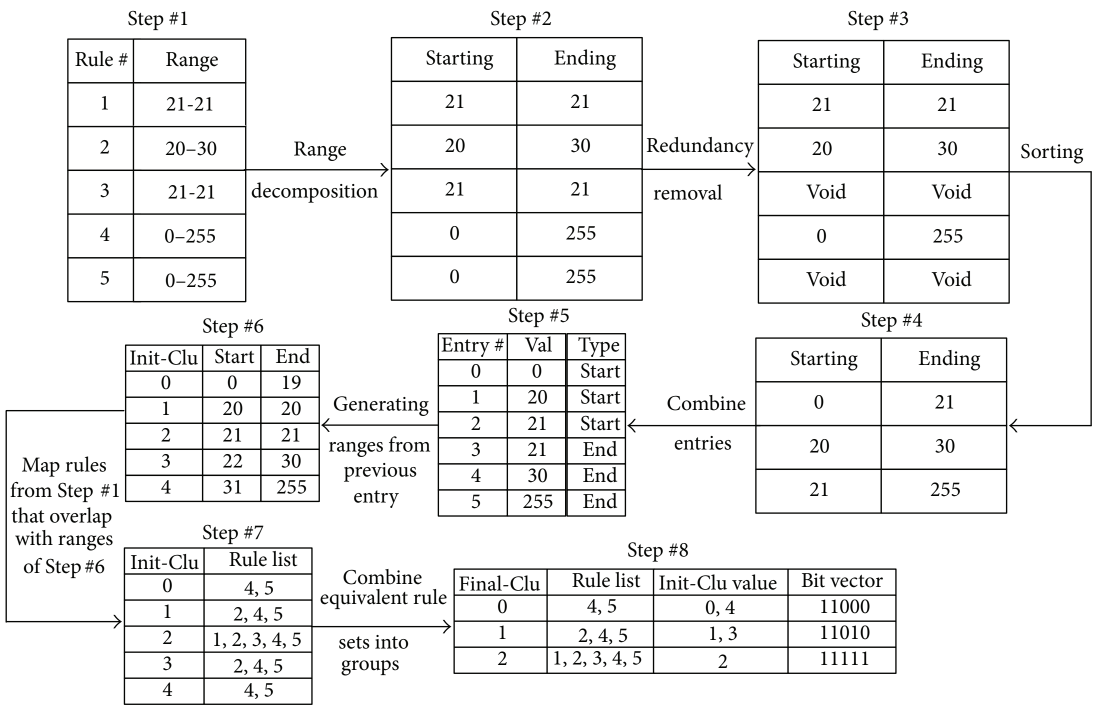
Step 2. For each table entry in Step 1, create a new table entry of the form (start, end) by decomposing the range into "starting" and "ending" values.

Step 3. For each table entry in Step 2, identify any duplicate entries. For example, rules 1 and 3 are represented by the same start and end values: (21, 21). Therefore, the table entry associated with rule 3 can be considered redundant. This is indicated in the table by the use of the keyword "void" in the row corresponding to rule 3. A similar situation exists for rules 4 and 5, which are also both represented by the same start and end values: (0, 255).

Step 4. First, remove all redundant table entries (marked void) identified in Step 3. Then sort into ascending order the remaining values in the "Start" and "End" columns, respectively.

Step 5. Merge (in ascending order) the values in the "Start" and "End" columns (found in Step 4) into a single (ordered) column. Then mark each value as either "Start" or "End."

Step 6. Use the "Start" and "End" values (in column type) from Step 5 to create a new set of clusters which we will refer to as Init-Clu. In practice, this is done by working from the top of the table towards the bottom of the table of Step 5 and using the information in consecutive rows of the table to generate the new "Start" and "End" values of the table in Step 6. For example, entry 0 of Step 5 has a "Start" value of 0, while entry 1 has a "Start" value of 20. These can be combined to form


Step #1
Range decomposition
Step #2
Redundancy removal
Step #3
Sorting
Step #4
Combine entries
Step #5
Generating ranges from previous entry
Step #6
Map rules from Step #1 that overlap with ranges of Step #6
Step #7
Combine equivalent rule sets into groups
Step #8

Step #1

Rule #	Range
1	21-21
2	20–30
3	21-21
4	0–255
5	0–255

Step #2

Starting	Ending
21	21
20	30
21	21
0	255
0	255

Step #3

Starting	Ending
21	21
20	30
Void	Void
0	255
Void	Void

Step #4

Starting	Ending
0	21
20	30
21	255

Step #5

Entry #	Val	Type
0	0	Start
1	20	Start
2	21	Start
3	21	End
4	30	End
5	255	End

Step #6

Init-Clu	Start	End
0	0	19
1	20	20
2	21	21
3	22	30
4	31	255

Step #7

Init-Clu	Rule list
0	4, 5
1	2, 4, 5
2	1, 2, 3, 4, 5
3	2, 4, 5
4	4, 5

Step #8

Final-Clu	Rule list	Init-Clu value	Bit vector
0	4, 5	0, 4	11000
1	2, 4, 5	1, 3	11010
2	1, 2, 3, 4, 5	2	11111

Figure 1: Preprocessing phase: steps of processing chunk #10 in the rule set of Table 1.

a new Init-Clu starting at 0 and ending at 19. Note that it is necessary to stop at 19, since 20 is a "Start" value not an "End" value. As a further example, consider entries 3 and 4 of Step 5. Entry 3 has an "End" value of 21, while entry 4 has an "End" value of 30. Since entry 3 is marked as an "End" value, it can be used to create a new Init-Clu that starts at "21," one more than 20.

Step 7. All Init-Clu formed in Step 6 will be preserved in Step 7. In this step the preprocessing stage will attempt to map rules from Step 1 to Init-Clu in Step 6. This is achieved by grouping all rules identified in Step 1 that overlap with the new "Start" and "End" Init-Clu formed in Step 6. For example, rules 4 and 5 (0 to 255) from Step 1 overlap with the "Start" and "End" values (0 and 19, resp.) of Init-Clu with index 0 in Step 6, thus giving rise to this Init-Clu in Step 7.

Step 8. Remove any redundant rule sets found in Step 7. A rule set is considered redundant if it obviously repeats.

(1) The number of *Final-Clu* created by this step is a function of the original rule set existing and also the overlap between the rules. The *Final-Clu* created in this step is a simple postprocessing step that attempts to combine all the Init-Clu formed in Step 7 that overlap in the rule list. It is important to notice that the total number of *Final-Clu* created in this step is three which is less than the total number of original rules (i.e., 5) found in Step 1. The newly created *Final-Clus* are stored in a memory (Mem$_1$).

(2) A bit vector contains all rules in the newly created cluster. The length of the bit vector corresponds to the original rule set size. In this example the original rule set contained 5 rules and therefore each bit vector has to be of length 5. A bit vector is created to signify the values belonging to the new *Final-Clu* formed in Step 8. These bit vectors are stored in yet a second memory (Mem$_2$). For example, since rules 4 and 5 are now part of *Final-Clu* with index 0, the bit vector 11000 is created.

The complexity of preprocessing a set of N rules is $\Theta(N)$ [1]. As a result of the steps described in the preprocessing phase, two memories (Mem$_1$ and Mem$_2$) will be populated with all the necessary information as shown in Figure 2. This information is used in the classification phase.

4.3. Classification Phase.

Following the construction of the 13 lookup tables and the corresponding bit vectors (as per the previous example), the lookup tables are ready for the process of classification, as seen in Figure 2. An incoming packet header is first divided into 13 chunks (of eight bits) and each is used as an address for its lookup table (Mem$_1$). Each lookup table (Mem$_1$) points to a specific bit vector (Mem$_2$) and, as a result, for a rule set containing N rules, 13 bit vectors of size N will be obtained. It follows then that the application of a logical "AND" operation to these vectors will produce the matched rule for the arriving packet.

4.4. Incremental Update Phase.

This phase can be further subdivided into two tasks—adding a new rule and deleting an existing rule. The former represents one of the main features of the PCIU algorithm, namely, that the algorithm

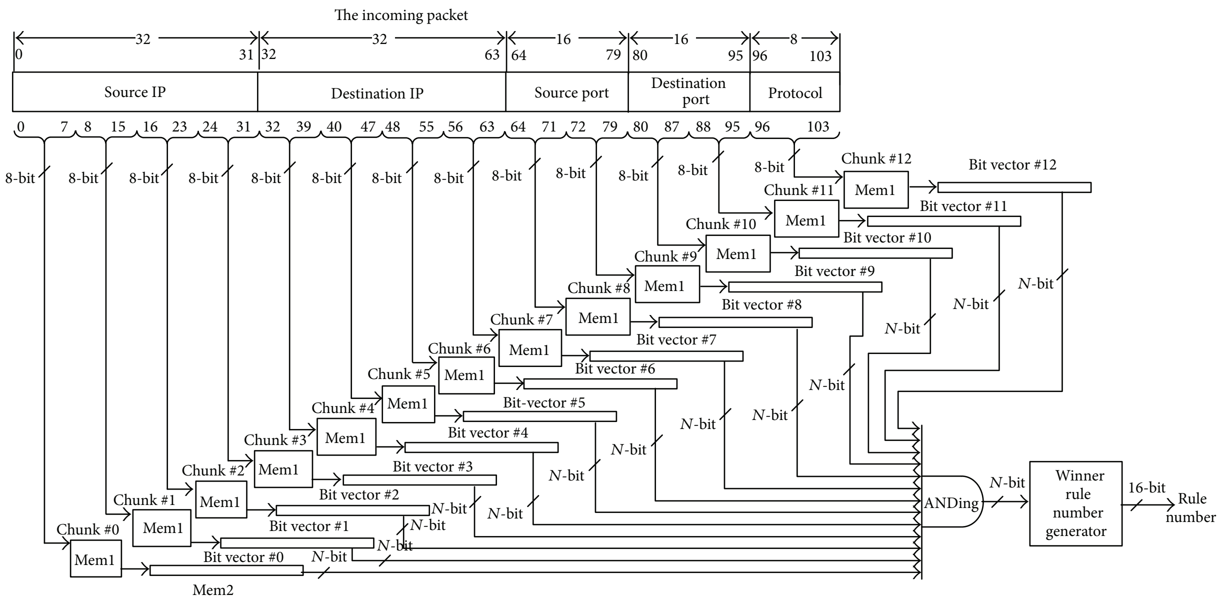


Figure 2: Classification phase of the PCIU algorithm.

can accommodate efficient incremental updates to the ruleset. It is, however, necessary to specify a design constraint on the capacity of the system (in terms of the number of rules). Although this value cannot be exceeded during the update phase, it is relatively simple to introduce a new rule through the following process.

(1) Find the first empty location in the rule set (designated Rule_{ID}) and insert the new rule at that location.

(2) Divide the new rule into 13 chunks of eight bits each.

(3) Assign a lookup table for each chunk. Each chunk will represent a range within the lookup table.

(4) For the range associated with each chunk, apply a bitwise logical OR operation of all bit vectors in this range with 2^{RuleID}.

The latter task (i.e., the deletion of an existing rule) can also be easily accomplished, according to the following process.

(1) Mark the location in the rule set (again designated as Rule_{ID}) as empty.

(2) Divide the rule to be deleted into 13 chunks, defining the ranges according to the same method described previously.

(3) For every bit vector in each range (from the lower bound to the upper bound), apply a bitwise logical "AND" operation between the vector and the complement of (2^{RuleID}).

Incremental updates (adding and/or deleting rules) might lead to internal fragmentation in the memory. Since preprocessing in PCIU is a fast and efficient task, fragmentation is not considered to be an issue since the packet classification engine can be reset and restarted. However, incremental update may cause some delay in the classification process since adding or deleting rules might take several clock cycles which will affect the real-time constraints imposed on the system.

4.5. The Experimental Design and Benchmarking Approaches. For our experiment we used ClassBench [8] as the source for the rule sets. These benchmarks have been used extensively by many researcher in the past to evaluate their proposed packet classification algorithms along with their hardware implementations. The advantage of using these specific benchmarks is to compare our proposed current work with previous work published in the literature. These benchmarks not only reflect real life scenarios but also have different characteristics since they were designed for miscellaneous applications. The amount of overlap regions varies from small to high which causes challenges for any new developed classification algorithm.

ClassBench consists of three tools, (i) Filter Set Analyzer, (ii) Filter Set Generator, and (iii) Trace Generator. Table 2 shows the size of each rule set and the trace file (i.e., testing packets) associated with it. The seeds used by filter sets to produce rule sets and the programs used to generate these rule sets are also taken from [8].

ClassBench was used to perform a battery of analysis on 12 real filter sets, provided by several Internet Service Providers (ISPs), a network equipment vendor, and other researchers working in the field. Each of these filter sets uses one of the following formats.

(1) *Access Control List (ACL).* This format is a standard for security, VPN, and NAT filters for firewalls and routers (enterprise, edge, and backbone).

(2) *Firewall (FW).* This is a proprietary format for specifying security filters for firewalls.

Table 2: Benchmark rule sets and traces.

Benchmark	ACL		FW		IPC	
Size	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

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

The PCIU algorithm was evaluated and compared to state-of-the-art techniques such as RFC and HiCut using several benchmarks in [1]. Results obtained indicate that PCIU outperforms these algorithms in terms of speed, memory usage, incremental update capability, preprocessing time, and classification time.

5. The Proposed Hardware Accelerator

While a pure software implementation can generate powerful results on a server machine, an embedded solution may be more desirable for some applications and clients. Embedded, hardware based specialized solutions are typically much more efficient in terms of speed, cost, and size than solutions that are implemented on general-purpose processor systems. This paper seeks to explore the design space of translating the PCIU algorithm into hardware by utilizing several optimization techniques, ranging from fine-grain to coarse grain and parallel coarse grain approaches. The methodology is presented in the following subsections and fully discloses the implementation tools, techniques, strategies for optimization, and results.

5.1. Experimental Setup and Impulse-C. Impulse-C [4] is a powerful ESL language that supports the development of highly parallel, mixed hardware/software algorithms and applications. It is an extension of ANSI C using C-compatible predefined library functions, with support for communicating process parallel programming models. Although these extensions are minor in terms of additional data types and predefined function calls, they allow multiple parallel program segments to be described, interconnected, and synchronized [23].

The Impulse-C CoDeveloper Application Manager Xilinx Edition Version 3.70.a.10 was used to implement the PCIU algorithm. Impulse-C CoDeveloper is an advanced software tool enabling high-performance applications on FPGA-based programmable platforms. In addition to its ability to convert a C-coded algorithm to HDL, CoDeveloper Application Manager also provides CoValidator tools to generate all necessary test bench files. These testing files can be run directly under ModelSim hardware simulation tools. The CoValidator provides simulation and HDL generation for design test and verification. Impulse CoDeveloper like many other ESLs provides verification capabilities to analyse parallel data flow

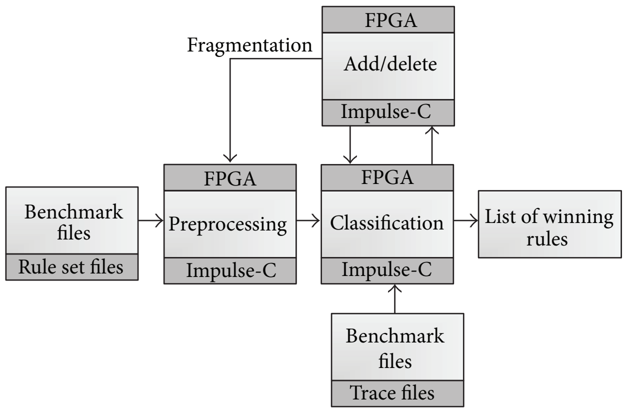


Figure 3: An overview of Impulse-C system (preprocessing and classification).

within any application. Even though the Impulse-C CoDeveloper is capable of speeding up the behaviour simulation of the application, since it is applied at a higher level of abstraction, it fails to provide accurate cycle simulation which is provided by tools such as Mentor Graphics ModelSim.

Figure 3 depicts the overall PCIU Impulse-C system organization. The CoDeveloper tool is used to feed the system with the preprocessed rule set and test packets and the Impulse-C development tools generate all of the files needed to synthesize and simulate the project using ModelSim. The ModelSim tool is used to determine the required number of clock cycles to classify the test packets during simulation. The overall architecture of the original PCIU algorithm is translated into hardware and mapped onto an FPGA using the Impulse-C CoDeveloper along with the Xilinx synthesis tools.

Since the graphical Stage Master Explorer tool performs an analysis of the Impulse-C implementation, it represents a rapid method by which design performance can be examined. This tool can be used to determine, on a process-by-process basis, the effectiveness of the compiler at the parallelization of the C-language statements. An estimate of the critical-path delay for the design is used to provide the user with a general idea of the system timing (or performance) during the implementation phase. Both the critical-path delay and the clock-cycle count play a critical role in optimizing the Impulse-C design. In our experiments, Xilinx ISE v12 was used to map the generated HDL files to a Virtex-6 (xc6vlx760) FPGA chip. The place and route reports and synthesis reports are both used to determine the critical-path delay, and ModelSim SE 6.6 is also used to simulate and count

Table 3: Preprocessing phase: number of clock cycles and overall time.

Size	Benchmark						Average	
	ACL		FW		IPC			
	No. of cycles	Time (ms)	No. of cycles	Time (ms)	No. of cycles	Time (ms)	No. of cycles	Time (ms)
0.1 k	382,423	2.58	362,496	2.44	391,704	2.64	378,874	2.55
1 k	3,247,248	21.87	2,808,021	18.91	3,340,964	22.50	3,132,077	21.09
5 k	15,052,497	101.38	15,771,953	106.22	15,218,285	102.50	15,347,578.33	103.37
10 k	32,452,174	218.57	31,339,626	211.07	30,663,219	206.52	31,485,006.33	212.05

the number of required clock cycles for each benchmark. It is important to bring to the attention of the reader that the proposed PCIU hardware accelerator can be mapped to other less expensive FPGAs such as Virtex-5LX as long as enough Block Ram is available to accommodate the bit vectors (stored in Mem_2) resulting from the preprocessing stage and used in the classification phase. An alternative is to use a platform with external memory that can be accessed by the hardware accelerator at the expense of performance achieved.

In the next few sections we will describe in detail the steps to transform the pure software implementation of the PCIU to hardware via the Impulse-C platform. First we will show the transformation of the preprocessing stage into hardware and this will be followed by a similar description for the classification stage of the original algorithm.

5.2. Preprocessing Stage Design. Figure 4 shows the different stages involved in the hardware implementation of the preprocessing stage based on the Impulse-C tool. The main phases of the hardware accelerator are based on bit vector generation, redundancy removal, and populating memories (Mem_1, Mem_2) with the corresponding vectors that are eventually used in the classification phase. The original PCIU's preprocessing C-code was mapped to the CoDeveloper to generate the baseline implementation. A few code transformations were applied to improve upon the baseline design. Even though the speedup achieved by mapping the preprocessing phase onto an FPGA is low as we will learn from the results section, the objective of mapping the preprocessing stage onto a reconfigurable computing platform along with the classification phase is to realize a single embedded system and thus lower power consumption, increase reliability, and reduce cost which is essential for such systems.

Table 3 presents the total clock cycles and preprocessing time for all benchmarks based on different sizes. The average across all benchmarks for a specific size is also presented in the last column of the Table. It is important to note that Figure 4 describes the processing of a single chunk of Table 1 introduced earlier.

The pure software implementation of the PCIU algorithm was executed on a state-of-the-art x86 Family 15 Model 4 Intel Xeon processor operating at 3.4 GHz. Table 4 summarizes the performance obtained by both the Xeon processor and hardware accelerator based on Impulse-C for preprocessing and evaluated in Rule/Sec. The average speedups achieved by the Impulse-C using either the rate of preprocessing rules (Rules/Sec) or total time (in milliseconds) are around 1.16x over all benchmarks for the 10 K rule. It is clear that the performance achieved by the hardware Impulse-C accelerator is limited and this can be attributed to the memory dependency of the preprocessing stage which translates to a limited amount of parallelism that can be exploited thus the modest speedup achieved.

(1) The hardware based preprocessing engine achieves on average 1.16x speedup over a 3.4 GHz single core Xeon processor.

(2) Since the preprocessing time was based on processing a single chunk, time can be reduced if the hardware accelerator is implemented in parallel by processing 13 chunks simultaneously.

(3) Further improvement of the preprocessing engine could be achieved if coarse grain and parallel coarse grain optimization steps were performed.

Since the preprocessing phase is only performed *once*, it would be an inefficient use of resources to perform further optimization on the hardware accelerator built. In fact, preprocessing can be implemented on a soft core Micro-Blaze embedded inside the FPGA if further resources are required for the more important classification phase. However, if the performance of the soft core does not meet the need for the application then the designer will have to map it to hardware as explained above. The incremental update capabilities of the PCIU running on an FPGA are still preserved. The rules in database can be either deleted or replaced by a new rule with no side effects. If new rules need to be added to the existing database then the designer has to make sure that enough space in the bit vector exists for such an addition. However, if not enough space is reserved a priori and new rules need to be added then the entire PCIU algorithm needs to be remapped and recompiled onto the FPGA.

5.3. Classification Stage Design. Since classification of packets is performed on a regular basis and given the importance of the classification phase, we will attempt to design a classification hardware accelerator engine by performing several optimization techniques on the baseline based module using fine grain, coarse grain, and parallel coarse grain approaches to construct an efficient and robust hardware module that can achieve at least an order of magnitude speedup over a

TABLE 4: Performance achieved in terms of preprocessing (Rule/sec) and time (ms).

Benchmark	Preprocessing (rule/sec)		Time (ms)		Speedup over desktop (x)
	Desktop	Impulse-C	Desktop	Impulse-C	Impulse-C
ACL (10 K)	38,412.00	43,936.51	250	218.57	1.14
FW (10 K)	39,621.28	44,112.83	235	211.07	1.11
IPC (10 K)	36,148.00	43,759.16	250	206.52	1.21
Average	38,060.43	43,936.16	245	212.05	1.16

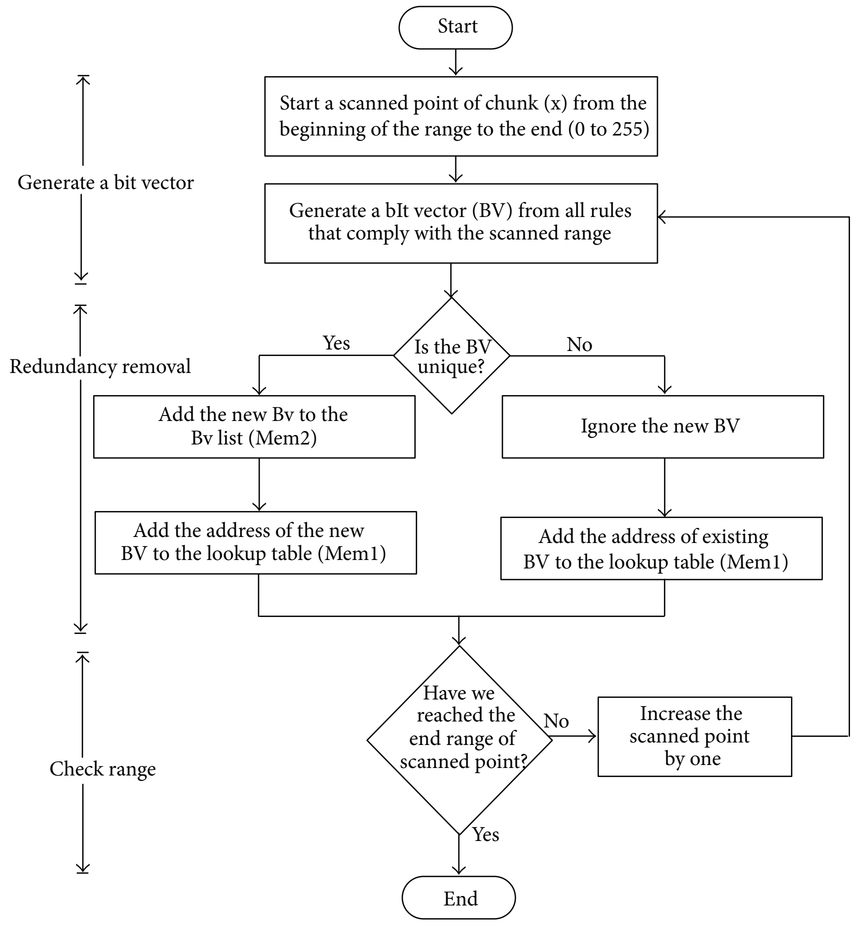

Start
Start a scanned point of chunk (x) from the beginning of the range to the end (0 to 255)
Generate a bit vector
Generate a bIt vector (BV) from all rules that comply with the scanned range
Redundancy removal
Yes
Is the BV unique?
No
Add the new Bv to the Bv list (Mem2)
Ignore the new BV
Add the address of the new BV to the lookup table (Mem1)
Add the address of existing BV to the lookup table (Mem1)
Check range
Have we reached the end range of scanned point?
No
Increase the scanned point by one
Yes
End

FIGURE 4: The PCIU Impulse-C preprocessing stages.

state-of-the-art Xeon processor. Figure 5 illustrates the main blocks of the PCIU Impulse-C implementation, including the Reference Generator, classifier, and Rule ID Generator.

The input stream is used both to supply the system with the preprocessed rule set and to feed the classifier with testing packets, while the output stream is used by the classifier to output the best match rule number for the incoming packet. The original PCIU's C-code was mapped to the CoDeveloper to generate the baseline implementation and required only the following code transformations:

(i) adding the stream reading/writing from/to the buses,

(ii) changing the variables to fixed sizes,

(iii) collapsing all functions to a single "main" function,

(iv) converting all dynamic arrays to local static arrays.

Although Impulse-C is an Electronic System Level design-based language, its implementation and optimization are different from other traditional ESLs. Whereas conventional ESLs are oriented more towards statement-level optimization, Impulse-C is oriented more towards system and streaming-level optimization similar to Xilinx Vivado HLS software [24]. Furthermore Impulse-C adopts a main block-based approach to optimize the target application, instead of using a timing model where each statement executes in a single clock cycle. In particular, Impulse-C provides three main pragmas—PIPELINE, UNROLL, and FLATTEN—to improve execution time. Using the Stage Master tools, these optimization pragmas provide an efficient method to improve the target design. Additionally, Impulse-C designs can take advantage of dual-port RAM optimization, allowing the design to access two (different) locations simultaneously and,

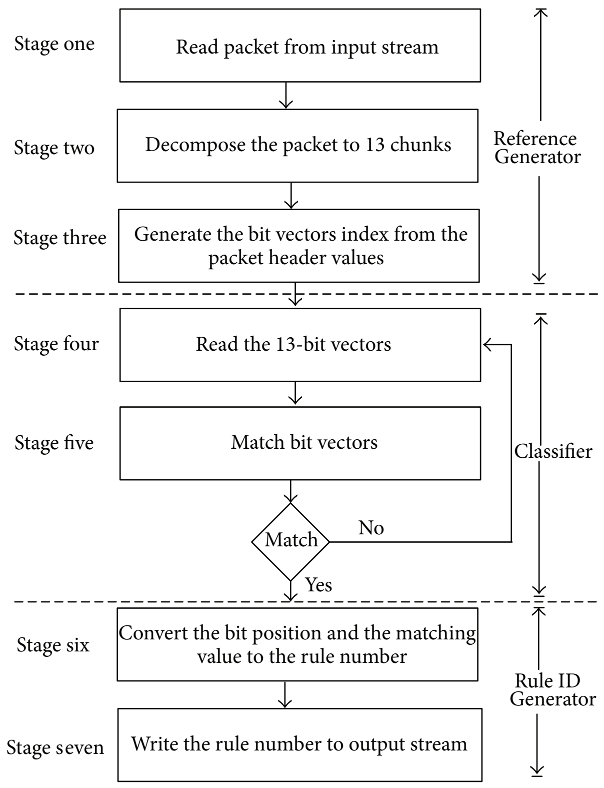

Stage one
Read packet from input stream
Stage two
Decompose the packet to 13 chunks
Stage three
Generate the bit vectors index from the packet header values
Reference Generator
Stage four
Read the 13-bit vectors
Stage five
Match bit vectors
Classifier
Match
No
Yes
Stage six
Convert the bit position and the matching value to the rule number
Stage seven
Write the rule number to output stream
Rule ID Generator

Figure 5: The PCIU Impulse-C classification stages.

consequently, helping to reduce the total number of clock cycles. As noted previously, the three different optimization strategies used to improve the PCIU code can be classified as either (a) fine grain, (b) coarse grain, or (c) parallel coarse grain, as will be described in the following sections.

5.4. Fine-Grain Optimization (FGO). The first step taken to improve the original Impulse-C implementation of the PCIU was by applying the PIPELINE and FLATTEN pragmas, separately, to all of the inner loops of the initial implementation.

The purpose of this optimization step is to convert the generated HDL block from a sequential block into either a pipelined or parallel block. The selection between the PIPELINE and FLATTEN pragma can be performed via the Stage Master exploration tool. The main optimization techniques applied in FGO are referred to as FGO-A, FGO-B, FGO-C, FGO-D, and FGO-E and will be described in detail in the following paragraphs.

(1) The FGO-A technique specifies that, since an outer loop cannot be pipelined in Impulse-C, the best option is to collapse the nested loops into a single pipelined loop. Algorithm 1 illustrates the conversion of a nested **FOR** construct into a single loop with the addition of a conditional statement for the outer loop. This optimization technique tends to reduce the number of clock cycles needed for the original (i.e., nested) loop by 50%.

(2) The FGO-B technique entails converting the **FOR** loops into **WHILE** loops as depicted in Algorithm 2. Although the Impulse-C compiler converts other loops into **WHILE** loops when generating HDL code, adding the FLATTEN pragma tends to improve the loop execution time.

(3) The FGO-C technique represents an attempt to improve the performance of the Rule$_{\text{ID}}$ Generator Block. An ideal approach to its optimization is to employ a modified technique that is more suitable for hardware than the original software approach used on a general-purpose processor. The modified code utilizes a series of **IF** statements as shown in Algorithm 3 in addition to a FLATTEN pragma which tends to reduce the number of clock cycles. It is important to distinguish between sequential code for classification phase running on a general-purpose processor and that running in parallel on hardware.

(4) The FGO-D technique takes advantages of the distributed dual-port RAM to ensure that multiple locations can be accessed simultaneously, without the need for memory duplication. Algorithm 4 illustrates the usage of dual-port RAM in the code. Although this technique tends to dramatically reduce the required clock cycles, it does often result in an increase in the length of the critical path.

(5) The final optimization technique, FGO-E, seeks to adapt the width/depth of streams. Since the stream width and depth effects have a huge influence on the system performance, the result of the optimization is prominent when the stream is reading or writing inside the pipeline loops. In this case, the depth of the stream (buffer size) has to be at least one more than the pipeline depth, to ensure that the stream does not become a system bottleneck. The stream width and depth improve the speed of reading/writing per cycle, which, when performed in place, adds the additional constraint that the pipeline must be kept full and running at maximum throughput.

5.4.1. FGO: Experimental Setup. In this section we intend to explore the effect of each optimization technique on the performance of the Impulse-C implementation of the PCIU algorithm. Accordingly, an extensive experimental analysis is performed, in which all possible combinations of optimizations are considered. If we assume that S is the set of five groups of enhancements (optimizations), our extensive analysis exploration process proposes the examination of each element of the power set of S (excluding the empty set), which entails that 31 different hardware versions are considered. It is important to note that each of these 31 versions is modified from the Impulse-C baseline implementation described previously. The entire process of generating all 31 versions, as well as the baseline, took a few months to complete.

Due to the deterministic nature of PCIU algorithm, and the fact that none of the optimizations techniques change the functionality of the algorithm, all 31 hardware versions

```
  /*Nested For Loop */
for(slice = 0; slice <  13; ++slice)
 {
  for(bv = 0; bv <  256; ++bv)
  {
   co_stream_read(InPut, &Read,32);
   Table[slice*256+bv]=Read;
  }
  }
```

```
  /*Nested For Loop with Pipeline in inner loop */
for(slice = 0; slice <  13; ++slice)
 {
  for(bv = 0; bv <  256; ++bv)
  {
   ♯ pragma CO PIPELINE
   co_stream_read(InPut, &Read,32);
   Table[slice*256+bv]=Read;
  }
 }
```

```
  /*Collapsing For Loop*/
bv=0;
for(slice = 0; slice <  13; bv++)
 {
   ♯ pragma CO PIPELINE
    co_uint16 idx = (slice ≪ 8) + bv;
   co_stream_read(InPut, &Read, 32);
   Table[idx]=Read;
   if(bv==255)slice++;
 }
```

Algorithm 1: FGO-A: nested FOR loop collapsing code example.

```
   /* For Code*/
for(i=0;i <  10;i++)
{
     MyRam[i]=i;
}
```

```
/* Do While Code*/
i=0;
do{
     ♯ pragma CO FLATTEN
       MyRam[i]=i;
       i++;
}  while(i <  10);
```

Algorithm 2: FGO-B: loop optimization (FLATTEN pragma).

```
      /* While Code*/
/* Search for the first set bit
 in the BV(value)*/
 /* Similar to Sequential search */
   p = 0;
   while(value)
   {
          p++;
           /* Shift Right by 1 */
          value=value≫1;
   }
```

```
  /* Serial IF Statement*/
p = 0;
 if(value){
   ♯ pragma CO FLATTEN
  if (value & 0xffffffffffffffff0000000000000000){
       p = 64; value ≫= 64; }
   if (value & 0xffffffff00000000){
       p += 32; value ≫= 32; }
  if (value & 0xffff0000){
       p += 16; value ≫= 16; }
  if (value & 0xff00) {
       p += 8; value ≫= 8; }
  if (value & 0xf0){
       p += 4; value ≫= 4; }
  if (value & 0xc){
       p += 2; value ≫= 2; }
  if (value & 0x2){
       p += 1; value ≫= 1; }
  p += (value & 0x1);
```

Algorithm 3: FGO-C: a **While** to **IF** statement conversion.

```
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 4: FGO-D: dual port RAM code optimization.

produce the same solutions for the same problem instances and, consequently, when comparing the hardware versions among themselves it is not necessary to consider solution quality. Contrarily, the different hardware versions need only to be compared with respect to run time.

Since Impulse-C requires the clock period for a design to be longer than the longest path through the combinational logic in the design, the run-time of each architecture can only be determined once the circuit has been placed and routed on the FPGA. After placement and routing, the clock rate (frequency) of the design can be determined from the reciprocal of the clock period, and the clock rate can then be multiplied by the number of clock cycles required to execute the design to determine the actual run-time of the algorithm mapped onto the FPGA.

Compiling each of the three benchmarks (i.e., ACL, IPC, and FW) of ClassBench [8] with one of four possible rule set sizes (i.e., 0.1 k, 1 k, 5 k, and 10 k) resulted in a total of 12 benchmarks for our experiment. All 31 versions of PCIU were tested using these 12 benchmarks, and the number of clock cycles in addition to the clock period was recorded. The total clock cycles along with the clock period were then used to calculate the speedup achieved over the baseline implementation, as demonstrated by

$$\text{SpeedUp}(x) = \frac{\text{num of clk cycles of baseline} * \text{clock period of baseline}}{\text{num of clk cycles for PCIU rev}_x * \text{clock period of rev}_x}. \quad (1)$$

5.4.2. FGO: Analysis of Results. Table 5 summarizes the expected effects of the three main block optimization pragmas on the chip area, in terms of the maximum critical-path delay and the number of clock cycles needed. It is important to highlight that the performance of different FGO combined techniques that will be evaluated in this section could vary based on the interaction and dependency among these techniques. In some cases the resultant speedup achieved from combining these different FGO optimization approaches could be attributed to the addition, product, or combination of each individual method.

Figures 6, 7, and 8 present the speedup performance achieved by all proposed FGO architectures with respect to the baseline Impulse-C architecture. Each figure describes a different evaluation metric as will be described below.

Figure 6 depicts the average classification speedup achieved for each of the three benchmarks (ACL, IPC, and FW) where one way indicates applying a single optimization step, two way a combination of two optimization steps, and so on. The first observation that can be made from Figure 6 is that consistent results are achieved for the individual and combined optimization techniques, across all benchmarks compiled with similar sizes. This indicates that changing the benchmark does not affect the speedup achieved for any FGO technique. This clearly indicates that the PCIU algorithm is robust and scales well with benchmark size. Therefore, we can conclude that neither the PCIU algorithm nor the optimization technique selection is affected by the complexity of the benchmark.

Table 5: Summary of impulse optimization pragma.

Pragma	Area	Frequency	Cycles
FLATTEN	Sharp increase	Decrease	Sharp decrease
PIPELINE	Increase	Increase	Decrease
UNROLL	Sharp increase	Decrease	Sharp decrease

Figure 7 presents the speedup achieved by different combinations of FGO averaged over all the benchmarks. It shows how the PCIU performs as the benchmark size increases from 100 to 10 K rules. Interestingly, no generalization can be made about whether the individual FGO techniques are positively or negatively affected by the size of the rule-set. Whereas FGO-A has speedups of 1.9x to 2.8x with the 0.1 k and 10 k rule sets, respectively, FGO-C has a 9x speedup with the 0.1 k rule set but will deteriorate to a 2x speedup when the size of the rule set increases to 10 k. The highest average speedup was achieved by FGO-C (5x)—an enhancement due to the replacement of the original search technique running on a general-purpose processor with modified approach in the form of flattened **IF** statement that is more suitable for hardware implementation that can take advantage of the FLATTEN pragma that exploits further parallelism in the system.

Table 6 presents the performance achieved by different FGO techniques along with resource utilization of the FPGA chip. It is evident from Table 6 that realizing the different FGO architectures on an FPGA entails a challenge since more than 80% of the resources are utilized. It should be noted that the different FGO techniques also differ in their usage of the FPGA chip resources—it is clear from Table 6 that FGO-E has the lowest resource consumption in terms of slices

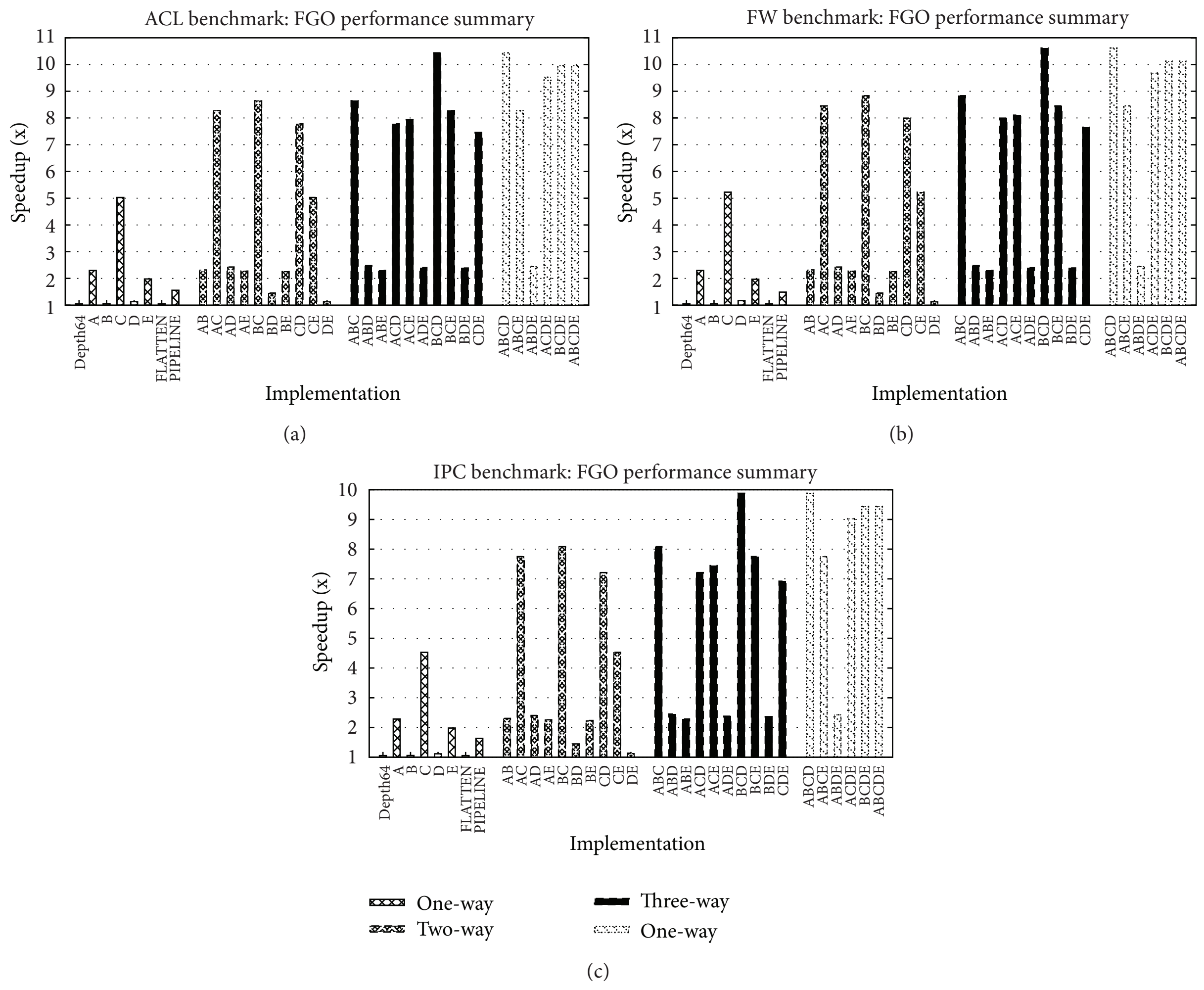


Figure 6: Speedup achieved by FGO techniques over the baseline (averaged over different benchmark sizes).

among the five techniques. However, FGO-E consumes a considerable amount of Block RAMs compared to other FGO implementations which poses some constraints on the type of FPGA chip that can be used to accommodate it. It is also important to notice that the resources required by FGO-D exceed the resources available in the Xilinx Virtex 6 (LX760). However, we still manage to report the maximum frequency and number of cycles from the place and route and ModelSim package. Furthermore, the implemented FGO techniques have different effects on the critical time delay and maximum frequency, and it is also clear that the PIPELINE pragma decreases the critical time delay (which, when performed in-place, tends to increase the maximum frequency).

Figure 8 presents the speedup achieved averaged over both benchmarks and sizes used in the evaluation process. This figure clearly highlights the performance of all FGO architectures along with their different combinations.

Based on the description of FGO-E earlier, the stream width and depth have substantial influence on the system performance. Referring back to Figure 3 it is clear that two types of streaming occur in the hardware Impulse-C proposed architecture. The first is used to fill up the memories (Mem_1 and Mem_2) with the preprocessed information in the form of Final Clusters and bit-vectors. The second streaming is in the form of trace packets that need to be classified and the generation of list of winning rules. Table 6 suggests that when the stream depth is modified from 2 to 64, fine grained optimization techniques tend to perform better in terms of classification. This is attributed to the reduction of cycles needed to fill the memory.

Figure 8 clearly demonstrates that when the FGO-B and FGO-C techniques were combined (designated FGO-BC), a classification speedup of 8.5x was achieved for all benchmarks—although the speedups achieved by combining (addition or multiplication) FGO-B and FGO-C are substantially less than 8.5x, if we consider the effect of pipeline pragmas in each FGO-B loop, we get a factor of nearly 8x. FGO-B alone does not achieve the effect of the pipeline pragma, due to the multiple nested loops. Furthermore, although FGO-AE has the highest memory filling speedup, it should be noted that this combined effect is not well represented by the sum of the individual effects associated with the FGO-A and FGO-E techniques, since FGO-E tends to reconstruct the size of the reading and, consequently,

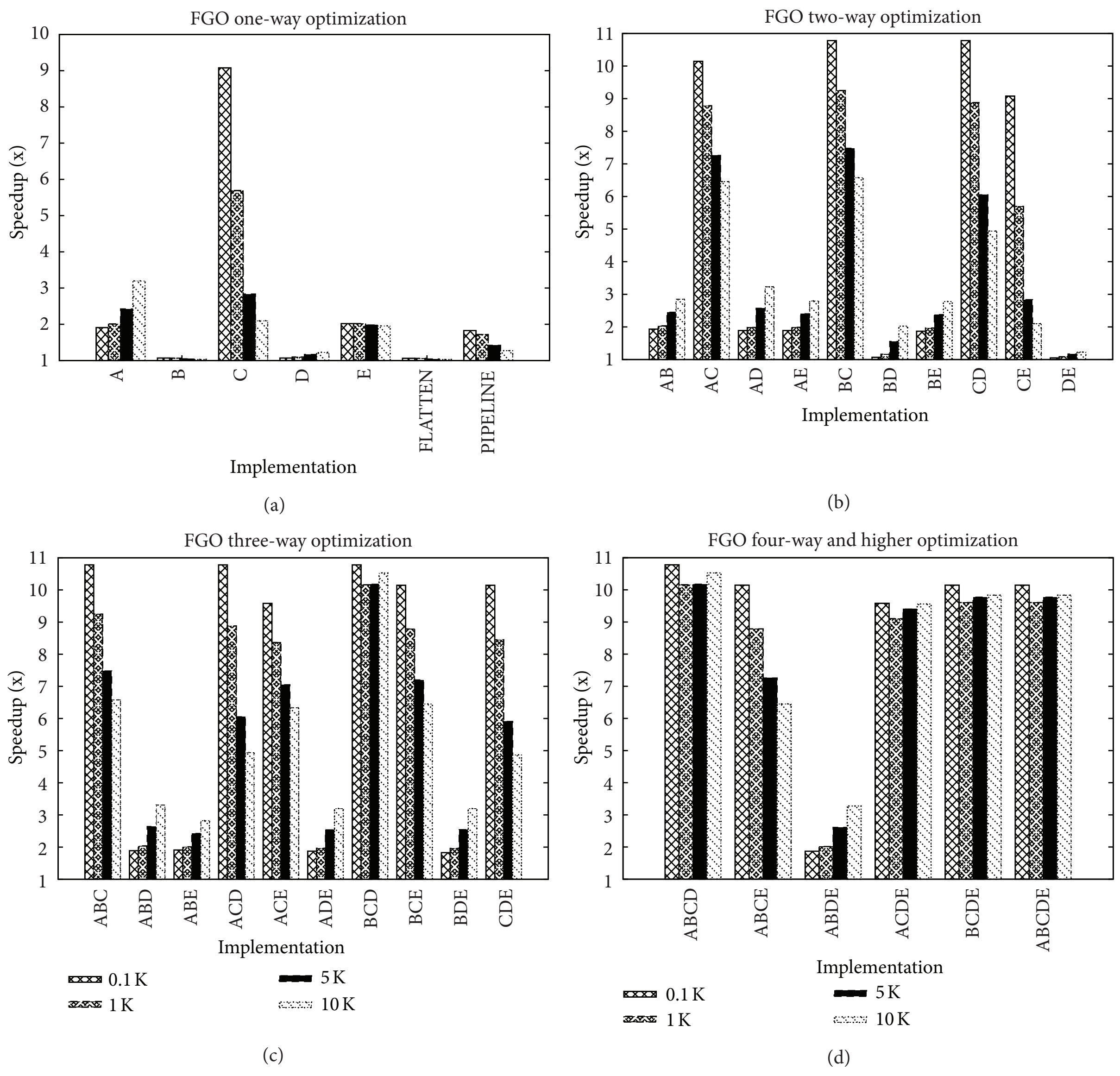


Figure 7: Speedup of FGO techniques over baseline (averaged over all benchmarks).

minimize the net effect of FGO-A. It could be concluded that the use of multiple fine-grain optimization techniques will often improve performance levels over the use of a single technique, but since additional combinations of techniques do not always result in a significant performance improvement, the techniques should be combined judiciously; although the combination of all five techniques produces a significant (but not optimal) speedup, it does require the most effort on the part of the developer.

Based on the 31 combinations of fine-grain optimization techniques, the greatest speedup was achieved by the FGO-BCD three-way technique. This enhancement can be largely explained by examining the individual contributions of each technique used.

(1) FGO-B seeks to reduce the number of clock cycles by converting **FOR** loops into **WHILE** loops, pipelining, and reducing deep logic (and, thus, increasing the clock frequency).

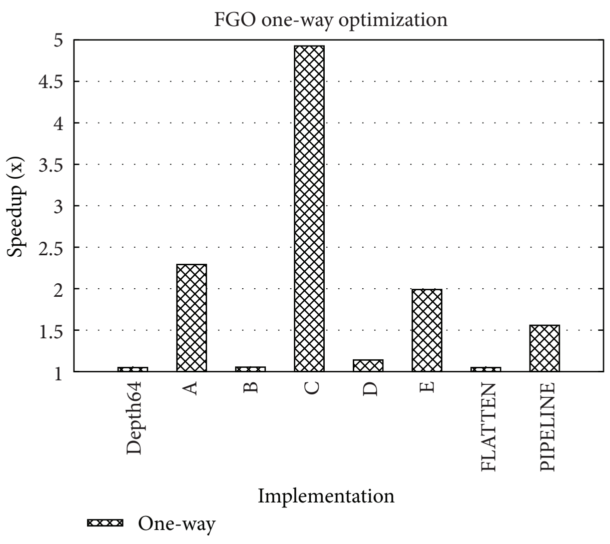
(2) FGO-C also seeks to reduce clock cycles by performing FLATTEN **IF** statements instead of loop search (reducing the critical-path delay).

(3) FGO-D tends to reduce the number of clock cycles by exploiting the dual-port-RAM.

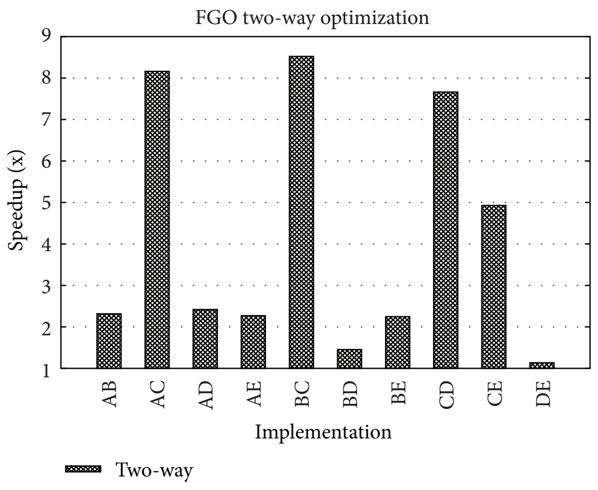
When applied individually, optimizations FGO-B, FGO-C, and FGO-D result in speedups of 1.05x, 4.92x, and 1.14x, respectively, but when these techniques are combined, abundant opportunities exist for the restructured and simplified statements produced by optimization FGO-B to be performed in pipeline with other statements. This leads to a significant reduction in the total number of clock cycles as well as a further improvement in clock frequency over the baseline implementation.

Table 6: Performance achieved/FPGA resource utilization for "one-way" FGO designs.

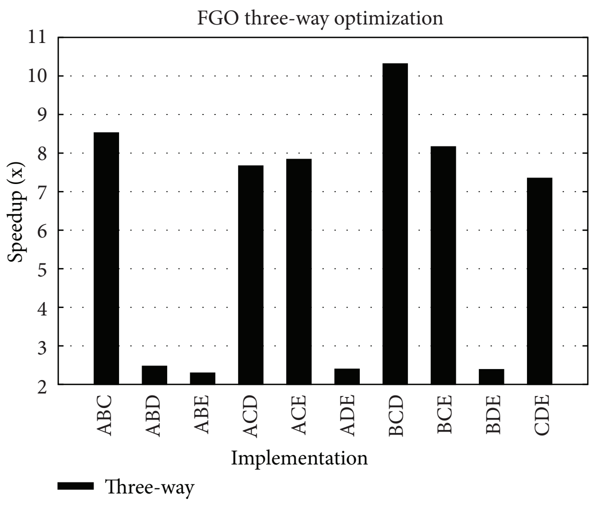
	Implementation						
	Baseline	Stream depth 64	A	B	C	D	E
Resources							
Block RAM	641	642	643	642	642	643	1668
Slice Reg	4,816	5,041	9,075	9,265	2,925	7064	3120
Slice LUTS	888,597	888,545	889,151	888,276	888,659	1,768,916	3,133
Slice LUT-FF	890,467	892,889	894,014	896,557	890,878	1,771,482	5,298
Performance							
Maximum Freq (MHz)	103.896	108.982	121.772	110.331	106.490	103.331	197.449
Num of Clocks ACL (10 k)	33,680,692	32,710,703	11,843,100	32,613,702	16,203,627	27,352,703	32,710,701
Time	324.18 ms	300.15 ms	97.26 ms	295.60 ms	152.16 ms	264.71 ms	165.67 ms
Speedup over previous	—	1.1x	3.3x	1.1x	2.1x	1.2x	2.0x



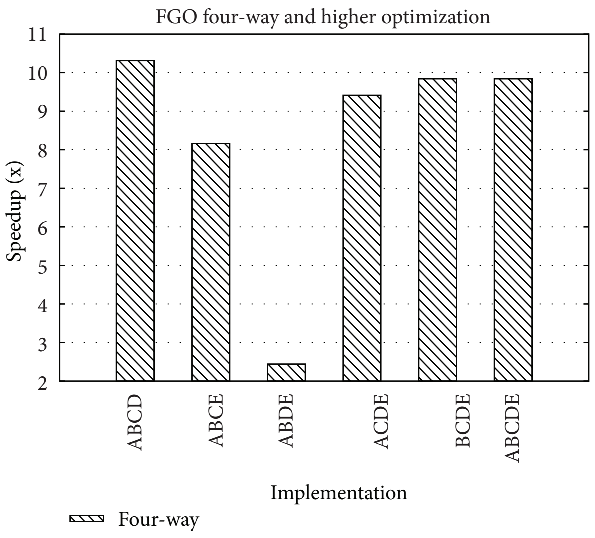
(a)



(b)



(c)



(d)

Figure 8: Speedup of FGO techniques over baseline (averaged over all benchmarks with different sizes).

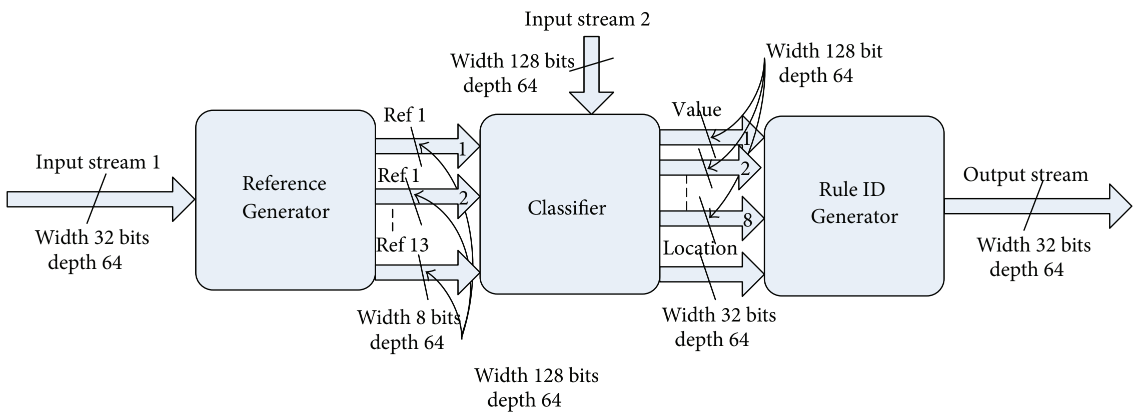


Figure 9: Parallel coarse grain optimization (PCGO).

These observations clearly indicate that fine-grain optimization can play a crucial role in enhancing the design in terms of reducing clock cycles, maximum delay time, and resource utilization. Moreover, adding more of the FGO techniques will not always lead to improvement of design, and the effects of individual FGO are not always additive.

5.5. Coarse Grain Optimization (CGO). One of the key methods of exploiting parallelism for the PCIU algorithm is to divide the system into three main, independent blocks (as shown previously in Figure 5). The PCIU system can be mapped to three processes operating in parallel and communicating via streams (for CGO the width of InputStream$_2$ is 32 bits and only two values are sent to the Rule ID Generator from the classifier) as seen in Figure 9. The InputStream$_1$ is used to feed the system with both the 13 referencing tables and the headers of the testing packet for classification, while InputStream$_2$ is used to supply the bit vectors both at system start-up and in the incremental updating stage. During the memory filling phase, both streams are used to supply the system with the needed lookup tables, and during the classification phase, the Reference Generator process reads the headers of the packet provided by InputStream$_1$.

The incoming packet header is decomposed into 13 chunks and each is used as an address to calculate the effective location (pointers) of the bit vector which will be used for matching. Since each bit vector maps to a 128-bit memory width, the bit vector will be decomposed into multiple chunks depending on the rule set size. A chunk from each of the thirteen bit vectors is subjected to a bitwise logical "AND" operation, and if the result is zero, the next chunk of the bit vectors is accessed. This operation will terminate when a nonzero value is obtained indicating a match.

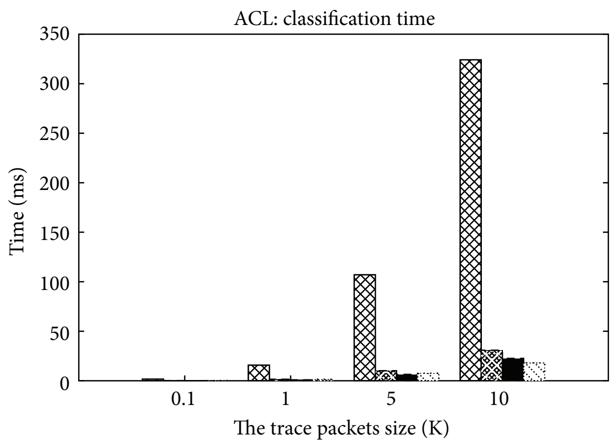
By utilizing the dual-port memory, the system can read two chunks of the bit vector simultaneously and perform the matching operation. Thus, two results can be generated in each cycle. The results are then sent to the Rule ID Generator process with the location of the first value (in case a nonzero value is found between them). The Rule ID Generator process reads Value_1 and Value_2 in addition to the location streams and generates the winning rule number, which is then written to the *Output Stream*. This organization of the PCIU system attempts to reduce the number of cycles needed by the system for completing the classification process. Since each process of the coarse grain optimization (CGO) still incorporates the fine grain optimization techniques, CGO achieves, on average, a speedup of almost 1.53x over the best FGO implementation.

5.6. Parallel Coarse Grain Optimization (PCGO). The final optimization phase, to further improve the Impulse-C PCIU implementation, was based on the idea of splitting the bit vector into four memory banks (instead of a single bank). By incorporating dual-port RAM, eight chunks of each of the 13 bit vectors can be read simultaneously from the memory and subjected to a matching operation in parallel to producing eight results. A winning rule is found when one of these results has a nonzero value. Figure 9 illustrates the PCGO configuration given that the InputStream$_2$ now has a width of 128 bits versus 32 bits for the coarse grain optimization flow. The PCGO classification process provides eight values to the Rule ID Generator (instead of two values in the CGO system), and this division of the memory tends to reduce the total clock cycles needed for the classification an improvement that can be attributed to the parallel application of the matching operation. Although the memory partitioning sharply reduces the number of clock cycles, it tends to increase the critical-path delay. The overall system performance can thus be expressed as the product of the number of clock cycles and the critical-path delay. In this regard, the PCGO technique did not achieve a dramatic speedup over the CGO technique, as one divides the memory further, the total amount of resources must be increased for accuracy, but the improvement in terms of speedup far outweighs the additional chip resource usage. Although PCGO achieves between 0.8x and 3x speedup (depending on the rule sets size) over the CGO implementation, the overall speedup achieved using FGO, CGO, and PCGO combined is significant.

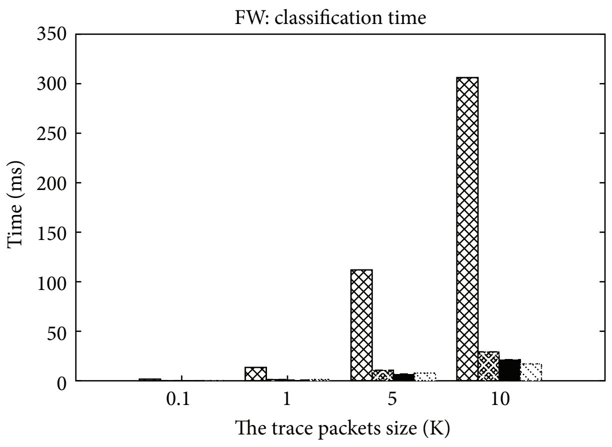
5.7. Comparative Evaluation. Figure 10 shows the execution time of the four implementations (i.e., the baseline

Table 7: Impulse-C: FPGA Resource Utilization for PCIU Design.

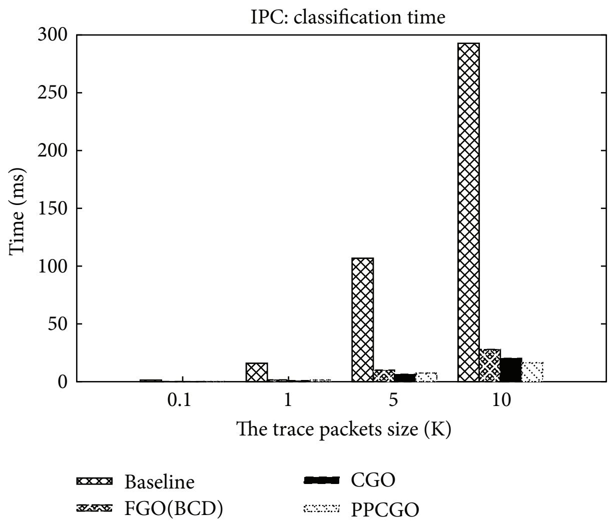
Resources	Implementation				
	Preprocessing	Classification			
		Baseline	FGO (BCD)	CGO	PCGO
Block RAM	3,077	641	642	647	711
Slice Reg	1,019	4,816	8,970	10,123	24,187
Slice LUTS	2,827	888,597	1,768,713	1,766,927	1,666,331
Slice LUT-FF	3,053	890,467	1,776,316	1,771,434	1,674,215
Maximum Freq (MHz)	148.478	103.896	107.66	112.717	65.542



(a)



(b)



(c)

Figure 10: PCIU's Impulse-C implementation: execution time.

implementation and the average performance of the FGO, CGO, and PCGO techniques). Although it is clear that PCGO achieves the best performance over the other implementations, it also has the highest chip resource utilization as seen in Table 7. Although PCGO requires less than half the number of clock cycles required by CGO, the increase in its critical-path delay leads to an overall speedup of 0.83x. Table 7 illustrates in detail the chip utilization associated with each of the implementations. It is clear that FGO (BCD) consumes almost twice the amount of resources as the baseline, except for the Block Ram. CGO, on the other hand, consumes almost the same resources as FGO (BCD) (because of the partitioning of the design into three subsystems and the use of the streams for communication) with a slight overuse

Table 8: Classification time and speedup achieved by different implementations.

Size	Baseline	FGO (BCD)		CGO		PCGO		Overall
	Time (ms)	Time (ms)	Baseline/FGO	Time (ms)	FGO/CGO	Time (ms)	CGO/PCGO	Baseline/PCGO
0.1 K	1.6	0.14	11.14x	0.09	1.52x	0.15	0.62x	10.67x
1 K	15.08	1.432	10.53x	0.87	1.65x	1.49	0.58x	10.12x
5 K	108.65	10.3	10.54x	6.60	1.56x	7.66	0.86x	14.18x
10 K	307.7	29.31	10.5x	21.35	1.37x	17.23	1.24x	17.86x
Average	—	—	10.67x	—	1.53x	—	0.83x	13.54x

of the Slice Register. The PCGO architecture requires (on average) an increase of 1.88x the chip utilization over the baseline in terms of lookup tables (LUTS) and Slice Registers.

6. Discussion and Comparison of Results

The previous results of this study show that a direct conversion from PCIU into Impulse-C does not produce a hardware accelerator implementation that executes with highest speed. However, by carefully employing language-level transformations to improve the hardware that is synthesized, it is possible to produce hardware that runs dramatically faster than software. The initial largest speedup of one-way based implementation (FGO-C) speedup (5x) (as seen in Figure 8) was achieved by converting the loops to flatten IF statement, collapsing nested complex loops to singular pipelined loops and then using dual-port-RAM to reduce the memory access time effect. Furthermore, the adoption of the stream width and depth surely has an additional effect on system performance in terms of both the critical-time delay and the number of clock cycles. Although the results obtained in this paper target the PCIU packet classification algorithm in particular, we believe that they will generalize to other implementations that use Impulse-C, as these FGO optimizations are very general and applicable across a wide range of implementations.

6.1. Incremental Improvement. Table 8 summarizes the performance achieved by the baseline, best FGO-BCD three-way approach, and CGO and PCGO implementations.

Table 8 also shows the amount of speedup achieved by: FGO over the baseline and CGO over FGO and finally the speedup achieved by PCGO over CGO. The average speedup of the PCGO implementation over the baseline over all benchmark sizes is almost 13x. However it is important to notice that the amount of speedup achieved increases as the rule set increases in size (i.e., 10 K). The key parameters of interest when examining the effectiveness of any implementation of PCIU are classification time, prprocessing time, and memory usage. The preprocessing time is considered to be of the least concern since it represents a one-time operation completed for a larger, continuous process (i.e., classification). That being said, improvements to preprocessing time are still sought especially in light of the incremental update capability of the PCIU. A shorter preprocessing time means less downtime of the system and a more versatile, resilient, and effective classification procedure overall.

6.2. GPP versus Hardware Accelerator. A pure software implementation on a general-purpose processor (desktop) has excellent results since it runs on a powerful processor (with several dedicated ALUs) and large amount of memory resources to use, but this hardware is generally not appropriate for anything but a server implementation, and in that respect a purely software approach is not directly comparable to the embedded alternatives. An “embedded system” tends to enhance power consumption and increase reliability at the expense of less flexibility in comparison to a server architecture. An alternative would be to utilize a soft processor like a Micro-Blaze or even a dedicated hard processor available on the FPGA to accomplish the task.

Table 9 presents the performance obtained for classification (evaluated in packet/sec) and the speedup achieved by the Impulse-C implementation over the Xeon processor. The average speedups shown in Table 9 are calculated either by using the ratio of the rate of classification (packets/sec) or total time (in milliseconds).

Figure 11 on the other hand presents the classification time of the final parallel coarse grained implementation along with that obtained by a general-purpose processor. Results obtained clearly indicate the following.

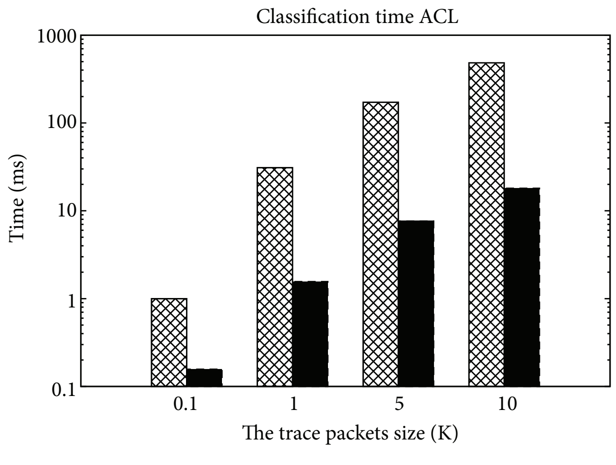
(1) The hardware based approach using Impulse-C achieves on average 27x classification speedup over the Xeon processor. In addition, the Impulse-C approach has the most straightforward way to convert any C-code to HDL compared to other ESLs and provides an easy way to improve any design and can be verified using the most popular verification CAD tools like ModelSim.

(2) Even with the vast resources and power of the desktop (32-bit WinXP running on Xeon 3.4 GHz with 2 G RAM) the performance of a general-purpose processor is inferior in terms of classification time to that obtained by the Impulse-C hardware accelerator.

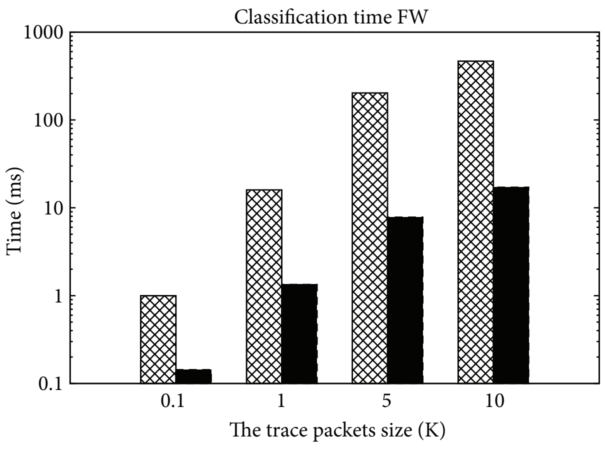
Table 10 presents a comparison between the current Impulse-C proposed architecture and the Handel-C implementation in [17] in terms of resources (Block Ram, LUTs, etc.) and design effort. It is clear from Table 10 that the Handel-C implementation is slightly faster than the Impulse-C design. The number of clock cycles in Handel-C is almost twice that of the Impulse-C yet the minimum period of the clock

Table 9: Performance achieved in terms of classification (packet/sec) and time (ms).

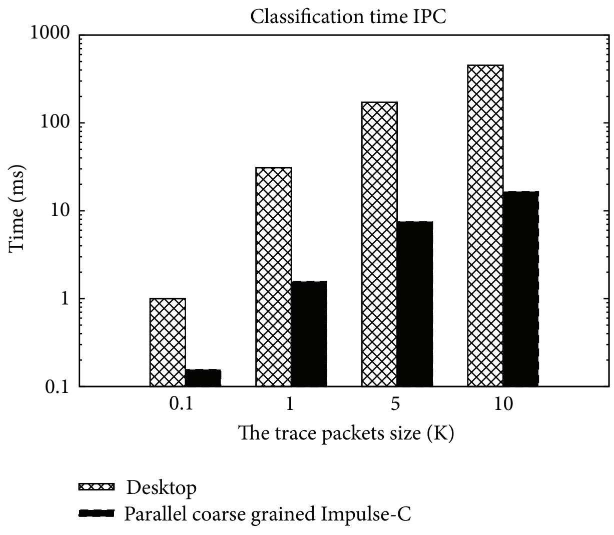
Benchmark	Classification (packet/sec)		Classification (time (ms))		Speedup over desktop (x)
	Desktop	Impulse-C	Desktop	Impulse-C	Impulse-C
ACL (10 K)	200,413.22	5,371,212.25	484	18.06	26.80
FW (10 K)	207,264.96	5,436,462.69	468	17.15	27.28
IPC (10 K)	214,128.04	5,503,311.31	453	16.47	27.50
Average	207,268.74	5,436,995.42	468	17.23	27.17



(a)



(b)



(c)

Figure 11: Software and hardware implementation of the PCIU: a comparison of classification time.

of Handel-C is almost half of that of the Impulse-C which translates to higher frequency of operation. Also, in terms of resources, Handel-C tends to occupy more LUTs than the Impulse-C; however the latter consumes more Block Ram. Finally, in terms of design effort the designer would spend less time modifying his/her original C implementation to port the code to Impulse-C environment than that spent mapping it to Handel-C. The advantage of the Handel-C is that every statement in the ported code takes exactly a single clock cycle and therefore the designer can anticipate the performance of the resulting hardware. The Impulse-C on the other hand requires much higher optimization effort since it requires more intervention from the designer to enhance the performance for each block of code.

6.3. Comparing the PCIU and GBSA Algorithms. Based on results obtained in [21] the GBSA algorithm achieves better performance over the proposed PCIU algorithm in terms of preprocessing and classification time when implemented

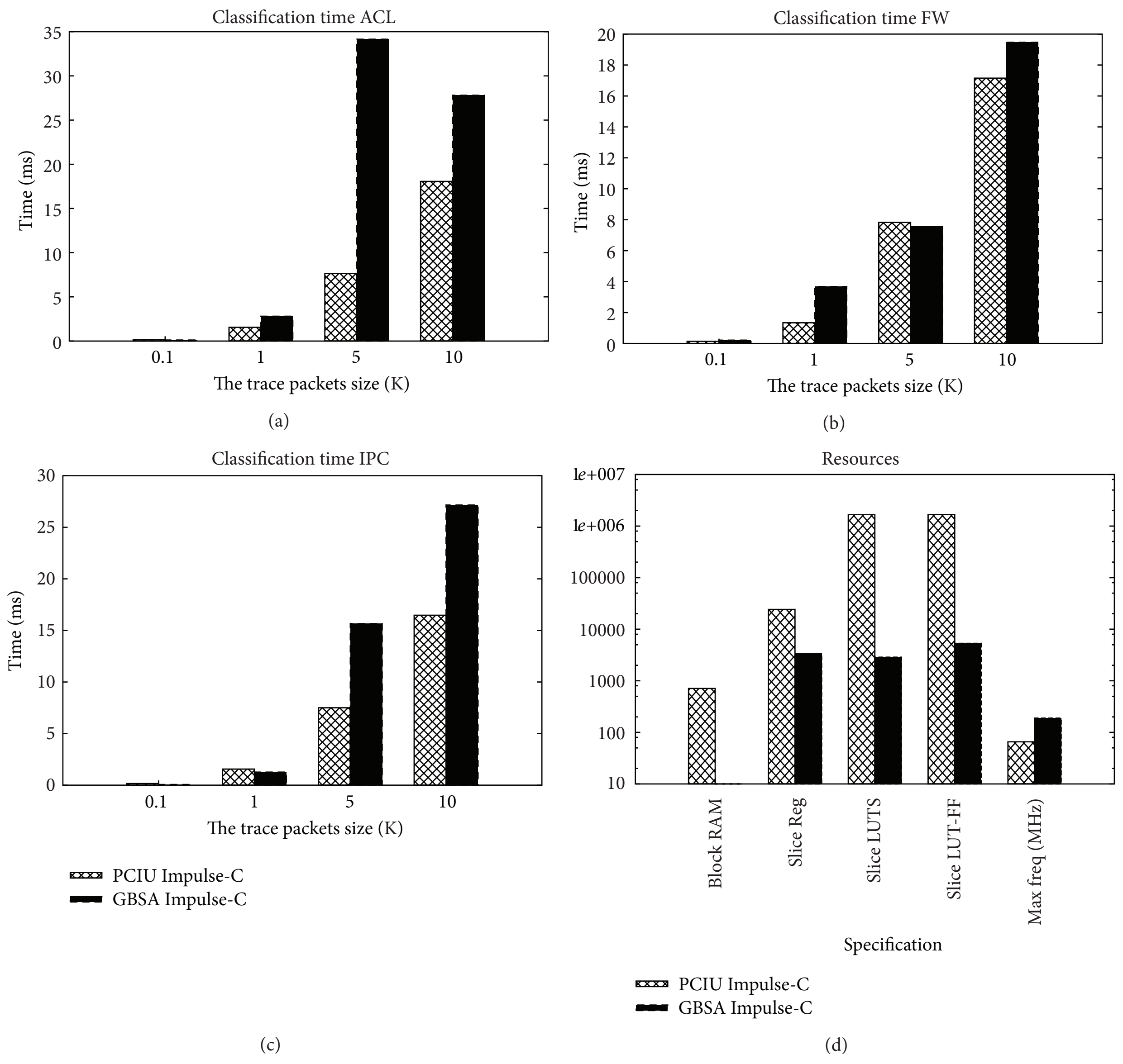


Figure 12: Impulse-C implementation of PCIU and GBSA: a comparison.

Table 10: PCIU: Impulse-C and Handel-C "PCGO" Design.

Resources	PCIU implementation (PCGO)	
	Impulse-C	Handel-C
Block RAM	711	0
Slice Reg	24,187	2,282
Slice LUTS	1,666,331	2,579,800
Slice LUT-FF	1,674,215	2,580,882
Maximum Freq (MHz)	65.542	111.435
Clock# ACL (10 k)	1,183,669	1,941,198
Time ACL (10 k) (ms)	18.06	17.42
Design (effort)	Medium	High
Optimization (effort)	High	Medium

on a general-purpose processor (Xeon 3.4 GHz with 2 G RAM). However, as the two implementations are mapped into hardware (RTL translation) using the Impulse-C tools the performance achieved by the PCIU algorithm surpasses that obtained by the GBSA algorithm. As can be seen in Figure 12 and Table 11 the PCIU algorithm Impulse-C based implementation achieves better performance than the GBSA implementation in terms of classification time. The average speedup achieved by the GBSA hardware implementation over its counterpart software implementation is almost 7.96x; however the amount of speedup achieved by the PCIU hardware implementation over the original software implementation running on a general-purpose processor is almost 26.8x. This is attributed to the memory dependency of the GBSA and also the sequential search nature which has a negative effect on the coarse grain implementation. However, the PCIU consumes extra resources since more parallelism has been exploited. This indicates that the PCIU is more suitable for mapping it into hardware even though the original software implementation on a general-purpose processor

Table 11: Software and hardware implementations of PCIU and GBSA algorithms: a comparison.

Benchmark	Desktop			Impulse-C		
	GBSA (ms)	PCIU (ms)	PCIU (speedup$_x$)	GBSA (ms)	PCIU (ms)	PCIU (speedup$_x$)
ACL (10 K)	192.1	484	0.4x	24.15	18.06	1.34x
FW (10 K)	163.9	468	0.35x	16.91	17.15	0.98x
IPC (10 K)	190.6	453	0.42x	23.58	16.47	1.43x
Average	182.2	468	0.39	21.55	17.23	1.25x

lagged in terms of performance when compared to the GBSA algorithm.

7. Conclusion and Future Work

The PCIU is a novel packet classification algorithm (with a unique incremental update capability) that has demonstrated powerful results. We have shown that this algorithm scales well for many different tasks and clients, and the incremental update capability allows it to change its rule set with minimal down-time. This allows implementations of PCIU to continue performing classification at a steady rate while remaining very adaptive and versatile. As we demonstrated in this paper, the PCIU is also an algorithm that greatly benefits from hardware acceleration and RTL translation and achieves greater performance boosts than competing algorithms—most of its shortcomings in terms of performance, when compared to other algorithms, are nullified by the incorporation of dedicated hardware. An extensive experimental analysis was performed in which all possible combinations of optimizations were considered. The study performed an extensive analysis for fine-grain optimization of Impulse-C. The analysis performed can be easily extended to similar applications that utilize ESL based approaches. The pure hardware based implementations using Impulse-C achieved on average a speedup of 27x over a pure software implementation running on a powerful, general-purpose processor. Our future work will target implementations of a pure RTL design based on VHDL against which we can compare our current results, in terms of area, power consumption, and maximum clock frequency. We also seek to compare our Impulse-C implementation with an Application Specific Instruction Set processor (ASIP) that is considered to be more flexible in terms of implementation changes and updates. Adopting both the PCIU and GBSA algorithms to operate under IPv6 is yet another goal that we seek to achieve in our future work. We also plan to extend the benchmarks that were used in this work (IPV4 ClassBench) to generate the IPv6 rule set and testing packets.

Conflict of Interests

The authors (O. Ahmed, S. Areibi, R. Collier, and G. Grewal) state that they do not have any personal or financial relationships with the above mentioned commercial identities (Impulse, Xilinx and Mentor Graphics) and, accordingly, there is no conflict of interests.

Acknowledgments

This work was partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC). The authors would like to thank Impulse Accelerated Technologies for allowing them to use their tools. The authors would also like to thank the Canadian Microelectronics Corporation for providing them with all the necessary CAD tools (Xilinx, and Mentor Graphics) used in this paper.

References

[1] 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.

[2] O. Ahmed and S. Areibi, "Software implementation of the PCIU algorithm," 2013, http://deimos.eos.uoguelph.ca/sareibi/PUBLICATIONS_dr/software_code_dr/PCIU_Software.html.

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

[4] IC, "Impulse accelerated technologies," 2011, http://www.impulseccelerated.com/.

[5] Calypto, "Catapult C synthesis," 2012, http://www.calypto.com.

[6] C. Bobda, *Introduction to Reconfigurable Computing: Architectures, Algorithms and Applications*, Springer, Dordrecht, The Netherlands, 2007.

[7] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: from prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[8] D. E. Taylor and J. S. Turner, "ClassBench: a packet classification benchmark," in *Proceedings of the 24th IEEE International Conference on Computer Communications (INFOCOM '05)*, pp. 2068–2079, Miami, Fla, USA, March 2005.

[9] 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, New York, NY, USA, December 2007.

[10] 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.

[11] G. S. 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.

[12] 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.

[13] Y.-K. Chang, C.-I. Lee, and C.-C. Su, "Multi-field range encoding for packet classification in TCAM," in *Proceedings of the 30th IEEE International Conference on Computer Communications (INFOCOM '11)*, pp. 196–200, Shanghai, China, April 2011.

[14] 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, September 2008.

[15] 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.

[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, April 2008.

[17] O. Ahmed, S. Areibi, K. Chattha, and B. Kelly, "PCIU: Hardware implementations of an efficient packet classification algorithm with an incremental update capability," *International Journal of Reconfigurable Computing*, vol. 2011, Article ID 648483, 21 pages, 2011.

[18] 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, usa, May 2010.

[19] 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.

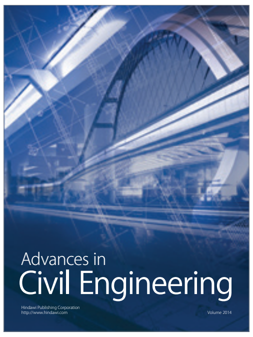
[20] W. Jiang and V. K. Prasanna, "Scalable packet classification on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, no. 99, pp. 1668–1680, 2011.

[21] O. Ahmed, S. Areibi, and G. Grewal, "Hardware accelerators targeting a novel group based packet classification algorithm," *Journal of Reconfigurable Computing*, vol. 2013, Article ID 681894, 33 pages, 2013.

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

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

[24] Xilinx, "C-based design: high level synthesis with vivado hls," 2013, http://www.xilinx.com/training/dsp/high-level-synthesis-with-vivado-hls.htm.



Hindawi

Submit your manuscripts at
http://www.hindawi.com