Chapter 5
Implementation & Results

5.1 Pre-processor:

To calculate the memory size of each AC-tree, we use information from Table 5-1, which shows the number of bytes for each sub-AC and for each original AC.

Table 5-1. Memory size of modified AC

		Class	Size (byte)
modified AC	AC 0	http + anycase	150612
	AC 1	Sql + anycase	207672
	AC 2	tcp:110,119,139,443,445,53,25 + anycase	72384
	AC 3	telnet, ssh, ftp + anycase	67416
	AC 4	Udp + anycase	55644
	AC 5	ip, icmp + anycase	60216
	AC 6	Other tcp service + anycase	86532
	total		700476
original AC			611772

To measure the reduction ratio of the original AC and the modified AC multi-pattern match algorithm we looked at three types of network behavior including an attack context, a normal context and a clean context. The attack profile is traffic which includes 313 Snort rules where each packet has more than two attack intensions, the normal case is real traffic from the Internet, and the clean traffic was sifted for all attack intentions from real traffic. The numbers of the original AC pattern match and numbers of the modified AC multi-pattern match are given in Table 5-2.

Table 5-2. Number of matching patterns

Total number of matches	Attack traffic	Real traffic	Clean traffic
original AC	4578349	1896436	1335860
modify AC	1334649	591039	452364
reduction rate	71.85%	68.83%	67.14%

After modifying the AC multi-pattern match algorithm, there is only a 10% memory space increase, and 60~70% of the unnecessary pattern match information is filtered. This substantially reduces the overhead of the post-processor.

5.2 Post-processor:

This system is implemented on an Altera development board Cyclone 1C20. The specification is as follows in Table 5-3 [33].

Table 5-3. The specification of development platform

FPGA	EP1C20FC400
Logic Element	20060
On-Chip memory	294 Kbits
I/ O pins	301
PLLs	2
micro-processor [34]	32-bits Nios processor (200 MHz)

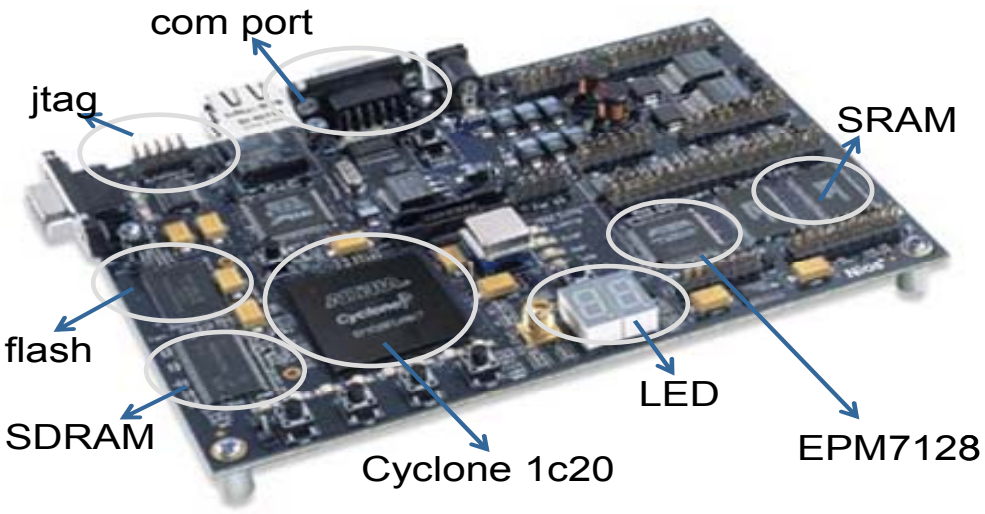

Figure 5-1. The diagram of Cyclone(1C20)

We implement software and hardware respectively.

SoC-based software:

Four functions:

1. Read data from input buffer and filter by EGFE (EventGroupFilter Engine) then forward to particular EventTable entry.
2. Read data from RuleResultBuffer.
3. Control CRME (correlation match engine) and other devices.
4, Receive the control command from UART interface.

These four functions are implemented by the GNU-C.

EGF (Event Group Filter) logic:

We attempt to implement the EGF algorithm using two approaches:

1. Software solution.
2. Customized instruction solution.

For the software-based solution, use the pseudo-code is as follows:

```
procedure CGF( var EventID, EventAddr of integer )
    if (EventID is first event) and (table not full)
        insert to table;
    else if (EventID not first event)
        find entry i from 1 to next_FEL-1 which FEL_GID[ i ]==EventID
            FEL_COUNT[ i ] ← FEL_COUNT[ i ] + 1;
end CGF;
```

The other solution is to add micro-instructions to the micro-processor. This means that when we want to carry out an EGF operation, we only need to call the macro “ALT_CI_EGFE (EventID, EventAddr);”. The user-defined logic has five

states in the finite state machine (FSM). The state translation diagram is shown in Figure 5-2.

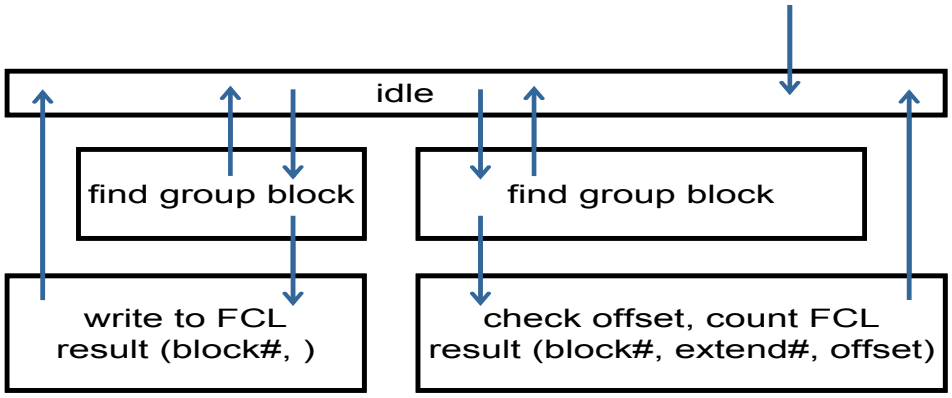

Figure 5-2. The state translation diagram of hardware-based EFG algorithm

We used a software solution and a customized instruction solution to measure the latencies for 500 EGF (EventGroupFilter) operations. This solution increased the speed of the operations up to 30 times (Table 5-4).

Table 5-4. The latency of two solutions

	Software	Customization Instruction	speed up
latency	4.414 ms	0.1471 ms	30.02

BCRME: (Binary Correlation Match Engine)

We designed nine states in the finite state machine to execute the Binary Correlation Match (BCRM) algorithm. The state translation diagram is shown in Figure 5-3.

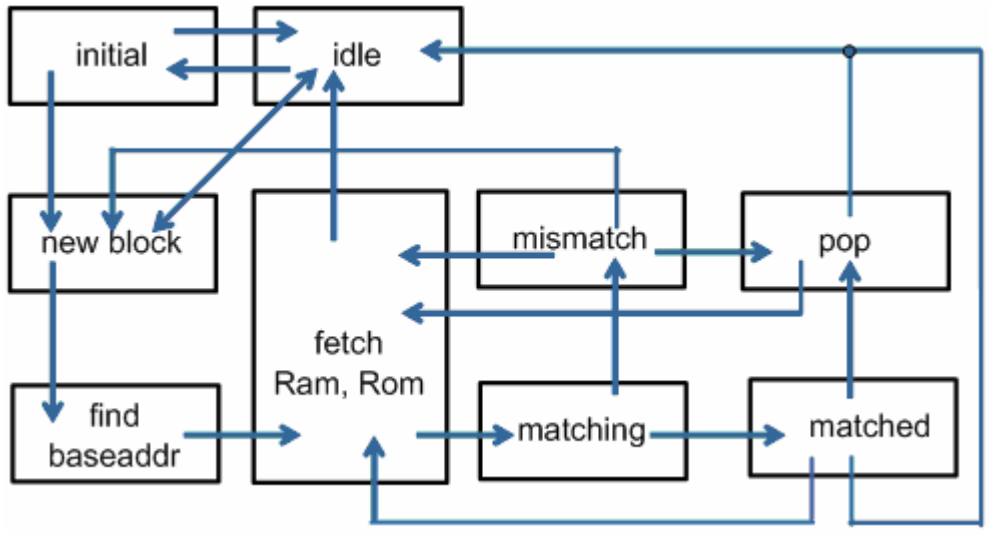

Figure 5-3. The state translation diagram of CRM algorithm

Consider the FPGA gate count; we can only partition the system into four parallel match engines. Four CRME can output RuleID to RuleResultBuffer at the same time, so the system needs an additional logic circuit to handle the order (Figure 5-4). Ordering the logic is a FIFO (first in first output) device, which schedules all RuleIDs that need to write to the RuleResultBuffer memory.

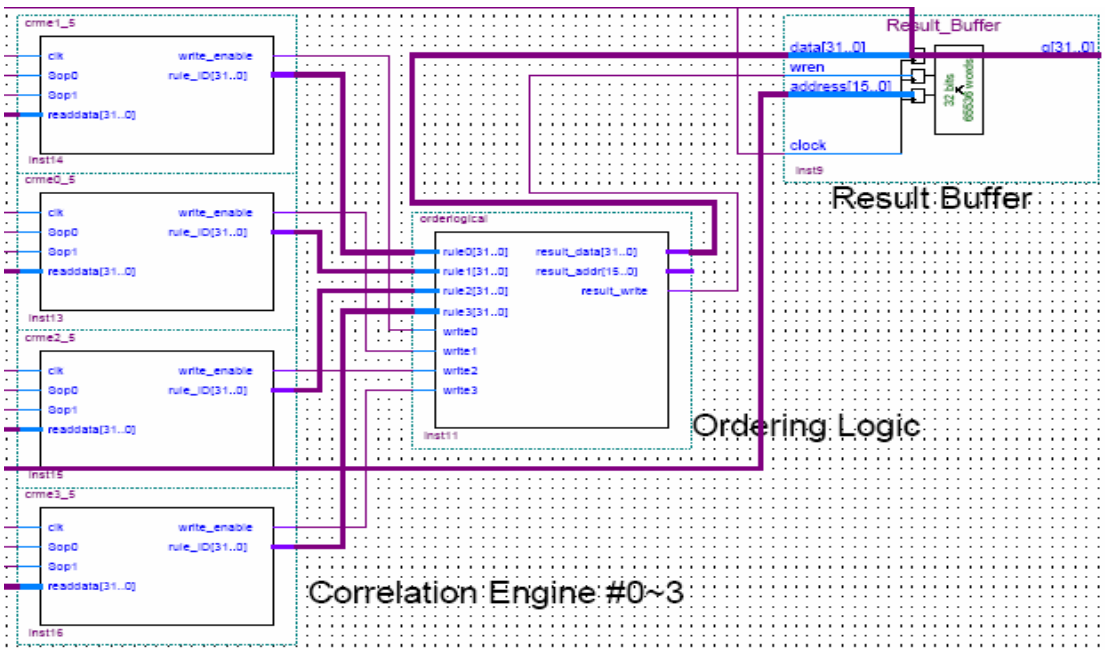

Figure 5-4. The logic circuit with CRME, ordering logic and Result Buffer

Total system:

The following table (Table 5-5) shows the resource utility rate of this thesis.

Figure 5-5 shows the SoC-based system critical diagram.

Table 5-5. The resource utility rate

	Quantity	Util. rate
Number of CRMEs	4	
Number of customization instructions	4	
LE	15146	75.50 %
On-chip memory	237696	80.85 %
I / O pins	176	58.47 %
PLLs	2	100 %

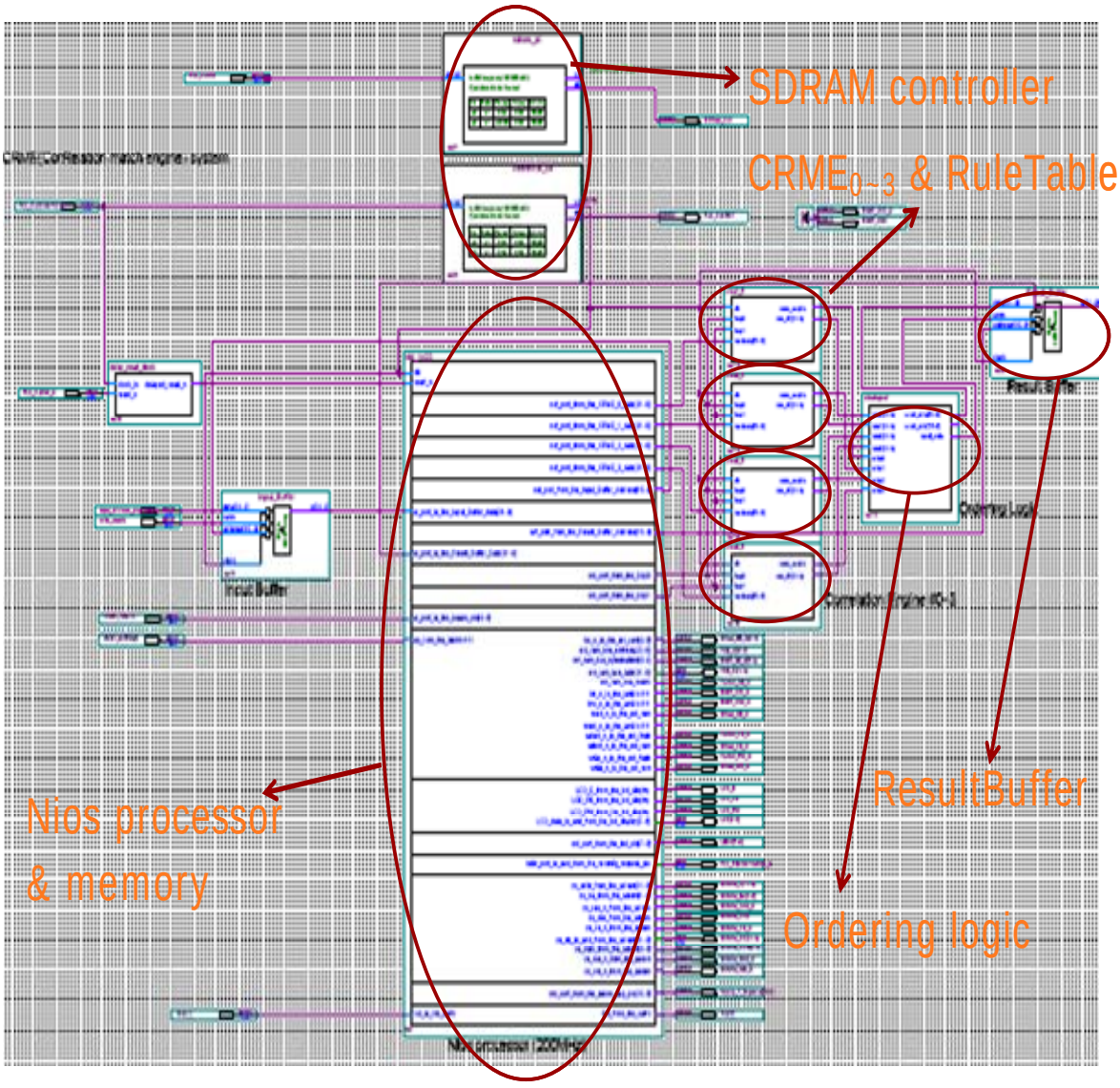

Figure 5-5. The SoC-based system critical diagram

5.3 Result:

We constructed two systems: a SoC-based system and a pure software-based system. This thesis uses the SoC-based system and the original Snort Detection Engine employs a pure software-based system. They use the same processor (200-MHz Nios processor) to execute correlation match operations.

First, we conducted an experiment to test the SoC-based system. We employed the open-source software Snort using constructed attack packets which included 313 Snort rules. We did this to observe whether the SoC-based system can thoroughly detect all rules which are similar to the Snort detection engine.

Second, we gathered the same amount of data under three types of network traffic:

Case1: Attack traffic. Each packet contains two or more rules.

Case2: Real traffic. This case was real traffic from Defcon9.

Case3: Clean traffic. Guarantee there never have the attack intention.

Third, in order to observe how system performance changes under more complicated conditions, we defined a variable to represent more complicated input data. The formula is as follows:

$$\text{ComplexRate} = \frac{|\ \text{ComplexGroup}\ |}{|\ \text{ComplexGroup}\ | + |\ \text{SimpleGroup}\ |}$$

Input buffer was partitioned into two different parts: simple blocks and complex blocks. Simple blocks collect all events which are in the ComplexGroup, complex block collect all events which are in the SimpleGroup. Therefore, we can use the access ratio of the two block types to control the complexity of input data. Figure 5-6 shows the relationship between input buffer and complex rate.

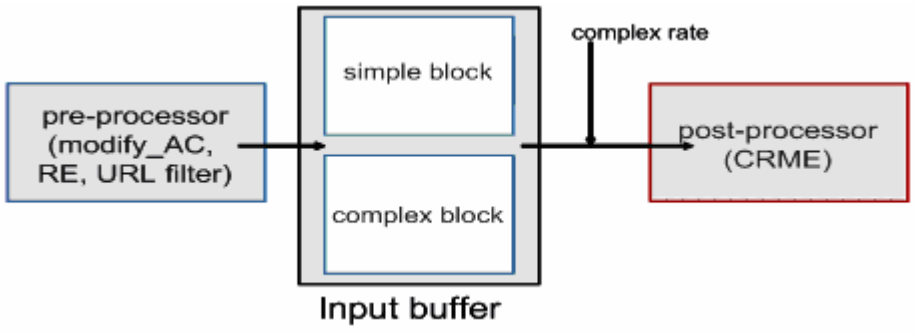

Figure 5-6. The relationship between input buffer and complex rate

Under the SoC-based system and pure software system, we measure the latency using three traffic types (Table 5-6(a), (b), (c), Figure5-7(a), (b), (c)), and to calculate the increased speed of the SoC-based system (Table 5-7, Figure 5-8).

Table 5-6(a). Case1 latency (ms)

system / complex rate	Software-based	SoC-based
0%	5527.9228	652.4575
12.5%	5547.2358	667.1900
25%	5773.4644	653.2501
37.5%	6000.5390	652.1861
50%	6371.2378	656.1399
67.5%	6773.7817	655.9780
75%	7311.4179	651.1978
87.5%	7895.2158	653.3099
100%	8710.4180	650.9666

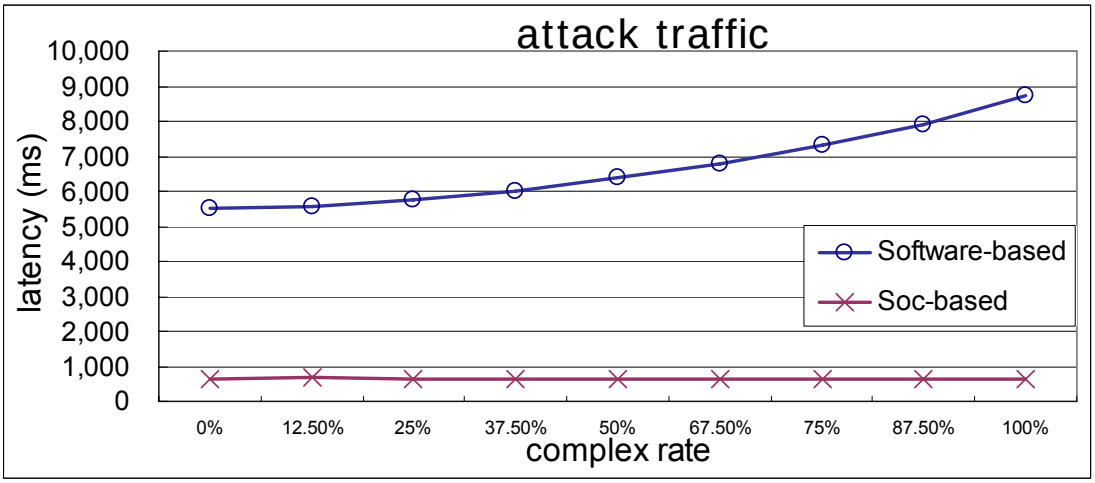

Figure 5-7(a). Case1 latency (ms)

Table 5-6(b). Case2 latency (ms)

system / complex rate	Software-based	SoC-based
0%	2436.3835	286.9169
12.5%	2459.3342	287.0807
25%	2584.4908	291.7757
37.5%	2726.6644	287.4409
50%	2816.0556	288.8588
67.5%	3030.4799	288.9143
75%	3341.3897	286.8515
87.5%	3567.2007	287.8539
100%	3956.4293	286.9140

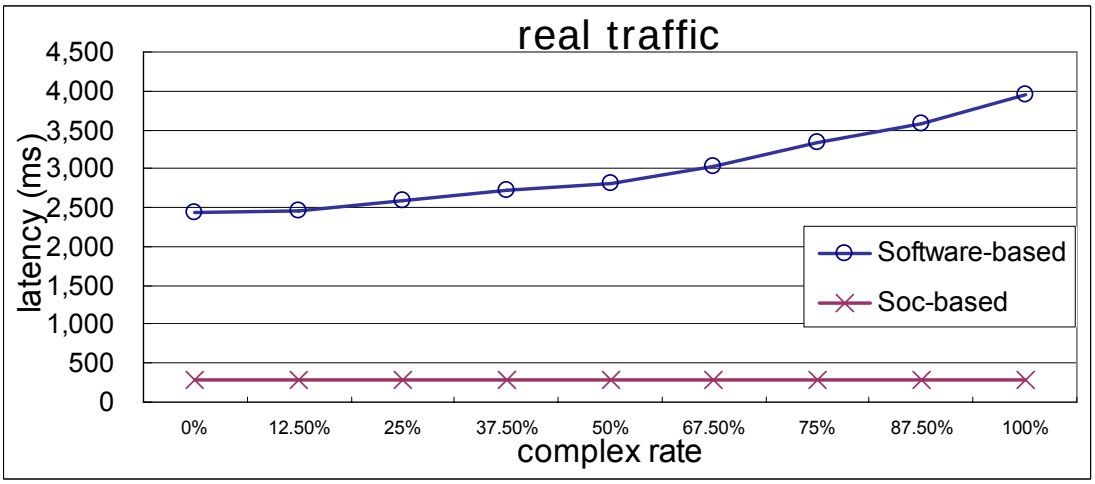

Figure 5-7(b). Case2 latency (ms)

Table 5-6(c). Case3 latency (ms)

system / complex rate	Software-based	SoC-based
0%	2158.7727	251.6503
12.5%	2181.9585	249.4379
25%	2253.5591	251.4102
37.5%	2424.2959	250.7783
50%	2575.8865	251.9172
67.5%	2718.8455	249.9839
75%	2950.3328	251.7770
87.5%	3173.9909	250.9199
100%	3500.6228	251.0464

Figure 5-7(c). Case3 latency (ms)

Table 5-7. Speedup: (SoC-based / Software-based)

traffic / complex rate	attack flow	Normal flow	clean flow
0%	8.4725	8.4916	8.5785
12.5%	8.3143	8.5667	8.7475
25%	8.8380	8.8578	8.9637
37.5%	9.2007	9.4860	9.6670
50%	9.7102	9.7489	10.2251
67.5%	10.3262	10.4892	10.8761
75%	11.2276	11.6485	11.7180
87.5%	12.0849	12.3924	12.6494
100%	13.3807	13.7896	13.9441

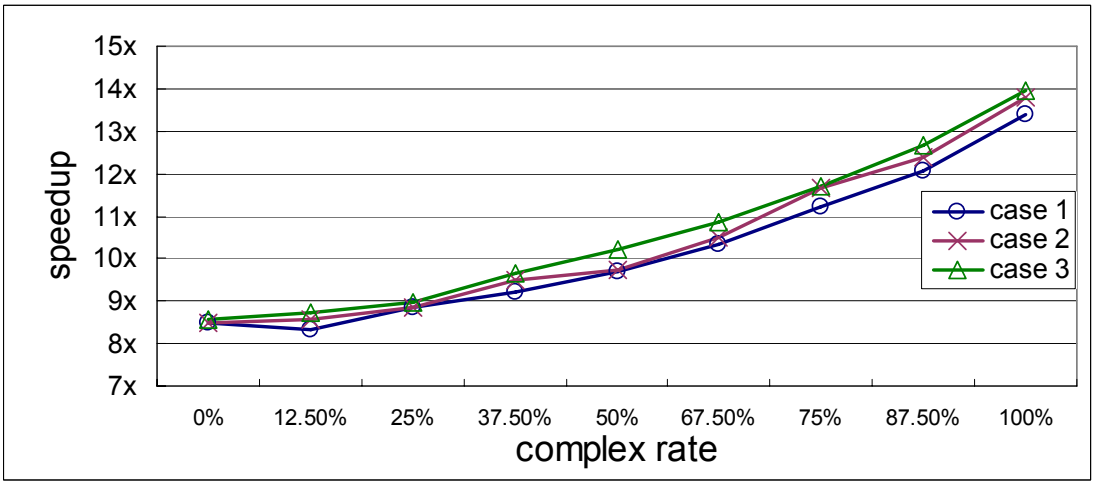

Figure 5-8. Speedup: (SoC-based / Software-based)

We observe three results:

1. Because the real traffic has limited attack intention, case 2 and case 3 are very similar.
2. On the pure software-based system, the event complexity is related to latency. On the SoC-based system, latency is not related to content complexity. Even with highly complex data, which contain group number increases, the performance is better.
3. The SoC-based system is 8~14 times faster.

Ignoring the pre-processor effect, we calculate the maximum throughput of the SoC-based system (Table 5-8). Max throughput is 400 Mbps, the lowest throughput being 153 Mbps. In comparison, a pure software-based system can only achieve 50Mbps.

Table 5-8. SoC-based performance (Mbps)

traffic / complex rate	attack flow	Normal flow	clean flow
0%	153.2667	348.5329	397.3768
12.5%	149.8823	348.3341	400.0901
25%	153.0807	342.7290	397.7563
37.5%	153.3305	347.8976	398.7585
50%	152.4065	346.8990	396.9558
67.5%	152.4441	346.1234	400.0276
75%	153.5630	348.6124	397.1769
87.5%	153.0667	347.3985	398.5336
100%	153.6177	348.5365	398.3327