시그내쳐 기반의 네트워크 침입 방지에서 고속의 패킷 필터링을 위한 시스템 구조

(A High-speed Packet Filtering System Architecture in Signature-based Network Intrusion Prevention)

김 대 영 † 김 선 일 †† 이 준 용 ††
(Dae Young Kim) (Sun Il Kim) (Jun Yong Lee)

요 약 네트워크 침입 방지에서 공격 패킷은 시그내쳐에 기반을 둔 방법에 의해 발견되어 제거된다. 패턴 매칭(Pattern Matching)은 공격 시그내쳐를 발견하기 위해 광범위하게 사용되고 있고, 또한 네트워크 침입방지 시스템에서 시간적으로 가장 많이 수행되는 부분이다. 네트워크 침입방지 시스템에 사용되는 패턴 매칭은 주로 하드웨어를 사용하여 가속화되며 회선 속도로 수행되어야 한다. 그러나 이것만으로는 충분치 않고 다음과 같은 조건들이 더 요구된다. 첫째, 패턴 매칭 하드웨어는 패턴 인덱스 번호와 패턴 발견 위치를 포함한 충분한 패턴 매칭 정보를 회선 속도에 맞게 제공해야 한다. 둘째, 불필요한 패턴 매칭을 줄이기 위한 패턴 그룹을 지원할 수 있어야 한다. 셋째, 패턴의 개수가 증가하더라도 최저 성능을 보장 할 수 있어야 한다. 마지막으로, 수행 중단 없이 몇분 또는 몇초 이내에 패턴 업데이트가 가능해야 한다. 본 논문에서는 위의 요구사항을 만족하는 시스템 구조를 제안한다. 이 시스템은 여러 개의 패턴 문자를 동시에 처리하고 파이프라인 구조를 사용하여 고속의 처리를 가능케 한다. Xilinx FPGA 시뮬레이션을 통해 제안된 시스템이 10Gbps 이상의 속도에서 동작하며 위의 모든 요구사항을 만족시킴을 보였다.

키워드 : 네트워크 침입 방지, 패턴 매칭, 내장형 시스템 구조, FPGA

Abstract In network intrusion prevention, attack packets are detected and filtered out based on their attack signatures. Pattern matching is extensively used to find attack signatures and the most time-consuming execution part of Network Intrusion Prevention Systems(NIPS). Pattern matching is usually accelerated by hardware and should be performed at wire speed in NIPS. However, that alone is not good enough. First, pattern matching hardware should be able to generate sufficient pattern match information including the pattern index number and the location of the match found at wire speed. Second, it should support pattern grouping to reduce unnecessary pattern matches. Third, it should always have a constant worst-case performance even if the number of patterns is increased. Finally it should be able to update patterns in a few minutes or seconds without stopping its operations, We propose a system architecture to meet the above requirement. The system architecture can process multiple pattern characters in parallel and employs a pipeline architecture to achieve high speed. Using Xilinx FPGA simulation, we show that the new system scales well to achieve a high speed over 10Gbps and satisfies all of the above requirements.

Key words : Network intrusion prevention, Pattern matching, Embedded system architecture, FPGA

1. 서 론

인터넷의 폭발적인 성장, 주문형 비디오, 전자 상거래, P2P를 응용한 파일 공유와 같은 새로운 애플리케이션의 등장은 네트워크 트래픽을 급격히 증가시켰다. 고속의 인터넷 접근과 높은 대역폭에 대한 요구를 만족시키기 위해 네트워크 속도 및 대역폭도 급격히 증가하고 있다. 이러한 추세로 인해 분산 서비스거부 공격(DDos), 이메

· 이 논문은 2006학년도 홍익대학교 학술연구진흥비에 의하여 지원되었음
† 학생회원 : 홍익대학교 컴퓨터공학과
hansol@cs.hongik.ac.kr
†† 종신회원 : 홍익대학교 컴퓨터공학과 교수
sikim@cs.hongik.ac.kr
jlee@cs.hongik.ac.kr
논문접수 : 2006년 1월 19일
심사완료 : 2006년 11월 13일

일 바이러스, 인터넷 웜(internet worm) 등의 악의적인 네트워크 공격 또한 급속히 빨라지고 더욱더 파괴적으로 변모해가고 있다. 예를 들어 Code Red Worm[1]이나 SQL Slammer Worm[2]은 수 시간 내지 수 분만에 전 세계로 퍼져서 네트워크를 마비시켜 막대한 피해를 입혔다.

이러한 네트워크 공격에 대한 방어 수단 중에서 최근 가장 각광받는 기술이 네트워크 침입방지 시스템(Network Intrusion Prevention System: NIPS)이다. NIPS는 방화벽(firewall)과 네트워크 침입탐지 시스템(Network Intrusion Detection System: NIDS)[3]을 결합한 형태로서, 패킷 헤더와 데이타(payloads)를 조사하는 NIDS의 기능과 공격 패킷을 차단하는 방화벽의 기능을 동시에 지니고 있다. 네트워크 침입방지 시스템은 네트워크 경로의 중간에 위치하여 실시간으로 패킷을 처리하기 때문에, 처리 속도가 전체 네트워크의 속도에 직접적인 영향을 미친다. NIPS의 패킷 처리 과정 중, 패킷 데이타에서 공격 문자열 패턴을 찾는 패턴 매칭이 핵심 기능이며 상당한 계산량을 요구한다. 예를 들면 NIDS/NIPS로 널리 사용되는 Snort[4]에서 패턴 매칭 수행 부분은 전체 수행 시간의 70%, 전체 명령의 80%를 차지한다[5].

NIPS의 패턴 매칭은 몇 가지 도메인 특성을 지닌다[6-8]. 첫째, 패턴의 개수가 매우 많고 계속 늘어난다. 둘째, 패턴의 길이는 대부분 32바이트 이하이지만, 전체적으로는 1에서 122까지 다양하다. 작은 길이의 패턴은 일부 패턴 매칭의 성능에 저하를 가져올 수 있다. 셋째, 대부분의 문자열 패턴은 대소문자를 구분하지 않는다. Snort에서 사용되는 문자열 패턴의 반 이상은 대소문자를 구분하지 않는다. 현재까지 도메인 특성의 일부나 전부를 만족시키는 패턴 매칭 하드웨어를 개발하는데 많은 연구가 집중되어왔다. 그러나 NIPS 하드웨어가 유용하고 효율적으로 동작하기 위해서는 빠르고 정확한 패턴 매칭이외에도 다른 중요한 특성들을 만족하여야한다. 이들 특성은 다음과 같다.

1) 회선 속도로 패턴 매칭 정보 제공 - 최소한 패턴 인덱스 번호와 패킷에서의 패턴의 위치 정보가 제공되어야 한다.

매칭되는 패턴이 발견되었을 때, 규칙 검사 소프트웨어가 다른 규칙 옵션이 만족되는지 조사한다. Snort는 관련된 규칙을 발견하기 위해 패턴 인덱스 정보를 사용하고, 발견된 패턴이 '*depth*'와 '*offset*'과 같은 옵션을 만족하는지 결정하기 위해 위치 정보를 사용한다. depth는 패턴을 찾기 위해 패킷 데이타를 어디까지 조사해야하는지를 나타내고, offset은 패킷에서 어디부터 조사를 시작 하는지를 나타낸다. 이 정보들은 매칭되는 모든 패턴에 대하여 회선 속도에 맞추어 제공되어야 한다. 그렇지 못하면, 발견된 패턴에 대한 이러한 정보를 처리하기 위해 속도가 늦어지고 결국에는 회선 속도에 맞출 수 없기 때문이다.

2) 패턴 그룹 지원 - 패킷이 속해 있는 규칙 그룹과 관련된 패턴들만 조사

Snort에서 규칙들은 프로토콜, 발신지 및 목적지 포트 번호를 기준으로 규칙 그룹으로 구분된다. 들어오는 패킷은 프로토콜, 발신지/목적지 포트에 따라 분류되어 해당 규칙 그룹에 속한 패턴에 대해서만 검사된다. 따라서 하드웨어에서 패턴 그룹을 지원하지 않으면, 해당 패킷과 관련성이 없는 많은 패턴들에 대해서 모두 검사하게 되므로, 결과적으로 불필요한 소프트웨어 수행을 야기한다.

3) 최저 성능 보장

이 요구조건은 [6,9]에서도 언급되어 있다. NIPS의 최저 성능은 네트워크 속도와 일치해야 한다. 그렇지 않으면, 공격자가 NIPS에서 처리가 가장 오랜 시간이 걸리도록 패킷의 내용을 조작하고, 계속해서 이렇게 조작된 패킷을 보내 NIPS가 과부하가 걸리도록 한다. 결과적으로 과부하가 걸린 NIPS는 다른 정상적인 트래픽까지 처리를 못 하게 된다. 또한 최저 성능은 새로운 패턴이 추가됐을 때도 일정해야 하며 그 성능을 예측할 수 있어야 한다. 그렇지 않다면, NIPS가 최저 속도 요구조건을 언제 만족시킬 수 없게 되는지를 예측할 수 없기 때문이다.

4) 수행 중단 없는 빠른 패턴 업데이트

SQL Slammer 웜과 같이 빠른 속도로 퍼지는 인터넷 웜으로부터 네트워크를 보호하는데 중요한 요구조건이다. NIPS에서 사용되는 Content 필터링은 새로운 웜이 발생한 후 몇 분 이내에 해당 웜을 걸러내기 시작해야 웜의 전파를 차단시킬 수 있다[10]. 따라서 패턴 매칭 하드웨어는 인터넷 웜의 차단을 위해서 몇 분 또는 몇 초 이내에 패턴을 업데이트 할 수 있어야 한다. 최근의 웜 출현으로 인한 피해를 고려해볼 때, 이 요구조건은 매우 중요하다. [9]에서 제시된 패턴 매칭 구조만이 이 요구 조건을 만족하고 있다.

FPGA[11-20]에 기반을 둔 대부분의 패턴 매칭 하드웨어는 최저 성능 보장조건을 잘 만족시키지 못한다. 패턴의 개수가 증가하게 되면, 상태 전이를 위한 조합 회로의 증가로 인해 FPGA 패턴 매칭 하드웨어의 동작 주파수가 줄어들게 된다. 이러한 점이 NIPS의 성능 예측을 힘들게 하고 결과적으로 NIPS의 성능이 네트워크 속도보다 느려지게 되는 요인이 된다. 또한 이런 방식들은 새로 추가되는 패턴을 위해 FPGA를 다시 프로그램하고 합성해야하는데, 많은 수의 패턴에 대해 이 작업을

수행하려면 보통 시간이 오래 걸리므로, 대부분의 경우 네 번째 요구조건인 수행중단 없는 빠른 업데이트를 만족시키지 못한다. Aho-Corasick 알고리즘[21]에 기반을 둔 Bit-split FSM 방식[9]은 상태 전이 테이블로 SRAM을 사용한다. 이 방식은 최저 성능과 수행중단 없는 빠른 업데이트를 만족시킬 뿐만 아니라, 성능도 뛰어나며 하드웨어 자원 활용도 우수하다. Shift-OR 알고리즘[22]를 하드웨어로 구현한 Shift-OR 패턴 매칭 가속기[8]는 Bit-split FSM처럼 패턴에 관한 정보가 테이블에 저장된다. 따라서 패턴이 추가되었을 경우에도 FPGA를 기반으로 하는 방법과 달리 최저 성능을 보장할 수 있다. 하지만 [8]에서는 수행 중단 없이 패턴을 업데이트하는 방법은 제시되어 있지 않다.

현재까지 제시된 패턴 매칭 하드웨어는 패턴 매칭을 하드웨어 자원을 가능한 적게 사용하며 빠른 속도를 성취하는데 초점을 두어왔다. 극히 일부 연구만이 최저 성능보장과 수행 중단 없는 빠른 패턴 업데이트를 할 수 있는 패턴 매칭 하드웨어를 제시하였다. 하지만 이것만으로는 위에서 언급된 네 가지 조건을 만족하는 효율적인 NIPS을 구성하지 못한다. 본 논문에는 회선 속도의 패턴 매칭 정보와 패턴 그룹을 지원하는 패턴 매칭 시스템 구조를 제안한다. [19,20]의 두 논문에서는 패턴 매칭 정보 요구조건과 관련된 쟁점을 언급하였고 pruned 우선순위 이진트리와 파이프라인 이진 OR 트리를 사용하여 시그내쳐 인덱스를 생성하는 유사한 두 구조를 제안하였다. 그러나 이 구조들은 동시에 발생하는 다중 매칭을 처리할 수 없고, 또한 패킷 데이타에서의 매칭이 일어난 위치에 대한 정보를 제공하지 못한다. Snort에서 사용되는 문자열 패턴을 분석해보면 많은 suffix 패턴 매칭이 존재함을 발견하였다. 동일한 suffix 매칭 그룹에서 패턴의 최대 개수는 5이다. 이는 주어진 입력 문자들에서 최대 5개의 다중 매칭이 일어날 수 있음을 의미한다. 본 논문에서 제시한 구조는 이러한 다중 매칭도 회선 속도로 처리 할 수 있다. 마지막으로 본 논문에서는 간단한 하드웨어 구조를 Shift-OR 패턴 매칭 가속기에 추가함으로써 수행 중단 없이 빠른 패턴 업데이트를 가능하게 하였다. 제시된 패턴 매칭 시스템 구조와 개선된 Shift-OR 패턴 매칭 가속기를 사용함으로써 앞에서 언급된 4가지 요구조건을 모두 만족하면서 회선 속도로 동작 할 수 있는 시스템을 구성 할 수 있다.

본 논문에서는 제안된 시스템 구조를 Xilinx FPGA 툴을 이용하여 평가하였다. FPGA 시뮬레이션을 통해 시간과 자원 결과를 얻었다. 2장에서는 Shift-OR 알고리즘[22]을 간략히 설명하고, 3장에서는 개선된 Shift-OR 패턴 매칭 가속기를 설명하고 4장에서는 패턴 그룹 하드웨어와 패턴 매칭 정보 시스템을 설명한다. 5장에서는 Xilinx FPGA 툴을 사용하여 평가하고, 6장에서 논문의 결론을 맺는다.

2. Shift-OR 패턴 매칭 알고리즘

여기서는 Shift-OR 패턴 매칭 가속기의 기본이 되는 다중 패턴에 대한 Shift-OR 패턴 매칭 알고리즘[22]에 대하여 간단히 소개한다. Shift-OR 알고리즘은 비트 연산에 기본을 두고 있고, 크기가 m (패턴의 길이)인 패턴 매칭의 상태를 나타내는 비트 벡터인 상태 벡터 R과 패턴에서 특정 문자 c의 위치를 나타내는 문자 위치 벡터 S_c가 있다. 예를 들면, 패턴 $P = p_0...p_{m-1}$와 입력 문자열 $X = ...x_{i+j}...$이 있으면, x_{i+j}를 처리하고 난후 $x_i...x_{i+j}$가 $p_0...p_j$와 일치하면 R의 j 번째 비트, $R[j]$=0 이고, 그렇지 않으면 $R[j]$=1 이다. 또 문자 위치 벡터의 경우 $P_i=c$ 이면 $S_c[i]$=0 이고 그렇지 않으면 $S_c[i]$=1 로 설정된다. 입력 문자 x_{i+j} 를 처리한 후 $R[j]$ 값을 얻으면 다음 입력 문자 x_{i+j+1}가 패턴의 P_{j+1} 위치에 나타나는가를 가지고 $R[j+1]$를 계산할 수 있다. $R[j+1]$ 값은 다음 수식과 같이 정의된다.

$$R[j+1] = \begin{cases} 0 & R[j]=0 \text{ and } S_c[j+1]=0, \text{ where } c = x_{i+j+1} \\ 1 & \text{otherwise} \end{cases} \quad (1)$$

$$R[0] = S_c[0], \quad \text{where } c = x_{i+j+1} \quad (2)$$

$R[m-1]$=0은 입력 문자열 $x_i...x_{i+m-1}$가 패턴의 $p_0...p_{m-1}$과 일치했다는 것을 나타낸다. 즉, 패턴과 일치하는 문자열을 발견한 것이다. 상태 비트 벡터 R을 계산하는 것은 위의 식에서 볼 수 있듯이 간단한 Shift와 OR 비트 연산으로 구성된다. 즉, 상태 벡터 R을 한번 Shift한 뒤, 문자 위치 벡터 S_c와 OR 연산함으로써 위의 식을 만족하는 새로운 상태 벡터 R을 계산하게 된다($SHIFT(R)$ OR S_c).

Shift-OR 알고리즘은 패턴에 유한 문자 집합, 부정 문자(complement symbol), don't care 문자가 있는 경우를 쉽게 다룰 수 있다. 만약 패턴의 위치 i에 문자 집합 $\{x, y, z\}$가 매칭될 수 있다면 $S_x[i] = S_y[i] = S_z[i] = 0$로 설정함으로써 이 경우를 처리할 수 있다. 부정 문자나 don't care경우도 같은 방법으로 처리될 수 있다. 따라서 대소문자 구분 없는 패턴 매칭의 경우 다른 오버헤드 없이 문자위치 벡터의 설정만으로 처리된다. 또한 Shift-OR 알고리즘은 쉽게 다중 패턴을 처리하도록 확장될 수 있다. 각 패턴에 대한 상태 벡터 R을 패턴의 순서대로 연결하여 합치고, 주어진 문자 c에 대하여 각 패턴의 문자 위치 벡터 S_c를 모두 연결하여 합쳐서 하나의 문자 위치 벡터를 만든다. 이 합쳐진 상태 벡터와 문자 위치 벡터에 Shift-OR 비트 연산을 적용하면 된다. 단지 각 패턴의 첫 번째 위치에 해당하는 R의

비트 값 $R[i]$를 계산할 때 $R[i-1]$은 무시하고 계산하는 것이 다를 뿐이다.

3. 패턴 매칭 유닛 (PMU): 개선된 Shift-OR 패턴 매칭 가속기

본 논문에서는 [8]에서 제시된 Shift-OR 패턴 매칭 가속기를 기반으로 수행 중단 없는 빠른 패턴 업데이트 기능을 보강하여 패턴 매칭 유닛(Pattern Match Unit: PMU)을 구성한다. 기존 Shift-OR 패턴 매칭 가속기에서는 패턴을 업데이트 할 때, 최소한 문자 위치 벡터를 저장한 테이블과 패턴 경계 벡터 레지스터가 변경되어야하는데 이에 대한 구체적인 방법이 제시가 안 되어 있다. 본 논문에서 제안하는 패턴 매칭 유닛에서는 매칭 위치 벡터 레지스터를 추가하고 다중 포트 SRAM을 사용하여 필요한 레지스터를 변경함으로써 패턴의 변경을 수행 중단 없이 실행하게 한다. 또한 문자 위치 벡터 테이블을 SRAM에 저장함으로써 내장형 메모리를 가지는 FPGA에 구현하기 쉽게 된다. 패턴 매칭 유닛은 본 논문에서 제안하는 패턴 매칭 시스템에서 패턴 매칭의 기능을 수행하며 최저 성능 보장과 수행 중단 없는 빠른 패턴 업데이트의 특성을 만족시킨다. 전체 패턴의 크기에 따라 패턴 매칭 시스템은 여러 개의 패턴 매칭 유닛(PMU)으로 구성되어 있다.

패턴 매칭 유닛은 다중 패턴에 대하여 Shift-OR 패턴 매칭 알고리즘을 수행한다. 그림 1은 패턴 매칭 유닛의 구성 요소들을 보여준다. PMU는 8비트 문자 하나에 대하여 256개의 문자 위치 벡터를 저장하는 다중포트 SRAM을 갖고 있다. 패킷 데이타(payload)로부터 여러 개의 문자를 읽어서 다중 포트 SRAM의 어드레싱에 사용한다. SRAM의 크기는 256 x W 비트이며, 여기서 W는 SRAM의 폭으로 PMU가 사용하는 문자 위치 벡터의 크기이기도하다. 문자 위치 벡터는 PMU에서 할당된 모든 패턴에 대하여 각각의 문자의 위치 벡터들을 하나로 연결하여 합친 것이다. 이 벡터들은 문자열 패턴으로부터 미리 계산하여 SRAM에 저장된다. SRAM의 포트 수 N은 동시에 처리할 수 있는 입력 문자수이다. PMU는 비트 벡터를 저장하기 위해 4개의 레지스터를 갖고 있다. 패턴 경계 벡터(B), 상태 벡터(R), 매칭 위치 벡터(M), 매칭 결과 벡터(F). 모든 벡터 레지스터의 크기는 W 비트로 동일하다.

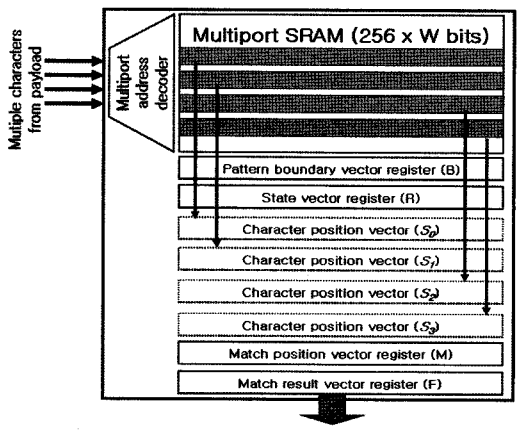

그림 1 패턴 매칭 유닛

우선 다중 포트 SRAM으로부터 읽어 들인 N개의 문자 위치 벡터 $S_0,...S_{N-1}$은 이전 주기(cycle)에서 계산된 상태 벡터(State Vector) R과 패턴 경계 벡터(Pattern Boundary Vector) B를 가지고 OR 연산을 하여 그림 2에서와 같이 새로운 상태 벡터 R을 계산한다. 패턴 경계 벡터는 비트 값 0으로 각 패턴의 경계를 표시하며, Shift-OR 계산 결과가 패턴 경계를 넘어가지 못하도록 방지하는데 사용된다. 처음에는 R과 B가 AND되고 $SHIFT$된 뒤에 S_0와 OR하여 중간 상태 벡터인 T_0를 만든다. T_0는 B와 AND되고 $SHIFT$된 뒤 S_1과 OR하여 다음 중간 상태 벡터인 T_1을 만든다. T_{N-1}이 만들어질 때까지 같은 방법의 계산이 각 단계에서 반복된다. T_{N-1}은 다음 계산 주기에 사용하기 위해 다시 R 레지스터에 저장된다. 이 계산 과정들은 다음 수식으로 표현된다.

$$T_k(0) = S_k(0) + 0 = S_k(0) \quad \text{for all } k \tag{3}$$

$$T_0(i) = S_0(i) + (R(i-1) * B(i-1)) \quad \text{for } i > 0 \tag{4}$$

$$T_k(i) = S_k(i) + (T_{k-1}(i-1) * B(i-1)) \quad \text{for } k > 0,\ i > 0 \tag{5}$$

그림 2에서 보듯이, $SHIFT$의 수행은 단지 i 번째 위치의 결과를 다음 단계의 $i+1$ 번째 위치의 OR 게이트의 한 쪽 입력에 연결함으로써 이루어진다. 각 단계의 계산은 Shift-OR 알고리즘에서 한번의 Shift-OR 연산과 동일하다. N번의 Shift-OR 연산은 한 주기에 처리

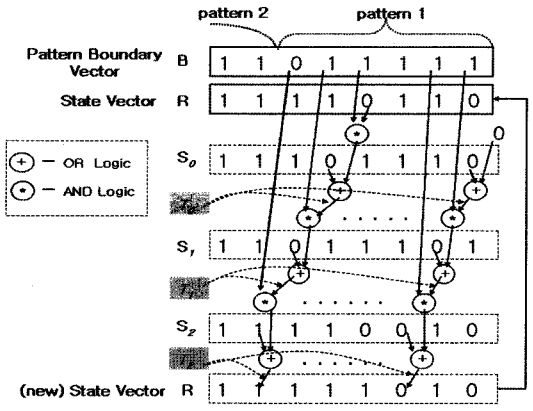

그림 2 Shift-OR 연산

되며, 모든 연산 과정은 조합 회로(Combinatorial circuit)로 구성되어 중간 계산 결과 T_0,... T_{N-1}은 생성되자마자 사용되기 때문에 따로 저장할 필요가 없다. 문자 위치 벡터 S_0,...S_{N-1}도 저장 될 필요 없이 다중 포트 SRAM에서 출력되어 계산에 직접 사용된다.

N개의 입력 문자들이 동시에 처리되는 동안 여러 개의 매칭이 발견될 수 있다. 그림 3은 모든 중간 상태 벡터(T_i)에서 모든 매칭을 발견해내는 구조를 보여준다. 중간 상태 벡터들의 같은 위치에 있는 모든 비트들이 AND된 뒤에, 매칭 위치 벡터 M의 같은 위치에 있는 비트와 OR된다. 매칭 위치 벡터는 각 패턴의 끝에 해당하는 위치가 비트 0으로 되어 있다. 결과는 매칭 결과 벡터인 F 레지스터에 기록된다. 매칭 결과 벡터에 0값을 갖는 비트(매칭 비트)가 있으면 이는 패턴이 매칭되었음을 의미한다. F 벡터 레지스터의 출력이 PMU의 출력이 된다.

PMU는 수행 중단 없는 빠른 업데이트를 할 수 있다. M벡터에서 패턴의 마지막 위치에 해당하는 비트를 1로 설정하면 해당 패턴을 간단히 무시할 수 있고 패턴을 삭제하는 것과 동일한 효과를 얻을 수 있다. 패턴을 추가하는 작업은 패턴 경계 벡터 B와 매칭 위치 벡터 M 레지스터들을 재설정하고, 문자 위치 벡터를 저장하고 있는 다중 포트 SRAM을 다시 로딩함으로써 이루어진다. B와 M레지스터를 업데이트하고 초기화하는 작업은 SRAM에 해당 벡터 값을 써놓고 그 값을 읽어서 각각의 레지스터에 로딩하면 된다. 이를 위해서는 별도의 SRAM을 사용할 수도 있고, 문자 위치 벡터를 저장하는 다중 포트 SRAM의 깊이를 증가시켜 사용할 수도 있다. 새로운 문자 위치 벡터를 다중포트 SRAM에 로딩하는데 걸리는 시간은 새로운 패턴의 길이에 해당하는 주기수 만큼이다. 새로운 패턴에 속한 문자들에 대해서만 문자 위치 벡터를 새로 쓸 필요가 있다. 다중포트 SRAM에 쓰기연산은 읽기 연산을 방해하지 않고서도 수행될 수 있다. 따라서 패턴 업데이트는 시스템을 멈추지 않고도 가능하다.

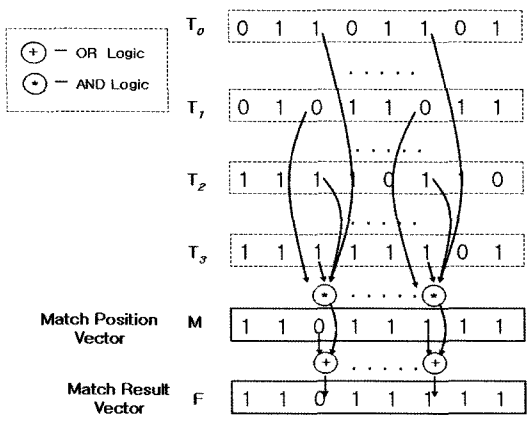

그림 3 매칭 결과 벡터 생성

4. 패턴 그룹 유닛(PGU)과 패턴 매칭 정보 유닛(PMIU)

본 논문에서 제시하는 NIPS를 위한 패턴 매칭 시스템은 PMU 외에도 불필요한 소프트웨어의 수행을 줄이기 위한 패턴 그룹을 지원하는 패턴 그룹 유닛(Pattern Group Unit: PGU)과 회선 속도로 패턴 매칭 정보를 제공하는 패턴 매칭 정보 유닛(Pattern Matching Information Unit: PMIU)으로 구성 되어 있다. PMU와 함께 이들 유닛들이 구성됨으로써 서론에서 제시된 4 가지 조건을 만족하는 시스템이 구성된다.

4.1 패턴 그룹 유닛(PGU)

Snort[4]에서 불필요한 패턴 매칭을 줄이기 위해 규칙들을 규칙에 명시된 프로토콜의 종류와 발신지 및 목적지의 포트 번호에 따라 규칙 그룹으로 분류하고 동일한 규칙 그룹에 속한 패턴들로 패턴 그룹을 형성한다. 들어오는 패킷은 프로토콜, 발신지/목적지 포트에 따라 분류되어 해당 패턴 그룹에 속한 패턴에 대해서만 검사된다. 예를 들어 다음 그림 4와 같은 세 개의 규칙이 있다고 가장 하자. 첫째와 둘째 규칙은 tcp 프로토콜과 목적지 포트가 80이고 데이타에 각각 "pattern1"과 "pattern2"의 문자열을 갖는 패킷에 적용된다. 셋째 규칙은 tcp 프로토콜과 목적지 포트가 50이고 데이타에 "pattern3"의 문자열을 갖는 패킷에 적용된다. "pattern1"과 "pattern2"는 같은 패턴 그룹 i에 속하고 "pattern3"는 다른 패턴 그룹 j로 분류 된다. 패킷의 들어 왔을 때 패킷의 헤더 정보를 보고 해당 패턴 그룹을 찾고 그 그룹에 해당하는 패턴들에 대해서만 패턴 매칭을 수행함으로써 불필요한 매칭을 줄 일 수 있다. 위의 예에서 패킷의 프로토콜이 tcp 이고 목적지 포트가 80인 경우 패킷의 데이타는 패턴 그룹 i 에 속한 패턴에 대해서만 검사가 된다. 패킷 헤더 정보에서 패턴 그룹을 찾는 것은 기존의 소프트웨어 또는 하드웨어 패킷 분류(Packet Classification)기법[23]들을 사용하여 수행 될 수 있다.

PGU는 패턴 그룹의 사용을 하드웨어적으로 지원한다. PGU에 있는 각각의 PMU는 동일한 패턴 그룹에

```
alert tcp any any -> any 80 (content: "pattern1")
alert tcp any any -> any 80 (content: "pattern2")
alert tcp any any -> any 50 (content: "pattern3")
```

그림 4 가상적인 규칙 예

속한 패턴들만을 처리한다. 예를 들면 그림 5에서 첫 번째, 두 번째 PMU는 같은 패턴 그룹 i 에 속한 패턴들만의 정보를 가지고 있고 마지막 PMU는 패턴 그룹 j 에 속한 패턴들만의 정보를 가지고 있는 것이다. 하나의 PMU가 패턴 그룹 i 와 j 의 정보를 동시에 갖지 않는다.

패킷이 들어오면 패킷의 프로토콜의 종류와 발신지 및 목적지의 포트 번호를 조사하여 해당하는 규칙 그룹의 번호를 얻어내고 이를 패턴의 그룹 인덱스로 사용한다. 패턴의 그룹 인덱스는 그림 5에서 보이는 것처럼 패턴 그룹 유닛에 접근할 때 사용된다. PGU에는 SRAM이 있어 특정 패턴 그룹에 대해 활성화 되어야하는 PMU들을 표시하는 벡터 값을 저장하고 있다. PGU는 해당 패턴 그룹을 처리하는 PMU만을 다음과 같이 활성화시킨다. PMU 작동 벡터는 패턴 그룹 인덱스를 통해 SRAM에서 읽혀지고, 벡터의 각 비트는 PMU를 활성화하거나 비활성화 하는데 사용된다. PMU가 비활성화 되면, PMU의 매칭 결과 벡터는 모두 1이 되어 매칭 결과에 영향을 끼치지 않는다.

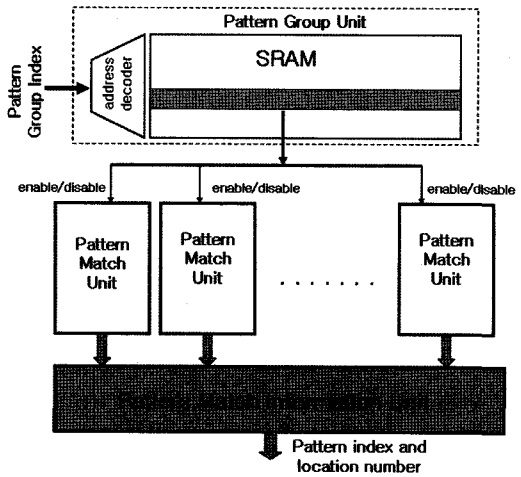

그림 5 패턴 매칭 시스템

4.2 패턴 매칭 정보 유닛(PMIU)

패턴 매칭 하드웨어는 보통 패턴이 매칭되었을 때 신호를 보낸다. 각 패턴 마다 신호가 필요함으로, Snort의 경우 패턴 매칭 하드웨어가 거의 2000개 이상의 신호를 필요로 한다. 소프트웨어가 한 번에 이 모든 신호를 읽을 수 없다. 그래서 소프트웨어가 패턴 매칭 뒤에 규칙을 더 조사를 하려면 신호 인덱스(또는 패턴 인덱스)가 제공되어 관련된 규칙을 쉽게 찾을 수 있어야 한다. 패턴 매칭 정보 유닛은 모든 PMU의 매칭 결과 벡터를 읽어서 모든 매칭 비트에 대해 패킷 데이타에서 발견된 매칭 패턴의 위치 정보와 패턴 인덱스를 생성한다. PMIU의 속도는 핵심 패턴 매칭 하드웨어의 처리 속도와 일치해야 한다. 그렇지 않으면 모든 매칭 결과가 처리될 때까지 패턴 매칭 하드웨어가 정지될 수 있기 때문이다.

그림 6은 PMIU 구조의 한 예를 보여준다. PMIU는 특수한 기능을 가진 파이프라인 구조를 갖는 우선순위 트리이다. 그림은 PMIU가 매칭 경과 벡터 레지스터로부터 매칭 결과 벡터를 어떻게 입력 받는지, 35번째 비트 위치에 해당하는 매칭 비트의 위치 인덱스를 어떻게 생성하는지 보여준다. 위치 인덱스는 패턴 인덱스로 사용될 수 있다. 우선, 입력 벡터 인코더(Input Vector Encoder: IVE)는 매칭 결과 벡터로부터 각각 4비트의 입력을 받아서 매칭된 모든 비트의 해당 4비트 내에서의 상대적인 비트 위치를 인코딩한다. 인코딩된 값들은 첫째 레벨의 FIFO에 저장된다. ID '00'인 첫째 레벨 FIFO는 35번째 비트 위치에 해당하는 매칭 비트의 상대적인 위치 인덱스를 갖고 있고 그 값은 '11'이다. 다음 레벨의 우선순위 경로 선택기(Priority Path Selector: PPS)는 4개의 첫째 레벨 FIFO중에서 FIFO '00'를 선택한다. 4개의 FIFO는 각각 32~35, 36~39, 40~43, 44~47번째의 각 4비트에서 매칭 비트의 상대적인 위치 인덱스를 갖고 있다. PPS는 첫 번째 FIFO의 ID('00')와 그 FIFO로부터 얻은 상대적인 위치 인덱스('11')을 합쳐서 위치 인덱스 '0011'을 생성해낸다. 이 위치 인덱스는 둘째 레벨의 FIFO '01'에 저장된다. 여기서 '0011'이란 값은 16개의 비트(32~47번째까지) 중에서 35번째 매칭 비트의 상대적인 위치를 나타낸다. 최상위 PPS는 두 번째 FIFO의 ID('01')과 그 FIFO로부터 얻은 위치 인덱스('0011')을 합쳐서 최종 위치 인덱스 '010011'을 생성한다. 다음 절에서는 PMIU의 3가지 주요 구성요소를 살펴본다.

4.2.1 FIFO

FIFO는 매칭 비트의 위치 인덱스를 저장한다. FIFO에는 3개의 제어 비트인 tag, dummy, empty와 인덱스 비트가 존재한다. 그림 6에서는 간결성을 위해 인덱스 비트만을 표시했다. 인덱스 비트는 매칭 비트 위치의 상대적인 위치 인덱스를 저장한다. 첫째 레벨에서, 인덱스 비트의 크기는 log_2N 이고 여기서 N 은 IVE가 처리하는 비트의 수이다. 레벨이 올라갈수록, 인덱스 비트의 크기는 PPS에 연결된 하위 레벨 FIFO의 log_2 개수만큼 씩 증가된다.

empty 비트는 해당 FIFO의 엔트리가 비어있는지의 여부를 나타낸다. tag 비트는 매칭 결과의 순서를 표현하는데 사용한다. tag 비트는 매칭 결과 벡터 레지스터와 연관되는데, 새로운 결과 벡터가 레지스터에 로드될

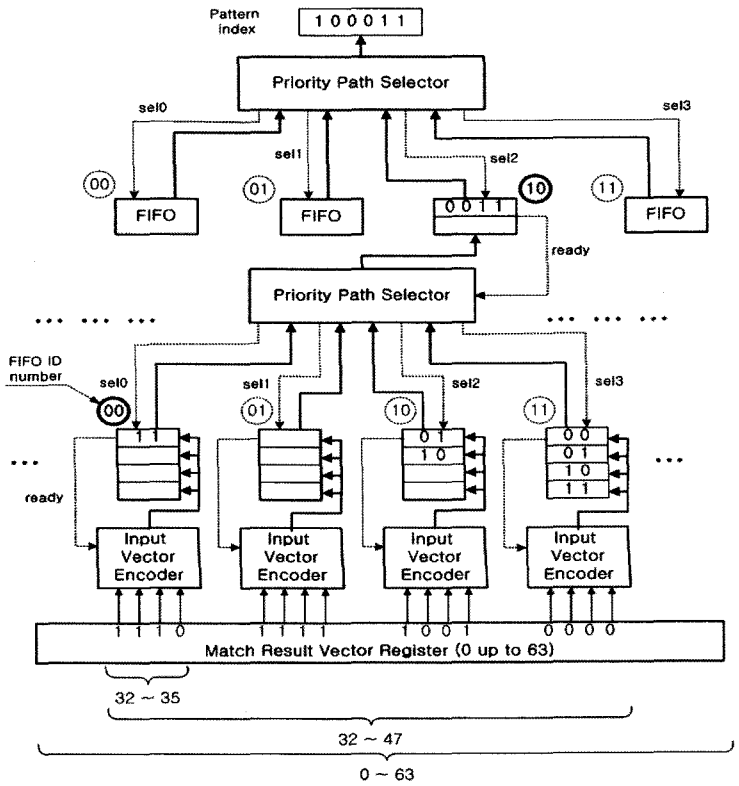

그림 6 패턴 매칭 정보 유닛 (PMIU)

때마다 0과 1값으로 번갈아가며 바뀐다. 모든 PMU는 동시에 새로운 결과 벡터를 생성한다. tag 비트 값은 매칭 결과 벡터의 인덱스 비트와 함께 올라간다. 최종 패턴 인덱스가 생성되었을 때, tag 비트의 변화 회수를 셈으로써 패턴의 위치 정보를 계산할 수 있다. PPS가 인덱스 비트를 선택할 때는, tag 비트를 사용하여 먼저 만들어진 매칭 결과 벡터에 대한 인덱스 비트가 그 뒤의 후속 매칭 결과 벡터에 대한 인덱스 비트보다 항상 먼저 상위 레벨로 올라가도록 한다. dummy 비트는 인덱스 비트의 유효 여부를 나타낸다. 매칭 비트가 하나도 존재하지 않을 경우에는 IVE는 dummy 비트가 설정된 인덱스 비트(dummy 인덱스 비트) 하나만을 첫째 레벨 FIFO에 저장한다. 이렇게 함으로써 동일한 태그 값을 갖지만 서로 다른 매칭 결과 벡터로부터 만들어진 인덱스 비트가 동시에 하나의 PPS의 입력으로 들어가는 것을 방지한다. 또한 필요 없어진 dummy 인덱스 비트는 중간에 PPS에 의해 제거되고, 매칭이 존재하지 않는 매칭 결과 벡터에 대해서는 최종적으로 하나의 dummy 인덱스 비트만 남게 된다. 자세한 PPS의 수행 과정은 다음 절에서 기술된다.

각 레벨의 FIFO는 파이프라인을 형성하는데, IVE에 연결된 첫째 레벨의 FIFO는 다른 레벨의 FIFO와 조금 다르다. IVE의 입력 비트 수가 첫째 레벨 FIFO의 깊이를 결정한다. 매칭 비트의 모든 상대적인 위치 인덱스는 한 클럭 주기(clock cycle) 안에 생성되어 FIFO에 저장된다. 나머지 레벨에서의 FIFO의 깊이는 최소 2이다. 2개의 엔트리는 파이프라인에서 버블이 발생하는 것을 방지하기 위해 필요하다. 2개의 엔트리를 갖는 FIFO에서는, 적어도 하나의 빈 엔트리가 생기자마자 새로운 상대적인 위치 인덱스가 연결된 하위 레벨의 PPS에 의해 생성되어 FIFO에 저장된다. 이것은 파이프라인에서 버블이 발생하는 것을 방지한다.

4.2.2 우선순위 경로 선택기(Priority Path Selector)

PPS는 연결된 입력 FIFO중에서 가장 우선순위가 높은 인덱스 비트를 선택하여 선택된 FIFO의 ID 번호와 입력 인덱스 비트를 합하여 새로운 상대적인 위치 인덱스를 생성한다. 선택된 FIFO 엔트리는 삭제되고, 새로 생성된 상대적인 위치 인덱스는 PPS에 연결된 출력 FIFO에 저장된다. 우선순위는 입력 FIFO의 우선순위뿐만 아니라 tag와 dummy 비트 값도 고려하여 결정한다. PPS의 동작은 한 주기에 수행된다.

PPS의 우선순위 모드는 1과 0모드가 있다. 주어진 우

선순위 모드에서, 모드와 동일한 tag 값을 갖는 입력 FIFO 엔트리만 먼저 선택되어 연산에 고려된다. 모든 입력 FIFO의 첫 번째 엔트리가 동일한 tag 값을 가질 때만, PPS의 모드는 해당 tag 값으로 바뀐다. 이는 모든 상대적인 위치 인덱스를 매칭 결과 생성과 동일한 순서로 계산하기 위함이다. dummy 인덱스가 아닌 엔트리가 적어도 하나 이상 존재하면, PPS는 그 중에서 하나의 엔트리를 선택하고 관련된 모든 입력 FIFO에 선택 신호를 보냄으로써 모든 dummy 인덱스 엔트리를 삭제한다. 모든 엔트리가 dummy 인덱스 엔트리이면 하나의 엔트리만 상위 레벨로 전달되고 다른 모든 dummy 인덱스는 삭제된다. 이렇게 함으로써, 최상위 레벨 PPS에 도달했을 때에는 단지 하나의 dummy 인덱스만 남게 된다. dummy 인덱스는 매칭이 존재하지 않은 매칭 결과 벡터를 위한 것이다.

4.2.3 입력 벡터 인코더(Input Vector Encoder)

입력 벡터 인코더(IVE)는 균등하게 나누어진 패턴 매칭 결과 벡터의 일부분을 입력 받아서 그 입력에 있는 모든 매칭 비트의 상대적인 위치 인덱스를 생성한다. 매칭 비트가 존재하지 않으면 단지 하나의 dummy 인덱스를 생성한다. 생성된 모든 상대적인 위치 인덱스는 한 주기 내에 첫째 레벨 FIFO에 저장되고 IVE의 모드는 매칭 결과 벡터의 태그 값과 동일한 값으로 설정한다. IVE의 모드가 매칭 결과 벡터와 동일한 한은 더 이상의 연산이 수행되지 않는다. 매칭 결과 벡터의 태그가 바뀌면, 즉 새로운 입력 패킷 데이타 바이트에 대해서 새로운 매칭 결과 벡터가 생성되고 벡터 레지스터에 로드되면, IVE는 상대적인 위치 인덱스를 다시 생성한다.

5. 평가

본 논문에서는 최신 Xilinx FPGA Virtex-4[24]을 사용하여 PMU를 설계하였다. Virtex-4는 Block RAM이라 불리는 가장 많은 수의 내장형 RAM을 갖고 있다. 본 설계에서는 Block RAM을 이용하여 멀티포트를 지원하는 SRAM을 구성하였다. Block RAM은 최대 듀얼 포트 메모리 설정만을 지원해서, 포트수가 2이상일 때에는 멀티 포트 SRAM을 시뮬레이션하기 위해 메모리 뱅크를 복제해야만 했다. 이 실험에서 PMU의 문자 위치 벡터의 크기를 256비트로 설정하였고, 따라서 하나의 PMU가 처리할 수 있는 전체 패턴의 크기는 256 바이트이다. Snort의 평균 패턴 크기는 대략 12바이트이기 때문에, 하나의 PMU는 평균적으로 약 20개 이상의 패턴을 처리할 수 있다.

그림 7은 Shift-OR 단계 수가 늘어남에 따라 변화되는 PMU의 처리 속도를 보여준다. X 축의 Shift-OR 단계의 개수는 $\log_2$의 스케일로 표시되었다. PMU는

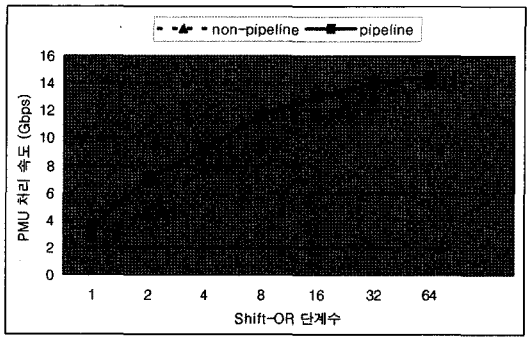

그림 7 PMU 처리 속도 vs. Shift-OR 단계 수

Shift-OR 단계의 수와 동일한 수의 입력 문자들을 한 번에 처리할 수 있다. 본 실험에서는 두 가지 버전의 설계를 사용하였는데, *'pipeline'*은 메모리 뱅크와 Shift-OR 연산 회로 사이에 파이프라인을 삽입 것이고, *'non-pipeline'*은 파이프라인을 삽입하지 않은 것이다. 그래프를 보면 Shift-OR 단계가 64개일 때, 두 버전의 PMU 처리 속도가 각각 14Gpbs, 14.5Gbps임을 알 수 있다.

pipeline 설계는 non-pipeline 설계에 비해 최대 58%의 성능 향상을 보여준다. 성능 차이는 Shift-OR 단계가 적을수록 더 두드러진다. non-pipeline의 경우 시스템 클럭(clock)의 속도는 메모리 접근 시간과 전체 Shift-OR 연산 시간에 의해 결정되는데 비하여, pipeline 경우는 메모리 접근 시간과 전체 Shift-OR 연산 시간 중에서 큰 쪽에 의해 결정 된다. 따라서 Shift-OR 단계의 수가 적을 때에는 메모리 접근 시간의 비중이 크기 때문에 pipeline 설계에서처럼 메모리 접근 시간과 Shift-OR 연산 시간을 겹치는 것이 중요하다. Shift-OR 단계의 수가 증가됨에 따라, 시스템 클럭 중 메모리 접근 시간의 비중이 줄어들어, 전체 Shift-OR 연산 시간이 주요 성능 지수가 된다. PMU의 속도는 기존 Shift-OR 패턴 매칭 가속기[8]와 근본적으로 같다. 추가로 첨가된 매칭 위치 벡터 레지스터의 접근에 걸리는 시간은 메모리 접근 시간이나 Shift-OR 연산 시간에 거리는 시간 보다 작기 때문에 시스템 클럭의 속도에 영향을 안 미친다.

그림 8은 RAM 뱅크, LUT, non-pipeline 설계에서의 플립플롭(flip-flop), pipeline 설계에서의 플립플롭과 같이 여러 하드웨어 자원에 대한 자원 사용량을 보여준다. 그림에서는 각각 RAM, LUT, FF(NP), FF(P)라고 이름 지었다. 자원 사용량은 한 패턴 문자를 처리하는데 필요한 자원의 개수를 나타낸다. RAM 뱅크의 개수가 증가하는 것은 멀티포트 메모리의 시뮬레이션에 기인한 것이다. Shift-OR 단계의 수가 증가함에 따라, 메모리

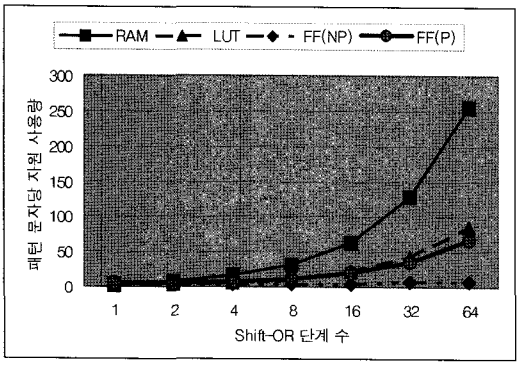

그림 8 패턴 문자당 자원 사용량 vs Shift-OR 단계 수

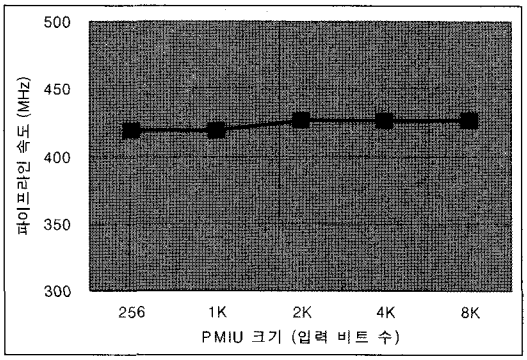

그림 9 PMIU 파이프라인 속도 vs PMIU 크기

읽기 포트가 더 필요하고, 늘어나는 메모리 포트를 지원하기 위해 메모리 뱅크가 더 복제 되었다. Shift-OR 단계가 증가 할수록 LUT의 사용도 증가 하였다. 이유는 Shift-OR를 구성하는데 LUT가 사용되기 때문이다. pipeline 설계에서는 플립플롭의 개수도 증가한다. 메모리 파이프라인을 위한 플립플롭의 개수는 Shift-OR 단계의 추가로 인한 메모리 포트 증가 때문에 증가한다. 반면에 non-pipeline 설계에서는 플립플롭의 개수가 항상 일정하다. 따라서 메모리가 충분히 빠르다면 많은 Shift-OR 단계가 필요한 경우에는 non-pipeline 설계가 좋은 선택이 될 수 있다. 기존의 Shift-OR 패턴 매칭 가속기와 비교하였을 때 추가로 사용된 매칭 위치 벡터 레지스터 때문에 플립플롭이 문자 당 하나씩 더 추가된다. 하지만 이것은 큰 문제가 안 된다. 전체 시스템이 구성되었을 때 플립플롭의 자원 사용률은 다른 자원에 비하면 나중에 나올 표 1에서 보듯이 낮기 때문이다.

전반적으로, Shift-OR 패턴 매칭 구조에 필요한 하드웨어 자원의 양과 처리 속도 사이에는 trade-off가 존재한다. 자원 효율성이 높은 non-pipeline 설계에서조차도 Shift-OR 단계를 늘릴수록 더 많은 읽기 포트, 메모리 뱅크, LUT가 필요하다. 그러나 하드웨어 자원이 넉넉하다면, Shift-OR 단계를 추가함으로써 동시에 처리되는 문자의 수를 늘려 PMU의 처리 속도를 향상시키는 것이 상대적으로 용이하다. 이것이 이전의 다른 패턴 매칭 하드웨어에 비해서 Shift-OR 패턴 매칭 구조가 가지는 중요한 장점이다.

본 논문에서는 256에서 8K 비트 입력을 갖는 PMIU를 구현하였다. 그림 9는 입력 크기 증가에 따른 PMIU의 파이프라인 속도를 보여준다. 테스트한 모든 PMIU의 크기에 대하여 파이프라인 속도는 420MHz 정도로 거의 일정하다. PMIU가 파이프라인 구조이기 때문에 그 크기가 증가하여도 시스템 클럭의 속도를 일정하게 유지하는 것을 보여 주고 있다. PMU의 최대 시스템 클럭의 속도가 1, 2단계 Shift-OR로 구성된 pipeline PMU에서 약 447MHz에 이르고 그림 7에서 보인 다른 모든 non-pipeline PMU와 pipeline PMU의 최대 시스템 클럭 속도가 356MHz 이다. 이는 Shift-OR의 단계 수가 늘어날수록 시스템 클럭의 속도가 떨어지기 때문이다. 따라서 PMIU는 10Gbps이상의 처리 속도를 보여주는 PMU들의 클럭 속도를 만족 시킨다. 그림 10은 PMIU의 다양한 크기에 따른 여러 하드웨어 자원(슬라이스, LUT, 플립플롭)의 자원 사용량을 보여준다. 자원 사용량은 PMIU의 입력 한 비트당 사용되는 자원량을 나타낸다. PMIU를 구성하는데 사용되는 하드웨어 자원은 입력 비트 당 일정하고, 결과적으로 전체 크기가 증가할수록 선형적으로 증가한다. 패턴의 수가 두배로 증가하여도 PMIU는 트리 형태의 파이프라인 구조이기 때문에 단지 우선순위 경로 선택기가 더 추가 될 뿐이기 때문에 입력 비트 당 자원량을 거의 일정하게 유지 할 수 있다.

마지막으로, 모든 시스템 구성요소를 하나의 Xilinx Virtex-4 칩에 구현하여 자원 활용도를 측정해보았다. 표 1은 PGU, PMU, PMIU의 자원 활용도와 Virtex-4 칩에서 사용된 자원의 백분율을 보여준다. 설계된 구조는 512×32 비트 크기의 PGU, 4단계 Shift-OR를 가진

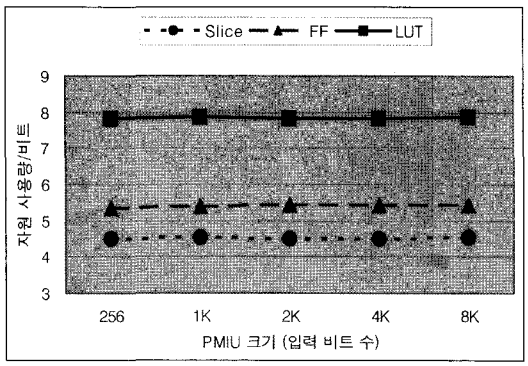

그림 10 PMIU 한 비트당 자원 사용량 vs PMIU 크기

표 1 PGU, PMU, PMIU의 자원 활용도

	PGU	PMU	PMIU	Utilization Percentage
슬라이스	18	25807	37040	94 %
플립플롭	32	24575	44366	54 %
4입력 LUT	0	49156	64263	88 %
Block RAM	1	512	0	92 %

32개의 PMU, 8K 비트 PMIU이다. 이 구조는 8K 바이트의 패턴 길이를 지원한다. 표에서 보듯이, 슬라이스가 가장 제한적인 자원이다. 슬라이스는 2개의 LUT와 2개의 플립플롭으로 구성되는데, 사용되지 않고 남은 플립플롭과 LUT가 많았다. 따라서 주문형 설계를 한다면 자원 활용도의 균형을 향상시킬 수 있을 것이다.

5. 결 론

네트워크 침입 방지 시스템(NIPS)은 최근에 DoS, 이메일 바이러스, 인터넷 웜과 같은 네트워크 공격에 대응하는 가장 유망한 기술로 부상하였다. NIPS는 패킷 헤더와 데이타를 회선속도로 조사하여 악의적인 패킷을 네트워크로부터 차단한다. 패턴 매칭은 패킷의 데이타에서 공격 패턴을 발견하는데 사용되고 NIPS의 수행 시간의 상당부분을 차지한다. 본 논문에서는, NIPS에 요구되는 중요한 4가지 사항(회선 속도로 패턴 매칭 정보 제공, 패턴 그룹 지원, 최저 성능 보장, 수행 중단 없는 빠른 업데이트)을 만족하는 패턴 매칭 시스템 구조를 제시하였다. 이 요구사항들은 NIPS에서 회선 속도로 매칭되는 공격 패턴을 찾는 만큼 중요하다.

본 논문에서는 제시한 패턴 매칭 시스템 구조를 Xilinx FPGA 툴을 사용하여 평가하였고 이 시스템은 10Gbps 이상의 빠른 속도를 성취할 수 있음을 보였다. 개선된 Shift-OR 패턴 매칭 가속기를 구현한 PMU는 Shift-OR 단계를 추가함으로써 손쉽게 목표한 패턴 매칭 처리 속도에 도달 할 수 있었다. PMIU 속도는 대부분의 PMU 수행 속도와 일치했고 420MHz 속도로 패턴 인덱스와 위치 정보를 생성할 수 있었다.

참 고 문 헌

[1] Code Red worm exploiting buffer overflow in IIS indexing service DLL. CERT Advisory CA-2001-19, Jan 2002.

[2] MS-SQL Server Worm. CERT Advisory CA-2003-04, Jan 2003.

[3] Xinyou Zhang, et al. Intrusion Prevention System Design. The Fourth International Conference on Computer and Information Technology. Sept 2004.

[4] Snort, url http://www.snort.org/

[5] S. Antonatos, K. G. Anagnostakis, and E. P. Markatos. Generating realistic workloads for network intrusion detection systems. ACM Workshop on Software and Performance, 2004.

[6] N. Tuck, T. Sherwood, B. Calder, G. Varghese: Deterministic Memory-Efficient StringMatching Algorithms for Intrusion Detection. IEEE INFOCOM 2004.

[7] Rong-Tai Liu, et al. A Fast String Matching Algorithm for Network Processor Based IntrusionDetection System. ACM Transaction on Embedded Computing Systems, Vol. 3, No. 3, August 2004.

[8] Sunil Kim. Pattern Matching Acceleration for Network Intrusion Detection Systems. SAMOS V. July 2005.

[9] Lin Tan, Timothy Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. ISCA June 2005.

[10] D. Moore, et al. Internet Quarantine: Requirements for Containing Self-Propagating Code. Proceedings of the IEEE INFOCOM Conference, April 2003.

[11] S. Dharmapurikar, et al. Deep Packet Inspection Using Parallel Bloom Filters. Symposium on High Performance Interconnects, Aug. 2003.

[12] B. L. Hutchings, and R. Franklin, D. Carver. Assisting Network Intrusion Detection with Reconfigurable Hardware. Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, September 2002.

[13] Reetinder Sidhu, and Viktor K. Prasanna. Fast Regular Expression Matching using FPGAs. Proceedings of The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, May 2001.

[14] James Moscola, et al. Implementation of a Content-Scanning Module for an Internet Firewall. Proceedings ofthe 11th Annual IEEE Symposiun on Field-Programmable Custom Computing Machines April, 2003.

[15] Maya Gokhale, et al. Granidt: Towards Gigabit Rate Network Intrusion Detection Technology. Proceedings of the Field-Programmable Logic and Applications, 12th International Conference, September 2002.

[16] Young H. Cho, et al. Specialized Hardware for Deep Network Packet Filtering. Proceedings of the International Conference on Field Programmable Logic and Applications, September 2002.

[17] Ioannis Sourdis, and Dionisios Pnevmatikatos. Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System. Proceedings of the 13th International Conference on Field Programmable Logic and Applications, September 2003.

[18] Ioannis Sourdis, and Dionisios Pnevmatikatos. Pre-decoded CAMs for Efficient and High-Speed

NIDS Pattern Matching.Proceedings of the Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines April 2004.
[19] Young H. Cho, W. H. Mangione-Smith. Programmable Hardware for Deep Packet Filtering on a Large Signature Set. Workshop on Architectural Support for Security and Anti-Virus. 2004.
[20] Young H. Cho, W. H. Mangione-Smith. Deep Packet Filter with Dedicated Logic and Read Only Memories. Proceedings of the 12th IEEE Symposium of Field-Programmable Custom Computing Machines, 2004.
[21] A. V. Aho, M. J. Corasick: E_cient String Matching: An Aid to Bibliographic Search. Communications of the ACM 18 (1975).
[22] Ricardo A. Baeza-Yates, and Gaston H. Gonnet. A New Approach to Text Searching. In the Proceedings of the Communications of the ACM, October 1992.
[23] David E. Taylor. Survey and Taxonomy of Packet Classification Techiniques. ACM Computing Survey, Vol 37, 2005.
[24] Xilinx, Inc. url http://www.xilinx.com/

김 대 영
1998년 홍익대학교 컴퓨터공학과(학사)
2001년 홍익대학교 전자계산학과(석사)
2001년~현재 홍익대학교 컴퓨터공학과 박사과정

김 선 일
1985년 서울대학교 컴퓨터공학과(학사)
1987년 서울대학교 컴퓨터공학과(석사)
1995년 University of Illinois at Urbana-Champaign(전산학 박사). 1995년~1999년. IBM, USA(연구원). 1999년~현재 홍익대학교 정보컴퓨터공학부 교수

이 준 용
1986년 서울대학교 공과대학 컴퓨터공학과(학사). 1988년 미국 미네소타 주립대(석사). 1996년 미국 미네소타 주립대(박사). 1996년~1997년 미국 IBM 연구원. 1997년~현재 홍익대학교 컴퓨터공학과 교수