검색을 지원하는 하드웨어 기반 XML 파서의 구현

김세운, 이규희*, 윤상균
연세대학교 컴퓨터정보통신공학부
e-mail: kimsaewoon@gmail.com, {powerpc*, skyun}@yonsei.ac.kr

Implementation of a Hardware-based XML Parser Supporting Search

Sae-Woon Kim, Kyu-Hee Lee*, Sang-Kyun Yun
Dept of Computer and Telecom. Engineering, Yonsei University

요 약

Extensible Markup Language(XML)는 데이터 저장, 웹 서비스, 팟캐스팅, 신디케이션과 같은 분야에서 널리 사용되는 언어이다. XML 문서를 사용하기 위해서는 반드시 XML 문서에 대한 파싱이 이루어져야 하기 때문에 여러 연구들이 제안되었다. 스트림 기반 파서인 Roll-Back Streaming XML(RBStreX) 파서는 파싱의 흐름을 되돌리는 Roll-Back을 제안하였지만, 특정 노드를 검색할 때 발생되는 오버헤드를 해결하지 못하였다. 본 논문에서는 특정 노드에 대한 검색 기능을 지원하는 새로운 구조의 하드웨어 스트리밍 파서를 제안하였고, 시뮬레이션을 통하여 하드웨어의 동작 검증을 하였다. 제안된 구조는 RBStreX 파서와 비교하여 소프트웨어 오버헤드가 없으며 하드웨어 사이클도 반 정도로 감소시키는 성능향상을 얻는다.

1. 서론

XML(eXtensible Markup Language)은 데이터나 정보 자체를 기술하기 위한 표준으로써, 잘 정의된 형태의 텍스트 기반 기술 언어이다. XML 문서들은 작성이 간단하며 모든 컴퓨터 플랫폼에서 지원되기 때문에, SOA (Service-Oriented Architecture)나 분산 컴퓨팅 환경에서 데이터의 전송과 저장을 위한 방법으로 널리 사용되고 있다.

XML 문서는 데이터와 양식 마크업(form markup)과 같은 일련의 문자들로 구성된다. XML에 저장된 데이터를 사용하는 응용프로그램들은 XML 문서를 읽고 구조와 내용에 접근하여 데이터를 내부적으로 사용 가능한 형태의 정보로 저장하는 파싱(parsing) 작업을 수행한다. 파싱 작업은 XML 문서의 처리 과정 중 가장 많은 시간이 소요되는 작업이기 때문에, 효율적인 XML 파서(parser)의 개발 및 사용이 요구된다.

현재 가장 널리 사용되는 파서들은 소프트웨어 기반 파서들 DOM(Document Object Model), 자바 XML API (SAX: Simple API for XML)과 StAX(Streaming API for XML) 등이 있으며, 하드웨어 파서는 몇몇 연구자들에 의해 연구되고 있다[1][2][3].

본 논문의 구성은 다음과 같다. 2장에서 XML 파서에 대한 관련연구를 기술하고, 3장에서 제안하는 구조에서 사용되는 XML 문서와 검증을 위한 시나리오를 명시한다. 4장에서는 제안된 하드웨어 기반 스트리밍 XML 파서의 구조를 설명한다. 5장에서는 제안된 구조를 평가하고, 6장에서 결론을 맺는다.

2. 관련연구

XML 파서는 XML 문서의 구조적 문법을 확인하여 파싱 결과를 메모리에 저장한다. XML 파서는 파싱 결과 전체를 트리 형태로 저장하는 트리 기반 파서와 새로운 개체가 나타날 때 마다 이벤트를 발생시키는 스트리밍 기반 파서로 나눌 수 있다. 트리 기반 파서는 문서를 읽고 모든 결과를 트리로 DOM(Document Object Model)로 저장한다. DOM을 사용하는 XML 파서는 모든 노드에 직접 접근할 수 있는 장점을 갖지만, 파싱된 모든 결과를 저장하기 때문에 메모리 부족 현상의 발생할 수 있어 대용량 XML 문서 처리에 적합하지 않다[4]. 스트리밍 기반 파서는 트리 기반 파서에 비해 메모리 효율적인 구조를 갖지만, 새로운 개체가 파싱될 때 마다 이벤트를 처리하기 때문에 데이터 처리를 위한 추가 시간이 발생한다.

RAX(Random Access XML)[2]는 Tarai에 의해 개발된 파싱 기술로써, 많은 기업에서 XML 파싱을 위한 하드웨어 가속기로 사용되고 있다. RAX는 XML 문서 파싱에서 발생되는 각 이벤트를 XPath로 생성하여 특정 데이터를 검색하기 위해 사용한다. XPath는 XML 문서를 검색할 수 있는 경로 표현 방법이다. RAX의 성능은 임의의 데이터를 한 번에 검색하여 접근하기 때문에 높은 성능을 내지만, XPath 연산을 위해 많은 메모리 자원이 요구되며, 구현 복잡도가 높다.

Virtual Token Descriptor for XML (VTD-XML)[3] 은 XimpleWare이 개발한 소프트웨어와 하드웨어에서 구현 가능한 파싱 기술이다. VTD-XML은 문서를 파싱하고 각 이

```
<parent>
  <name>Old John</name>
  <age>50</age>
  <child>
    <name>Tommy</name>
    <age>17</age>
  </child>
  <child>
    <name>Mary</name>
    <age>15</age>
  </child>
</parent>
<parent>
  <name>Old Tom</name>
  ....
....
```

그림 1. 시나리오를 위한 XML 문서

벤트 별로 64 비트 바이너리 형태의 VTD 레코드(토큰)를 생성한다. 이 구조는 VTD 레코드에 임의 접근이 가능하도록 구현되어 속도 향상을 시켰지만, VTD 레코드는 DOM과 달리 객체지향을 지원할 수 없다.

Chee[1] 등은 RBStreX(Roll-Back Streaming XML Parser)는 스트리밍 기반 파서로써, XML 데이터를 추출하는 기능만 지원한다. Chee 등은 파싱의 흐름을 되돌릴 수 없는 스트리밍 기반 파서의 단점을 roll-back 기능으로 해결하였다. roll-back은 현재 처리된 element가 조건에 만족하는 경우 현재의 상위 element로 직접 이동하는 기능이다. 이 구조는 적은 양의 메모리 자원을 사용하기 때문에 임베디드 시스템에 적합한 형태이다. 그러나, RBStreX는 수행하는 XML 문서의 형태에 따라 가변적인 파싱 시간을 갖고, 발생되는 모든 이벤트와 파싱 명령어를 응용프로그램에 송/수신하는데 오버헤드가 발생한다.

본 논문에서는 이전에 연구된 스트리밍 파서들의 송/수신 오버헤드를 줄이고 roll-back 기능을 갖춘 하드웨어 기반 스트리밍 파서 구조를 구현한다. 모든 이벤트들과 명령어들에서 발생되는 송/수신 시간을 줄이기 위해, 본 논문에서는 스트리밍 XML 파서에 검색 기능을 추가한 새로운 구조를 제안한다.

3. XML 파서 구현을 위한 시나리오

제안하는 구조의 XML 파싱 절차들을 기술하고 동작 검증 및 성능 비교를 위해, 그림 1의 XML 문서를 이용한다. RBStreX 파서는[1] 그림 1의 XML 문서를 파싱하기 위해 get_next 명령어를 사용하여 <parent>나 </name>과 같은 element와 “Old John"과 같은 content 단위로 순차 처리한다.

제안된 구조에 적용하는 시나리오는 다음과 같다.

1. 이름이 "Mary"인 데이터 찾기
2. "Mary" 부모의 모든 자식들에 대한 나이를 제공

일반적인 스트리밍 파서에서 roll-back이 기능이 없다면 조건 1이 만족할 때 까지 get_next 명령이 수행되고, 조건 2를 수행하기 위해 라인 1부터 문서를 다시 파싱하여 결과를 도출해야 한다. 만일 그림 1의 문서와 달리, "Mary" 가족 이전에 다른 가족에 대한 정보가 여러 개 존재한다면 비효율적인 작업이 수행된다. RBStreX의 roll-back 기능은 조건 2를 수행하기 위해 직접 "Mary"의 parent에 접근할 수 있다. 이 구조는 roll-back 이후parent element부터 get_next를 수행하기 때문에 다른 파서들에 비해 오버헤드가 적다.

4. 검색을 지원하는 XML 파서 구조

XML 문서를 파싱하기 위해 사용되는 get_next 명령은 수행 결과를 항상 호출한 곳으로 전달한다. 앞서 언급한 바와 같이, 이는 XML 문서 파싱에서 불필요한 오버헤드를 야기한다. 본 장에서는 get_next, roll_back 명령어들에서 발생되는 오버헤드를 해결하기 위한 검색을 지원하는 하드웨어 기반 스트리밍 XML 파서의 구조를 제시한다. 본 논문에서 제안하는 XML 파서는 FPGA(Field Programmable Gate Arrays)를 사용하여 하드웨어 가속기로 구현하며, 이에 대한 기본 구조는 그림 2와 같다.

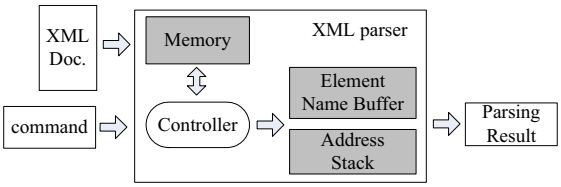


그림 2. 제안된 하드웨어 XML 파서의 기본 구조

XML 문서는 파일이거나 실시간으로 입력되는 스트림일 수 있는데, 이 데이터는 먼저 메모리에 저장되며 XML 파서는 주소를 사용하여 데이터를 바이트 단위로 읽는다. 응용 프로그램에서 XML 파서에 명령어를 공급하면, 메모리에 저장된 XML 문서 데이터를 순차적으로 읽으면서, 지시한 명령어에 대한 동작을 수행한다. 본 XML 파서는 get_next, roll_back과 search의 3가지 명령을 다음과 같이 수행한다.

- **get_next** : get_next 명령어를 받으면, 컨트롤러는 XML 파싱 상태머신에 의해서 구문을 해석하여 start element, end element, 또는 content에 속하는 토큰을 찾고, (eventType, eventData) 형식의 결과를 반환한다. eventType은 찾은 토큰의 종류를 나타내며, eventData는 element 또는 content 내용이다. 파싱동안 element 또는 content 내용은 Element Name Buffer(ENB)에 저장되어 결과를 반환할 때에 사용된다. get_next를 수행하는 동안 roll_back에 대비하여 부모 element가 저장된 메모리 주소를 Address Stack에 저장한다.
- **roll_back** : roll_back 명령어는 XML 메모리 주소를 현재의 element의 부모 element가 저장된 위치로 되돌리는 동작을 수행하고 roll_back 성공/실패 여부를 출력으로 제

시한다. roll_back 성공시, Address Stack에서 roll_back을 위한 주소를 pop하여 XML 메모리 주소를 다시 설정하여 해당 위치에서 파싱을 재시작할 수 있게 한다.

• **search** : 일반적인 XML 스트리밍 파서에서 발생되는 오버헤드를 제거하기 위해 제안하는 search 명령어는 검색 조건인 (element, content)의 쌍과 함께 입력되며 그림 3과 같은 검색용 키워드 레지스터에 저장된다. 컨트롤러는 XML 파싱동안 발견되는 element나 content 토큰을 주어진 검색 값과 비교하여 두 값이 모두 일치할 때에 성공을 반환하고, 찾지 못하면 실패를 반환한다. 이 명령어를 수행하는 동안에도 roll_back을 위하여 Address_Stack에 부모 element가 저장된 메모리 주소를 저장한다.

검색 조건인 (element, content) 쌍은 검색용 키워드 레지스터에 그림 3과 같이 "name\0Mary\0"같은 형식으로 저장된다. '\0'은 널문자를 의미하며 element와 content 문자열의 끝을 표시하는 구분자로 사용된다. 두 요소는 순차적으로 나타나므로 search_addr를 계속해서 증가시키면서

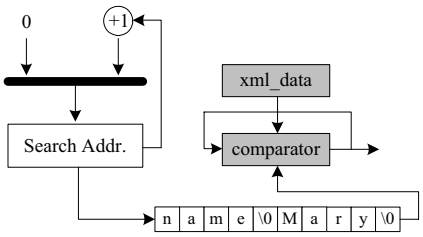


그림 3. 제안된 하드웨어 search 구조

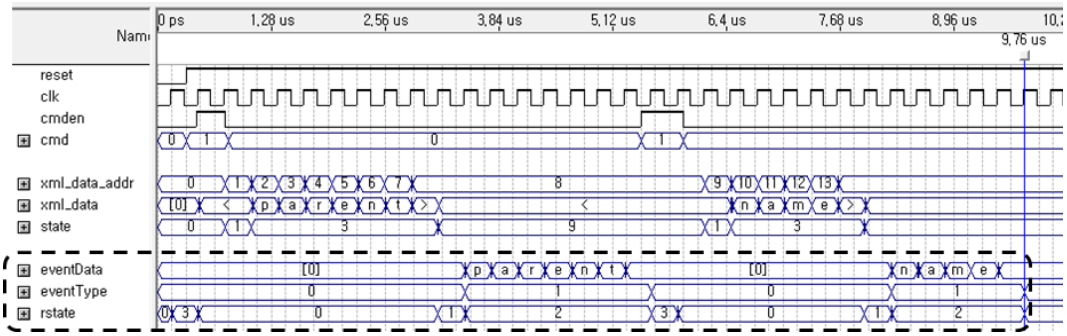
element와 content를 연속하여 검색할 수 있다.

.

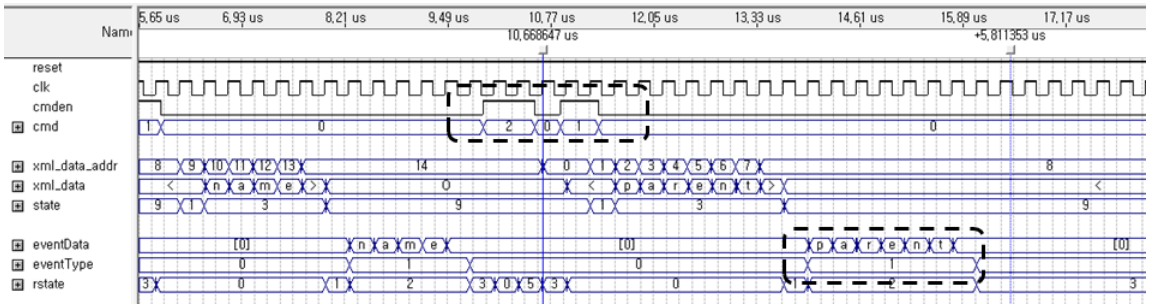
5. 평가분석

5-1. 동작 검증

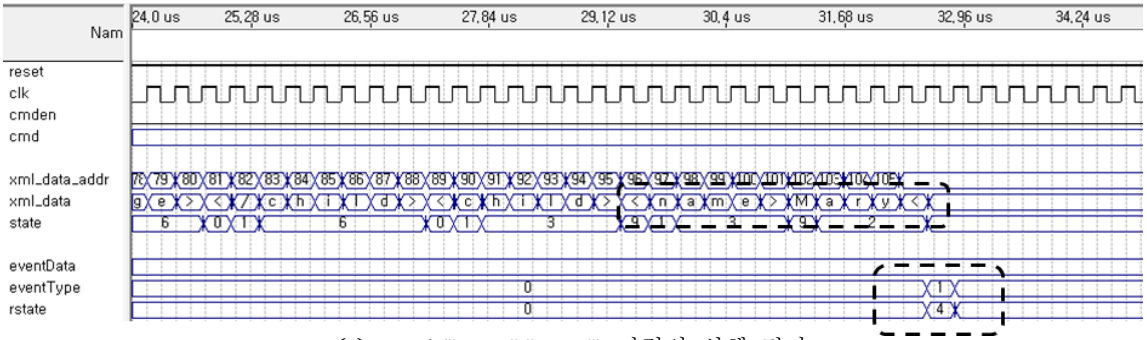
본 논문에서 제안된 하드웨어 구조는 Verilog HDL로 설계하여 Altera의 QuartusII 9.1 합성 도구를 사용하여 FPGA로 합성하였고 Quartus II의 시뮬레이터로 동작을 검증하였다. 그림 4는 동작 검증을 위한 시뮬레이션 파형도이다. 제안된 구조의 동작 검증을 위해 그림 1의 XML 문서를



(a) get_next 명령어 실행 결과



(b) roll_back 후 get_next 명령의 실행 결과



(c) search("name","Mary") 명령의 실행 결과

그림 4. 제안한 구조의 동작 검증 결과

적용하였다.

그림 4(a)는 처음에 get_next 명령(cmd=1)을 2번 연속하여 실행하였을 때의 파형이다. {start_element(1), "parent"}와 {start_element(1), "name"}의 결과가 연속하여 제공되어 get_next 명령이 정상적으로 동작함을 확인할 수 있다.

그림 4(b)는 </name>을 읽은 다음에 roll_back 명령(cmd=2)을 수행한 후 get_next 명령(cmd=1)를 수행한 결과이다. 그림 4(b)를 살펴보면, roll_back을 수행하기 전에는 메모리 주소가 <name>의 content인 “Old John"의 시작위치(14번지)를 가리켜서 데이터가 ‘O’가 출력되었는데, roll_back을 수행한 후에 부모 element인 <parent>가 저장된 시작위치(0번지)로 변경되어 roll_back이 정상적으로 동작함을 확인할 수 있다. 그리고 get_next을 뒤이어 실행했을 때에 {start_element(1), "parent"}의 결과가 제공됨을 확인할 수 있다.

그림 4(c)는 검색 값으로 {“name","Mary"}를 사용하여 search 명령(cmd=3)을 실행시킨 결과이다. 이 그림에는 검색 성공 결과 파형만 나타내있으며, 명령어를 제공하는 부분은 훨씬 앞에 있어서 포함되지 않았다. 그림 4(c)를 살펴보면 "...<name>Mary<"의 '<'가 입력될 때에 content가 Mary임이 확인되어 검색 성공(1)의 결과가 제공되어 search 명령이 정상적으로 동작함을 확인할 수 있다.

5-2. 성능 분석

구현된 XML 파서의 각 명령어의 수행시간은 표 1과 같다. get_next 명령은 토큰 문자열의 길이가 n이라고 할 때에 명령어 제공에 1 cycle, XML 메모리에서 데이터를 읽는 시간은 n+a cycle (a는 start_element는 <와 >를 포함하여 2, end_element는 </와 >를 포함하여 3, content는 0), 결과를 받는 데 n+1 사이클 (문자열 끝에 null 추가), 토큰을 찾은 후 결과출력 준비에 1 cycle이 소요되어 적어도 총 2n+a+3 cycle이 소요된다. 그림 4(a)의 앞부분에서 start_element인 “parent"를 결과를 얻는 데 17 cycle이 소요됨을 확인할 수 있다.

roll_back 명령은 명령어 제공에 1 cycle, roll_back 수행에 1 cycle이 소요되어 총 2 cycle이 소요된다. 그림 4(b)에서 이를 확인할 수 있다.

search 명령은 검색 키워드의 길이를 m이라고 할 때에 명령어 제공에 m cycle이, 검색 결과를 받는 데 1 cycle이 소요된다. 그리고 검색이 성공한 다음의 ‘<’까지의 문자열의 길이를 N이라고 하면 XML 메모리에서 데이터를 읽는 시간이 N cycle 소요되므로 총 m+N+1 cycle이 소요된다. 그림 4(c)에서 검색이 성공할 때까지 XML 메모리 데이터가 연속적으로 읽히는 것을 확인할 수 있다.

본 논문에서 제시한 search 기능이 없는 RBStreX 구조에서는 get_next를 연속하여 실행하여 얻은 결과를 소프트웨어로 판단하여 (element, content)쌍의 검색을 수행해야 한다. 검색이 성공할 때까지 수행한 get_next 명령어 횟수를 k라고 하고, 소프트웨어에서 get_next 결과를 판단하는 데 소요되는 시간을 약 S라고 하면, 전체 소요 시간은 다음과 같이 나타낼 수 있다.

$$\sum_{i=1}^{k}(2n_i + a_i + 3 + S) > 2\sum_{i=1}^{k}(n_i + a_i) + kS \cong 2N + kS$$

여기서 $\sum(n_i + a_i)$는 검색이 성공할 때까지 입력받은 문자열 길이 N이며, a_i는 0, 2 또는 3이므로 위의 식이 성립한다.

제안한 구조와 RBStreX의 결과를 비교하면 검색이 성공할 때까지의 문자열이 N일 때에 제안한 구조는 m+N+1 cycle이 소요되는 데 비해서, RBStreX는 2N+kS cycle이 소요되어 m<<N이라고 가정할 때에 약 2배의 하드웨어 사이클이 소요되며 kS 사이클의 소프트웨어 실행 오버헤드가 추가된다. 따라서 제안한 구조가 RBStreX에 비해서 검색 동작을 수행해야 할 때에 훨씬 효율적으로 수행할 수 있다.

표 1. 제안된 구조의 명령어 수행시간

수행 명령	소요시간	비고
get_next	2n+a+3	n: 토큰 문자열 길이 a: 2(start_elem),3(end_elem) 0(content),
roll_back	2	
search	m+N+1	m: 검색 키워드 길이 N: 검색성공 다음까지의 입력 문자열 길이

6. 결론

본 논문에서는 하드웨어 XML 파서 구현에서 XML 문서 처리 시 빈번히 발생하는 특정 노드 검색 수행 시에 RBSteX에서와 같이 get_next 명령을 반복 사용하는 것으로 인한 비효율적인 구조를 개선하기 위해 특정 노드를 검색할 수 있는 새로운 search 명령을 제안하였다. 이 명령어를 사용하면 한 번의 명령어 호출로 XML 파서에서 검색을 수행하므로 소프트웨어 오버헤드가 없으며 하드웨어 사이클도 반 정도로 감소시키는 성능향상을 얻을 수 있다.

참고문헌

[1] C. Chang, F. Mohd-Yasin, and A. Mutspha, "RBStreX: Hardware XML Parser for Embedded System", in *Proc. of* ICITST, 2009.

[2] Tarai Inc., "RAX : Random Access XML", white paper, Apr 2004.

[3] XimpleWare, "VTD-XML: The Future of XML Processing", http://vtd-xml.sourceforge.net.

[4] Tong, T.etal, "Rules about XML in XML", *Expert Systems with Applications*, Vol. 30, No.2, 2006, pp. 397-411.

[5] Zdfu Dai, Nick NI, Jianwen Zhu, "A 1 Cycle-Per-Byte XML Parsing Accelerator", in *Proc of 18th ACM/SIGDA Int. Symp. Field Prog. Gate Arrays* (FPGA'10), 2010.