XML Document Parsing: Operational and Performance Characteristics

Parsing is an expensive operation that can degrade XML processing performance. A survey of four representative XML parsing models—DOM, SAX, StAX, and VTD—reveals their suitability for different types of applications.

Tak Cheung Lam and Jianxun Jason Ding
Cisco Systems

Jyh-Charn Liu
Texas A&M University

Broadly used in database and networking applications, the Extensible Markup Language is the de facto standard for the interoperable document format. As XML becomes widespread, it is critical for application developers to understand the operational and performance characteristics of XML processing.

As Figure 1 shows, XML processing occurs in four stages: *parsing*, *access*, *modification*, and *serialization*. Although parsing is the most expensive operation,[1] there are no detailed studies that compare

- the processing steps and associated overhead costs of different parsing models,
- tradeoffs in accessing and modifying parsed data, and
- XML-based applications' access and modification requirements.

Figure 1 also illustrates the three-step parsing process. The first two steps, *character conversion* and *lexical analysis,* are usually invariant among different parsing models, while the third step, *syntactic analysis,* creates data representations based on the parsing model used.

To help developers make sensible choices for their target applications, we compared the data representations of four representative parsing models: document object model (DOM; www.w3.org/DOM), simple API for XML (SAX; www.saxproject.org), streaming API for XML (StAX; http://jcp.org/en/jsr/detail?id=173), and virtual token descriptor (VTD; http://vtd-xml.sourceforge.net). These data representations result in different operational and performance characteristics.

XML-based database and networking applications have unique requirements with respect to access and modification of parsed data. Database

Published by the IEEE Computer Society

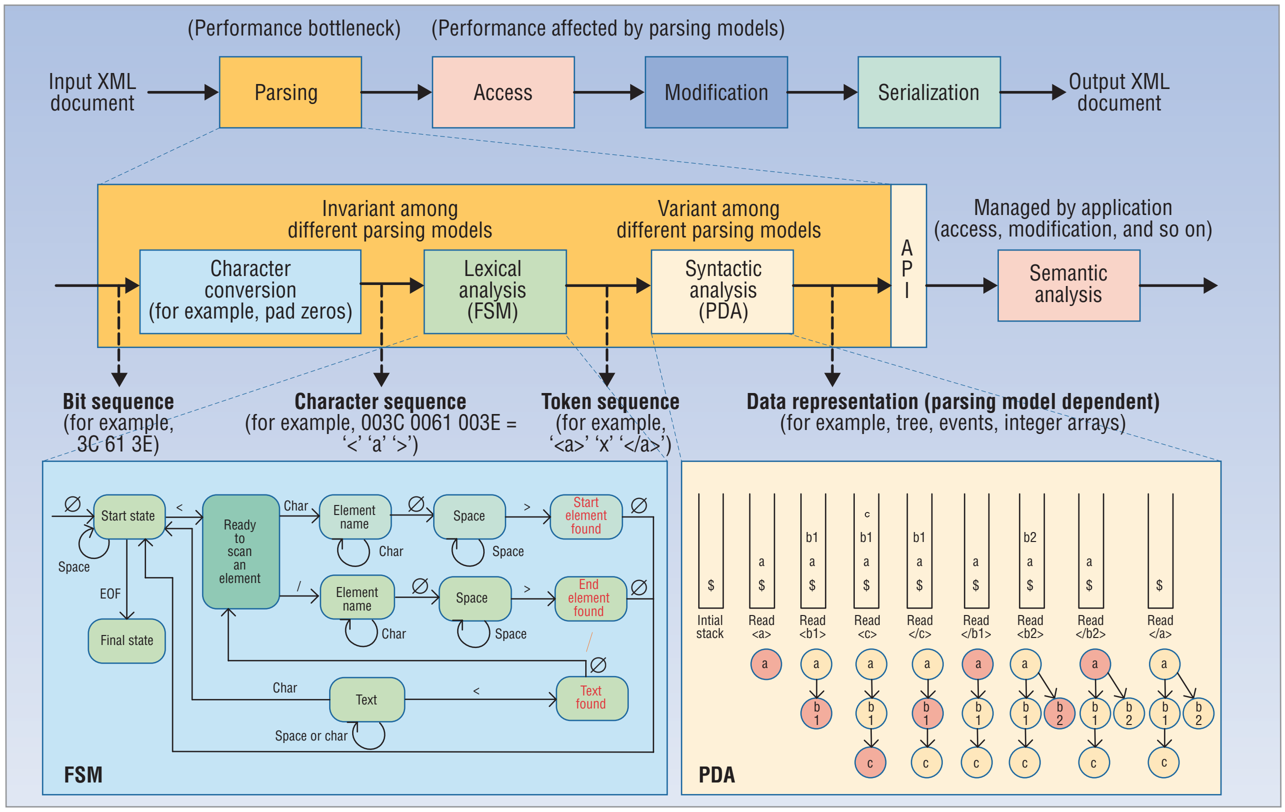

(Performance bottleneck)
(Performance affected by parsing models)
Input XML document
Parsing
Access
Modification
Serialization
Output XML document
Invariant among different parsing models
Variant among different parsing models
Managed by application (access, modification, and so on)
Character conversion (for example, pad zeros)
Lexical analysis (FSM)
Syntactic analysis (PDA)
API
Semantic analysis
Bit sequence (for example, 3C 61 3E)
Character sequence (for example, 003C 0061 003E = '<' 'a' '>')
Token sequence (for example, '<a>' 'x' '</a>')
Data representation (parsing model dependent) (for example, tree, events, integer arrays)
Start state
Ready to scan an element
Element name
Space
Start element found
End element found
Final state
Text
Text found
Char
Space
EOF
Space or char
FSM
Intial stack
Read <a>
Read <b1>
Read <c>
Read </c>
Read </b1>
Read <b2>
Read </b2>
Read </a>
PDA

Figure 1. XML processing stages and parsing steps. The three-step parsing process is the most expensive operation in XML processing.

applications must be able to access and modify the document structure back and forth; the parsed document resides in the database server to receive multiple incoming queries and update instructions. Networking applications rely on one-pass access and modification during parsing; they pass the unparsed document through the node to match the parsed queries and update instructions reside in the node.

XML PARSING STEPS

An XML parser first groups a bit sequence into characters, then groups the characters into *tokens*, and finally verifies the tokens and organizes them into certain data representations for analysis at the access stage.

Character conversion

The first parsing step involves converting a bit sequence from an XML document to the character sets the host programming language understands. For example, documents written in Western, Latin-style alphabets are usually created in UTF-8, while Java usually reads characters in UTF-16. In most cases, a UTF-8 character can be converted to UTF-16 by simply padding 8-bit leading zeros. For example, the parser converts "<" "a" ">" from "3C 61 3E" to "003C 0061 003E" in hexadecimal representation. It is possible to avoid such a character conversion by composing the documents in UTF-16, but UTF-16 takes twice as much space as UTF-8, which has tradeoffs in storage and character scanning speed.

Lexical analysis

The second parsing step involves partitioning the character stream into subsequences called tokens. Major tokens include a *start element*, *text*, and an *end element*, as Table 1 shows. A token can itself consist of multiple tokens. Each token is defined by a regular expression in the World Wide Web Consortium (W3C) XML specifications, as shown in Table 2. For example, a start element consists of a "<", followed by an element name, zero or more attributes preceded by a space-like character, and a ">". Each attribute consists of an attribute name, followed by an "=" enclosed by a

Table 1. XML token examples.

Token	Example
Start element	**<Record>**John</Record>
End element	<Record>John**</Record>**
Text	<Record>**John**</Record>
Start element name	<**Record** private = "yes">
Attribute name	<Record **private** = "yes">
Attribute value	<Record private = "**yes**">

Table 2. Regular expressions of XML tokens.

Token	Regular expression
Start element	'<' Name (S Attribute)* S? '<'
End element	'</' Name (S Attribute)* S? '<'
Attribute	Name Eq AttValue
S	(0×20 |0×9|0×D |0×A)+ *Space-like characters*
Eq	S? '=' S? *Equal-like characters*
Name	Some other regular expressions
AttValue	Some other regular expressions

* = 0 or more; ? = 0 or 1; + = 1 or more.

zero or one space-like character on each side, and then an attribute value.

A finite-state machine (FSM) processes the character stream to match the regular expressions. The simplified FSM in Figure 1 processes the start element, text, and the end element only, without processing attributes. To achieve full tokenization, an FSM must evaluate many conditions that occur at every character. Depending on the nature of these conditions and the frequency with which they occur, this can result in a less predictable flow of instructions and thus potentially low performance on a general-purpose processor. Proposed tokenization improvements include assigning priority to transition rules,[2] changing instruction sets for "<" and ">",[3] and duplicating the FSM for parallel processing.[4]

Syntactic analysis

The third parsing step involves verifying the tokens' well-formedness, mainly by ensuring that they have properly nested tags. The *pushdown automaton* (PDA) in Figure 1 verifies the nested structure using the following transition rules:

1. The PDA initially pushes a "$" symbol to the stack.
2. If it finds a start element, the PDA pushes it to the stack.
3. If it finds an end element, the PDA checks whether it is equal to the top of the stack.

- If yes, the PDA pops the element from the stack.
  If the top element is "$", then the document is "well-formed." Done!
  Otherwise, the PDA continues to read the next element.
- If no, the document is not "well-formed." Done!

In the complete well-formedness check, the PDA must verify more constraints—for example, attribute names of the same element cannot repeat. If schema validation is required, a more sophisticated PDA checks extra constraints such as specific element names, the number of child elements, and the data type of attribute values.

In accordance with the parsing model, the PDA organizes tokens into data representations for subsequent processing. For example, it can produce a tree object using the following variation of transition rule 2:

If it finds a start element, the PDA checks the top element before pushing it to the stack.

- If the top element is "$", then this start element is the root.
- Otherwise, this start element becomes the top element's child.

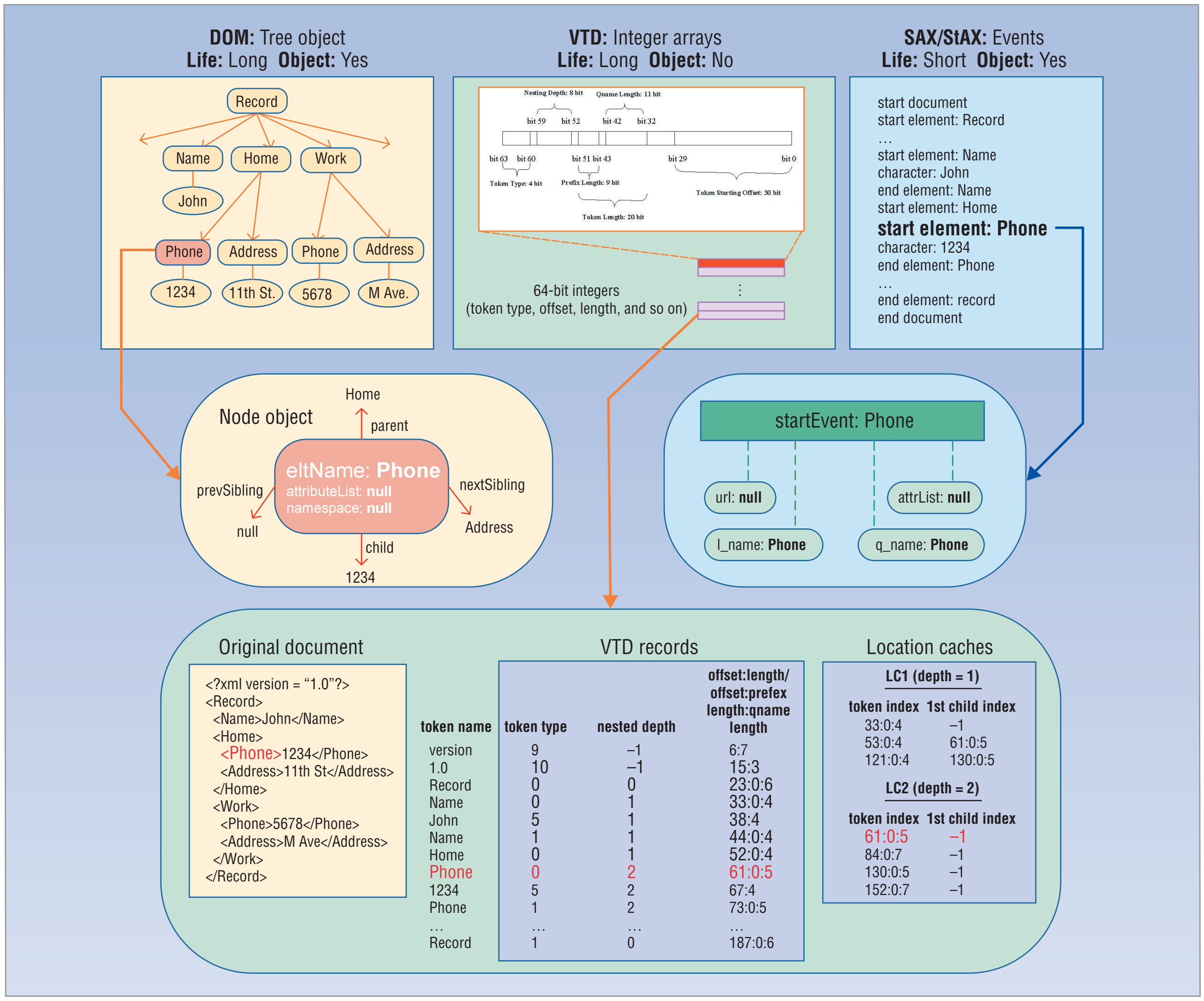
After syntactic analysis, the data representations are available for access or modification by the application via various APIs provided by different parsing models, including DOM, SAX, StAX, and VTD.

PARSING MODEL DATA REPRESENTATIONS

XML parsers use different models to create data representations. DOM creates a tree object, VTD creates integer arrays, and SAX and StAX create a sequence of events. Both DOM and VTD maintain long-lived structural data for sophisticated operations in the access and modification stages, while SAX and StAX do not. DOM as well as SAX and StAX create objects for their data representations, while VTD eliminates the object-creation overhead via integer arrays.

DOM and VTD maintain different types of long-lived structural data. DOM produces many node objects to build the tree object. Each node object stores the element name, attributes, namespaces, and pointers to indicate the parent-child-sibling relationship. For example, in Figure 2 the node object stores the element name of Phone as well as the pointers to its parent (Home), child (1234), and next sibling (Address). In contrast, VTD creates no object but stores the original document and produces arrays of 64-bit integers called *VTD records* (VRs) and *location caches* (LCs). VRs store token positions in the original document, while LCs store the parent-child-sibling relationship among tokens.

While DOM produces many node objects that include pointers to indicate the parent-child-sibling relationship, SAX and StAX associate different objects with different events and do not maintain the structures among objects. For example, the start element event is associated with three String objects and an Attributes object for the namespace uniform resource identifier (URI), local name, qualified name, and attribute list. The end element event is similar to the start element event without an attribute list. The character event is associated with an array of characters and two integers to denote



```
<?xml version = "1.0"?>
<Record>
  <Name>John</Name>
  <Home>
    <Phone>1234</Phone>
    <Address>11th St</Address>
  </Home>
  <Work>
    <Phone>5678</Phone>
    <Address>M Ave</Address>
  </Work>
</Record>
```

token name	token type	nested depth	offset:length/ offset:prefix length:qname length
version	9	–1	6:7
1.0	10	–1	15:3
Record	0	0	23:0:6
Name	0	1	33:0:4
John	5	1	38:4
Name	1	1	44:0:4
Home	0	1	52:0:4
Phone	0	2	61:0:5
1234	5	2	67:4
Phone	1	2	73:0:5
…	…	…	…
Record	1	0	187:0:6

LC1 (depth = 1)

token index	1st child index
33:0:4	–1
53:0:4	61:0:5
121:0:4	130:0:5

LC2 (depth = 2)

token index	1st child index
61:0:5	–1
84:0:7	–1
130:0:5	–1
152:0:7	–1

Figure 2. Data representation example. The start element of Phone is represented by "0, 2, 61:0:5" in VTD records. This entry indicates that there is a token of type 0 (start element) at nested depth 2, and this token's first character is located at the 61st position of the original document. This token has a prefix name of length 0, indicating that the token does not use a namespace, and a qualified name of length 5. The token indices (offset: prefix length: qname length) of all start elements are stored in location caches at certain nested depths. For example, LC level 2 (LC2) stores the token indices by its first 32-bit field for all start elements at nested depth 2. The second 32-bit field stores the index of its first child. A token with no child has "–1" in this field. For example, the start element of Phone is recorded in LC2 as "61:0:5, –1".

the start position and text length. In Figure 2, Phone's start element has no attribute and namespace, so SAX and StAX associate it with two String objects to store its local and qualified names.

OPERATIONAL AND PERFORMANCE CHARACTERISTICS

Different data representations result in different operational and performance characteristics, as summarized in Tables 3 and 4, respectively. They also affect the choice of parsing models for various applications, as indicated in Table 5. We focus on how different data representations impact three XML processing capabilities: streaming, access and modification, and hardware acceleration.

Streaming capability

Streaming requires low latency and memory usage, and usually the parser only needs to extract a small portion of the document sequentially without knowing the entire document structure. To understand parsing models' impact on streaming capability, it is important to understand how the parser and application interact during data access.

DOM and VTD. As Figure 3a shows, DOM and VTD can access data only after parsing is complete—that is, when the loop inside the parser program can draw no more tokens from lexical analysis to construct the tree or VRs. A large document will significantly delay data access. Moreover, the two models' long-lived data

Table 3. XML processing operational characteristics.

XML processing stage	DOM	SAX (push)	StAX (pull)	VTD
Parsing	1. Extract token as objects.	1. Extract token as objects.	1. Extract token as objects.	1. Do not extract token as objects (use integers instead).
	2. Build tree by objects (for example, Nodes).	2. Create events by objects (for example, Strings).	2. Create events by objects (for example, Strings).	2. Build location cache and 64-bit VTD records.
	3. Not ready for access.	3. Ready for access—go to step 8 (application handles event).	3. Ready for access—go to step 8 (application handles or skips event).	3. Not ready for access.
	4. Do not destroy any objects.	4. Destroy objects after handling the event.	4. Destroy objects after handling or skipping the event.	4. Do not destroy any objects.
	5. Repeat from step 1 until all tokens are processed.	5. Repeat from step 1 until all tokens are processed.	5. Repeat from step 1 until all tokens are processed.	5. Repeat from step 1 until all tokens are processed.
	6. (Optional) Destroy the original document after building the entire tree.	6. (Optional) Destroy the original document after handling all events.	6. (Optional) Destroy the original document after handling or skipping all events.	6. Keep the original document in memory.
	7. Ready for access.	7. Access is complete—go to step 9.	7. Access is complete—go to step 9.	7. Ready for access.
Access	8. Back-and-forth access: Parsing provides sufficient data structures (tree).	8. Sequential access (no skip): The application creates its own data structure if more advanced access or modification is required (go to step 4).	8. Sequential access (skip forward): The application creates its own data structure if more advanced access or modification is required (go to step 4).	8. Back-and-forth access: Parsing provides sufficient data structures (VTD records and location caches).
Modification	9. Update the tree.	9. Update the data structure from step 8.	9. Update the data structure from step 8.	9. Update by making new copy of the document.
	10. Write the tree in XML format.	10. Write the data structure from step 9 in XML format.	10. Write the data structure from step 9 in XML format.	10. The document is already in XML format.
	11. Destroy the tree.	11. Destroy the data structure.	11. Destroy the data structure.	11. Destroy VTD records and location cache.

Table 4. XML processing performance characteristics.

Category	DOM	SAX (push)	StAX (pull)	VTD
Output	Tree object	Events (all tokens)	Events (interested tokens)	Integer array
Parsing (CPU)	High	Medium	Medium	Low
Parsing (memory)	Intensive	Low	Low	Medium
Access (navigation)	Fast (back and forth)	Slow (sequential: no skipping)	Medium (sequential: skip forward)	Fast (back and forth)
Modification (update)	Medium (not incremental)	Depends (template/forward)	Depends (template/forward)	Fast (incremental)
Estimated[5] throughput, small file (1 Kbyte-15 Kbytes)*	~10 Mbytes per second	~20 Mbytes per second	~20 Mbytes per second	~50 Mbytes per second
Estimated[5] throughput, large file (1 Mbyte-15 Mbytes)*	~5 Mbytes per second	~20 Mbytes per second	~20 Mbytes per second	~40 Mbytes per second
Estimated memory,* large file (1 Mbyte-15 Mbytes)	~7 Mbytes	Does not depend on document size	Does not depend on document size	~1.5 Mbytes

* The test platform is a Sony VAIO laptop with a Pentium M 1.7-GHz processor (2-Mbyte integrated L2 cache) and 512-Mbyte DDR2 RAM. The front bus is clocked at 400 MHz. The OS is Windows XP Professional Edition with Service Pack 2, and the Java virtual machine is version 1.5.0_06.

Table 5. XML processing capability and applications.

Capability	DOM	VTD	SAX/StAX
Parsing output	Tree object (long-lived)	Integer arrays (long-lived)	Events (frequently destroyed)
Streaming	No	No	Yes
Access	Back and forth repeatedly	Back and forth repeatedly	One pass sequentially
Modification	Directly modify the tree (frequent and complex is OK)	Copy and paste to new array (only OK if simple and rare)	Get and put into template (only OK if simple and rare)
Desirable application	Database	In between	Networking

representations make memory usage grow with document size, which is undesirable for streaming.

SAX and StAX. In contrast, SAX and StAX interlace parsing and access, so the application can access partial data before parsing is complete. Because the objects associated with events can be destroyed regularly, memory usage does not grow with document size. Therefore, although SAX and StAX are two times slower than VTD due to frequent object allocations,[5] they are still better candidates for streaming applications.

SAX and StAX are both designed for streaming applications, but their parser-application interactions are different. As Figure 3b shows, SAX adopts the push model, which uses callback functions to report events from the parser to the application. The parser has a loop to continuously check tokens produced from lexical analysis. When it finds a token, the parser invokes a callback function based on the token type such as startElement(…), endElement(…), characters(…).

In contrast, StAX adopts the pull model, as shown in Figure 3c. An application in the pull model can skip uninterested events by calling nextEvent(), whereas an application in the push model must handle all events fed from the parser. The pull model does not need to maintain states between callback functions to decide correct actions, making the programming flow more natural and maintainable.

A common misconception is that pull parsers are always faster than push parsers because they save effort by skipping uninteresting events. However, numerous studies reveal that this is not always true.[5-8] Although the application can skip events by calling nextEvent(), the parser still creates the events sequentially without skipping them. Performance therefore depends on the application needs. If the application has to navigate through the entire document, the pull model has little advantage over the push model, but if it can stop parsing after accessing certain uninteresting data, the pull model is faster.

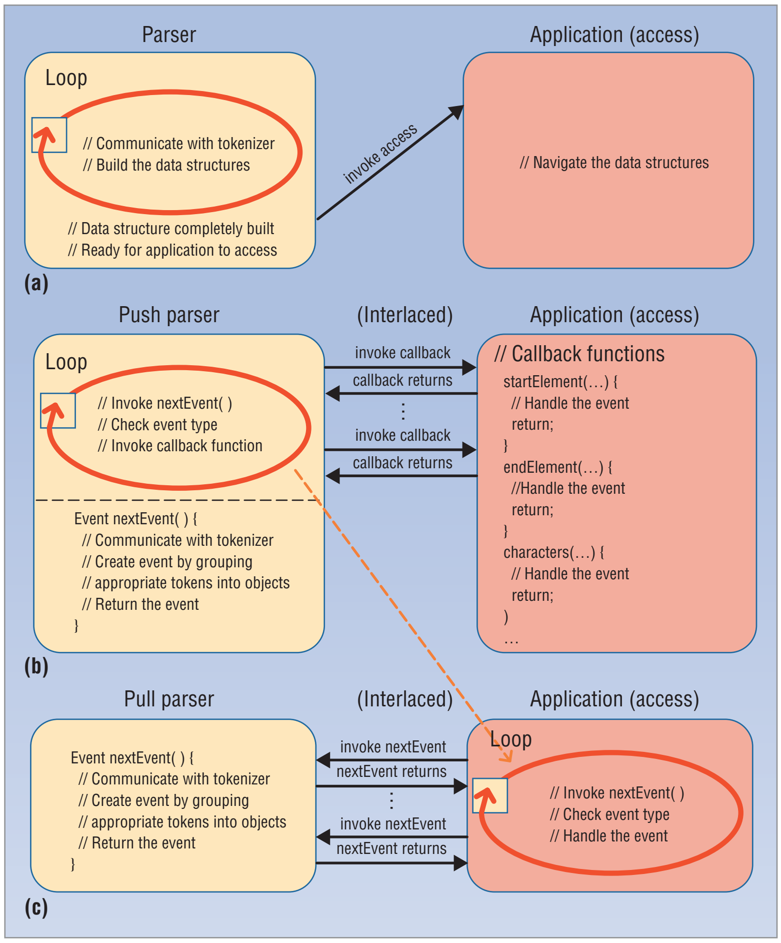


Figure 3. Parser-application interactions. (a) DOM and VTD: Access after parsing completed; (b) SAX: Interlaced parsing and access (parser driven); (c) StAX: Interlaced parsing and access (application driven).

Access and modification capability

To compare the four parsing models' ability to access and modify data back and forth, we considered two scenarios based on the input XML document shown in Figure 2, in which the root Record has a child Name with several siblings, each of which has multiple Phone

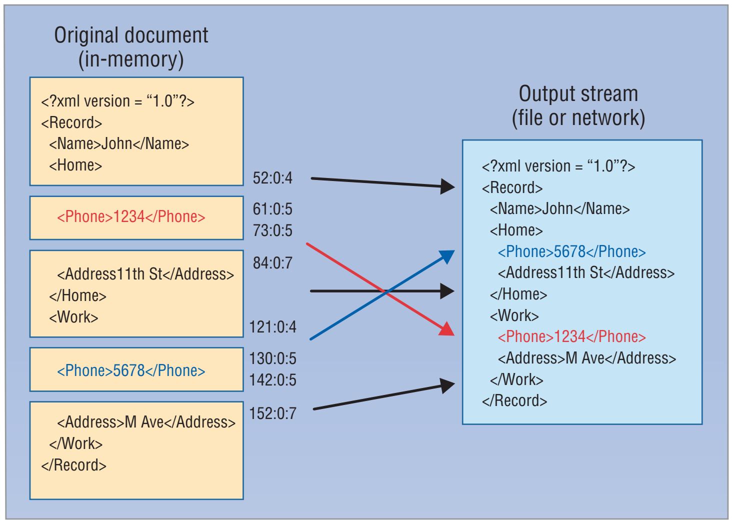


Figure 4. VTD's "copy and paste" approach. VTD rearranges the original document into a new buffer, a modification that requires three processing steps in DOM. The approach makes VTD more suitable than DOM for incremental updates.

children. The siblings can appear before Name (not shown in the figure). The access scenario calls for printing the texts of all Phone elements if the text of Name is John, while the modification scenario calls for switching the first and last Phone elements in the document.

SAX and StAX. SAX and StAX do not maintain long-lived structural data and are limited to sequential access. Thus, in the access scenario, the application must either buffer the texts of all Phone elements before the FSM matches John or parse the document again after the FSM matches John. Memory consumption depends on Name's location in the document. In the modification scenario, the application must buffer the entire document before it can switch the last Phone element with the first one. SAX and StAX thus do not have an advantage in terms of memory consumption as they do in streaming applications. For this reason, SAX and StAX are typically used for forward-only applications or simple modifications via template such as Extensible Stylesheet Language Transformations (XSLT).

DOM and VTD. In contrast to SAX and StAX, DOM and VTD maintain parent-child-sibling information in their long-lived structural data. Preparing this data incurs more overhead, but the simple-to-navigate tree or LCs ease access.

VTD consumes far less memory than DOM—1.3 to 1.5 times the original document size, which is 3 to 5 times smaller than the DOM tree—and it parses 5 to 10 times faster than DOM.[5] This performance difference is primarily attributable to DOM object creations. Many object-oriented programming languages incur small memory overhead per object allocation, and VTD is immune to this overhead because it uses integer arrays instead of objects. Moreover, VRs and LCs are constant in length and thus the VTD implementation can store them in large memory blocks. By allocating a large integer array for 4,096 VRs, VTD has a per-array allocation overhead of only 16 bytes, significantly reducing the per-record overhead.

Although VTD outperforms DOM in throughput and memory usage, it cannot replace DOM because of their different modification capabilities. DOM is more suitable for massive and frequent updates, much as a linked-list is more suitable than an array for update operations. It is possible to add or delete a node to or from the DOM tree by simply manipulating the pointers between tree nodes. The modified tree is then ready for further updates.

On the other hand, when adding or deleting a record to or from the integer arrays in VRs, VTD might need to rebuild many VR and LC entries to process the next update operation. VTD is thus more suitable for incremental updating. As Figure 4 shows, it employs a "copy and paste" approach in the modification scenario that rearranges the original document into a new buffer. Such a modification requires three processing steps in DOM: building the in-memory tree, navigating and updating the tree, and translating the updated tree back into XML format. These steps involve many string concatenations, buffer allocations, and character conversions, making DOM less efficient for simple and rare modifications.

Hardware acceleration capability

Hardware acceleration can boost XML parsing performance that general-purpose processors otherwise cannot achieve. As Table 4 shows, XML parsing software throughput on a general-purpose processor ranges from 5 to 50 Mbytes per second, but enterprise software servers call for gigabit-per-second throughput rates.[9,10] To explore the different parsing models' hardware acceleration capabilities, it is necessary to understand their data structures.

VTD's symmetric data structure makes it or other similar flat-array XML representations ideal candidates for hardware acceleration. The flat array allows efficient memory management before and after acceleration and eases the crossing between the hardware acceleration and the application logic spaces.

In contrast, the parsing and application logic in SAX and StAX are interwoven via callbacks or events intensively. The high frequency and associated overhead of entering and leaving the hardware acceleration space to and from the application logic space make these models unsuitable for hardware acceleration.

DOM's tree structure also poses serious challenges to hardware acceleration. Implementing circuitries to operate and manage dynamic tree representations is far more complex than typical field-programmable gate array (FPGA) and application-specific integrated circuit (ASIC) designs, making DOM less favorable for hardware acceleration.

No commercial VTD hardware is yet available on the market, so at this point it is impossible to evaluate the performance of hardware-accelerated VTD versus software-based DOM on frequently updated operations.

Our analysis of parsing models' data representations and their impact on XML processing yielded the following conclusions. Both DOM and VTD are good for back-and-forth data access. VTD parses faster than DOM and consumes less memory. VTD is better for simple and rare modifications, while DOM is better for complex and frequent ones. SAX and StAX are appropriate for applications with extremely restrictive memory but not for back-and-forth access or modification.

In a nutshell, DOM is most suitable for database applications, while SAX and StAX are more appropriate for streaming applications. VTD is a good candidate for hardware acceleration based on its symmetric array structure, but its effectiveness in real-world applications using a commercial hardware accelerator remains an open question.

References

1. M. Nicola and J. John, "XML Parsing: A Threat to Database Performance," *Proc. 12th Int'l Conf. Information and Knowledge Management* (CIKM 03), ACM Press, 2003, pp. 175-178.
2. J. van Lunteren et al., "XML Accelerator Engine," *Proc. 1st Int'l Workshop High Performance XML Processing*, 2004; www.zurich.ibm.com/~jvl/xml2004.pdf.
3. L. Zhao and L. Bhuyan, "Performance Evaluation and Acceleration for XML Data Parsing," *Proc. 9th Workshop Computer Architecture Evaluation Using Commercial Workloads* (CAECW 06), 2006; www.cs.ucr.edu/~zhao/paper/caecw06_xml.pdf.
4. Y. Pan et al., "Parallel XML Parsing Using Meta-DFAs," *Proc. 3rd IEEE Int'l Conf. e-Science and Grid Computing* (e-Science 07), IEEE CS Press, 2007, pp. 237-244.
5. J. Zhang, "Simplify XML Processing with VTD-XML," *JavaWorld,* 27 Mar. 2006; www.javaworld.com/javaworld/jw-03-2006/jw-0327-simplify.html.
6. Y. Oren, "SAX Parser Benchmarks," 2002; http://piccolo.sourceforge.net/bench.html.
7. B. Nag, "A Comparison of XML Processing in .NET and J2EE," *Proc. XML Conf. and Exposition 2003* (XML 03), 2003; www.idealliance.org/papers/dx_xml03/papers/06-01-03/06-01-03.pdf.
8. A. Slominski, "On Performance of Java XML Parsers"; www.cs.indiana.edu/~aslom/exxp.
9. A. Waheed and J. Ding, "Benchmarking XML Based Application Oriented Network Infrastructure and Services," *Proc. 2007 Int'l Symp. Applications and the Internet* (SAINT 07), no. 15, IEEE CS Press, 2007.
10. J. Ding and A. Waheed, "Dual Processor Performance Characterization for XML Application-Oriented Networking," *Proc. 2007 Int'l Conf. Parallel Processing* (ICPP 07), no. 52, IEEE CS Press, 2007.

***Tak Cheung (Brian) Lam** is a software engineer in the Wireless and Security Technology Group at Cisco Systems. His research interests include applied cryptography, network security, and performance optimization. Lam received a PhD in computer science from Texas A&M University. He is a member of the IEEE Computer Society. Contact him at brlam@cisco.com.*

***Jianxun Jason Ding** is a senior technical leader in Cisco's Open Platform Software Technology Center. His research interests include performance testing, evaluation, and optimization of both hardware and software systems. Ding received a PhD in computer science from Texas A&M University. He is a member of the IEEE Computer Society. Contact him at jiding@cisco.com.*

***Jyh-Charn Liu** is a professor in the Department of Computer Science at the Dwight Look College of Engineering, Texas A&M University. His research interests include real-time distributed computing systems, network performance and security, and medical informatics. Liu received a PhD in electrical and computer engineering from the University of Michigan. He is a member of the IEEE Computer and Communications societies. Contact him at liu@cs.tamu.edu.*