Expeditious XML processing

Kelvin Yeow[1], Nigel Horspool[2], and Michael Levy[3]

Abstract

The efficiency of an XML processor is highly dependent on the representation of the XML document in the computer's memory. We present a representation for XML documents, derived from Warren's representation of Prolog terms in WAM, which permits very efficient access and update operations. Our scheme is implemented in CXMLParser, a non-validating XML processor. We present the results of a performance comparison with two other XML processors which show that CXMLParser meets its high-performance goal.

Key Words: general tree; parsing; WAM; XML processor

Approximate Word Count: 4000

[1] Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada V8P 5C2. Email: whyeow@csc.uvic.ca. Web: http://www.csc.uvic.ca/~whyeow
[2] Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada V8P 5C2. Email: nigelh@csr.uvic.ca. Web: http://www.csc.uvic.ca/~nigelh
[3] newheights, 1006 Government Street, Victoria, British Columbia, Canada V8W 1X7. Email: mlevy@newheights.com. Web: http://www.newheights.com

Expeditious XML processing

INTRODUCTION

An XML processor is a software module that usually exists within a larger system. An XML processor provides the facility to access, modify, parse or create an XML document, usually through the API (Application Program Interface). The core data structure of an XML processor is its representation of the XML document. The efficiency of an XML processor is critically dependent on this representation.

There are many ways an XML document can be represented. After all, it is a general tree, where open and close tags delimit subtrees. The choice of representation depends on the context of use and on various performance trade-offs.

We present CXMLParser, a non-validating XML processor based on the techniques used in WAM (Warren Abstract Machine). CXMLParser is implemented in C++. It conforms to the W3C XML 1.0 Recommendation [7], specifically the well-formedness constraint. The conformance is exercised to a limited extent but sufficient for the proof of concept. It recognises the three common XML constructs: attributes, elements and text data. The sequential memory technique for representing the XML document originates from Warren's representation of terms in WAM. We adapted the technique to cater for runtime insertion and deletion.

To gauge the relative time efficiency of CXMLParser, we conducted a performance comparison with two other XML processors. The two candidate XML processors are Xerces C++ (Apache XML project) [5] and XML4J implemented in Java (IBM alphaWorks) [6]. The performance comparison tests four common operations: parse, delete, insert and traverse on three distinct sized XML documents. We present the methodology and the results of the performance comparison.

BACKGROUND

In this section, we discuss the correlation of XML, WAM, and general tree – which serves to illustrate the direction of our approach that warrants the implementation of CXMLParser.

XML as a general tree

An XML document contains data and the relationship between data. The relationship between data is expressed by means of nesting. The inherent hierarchical structure of an XML document can be likened to the tree structure.

A tree structure describes branching relationship between nodes [1]. A node is the atomic unit of a tree. Further, the relationships between nodes are described in terminologies borrowed from biological trees [2]. The root of a tree is the single node having no parent node. For any other given nodes, the node immediately above it is known as its parent and the nodes directly below it are its children. A tree is also called a *general tree* to distinguish it from more specialized kinds of trees. Any node in a general tree can have zero or more children. We will use the terms general tree and tree interchangeably in the rest of this paper.

There are many ways a general tree can be represented inside a computer. The type of representation is determined by the way the relationship or edge is established and the way the memory is allocated for the storage of the data.

The edges are established either implicitly or explicitly. Implicit edges are established through *adjacency*. The sequential memory representation of a binary tree that uses implicit edges allows a parent node to have direct access to its left and right child through the index of the parent node [2]. Implicit edges consume no space but it requires the arity be known a priori. In the case of a binary tree, there will be wastage if the tree is unbalance. Explicit edges are established through pointers similar to the linked-list data structure. Explicit edges consume more space but it is more intuitive and allows for efficient insertion.

There are two memory allocation techniques: sequential memory and dynamically allocated memory. The term dynamic implies non-contiguous memory locations for nodes whereas the term sequential implies contiguous memory locations for nodes. The dynamically allocated memory techniques allow for efficient insert and delete operations. However, as the term dynamic implies, the memory is allocated on need and individual basis. On some machines, it involves the operating system intervention. It is a double-edge sword – a parser, for a XML document that receives a stream of characters as input, will need to invoke memory allocation for every instance of a recognized node – a potentially expensive alternative compared to the sequential memory techniques. On the other hand, the sequential memory techniques are cumbersome if the tree structure is subjected to changes in shape and size at runtime.

Warren's representation of terms

In 1983, David H. D. Warren designed an abstract machine for the execution of Prolog consisting of a memory architecture and an instruction set [3]. The memory architecture thrives on *terms*. A term is either a variable, constant or structure. Of particular interest is the representation of the term structure in the form of $f(t_1, ..., t_n)$ where f is a symbol called a *functor*, and the t_i's are the *subterms*. The number of subterms for a given functor is predetermined and called its *arity*. Recursively, a subterm can be a variable, constant or structure. To recapitulate this in the context of a tree structure, a non-leaf node and its children can be viewed as a functor and its subterms respectively. The number of children of a parent node is then known as its arity.

As part of the illustration of the WAM memory architecture with respect to XML, we define a simplified representation for terms consisting of a global block of storage in the form of an addressable heap called HEAP, which is an array of data cells. A heap cell's address is its index in the array HEAP. There are two types of data to be stored in HEAP: constant (denoted by a capitalized identifier) and structure (denoted by an identifier starting with a lower-case letter).

The heap format used for representing a structure $f(t_1, ..., t_n)$ will consist of $n + 2$ contiguous heap cells. The first two of these $n + 2$ cells stores the functor and its arity. The n other cells contain references to the root of the n subterms in proper order. If HEAP[k] = f then HEAP[$k + 1$] will refer to its arity. Hence, HEAP[$k + 2$] will refer to the first subterm t_1 and HEAP[$k + n + 1$] will refer to the n-th (and last) subterm t_n.

Figure 1 shows the postorder heap representation of the functor $f(A, g(B, C, D))$ that starts at the root, f. Here A, B, C and D are the constants. f and g are the functors. It is interesting to note that this technique has a combination of implicit and explicit edges. The cells that contain the constants need not be contiguous while the references to the constants belonging to a structure are strictly contiguous. Traversals are guided by the arity. Note that the illustrations have been simplified to suit the context of discussion and may not mirror Aït-Kaci's [3] illustrations.

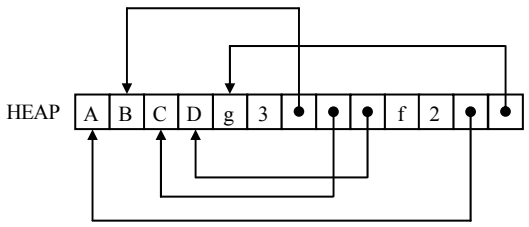

Figure 1. The postorder heap representation of the functor $f(A, g(B, C, D))$

A term as a general tree

A *term* in WAM is similar to a general tree. The term structure is a parent node. The term constants are the leaf nodes. The representation of a term in WAM is a form of sequential memory technique. One attractive property of this representation is that a parent node has direct access to its children nodes. However, this technique assumes that the arity is static; hence, it is particularly inadequate for insertion at runtime. We will show how this technique can be extended to overcome this limitation under the implementation section.

XML as a term

It then follows that any XML document can be expressed in a similar fashion. The XML element corresponds to the term structure. The XML attribute corresponds to a term structure with an arity of two at all times – a key and value pair. The text node corresponds to the term constant.

ALTERNATIVE REPRESENTATIONS

There are a number of well-documented general tree representations applicable to the XML document. The differences between the representations manifest in the way the relationship is established and the way the memory is allocated for the storage of the data. We review them here in the context of the effort required to perform common operations such as insert, delete, and access by position.

First Child - Right Sibling (FCRS)

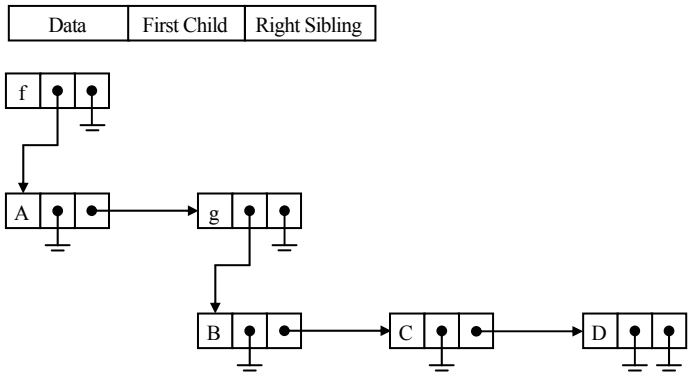

Figure 2. First Child - Right Sibling

This representation uses explicit edges to establish relationship between the nodes. The data of each node is stored in a dynamically allocated memory. This representation is similar to the linked representation of a binary tree. Each tree node contains its data, a pointer to its first child and a pointer to its right sibling. If a node has a null First Child, it is a leaf node. With the exception of the root node, a node with a null Right Sibling is the last child of its parent node. The amount of storage space for *n* nodes is *3n*.

Postorder with Degrees (PWD)

This representation uses implicit edges to establish relationship between the nodes. The data storage is in the form of sequential memory. Each tree node contains the data and its *degree*. The degree of a node is the number of subtrees of the node [1]. The amount of storage space for *n* nodes is *2n*.

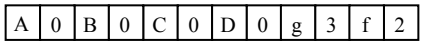

Figure 3. Postorder with Degrees

List of Children (LOC)

This representation uses explicit edges to establish relationship between the nodes. Each tree node contains data and a list of pointers to its children nodes. The main physical structure is usually an array while the list of pointers to children nodes is a singly linked-list. The amount of storage space for *n* nodes is *4n – 2*.

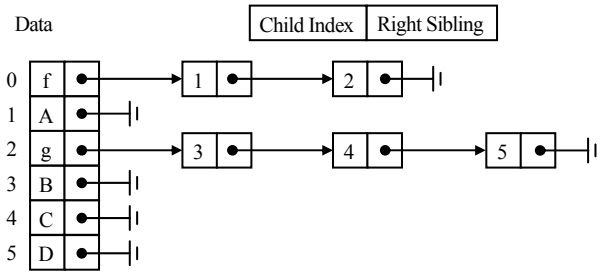

Figure 4. List of Children

Discussion of alternatives

The usage of explicit edges in FCRS and LOC allows for efficient insertion and deletion especially on individual basis during runtime. However, during a parse operation, frequent calls for memory allocation can be expensive. Table 1 shows the order of magnitude of effort required for basic operations on each representation.

Operations \ Representations	FCRS	PWD	LOC
Insertion	*O(1)*	N/A	*O(1)*
Deletion	*O(1)*	N/A	*O(1)*
Access by position	*O(n)*	*O(1)*	*O(n)*

Table 1. Orders of magnitude of effort required to perform basic operations

Our technique closely resembles PWD. The variation lies in the location of the children nodes. PWD requires the children nodes to be adjacent while our technique keeps track of the children nodes through the children node pointers.

IMPLEMENTATION

This section consists of three parts: first, we describe the core data structure of CXMLParser. We then describe the nodes recognized by CXMLParser and their representation. Based on the data structure, we describe four operations: insert, delete, parse and traverse. In this section, the data will be termed as the character literals and the relationship between data will be termed as the structure.

Core Data Structure

The structure of the XML document will be stored in a block of contiguous memory in the form of an oversized integer array. The character literals of the XML document are stored in an oversized character array. Throughout the rest of this paper, the integer array and the character array will be known as HEAP and CHAR respectively.

For each array, a stack pointer points to the next available cell for consumption. The stack pointers are assumed to point to the next available cell at the end of the used portions of the array. Hence, they will not be shown in the diagrams.

Nodes

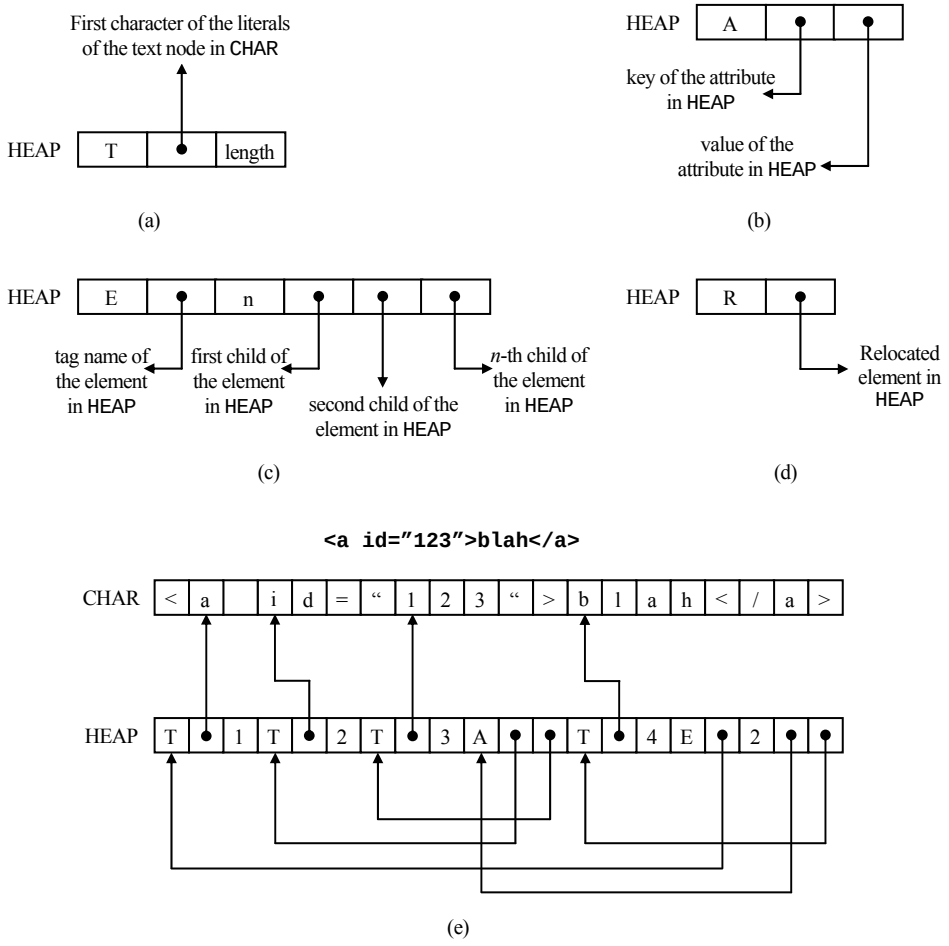

Figure 5. Representation of (a) a text node, (b) an attribute node, (c) an element node, (d) a redirection node, and (e) an example of an XML document in CXMLParser

There are four types of node: text, attribute, element, and redirection. The type of the node will be encoded in the HEAP using an enumeration type constant.

The text node consists of its literals stored in CHAR and its structure stored in HEAP. The heap format used for representing a text node consists of 3 contiguous heap cells as shown in Figure 5(a). The contiguous cells store the type of the node denoted by T for text node, a pointer to the first character of its literal in CHAR, and the number of characters (or length).

An attribute has a key and its associated value; both the key and the value are represented as the text nodes. An attribute node has two child nodes at any instance. The heap format used for representing an attribute will consist of 3 contiguous heap cells as shown in Figure 5(b). The contiguous cells store the type of the node denoted by A for attribute, a pointer to the key and a pointer to the value. If HEAP[k] = id then HEAP[k + 1] contains the pointer to the key and HEAP[k + 2] contains the pointer to the value.

An element is denoted by an open and close tag. The content of an element node can be other elements, attributes or a text node. The heap format used for representing an element will consist of n + 3 contiguous heap cells as shown in Figure 5(c). The first two of these n + 3 cells stores the type of node denoted by E for element, a pointer to its tag name and arity. The n other cells contain references to the root of the n children nodes. If HEAP[k] = a then HEAP[k + 1] contains the pointer to the tag name and HEAP [k + 2] contains the arity of the element. Hence, HEAP[k + 3] contains the pointer to the first child node and HEAP[$k + n$ + 2] contains the pointer to the n-th (and last) child node.

A redirection node allows the element nodes to relocate during an insertion. A redirection node consists of the type of the node denoted by R for redirection and a pointer to the relocated element as shown in Figure 5(d).

Figure 5(e) shows an example of an XML document and its representation in HEAP and CHAR. Figure 6 shows the usage of the redirection node during an insertion.

Operations

The arity of a term must be known before building its representation on the heap. In addition, once built, the arity is assumed static. This implies that the technique for representing the terms in WAM is not meant for modification of the tree at runtime whether in shape or size.

Depending on applications, a generic XML processor should allow for modifications at runtime. In the case of an insertion, we observe that if a child node is inserted at runtime, the changes to HEAP should be (1) the arity of the parent node increases by one and (2) an additional pointer to the new child node is needed. While (1) s a simple increment, (2) requires an additional adjacent cell. Instead of moving all cells to the right by one, our solution is to copy and append the structure of the parent node to HEAP, and place a redirection node at the previous location so that any references to the previous location can be redirected to the new location. Figure 6(a) and 6(b) show the state of HEAP and CHAR before and after inserting a node to an element.

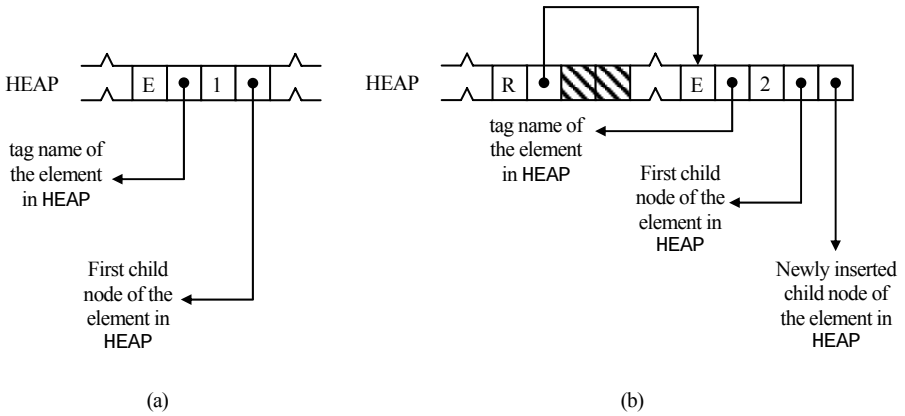

Figure 6. The state of the HEAP (a) before and (b) after an insertion

A redirection node requires two cells. Any request for the parent node through the previous location will be redirected to the new location. The immediate failing of this method is that the redirection process can be expensive if the parent node is relocated (i.e. insertion) frequently. For example, if 5000 nodes are added to the same parent node, a request for the parent node through the initial location requires processing 5000 redirections. Our solution to this problem is to process the redirection once and update the requester. This lazy approach makes best use of the redirection technique, which is a singly linked-list.

Deletion is achieved by decrementing the arity of the parent node. Continuing from the insertion example, Figure 7 shows the state of the HEAP after removing the newly added child node. If the removed child node is not the last child in the list, then move all subsequent child pointers one cell to the left.

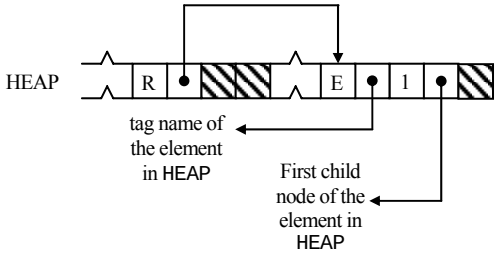

Figure 7. The state of the HEAP after a deletion

The parse operation is a pushdown automaton. The XML document is copied into the memory as CHAR and then annotated. The parse operation is fast because our technique

minimizes memory allocation calls on two fronts: both HEAP and CHAR are allocated once.

The tree traversal operation is guided by the arity of each element nodes. An iteration through the children nodes of an element always starts at HEAP[k + 3] through to HEAP[$k + n + 2$] where k is the index of the element in HEAP and n is the arity of the element. Our technique allows direct access to any children nodes from the parent node.

PERFORMANCE TESTING AND COMPARISON

Input

The three XML documents used for the performance testing are small.xml, medium.xml and big.xml. The goal was to emulate a table with records and data fields. The following shows a snapshot of small.xml:

```
<r>
  <record>
    <name>foo</name>
    <tel>123456789</tel>
    <height>170</height>
    <weight>100</weight>
    <language>english</language>
    <alphabets>abcdefghijklmnopqrstuvwxyz</alphabets>
  </record>
  …
</r>
```

The three documents differ by the number of records. The tree height is three for all of the XML documents. Table 2 shows the properties of the three XML documents.

	small.xml	medium.xml	big.xml
Size in bytes	103,012	1,030,012	5,150,012
Number of records	500	5,000	25,000
Total leaf nodes	3,000	30,000	150,000

Table 2. The properties of the three XML documents

Operations

Delete	Parse file
	Start timer
	Get root node
	For each record
	  For each field
	  Remove field from record
	  Remove record from root
	Stop timer

Insert	Start timer
	Initialize document object
	Get root node
	For 1 to Number of records
	  Create record
	  Create fields
	  Link fields to record
	  Link record to root
	Stop timer

Parse	Start timer Parse file Stop timer

Traverse	Parse file
	Start timer
	Get root node
	For each record
	  For each field
	  If field = “name” AND if field.getText() =”mambo”
	  break
	Stop timer

Table 3. The pseudocode of each of the operation

Four operations were tested: delete, insert, parse and traverse operations. These operations are applied to the three XML documents. Table 3 describes the operations in details.

Environment

The performance test was run on a Pentium III 650 Windows 2000 machine with 128 MB RAM and a virtual memory of 192 MB. No background activity was demanded from the CPU while the tests were conducted. The C++ compiler used is Microsoft Visual C++ version 6.0 Enterprise Edition. The Java interpreter and virtual machine are bundled as Java 2 SDK version 1.3.1.

XML Processors

The version of Xerces-C and XML4J used is 1.4 and 1.1.16 respectively. All of the XML processors were run in non-validating mode with namespace processing disabled. In addition, the XML processors were required to build a tree representation of the XML document. The performance was measured with the _ftime() function available from time.h while in Java, the System.currentTimeMillis() function was used.

Results

	CXMLParser		Xerces-C++		XML4J	
	ms	byte/ms	ms	byte/ms	ms	byte/ms
small.xml	3	34337	20	5151	33	3122
medium.xml	200	5150	264	3902	381	2703
big.xml	4573	1126	2526	2039	7033	732

Table 4. The time performance of the delete operation

	CXMLParser		Xerces-C++		XML4J	
	ms	byte/ms	ms	byte/ms	ms	byte/ms
small.xml	20	5151	40	2575	160	644
medium.xml	1182	871	377	2732	951	1083
big.xml	N/A	N/A	1900	2711	4757	1083

Table 5. The time performance of the insert operation

	CXMLParser		Xerces-C++		XML4J	
	ms	byte/ms	ms	byte/ms	ms	byte/ms
small.xml	17	6060	73	1411	367	281
medium.xml	130	7923	748	1377	2510	410
big.xml	691	7453	3825	1346	11443	450

Table 6. The time performance of the parse operation

	CXMLParser		Xerces-C++		XML4J	
	ms	byte/ms	ms	byte/ms	ms	byte/ms
small.xml	7	14716	10	10301	17	6060
medium.xml	60	17167	130	7923	80	12875
big.xml	337	15282	664	7756	317	16246

Table 7. The time performance of the traverse operation

Discussions of results

The design goal of CXMLParser is to be fast at the expense of memory. We have proven that our technique is fast for insert, parse, delete and traverse operations. The declining cost of memory allows us to place higher premium on speed. The result for insertion shows that CXMLParser is very fast for small input, slower as input size increases and unacceptable for large input. An impromptu investigation was conducted to confirm the cause for poor performance in processing medium.xml for insertion. About 60-68% of the total time was spent on reallocation of HEAP while 28-35% of the total time was spent on transferring the parent node from the old to the new location within HEAP.

FUTURE WORK

CXMLParser can be improved on two fronts: memory consumption and minimizing movement of the parent node during insertion.

The pattern of memory consumption eventually affects the time performance as we observed the results of insertion in medium.xml and large.xml. When there is insufficient space in HEAP, it will be reallocated to twice the previous size. This reallocation must be minimised. An immediate solution to this is to implement a list of multi-sized free cells. Knuth [1] describes a dynamic storage allocation algorithm for reserving and freeing variable-size blocks of memory from a larger storage area. This algorithm is then refined into different placement strategies such as best fit, first fit and liberation.

The observation of HEAP reveals that the only change to a parent node during an insertion is an additional pointer to the new child. Instead of moving all cells on the right by one, we choose to relocate the parent node to the next available space in HEAP. This process of relocating is expensive if insertions are done in batches. There are many ways to reduce or prevent the parent nodes from relocating. A linked-list of child pointers can be incorporated without relocating the parent node. This can be achieved by including a redirection flag that redirects to the new child pointer of the parent node.

One of the advantages of our technique is the direct access of any children nodes from the parent node if the position of the child node relative to the parent node is known. This advantage can be potentially useful if CXMLParser supports validation. With the schema (i.e. DTD or XML Schema), direct access is possible via tag names without going through each children node.

CONCLUSION

CXMLParser performs at comparable speed for all of the tested operations. CXMLParser is not a finished product; our intention is to prove that our technique for representing the XML document is time efficient. We have provided the proof of concept. The outcome of the performance testing proves that the compact representation of the terms, as first conceived by Warren, can be improvised to represent the XML document while preserving the efficiency.

ACKNOWLEDGEMENTS

Financial support from the Natural Sciences and Engineering Research Council of Canada is gratefully acknowledged. Also thanks to newheights for the hardware support.

REFERENCES

[1] Knuth, D. E. The art of computer programming: fundamental algorithms. Addison-Wesley, Reading, MA (1997).

[2] Headington, M. R., and Riley, D. D. Data Abstraction and Structures using C++. D. C. Heath and Company, Lexington, MA (1994).

[3] Aït-Kaci, Warren's Abstract Machine: A tutorial reconstruction. MIT Press, Cambride, MA (1991).

[4] Howe, D. Free Online Dictionary of Computing [Online]. Available: http://foldoc.doc .ic.ac.uk /foldoc/index.html (1993).

[5] Apache XML Project. Xerces C++ [Online]. Available: http://xml.apache.org/xerces-c/index.html (1999).

[6] IBM alphaWorks. XML Parser for Java [Online]. Available: http://www.alphaworks. ibm.com/ (1999).

[7] World Wide Web Consortium (W3C). XML version 1.0: W3C Recommendation [Online]. Available: http://www.w3.org /TR/REC-xml (1998).

[8] Yeow, K. W. H. XML Processor: Design, Implementation and Contemporaries. Internal report, UVIC (2001).