UNIVERSITY OF LEICESTER

XXML

Handling Extra-Large XML Documents

Andreas Poyias, LGIP Intern

2/1/2013

This White Paper introduces the SiXML project, developed by O'Neil Delpratt, Stelios Joannou, Naila Rahman and Rajeev Raman at the University of Leicester. SiXML uses the novel technology of succinct data structures to process unaltered XML documents in main memory with a very low memory footprint. SiXML greatly improves the scalability of XML processing, both In terms of volume of data that can be handled and processing time.

1. Introduction

1.1. XML

Extensible Markup Language (XML) is a text-based format markup language, used for storage, transmission and manipulation of data. XML uses meta-data (tags) to demarcate and give meaning to unstructured textual data. The addition of these meta-data to the unstructured textual data causes the file size to increase heavily, a phenomenon referred to as *XML bloat*.

1.2. DOM

XML documents are often held in main memory (RAM) and processed via standard interfaces such as the W3C standard Document Object Model (DOM). DOM represents an XML document as a tree structure (the *Node-Tree*) in RAM, and allows navigation, access and modification operations to be performed on the document. The RAM usage of normal DOM implementations (e.g. Apache Xerces, the reference implementation of DOM) is usually an order of magnitude greater than the (already bloated) XML document (see Figure 1). This greatly affects the scalability of XML processing software. If the memory needed to represent an XML document is greater than the physical RAM on the machine, then the document may not load, or a substantial part may be stored on *Virtual* Memory (VM), which is located on slow storage media such as magnetic or solid-state disk, making XML processing much slower. In other cases VM is either limited (e.g. in Java VM is at most 2GB), or not available at all (e.g. the Android operating system).

The SiXML project offers a solution to this problem by providing libraries for XML developers to be able to rapidly process XML documents in-memory using a very low memory footprint.

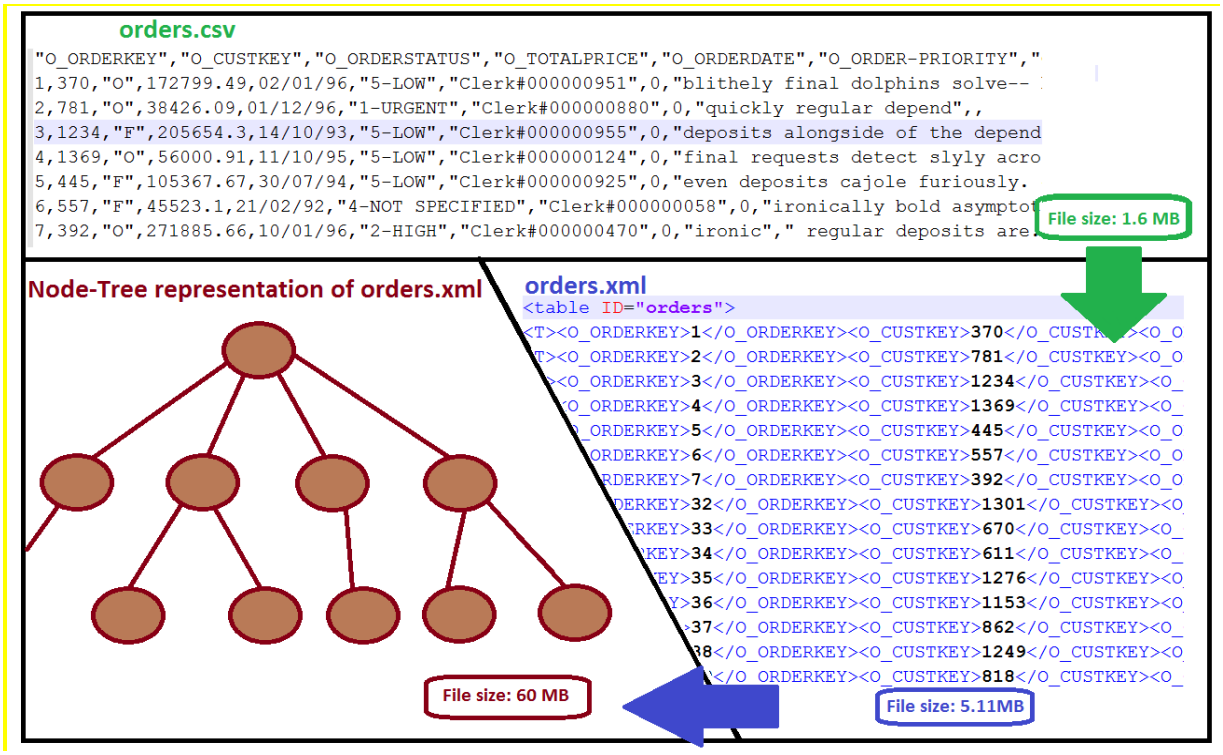

Figure 1 - XML Bloat

2. Details of Implementation/Functionalities

2.1. DOM API

Programmers that use the DOM API can build XML documents, navigate in any direction in the node-tree (e.g. move from a node to its first child, its siblings or its parent), add, delete or modify elements and content in an xml document. They can filter specific types while traversing a document (Element nodes, Text nodes, Attributes). Therefore, DOM API provides users the freedom to manipulate a document in any way they want as long as their machine has enough memory to load the document.

One reason why DOM representations of XML documents use a lot of RAM is the use of pointers. For example, the Xerces DOM implementation uses 5 pointers per node solely for representing the tree structure (and many other pointers for attributes etc.). On a 64-bit machine each pointer costs 8 bytes, giving a total of 40 bytes (320 bits) per node just for tree representation.

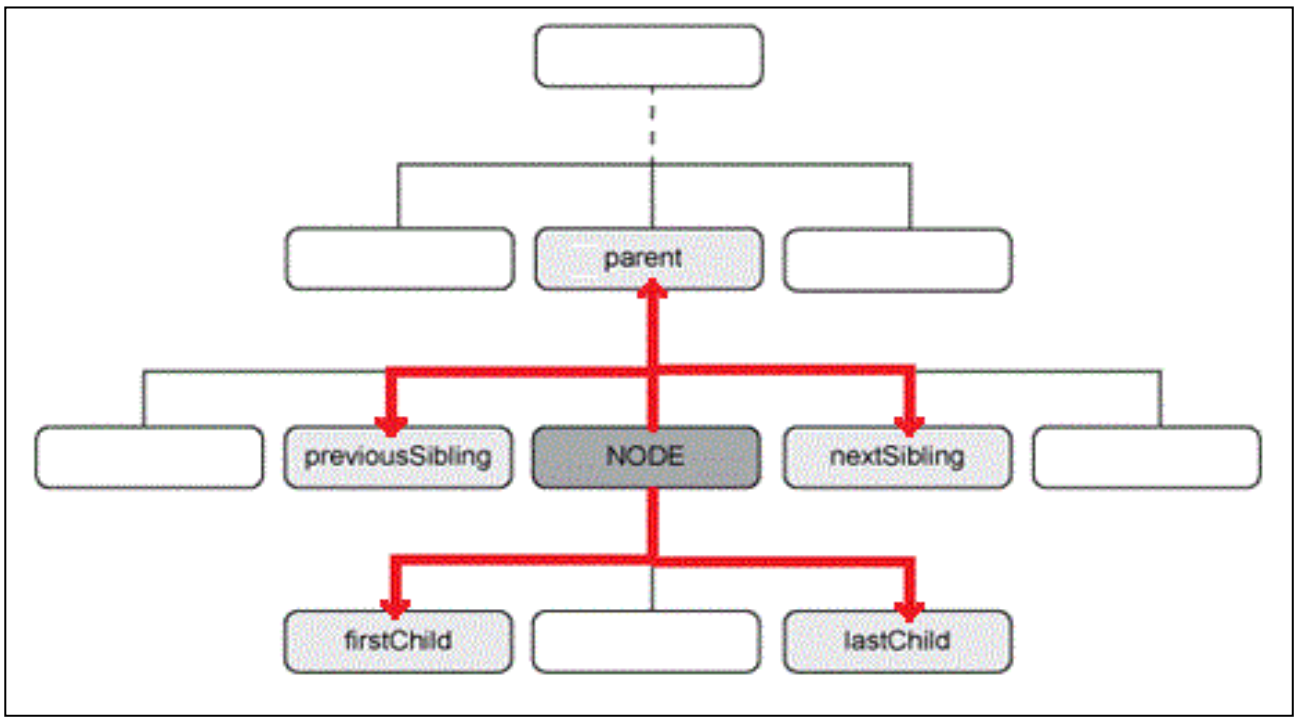

Figure 2 - Xerces Node Tree representation

2.2. SiXDOM principles

SiXDOM is an implementation of DOM based on *succinct data structures*, which are highly space-efficient data structures that support operations rapidly, and were pioneered by Jacobson [1]. Succinct data structures use novel ideas to store data in RAM in close to the information-theoretic minimum, while permitting a wide range of operations on the data. According to information theory, a DOM node-tree can be represented in just 2 *bits* per node, as opposed to the 40 bytes used by standard implementations.

For example, the node tree is represented as a balanced parenthesis sequence, as displayed in Figure 3. Each coloured tag is represented by the corresponding node in the tree. Each opening tag is an opening parenthesis and each closing tag is a closing parenthesis represented by 1 bit each, thus using 2 bits per node. However, navigating a tree with hundreds of millions of nodes using the parenthesis string would be very slow. Based upon several years of research into

succinct data structures, we store additional information to guide navigation among the parentheses. The resulting tree representation requires less than 3 bits per node, and supports navigation at speeds *comparable to a pointer-based representation*. SiXDOM uses similar "pointer-less" succinct data structures to represent all the other information in an XML document.

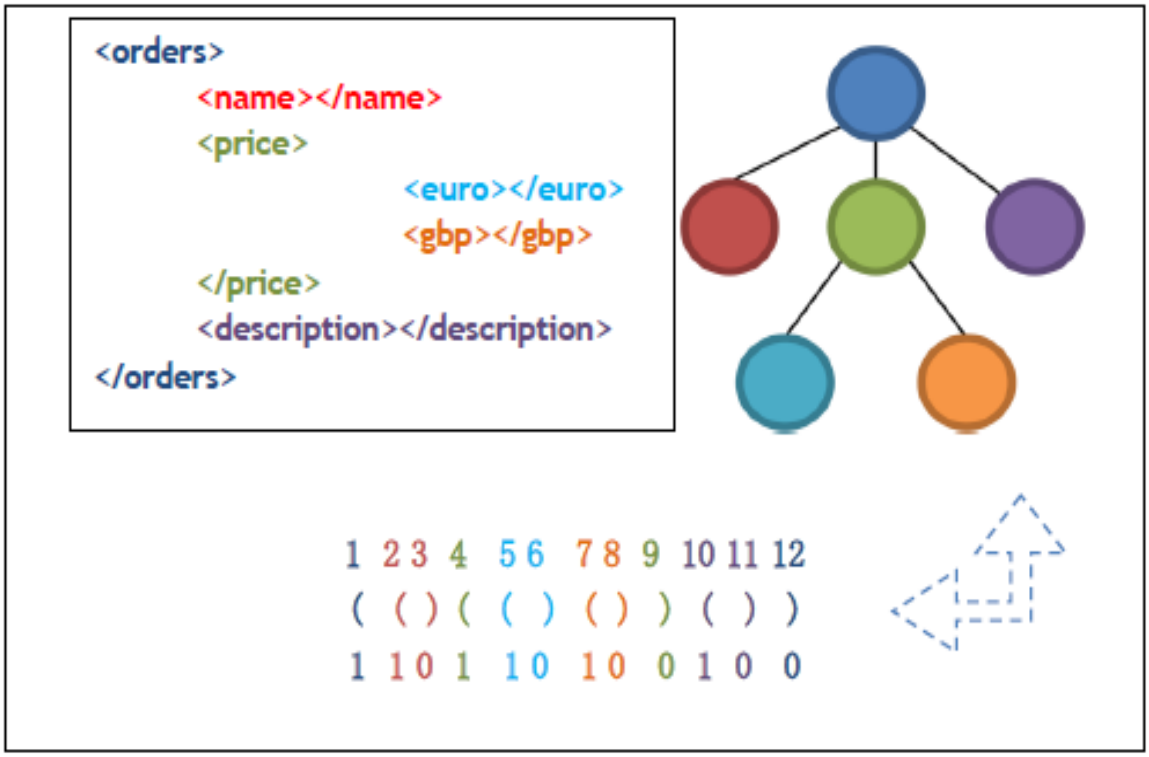

Figure 3 - SiXDOM Node Tree representation

2.3. SiXDOM status

The latest version is SiXDOM v1.2, which is a native C++ implementation of DOM Core Level 2 and a partial implementation of Level 3. In essence SiXDOM 1.2 supports all DOM methods except those that modify the document like add node, delete node or modify a document in memory. SiXDOM implementation is available to be used in Java and Python as well with the use of wrapper classes. SiXDOM libraries are available for download from http://www.cs.le.ac.uk/SiXML. In Sections 3 and 4 we discuss the performance of SiXDOM, and show that it outperforms Xerces by an order of magnitude in both memory and speed for largish XML documents.

2.4. Node vs Treewalker Interface

The SiXDOM implementation is able to use both the Node and the Treewalker interface while traversing. Our recommendation is the use of the Treewalker interface. A key difference between SiXDOM and (say) Xerces is that in SiXDOM, the Node objects in the node-tree are created on demand. Thus, traversing using the Node interface results in the creation of Node objects, which are garbage collected only at the time of deletion of the SiXDOM document collection. This could lead to memory leaks, if the Node objects created as a by-product of traversal are not in fact needed on a long-term basis. These Node objects can take up a lot of memory (see Section 4.4 below). However, navigating an XML document in SiXDOM through Treewalker does not create any new objects during traversal and avoids this kind of memory leak. The experiments that we discuss in Section 3 will be using the Treewalker interface.

SOURCE CODE EXAMPLE using TreeWalker and Node Interface

```
void TreeWalkerInterfaceCode(){
  int check = 0;
  while (current != NULL){
    if(check < 2){
     if(current -> getNodeType() == ELEMENT){
       myElemCount++;
     }else{
       if (current -> getNodeType() == TEXT)
         myTextCount++;
     }
       current = myWalker -> getFirstChild();
     if(current != NULL){
       check = 0;
     }else{
       current = myWalker -> getNextSibling();
       if(current != NULL){
        check = 1;
       }else{
        current = myWalker -> getParentNode();
        check = 2;
       }
    }
   }else{
     current = myWalker -> getNextSibling();
     if(current != NULL){
        check = 1;
     }else{
        current = myWalker -> getParentNode();
        check = 2;
     }
   }
  }
}
```

```
void NodeInterfaceCode(){
  myNode = rootNode;
  SDOM_Node* current = myNode;
  int check = 0;
  while (myNode != NULL){
   if(check<2){
    if(myNode -> getNodeType() == ELEMENT){
      myElemCount++;
    }else{
      if(myNode -> getNodeType() == TEXT){
        myTextCount++;
      }
    }
    current=myNode->getFirstChild();
    if(current != NULL){
      myNode = current;
      check = 0;
    }else{
      current = myNode -> getNextSibling();
      if(current != NULL){
        myNode = current;
        check = 1;
       }else{
         myNode = myNode -> getParentNode();
         check = 2;
       }
      }
   }else{
     current = myNode -> getNextSibling();
     if(current != NULL){
       myNode = current;
       check = 1;
      }else{
       myNode = myNode -> getParentNode();
       check = 2;
      }
    }
  }
}
```

Figure 4 - Source code snippet for non-recursive document-order traversal, counting the number of element and text nodes, using the Treewalker interface (recommended, left) and Node interface (right). In the Node interface, the highlighted lines result in new Node objects being allocated.

3. Experimental Evaluation

3.1. Experimental setup

The experiments were carried out on 64-bit Linux machine of 8GB RAM. We parsed different XML documents of different properties in memory taken from [2] and traversed them. At the same time, we measured the memory required for each XML document, the parse time and the traverse time. In Table 1, the file properties of the different benchmark XML documents are displayed. The different characteristics of each file affect in their own way the memory usage or runtime of the implementations that took place.

xmlFile	File size (MB)	Node Count	Text Space Usage
mondial-3.0.xml	1.7	57372	0.38MB
treebank_e.xml	82	7312612	57.38MB
SwissProt.xml	110	8409225	35.39MB
proteins.xml	683	58446013	322.11MB
dblp.xml	840	64414065	396.43MB
factor9.6.xml	1073	45139212	746.66MB
factor19.2.xml	2150	90311953	1494.06MB
factor38.4.xml	4300	180569240	2986.16MB

Table 1 - File Properties

We used "proc/self/stat" file to fetch the Virtual and Resident Memory used during the execution of the experiments. We were also able to measure the wall clock time to get the performance in terms of parse and traverse speed. Using the values taken we were able to compare SiXDOM with Xerces-C++ version 2.8.0 which is the reference implementation of DOM API. These experiments were also carried out in Java and Python comparing SiXDOM with “Xerces2 Java version 2.11.0” and “miniDOM” respectively. Java implementation was firstly compiled in bytecode to be executed.

4. SiXDOM Vs Xerces-C++

4.1. Memory Usage

We were able to parse all XML documents shown in Table 1 by using the SiXDOM implementation. The VM usage was on average 100% of the file size. The only XML file that exceeds the 100% range is "mondial-3.0.xml" which is our smallest file in size (1.7MB). This is because in SiXDOM there is a fixed memory use cost to pay (3-4 MB). This fixed cost is *per SiXDOM document collection instance*, and is negligible for large documents, or if processing a large number of small documents (which should cover most use cases of SiXDOM!).

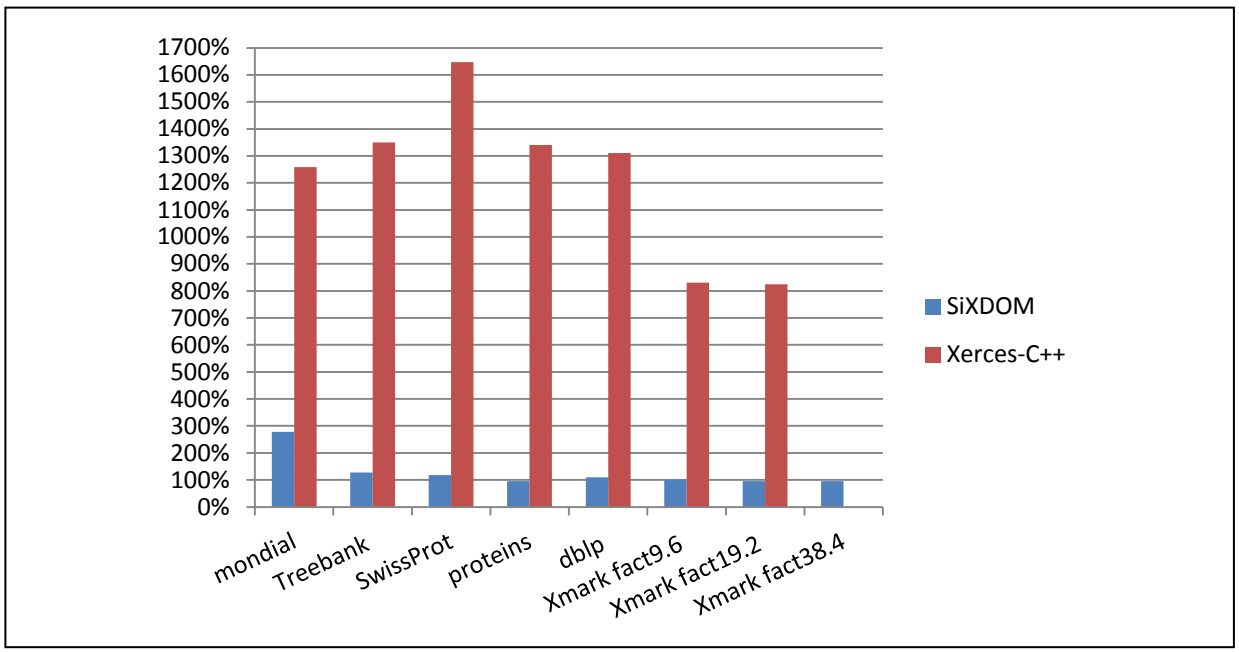

Figure 5 - SiXDOM Vs Xerces-C++ VM usage given as percentage of file size

Xerces on the other hand was not able to parse the last xml document "factor38.4.xml" of file size 4GB. The memory usage was on average 1200% (without the XMark documents 1400%) of the file size including smaller documents like "mondal-3.0.xml". Xmark documents are text-centric (Table 1) and contain relatively little markup. Since SiXDOM 1.2 does not compress textual data, it gains no advantage over Xerces for textual data. As the proportion of textual data increases, the space advantage of SiXDOM will decrease. These properties helped Xerces-C++ performance in terms of memory usage dropping the percentage from an average of 1400% to 1200% since the two Xmark documents that were achieved to be parsed required only 830% of the file size.

4.2 Traverse time

After we parsed the XML documents we traversed them in a document order traversal. As previously stated we measured the elapsed time of this procedure for both implementations. Although the Xerces implementation is about twice as fast as SiXDOM for documents of size up to 110MB, the absolute time difference for such small documents is minimal being less than 1 second. For bigger XML documents there is a huge difference in the elapsed time required for the traversal where SiXDOM is 900%-4300% faster (Figure 7). The difference is due to the memory usage.

For XML documents of size 680MB or larger Xerces-C++ used all the physical memory available (8GB) and at the same time huge amount of virtual memory (Figure 6). As the physical RAM was not enough for the in-memory representation of the XML documents, this affected badly the traverse speed of the Xerces-C++ execution.

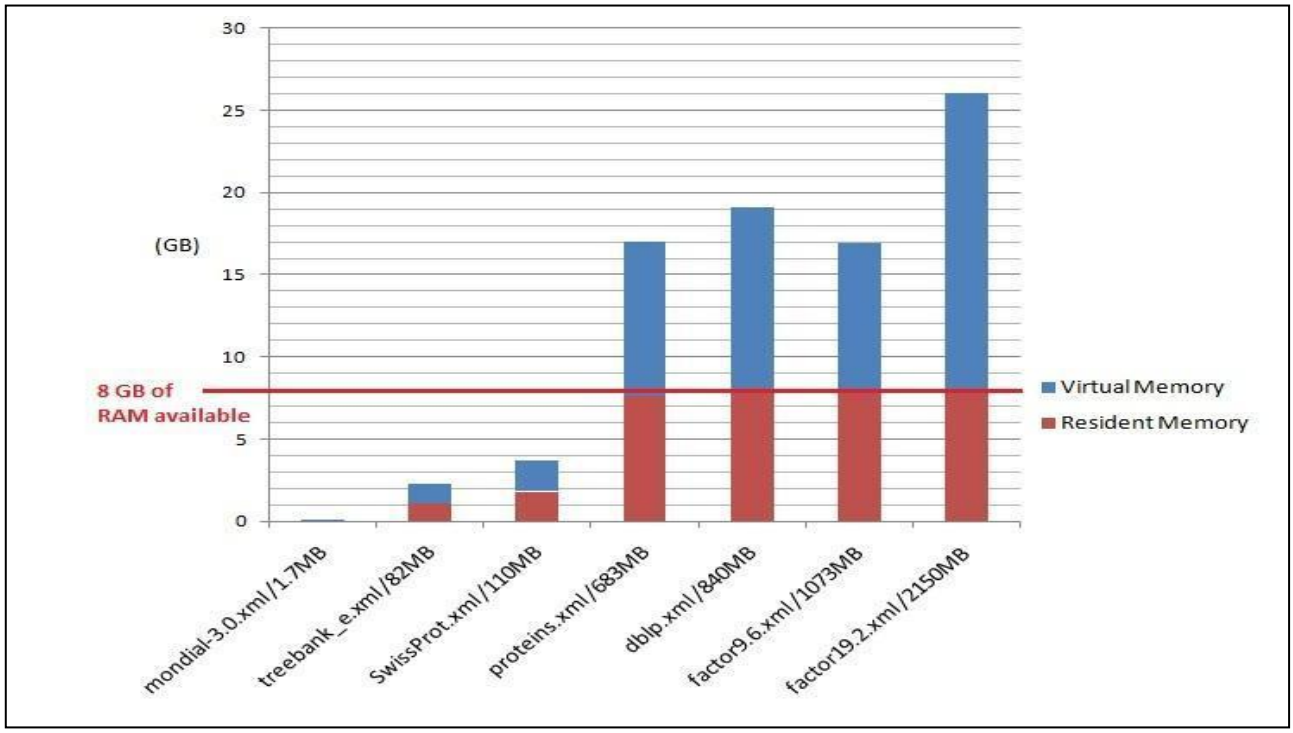

Figure 6 - Xerces-C++ RES and VM usage.

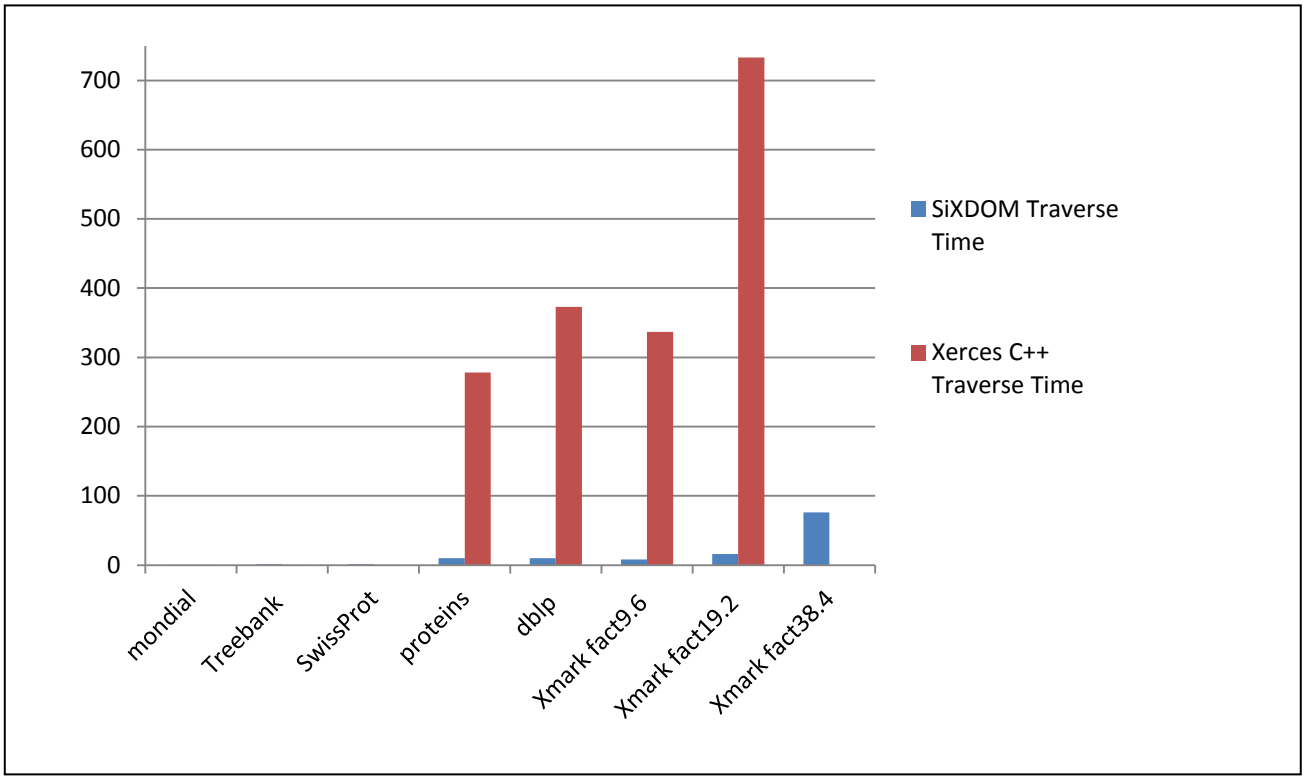

Figure 7 - Xerces Vs Xerces-C++ Traverse speed (Wall Clock Time, seconds)

4.3. Parse time

For the parse time we have a similar scenario like traverse time. Xerces-C++ is 100% faster for small files. SiXDOM performance was much better when it was parsing big files like proteins or bigger. The smaller the document the less time it requires to be parsed. Therefore, the impact on a slow parse for a small file like treebank_e would be minimal making the difference between SiXDOM and Xerces C++ to be less than 3 seconds. On the other hand the impact on bigger files like proteins is much larger; where the SiXDOM is faster for approximately 75 seconds.

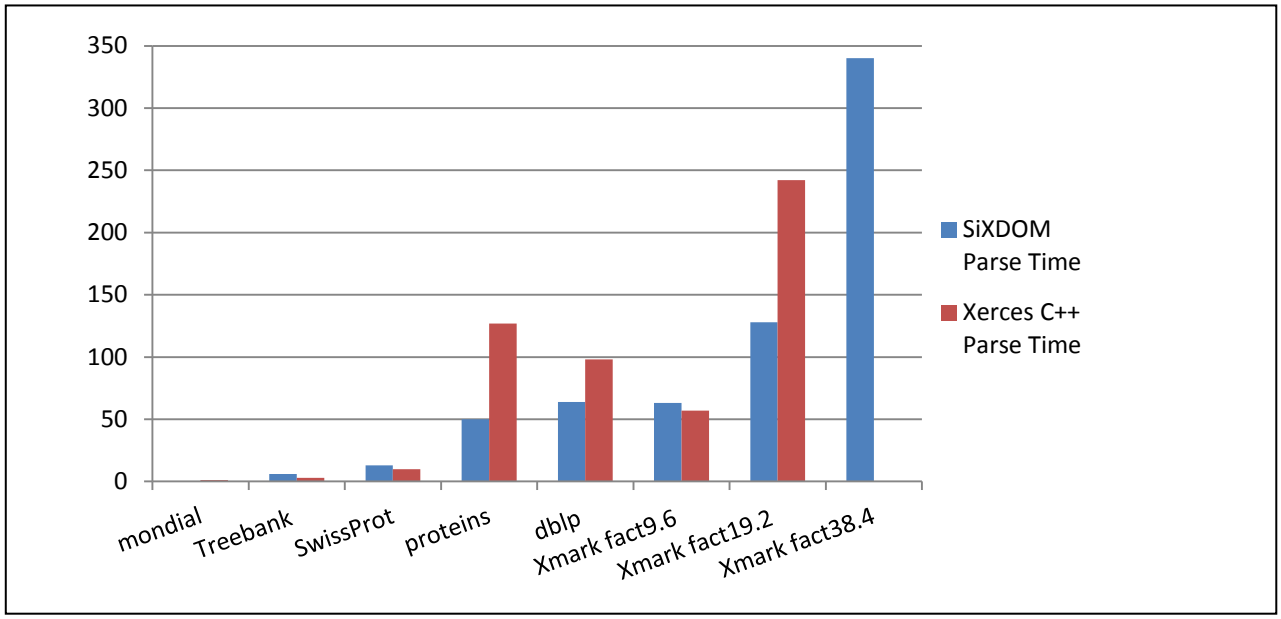

Figure 8 - SiXDOM Vs Xerces-C++ Parse speed (Wall Clock Time, seconds)

4.4 Using Node vs. Treewalker

As mentioned in Section 2.4, we should avoid using the Node interface to perform traversals when using SiXDOM. Because SiXDOM creates Node objects on demand, performing a traversal of the tree using the Node interface dynamically creates (during traversal) all the Node objects encountered. If these Node objects are not required, this is a memory leak, and since SiXDOM has a very low memory footprint, these unwanted Node objects affect the memory usage greatly as can be seen from Table 2. On the other hand, memory usage is stable pre-and-post traverse using the TreeWalker interface.

	mondial	Treebank-e	SwissProt	proteins	dblp	factor9.6	factor19.2
Pre-Traverse(MB)	8.7	106	130	636	840	1073	2150
Treewalker Post-Traverse (MB)	8.7	106	130	636	840	1073	2150
Node Post-Traverse (MB)	11	539	630	4228	4705	3651	7299

Table 2 - Treewalker Vs. Node Interface.

5. SiXDOM Vs. Xerces2 Java

5.1. Memory Usage

SiXDOM successfully parsed all the documents at a ratio of 110% of file size in memory. This ratio is very close to the C++ implementation. This is happening since we use Java wrapper class to bind Java interface with the C++ implementation. The Xerces2 Java implementation was able to parse only the three smallest files used for our experiments. Xerces2 Java is fully a Java implementation and this limits it to the 2 GB of VM whereas we are able to use all RAM available. The biggest file that Xerces2 Java was able to parse is Swissprot.xml (109MB) where it required 1.1 GB of resident memory.

	mondial	Treebank	SwissProt	proteins	dblp	factor9.6	factor19.2	factor38.4
File size (MB)	1.7	82	110	683	840	1073	2150	4300
SiXDOM	300%	134%	131%	96%	112%	104%	103%	105%
Xerces2 Java	1200%	814%	1033%	FAILED	FAILED	FAILED	FAILED	FAILED

Table 3 - SiXDOM Vs Xerces2 Java RES memory usage given as percentage of file size

5.2. Traverse time

SiXDOM java traverse time was the same with the SiXDOM C++ implementation. Xerces2 Java was 700-1000% slower. One reason is that SiXDOM has ready compiled classes to run with the Java interface whereas Xerces2 Java is a complete Java implementations therefore is interpreted. SiXDOM required 1 second for "treecank_e" an SwissProt" but Xerces neede 7 and 10 seconds respectively.

5.3. Parse time

Xerces2 Java parse time was much slower than SiXDOM. It wasted huge amount of time because of Garbage Collections (GC). It was 400% slower for “mondial-1.0.xml”, it required more than 150 seconds to parse “Swisprot.xml” and “treebank_e.xml” making it more than 5500% slower.

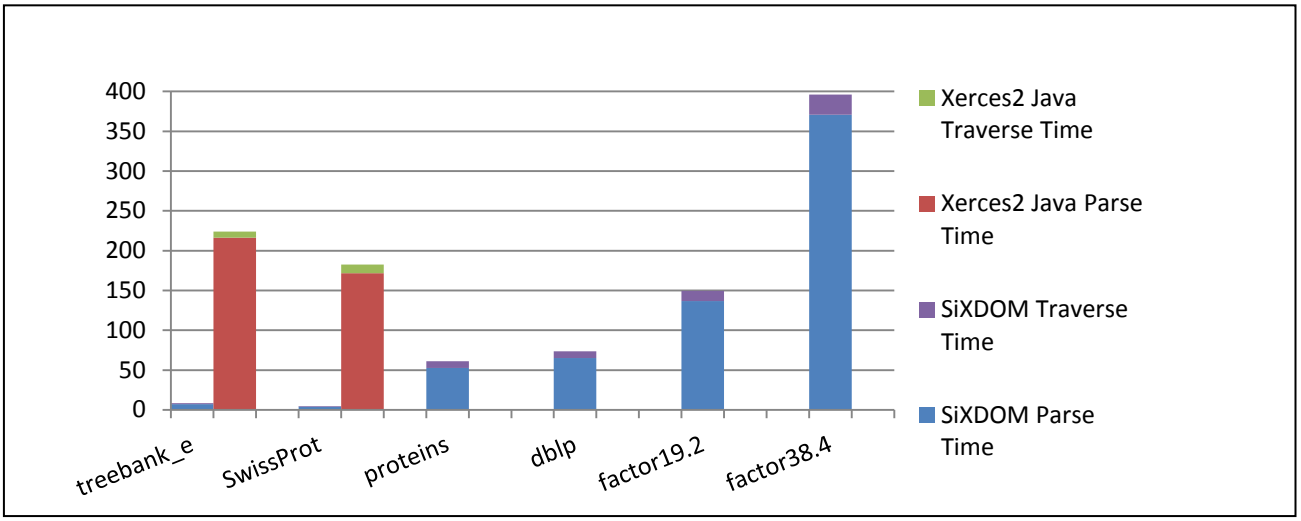

Figure 9 - SiXDOM (Java)Vs Xerces2 Java Parse and Traverse speed (Wall Clock Time, secs)

6. SiXDOM Vs. miniDOM Python

6.1. Memory Usage

SiXDOM was able to parse all the xml files available using up to 120% of file size in memory. This ratio is close to C++ implementation because of the Python wrapper class as explained previously with Java. miniDOM was able to parse only the two smallest XML douments. The biggest document that miniDOM was able to parse was “Treebank-e.xml” (82 MB) and it required 7.3GB of resident memory raising the ratio up to 8900% of the file size.

6.2. Traverse time

SiXDOM python was 100% faster than miniDOM for the two files that miniDOM was able to parse.

6.3. Parse time

SiXDOM parse time was much better than miniDOM. It required for “mondial-3.0.xml” and “treebank_e.xml”, 0.2 and 7.1 seconds to parse, whereas miniDOM required 1.5 and 100 seconds respectively.

References

[1] Jacobson, G. 1989. Space-efficient static trees and graphs. In Proc. of the 30th Annual Symposium on the Foundations of Computer Science (NC, USA, 10/30/1989-11/01/1989). FOCS ‘89. IEEE Computer Society, Washington, DC, vol. 225, Cat. No. 89CH2808-4, pp. 549-554, 1989.DOI=10.1109/SFCS.1989.63533

[2] University of Washington. XML Data Repository.
http://www.cs.washington.edu/research/xmldatasets/www/repository.html ,1998