

# XVM – A Hybrid Sequential-Query Virtual Machine for Processing XML Languages

Anguel Novoselsky, Zhen Hua Liu

Oracle Corporation

400, Oracle Parkway

Redwood Shores, CA 94065

USA

{anguel.novoselsky, zhen.liu}@oracle.com

## ABSTRACT

As XML establishes itself as a universal data model for data exchange and presentation, a number of high-level declarative languages, such as XPath, XQuery, XQueryP, XSLT, have been proposed to effectively query, transform and manipulate the XML data. There have been considerable efforts, both in academic and industry communities of providing optimization and efficient implementation of these languages. However, majority of such efforts have been directed on optimizing these languages as database query languages using iterator streaming based execution model with XML indexing techniques. While iterator based model is effective for searching and pipelining large XML data, it may not be efficient for processing XML data with procedural oriented constructs, such as modules and user defined functions in XQuery, variable assignments, sequential execution expressions in XQueryP and template matching constructs in XSLT. In this paper, we present an XML Virtual Machine (XVM) that is able to handle a family of XML languages like XQuery, XPath, XQueryP and XSLT by using a sequential processing model implemented as a classical stack based virtual machine (XVM). Furthermore, we show how the XVM sequential processing model can be integrated with the iterator query processing model through the notion of *iterator sequence* and *query fragment push-down* so that the best part of both models can be embraced for the purpose of building an efficient XML application environment.

## 1. Introduction

As XML has become the flexible self-describing data model for exchanging, presenting and storing data, there has been several XML based declarative languages from W3C, such as XQuery/XPath [1], XSLT [2] to manipulate XML. Furthermore, recent effort of positioning XML as the universal data model and XQueryP [3][4] as an XML application development language has resulted W3C's effort of extending XQuery with script [5]. Since XQuery was originally incubated from people with strong database background, The processing model of database query languages, such as SQL or OQL, has mainly influenced the design of XQuery language, its compiler optimizer techniques and its execution model. This is why the iterator

based execution model [6] combined with various XML indexing techniques [12][26] was adopted as the most common XML processing model. Many industrial implementations have been focusing on processing XQuery over large XML dataset stored in database using XML indexing and iterator model of evaluation. Typical examples are BEA's streaming mid-tier XQuery engine [9][10] and Oracle XMLDB's native XQuery engine that rewrites XQuery into extended relational algebra with SQL/XML construct [16] in order to fully leverage the iterator based execution of the underlying database engine [7][8]. Furthermore this iterator-based model has also been applied to XSLT processing in the context of database to achieve optimal performance for XSLT processing [11]. While iterator is ideal for searching and pipelining large amount of data, it may not be cost effective when working with full blown XQuery programs with many interleaved XQuery user defined function dealing with moderately sized XML data. This is because the iterator model needs to keep the entire expression execution tree available until the last item in the sequence is computed and fetched by the client. Since the intermediate data sequence from each expression is not fully materialized but rather lazily evaluated, all the intermediate execution states for each expression tree have to be kept active. When user defined XQuery function is evaluated in iterator manner and the return value of the function is a lazily evaluated, this causes the intermediate states for all expression trees used in the function call chain to be kept available. Such an extensive state tracking doesn't scale well especially where there is a long chain of function calls, which is common for XQuery when it is used beyond just as a database query language. For example, XBRL [28] has developed and used XQuery functions and modules extensively to process XBRL XML data.

When XQuery is extended with the notion of sequential expressions (essentially statement concept in imperative languages), blocks and variable assignments in XQueryP [3][4], the iterator model becomes even less attractive for an overall execution model. Instead a hybrid approach of using the classical sequential model as the main execution model, (already widely adopted by imperative languages) while keeping the iterator for the *query* part of expressions becomes more promising.

In this paper, we explore the sequential approach of executing XQuery, XQueryP and XSLT. We have built both an

*XCompiler* and an *XML Virtual Machine (XVM)*. The *XCompiler* compiles XQuery, XqueryP, XSLT into one unified byte-code so that it can be executed in one virtual machine framework. This is in principle the same as Java approach, where Java programs are compiled into byte-code and executed by JVM. The XVM is a classical stack based machine where each virtual instruction pops its operands from the stack and pushes the result back into the stack. Though the virtual machine is based on sequential execution model just as an imperative programming language processor, it has the capability of interacting with external query processors via *iterator-sequence* data objects and the *query-pushdown* compiler technique.

Although both *XCompiler* and *XVM* can run standalone, they can be embedded into different XML platforms, such as XML database servers, application servers and web servers where XQuery, XqueryP or XSLT processing is needed. We provide both compile-time and run-time APIs to facilitate such an integration.

Both, *XCompiler* and *XVM* are designed to be XML storage representation independent. *XVM* works with XML node tree via *extended DOM interface*, where the embedding XML platforms can override the default interface implementation with their own efficient interface implementation on top of their platform specific XML physical representation. Furthermore, the embedding XML platforms can couple their XQuery/XPath compilers with *XCompiler* by providing an *external query-optimizer* and integrate their XML processors with *XVM* via an *external iterator-executor* interface.

In summary, the main contributions of this paper are:

- We show that XQuery, XqueryP and XSLT can be compiled into common byte-code so that a virtual machine (*XVM*) can execute all of these XML programming languages. The byte-code is platform independent so that XML applications using XML programming language can be *written once and run everywhere*. This becomes extremely useful in distributed environment, where both XML data and byte-code can be migrated easily.
- We show that the *XCompiler* can integrate with *external query-optimizer* to compile the query program parts into iterator plans so that *XVM* can integrate with the corresponding *external iterator-executors* in order to achieve optimal execution performance.
- We show that *XCompiler* and *XVM* provide rich set of interfaces so that they can work with variety of XML processing platforms.

The rest of the paper is organized as follows: section 2 discusses the design rationale of *XCompiler* and *XVM* with an example of *XVM* byte code and query push down optimizations. Next three sections describe the design and implementation of *XVM* byte code, the *XCompiler* and the *XVM*. Section 6 discusses performance evaluation. Section 7

does related work comparison. Section 8 discusses the challenge and future work. Section 9 concludes the paper.

## 2. Design Rationale of XCompiler and XVM

Traditionally, SQL like database **query languages** deals with huge (potentially infinite) volumes of data. Collecting all of the queried data in one step often requires a lot of memory and processor resources and in cases with infinite input streams is even impossible. This is why most of query processors use *lazy evaluation* (to retrieve only one data item in a step) as a main evaluation paradigm. One popular and quite common way to implement lazy evaluation is with '*iterators*'. Queries are compiled into tree-like structures (called '*execution plan*' in databases) with iterator nodes. Each iterator node implements *open-fetch-close* methods for obtaining the current expression result. Query step execution starts from the root node and spreads down the query tree nodes until a new data item is received in the root node. All intermediate execution states have to be kept until the last data item is consumed.

**Imperative languages** (like C and Java) on the other hand, explicitly describe the computation as a sequence of steps called 'statements'. Since most of imperative languages use procedures or functions, they are also called procedural. The imperative language programs are compiled into a sequence of atomic instructions. At each step only one instruction is executed after the previous instruction execution is finished with all of its result data is collected and stored. Program state is changed as each instruction is executed. Imperative languages are designed for data processing - their data querying capabilities are quite limited.

XQuery, XqueryP and XSLT have both procedural (imperative) and query (declarative) aspects. The imperative constructs are modules, user defined functions, variable assignment, sequential expressions (statement concept), blocks, exception handlings ... etc. The declarative nature of XQuery shows in its composed expressions, in particular the FLOWR clause, which is very much like the SQL one. The straightforward execution strategy for imperative parts of the language is to compile them into a (virtual) machine code and execute it in sequential execution mode. The XQuery declarative constructs can be compiled into an execution plan and executed in iterator execution mode, which delivers a scalable solution for large data sizes and integration with any XML indexing strategies.

In order to embrace both style of execution mode in *XVM*, we provide query push-down interface in the *XCompiler* that allows one to plug in *external query optimizers*. The external query optimizer has a detailed knowledge of the underlying XML data and its potential indexing so it decides what part of query fragment shall be compiled into an optimal iterator plan. The corresponding external iterator executor invoked by the *XVM* as an external function performs the iterator plan execution. Typical external optimizer starts from the bottom part of the intermediate expression tree generated by the *XCompiler*, walks the tree up and tries to identify the declarative part according to the property of the underlying XML data. Once the declarative part is determined, the

external query optimizer compiles it into a query plan that is to be executed in iterator fashion at run time. The XCompiler simply generates a *iterator-executer-function-call* instruction that causes XVM to invoke the external iterator executer.

The integration and interaction between the sequential model and iterator model is realized through iterator data sequences during run-time. The iterator data sequences are “virtual”-they contain the open-fetch-close function pointers and a pointer to an opaque iterator state object. The external iterator executer “knows” how to fetch data item by item by using and maintaining the state object passed in as a function parameter every time open or fetch or close functions is called. The real iterator execution of the child expression may not be realized until the parent expression executed by XVM needs the data from the child expression.

Table 2 shows an example of XVM byte code (discussed in section 3) generated for a simple XQuery that finds all employees with a family “Smith”. The XQuery is shown in Table 1. It expects an XML node passing as external bound variable “\$db”. The byte code generated by XCompiler (discussed in section 4) shown in Table 2 is based on sequential execution model without invoking any external query optimizers.

```
declare variable $db node() external;
for $i in $db/employee
where $i/lastname = "Smith"
return
  <person> {$i/firstname, $i/lastname} </person>
```

**Table 1 - XQuery Example**

Since there is no external query optimizer involved, the XVM interfaces the XML node via extended DOM interface (discussed in section 5.4) at run-time. For example, the execution of the instruction ‘CHILD employee’ invokes getChild() DOM method to match ‘employee’ tag. The execution of the instruction ‘ELEMLIT’ invokes DOM element node construction method.

```
1. declare variable $db node() external;

5  LOAD      mem: @G[6]

2. for $i in $db/employee

9  CHILD     employee
12 PUSHCTX
13 FOREACH   code: @-30

3. where $i/lastname = "Smith"

15 PUSHCTXVAR 0
17 CHILD      lastname
20 EQIMM      'Smith'
22 BNO        code: @9
```

```
4. return
5. <person> {$i/firstname, $i/lastname} </person>

24 ELEMLIT    'person' 'person' "
28 PUSHCTXVAR 0
30 CHILD      firstname
33 PUSHCTXVAR 0
35 CHILD      lastname
38 UNION
39 ATTACH
40 ENDELEM
41 BRA        code: @28
43 END
```

**Table 2 – Example of XVM Byte Code without query push down optimization**

Table 3, on the other hand, shows the byte-code with XML index specific optimizations applied. The XCompiler invokes external query optimizer. The external query optimizer finds that the underlying external XML has an index for employee names and as a result the compiler replaces the loop initialization and the filter with an external function call. During execution time, the *iterator-executer-function-call* instruction invokes the external iterator-executer, which returns an iterator sequence data object. The iterator sequence objects provide an “open-fetch-close” interface for fetching all employees with name “Smith”. The FOREACH instruction loops through the iterator sequence by calling the iterator fetch method until “null” is returned.

```
1. declare variable $db node() external.;
2. for $i in $db/employee
3. where $i/lastname = "Smith"

4

5  ITERATORFUNCALL
13 PUSHCTX
14 FOREACH      code: @-21

4. return
5. <person> {$i/firstname, $i/lastname} </person>

16 ELEMLIT    'person' 'person' "
20 PUSHCTXVAR 0
22 CHILD      firstname
25 PUSHCTXVAR 0
27 CHILD      lastname
30 UNION
31 ATTACH
32 ENDELEM
33 BRA        code: @19
35 END
```

**Table 3 – Example of XVM byte code with query push down optimization**

Having studied the example of XVM byte code, we now illustrate the XVM byte code in details.

### 3. XVM Byte Code

XVM byte-code is a platform independent sequence of two byte units. It has a header and a body. The header contains byte-code description XQuery or XSLT or XPath version, lengths and offsets to each body section.

The body contains sections for the byte-code itself (which are referred as XVM instructions), strings, numbers (as strings), string-tables, types, patterns, pattern tables, external function tables, ... etc. All byte-code references are in the form of relative offsets (as a number of units) from the beginning of the corresponding section or offsets from the current address.

When the XVM loads the byte-code, it converts all numbers from their string representations into digital ones (decimal, float, double or integer). Numbers reside in tables and XVM instructions refer to them with their table offsets calculated in compile-time. XML Schema pointers to user-defined types are obtained from the schema objects and put in type table exactly like the numbers before. Also, the XVM populates tables with built-in, user-defined and external function addresses. There is one table for each function namespace. Again, XVM instructions refer to functions with their table offsets prepared in compile-time. The XVM approach for dynamically linking external entities is quite similar to DLL dynamic linkage in Windows.

XVM instructions are classified into the following groups based on their tasks:

- **XPath step** instructions;  
Execution of these instructions calls the proper node navigation methods in the extended DOM interface.
- **XML nodes creation** instructions;  
Execution of these instructions calls the proper node construction methods in the extended DOM interface.
- **Arithmetic and Comparison** instructions;  
By default, these instructions are type polymorphic, that is, they do arithmetic and comparison based on the type of the operands. However, the XCompiler can generate non-polymorphic instructions when XCompiler can determine the types of these operands via static type analysis.
- **Data transfer** (load, store, push, pop) instructions;  
These instructions move XVM sequence objects among XVM stack, context stack.
- **Type checking & type conversion** instructions;  
These instructions implements XQuery run time type checking and value casting.
- **Control transfer** (branch) instructions;
- **Loop** (for-each) instruction;
- **Match** instructions;  
These instructions are for XSLT template matching
- **Function or Template invocation** instructions;  
These instructions call XQuery functions or XSLT templates. Both built-in XQuery functions and operators and user-defined functions are invoked this way.

- **iterator-executer-function-call** instruction  
This instruction invokes the external iterator-executer.

Having looked at the bye code, we now show how XCompiler generates the byte code.

### 4. XCompiler

XVM compilation consists of four processing phases:

- Parsing and semantic analysis.
- Platform-independent optimizations.
- Platform-specific analysis and optimizations.
- Code generation.

Figure 1 shows the multi-phased XCompiler.

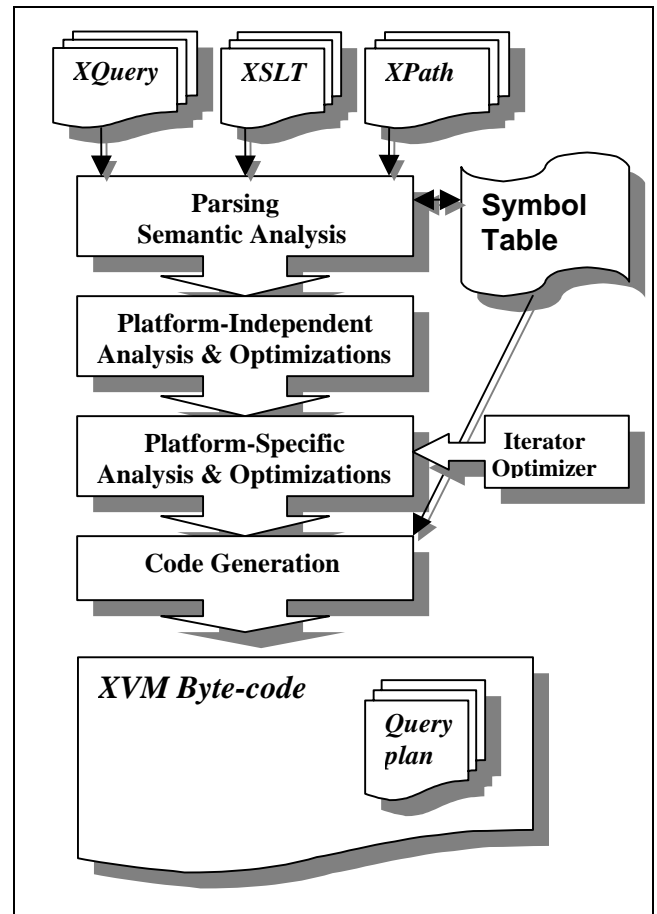


Figure 1 - XCompiler Architecture

During the first phase, the input program is parsed, semantically analyzed and converted into an intermediate language (IL) representation. The IL forms a graph, where graph nodes represent operations and semantic entities, forward arches represent control flow and backward arches stand for function and variable references. All function, variable and other declarations found in the first phase are added to the Compiler's Symbol Table. Both XSLT and

XQuery are statically scoped languages, which means that their inner declarations hide the outer ones. This is why the Symbol Table has a stack-based design in order to push and pop inner scopes of visibility. For example, all XPath built-in function declarations and built-in types are pushed first and stay at the bottom of the Symbol Table stack during the compilation. Both Symbol Table declaration entries and their IL graph counterparts cross-reference each other. The first phase performs most of the type propagation and some constant folding. Since all program variables and function parameters reside in the XVM stacks, the compiler calculates their stack offsets (used for referencing) during the semantic analysis. The top-level variable offsets start from the bottom of the stack. Local function or template variable offsets are calculated from the beginning of the current function or template stack's frame. Also, since both XSLT and XQuery allow forward referencing (some entities can be referenced before they physically appear in the text) all such unresolved references are collected and processed afterwards.

The second compilation phase applies platform independent data-flow analysis and optimizations on the IL graph. They include: loop optimizations; let clause variable references replacement by the variable value; function actual argument and variable assignments type casting (if needed); XPath expression analysis and simplification; ... etc.

By default the compiler converts the whole IL graph into XVM byte-code but for some types of the XML processing it is better to use non-imperative processing strategies provided by the external embedding XML platforms. For example, if the embedding platform is an XML database, then it is better to execute some of XPath/XQuery inside the database through XML indexing and iterator execution strategy. If the XML source is a stream, then it is better to execute a simple XPath expression directly on the stream then using the extended DOM interface. The decision, which IL sub-graphs are to be compiled into byte-code and which sub-graphs are to be compiled as iterators is made in the third compilation phase. The phase includes built-in and external query optimizers. For example built-in optimizers recognize all XPath expressions that can work on input XML streams. External query optimizers are used to find IL parts that are better to be executed inside XML database if the external embedding XML platform is an XML database. In both cases, the identified IL sub-graphs are marked as *iterator-sub-graphs* so an *iterator-executer-function-call* instruction to be generated during the code generation phase. XQueryP has the notion of sequential expression, which is essentially statement in the imperative languages. There are optimization opportunities exist to group and optimize sequential expression as one non-sequential expression or vice versa by converting non-sequential expression into sequential expression.

The fourth phase does the byte-code generation. The code generation process is quite straightforward because all of the important decisions were made by the previous two optimization phases. The only complicated cases are XPath predicates and element constructors. For predicates XCompiler generates LOOP instructions and it should take care to handle properly embedded predicates. When the output is streamed it is not efficient to build intermediate DOM fragments, stream them afterwards and destroy at the

end. This is way the code generator analyses the element constructor context and tries to make the DOM tree built in top-down fashion.

Having discussed compilation process, we now show how the XVM engine works in run time.

## 5. XML Virtual Machine

### 5.1 XQuery Data Model – XVM Sequence Object

As it was already mentioned XVM has a stack-based architecture, each machine instruction pops its input operands from the stack and replaces them with the result. The operands are conceptually the XQuery data model sequences, which are pushed and popped on the **XVM stack** during execution. Although the XVM sequence object is a composite data type, to provide better performance and ease of memory management, XVM has implemented different concrete types for sequence object.

#### Singleton Sequence - item

For a singleton sequence, XVM has the notion of item object (scalar). The item object is essentially the union of all built-in data objects with scalar type (such as xs:decimal, xs:double, xs:string supported by XQuery) and the XML node object. The XML node object is a pointer to a DOM node whose underlying representation is opaque to XVM. All scalar data objects that have fixed size are pushed onto the XVM stack directly without any dynamical memory allocation. However, the content of scalar object with variable size (such as xs:string) is pushed onto a separate **string-stack** and only the pointer to the content resides in the XVM stack as a string object. As string objects are pushed and popped onto the XVM stack, the corresponding string content space is pushed (equivalent to allocate) and popped (equivalent to free) so the allocation and free string operations become relatively inexpensive compared to allocation and free from a general purpose heap.

#### Multi-item Sequence

We divide multi-item sequences, into a homogeneous node sequences consisting of XML nodes only, which we call **node-sets** and heterogeneous sequences consisting of mixture of nodes and items, which we call **item-sets**. The compiler static type analysis can figure out, for which one of the two to generate byte-code in most of the cases. Similar to the design of the string-stack, the actual contents of the node-set sequence and item-set sequence are stored in a **node-stack** and an **item-stack**. Only the pointers to the part of the node stack or item stack reside in the XVM stack. Again node-stack and item-stack grow and shrink logically as sequence object are pushed to and popped off the XVM stack. This way of managing memory for multi-item sequence is much cheaper than doing so with a general-purpose heap.

#### Iterator Sequence

Some sequences do not have to be fully materialized, in particular, when they are computed from an external iterator-executer. We have the notion of iterator sequence whose

content consists of function pointers to ‘open-fetch-close’ methods and a pointer to an opaque state where the external iterator-executer stores its execution state.

## 5.2 Stacks in XVM

XVM uses the following main stacks for execution.

- **XVM-stack** – Conceptually this is the main stack that all programs have. Frames for XQuery functions, XSLT templates are pushed and popped onto the XVM stack. Each frame contains currently computed XVM sequence objects for expression operands, local variables and parameters.
- **Context-stack** – contains XVM sequence objects for loop and context variables. For example, when computing loop expression, each looping sequence has to reside in the context-stack so that the result from the loop execution can be pushed onto the top of the XVM- stack.
- **DOM-stack** – contains currently created node and its predecessors. The stack can contain intermediate fragments of more than one DOM trees.

As discussed in section 5.1, for better performance and memory management, both the XVM-stack and the Context stack use three sub-stacks: item-stack, node-stack and string-stack for holding variable length objects.

The sub-stacks grow and shrink in sync with the main stacks. They are implemented as linked lists of dynamically allocated-on-demand segments.

The XVM stack layout is shown in Figure 2.

## 5.3 Executing instructions in XVM

XVM instructions have the following format:

**opcode [mode] [operand ]\***

Depending on the opcode and mode, the operands are treated as one of the following entities:

- Stack offset – variable or parameter address.
- Constant table offset – for names, string literals, numbers, types, ... etc.
- Byte-code offset – for ‘branch’ or ‘call’ VM instructions.
- Immediate value – usually an integer.

The XVM execution architecture is quite simple. There is a set of functions, one for each instruction, implementing the instruction semantics. The VM main loop moves the instruction pointer over byte-code instructions and calls the corresponding function. The default instruction pointer step is one instruction. Only instructions like ‘branch’ or ‘call’ can change the instruction pointer according to their operand values. Each instruction takes its operands from the VM-stack

and pushes back the result. When a function is called or a template is activated, the corresponding function or template stack frame is pushed into the XVM-stack. The frame contains the return address, current stack pointers, current node, a descriptor address plus (if needed) slots for parameters and local variables.

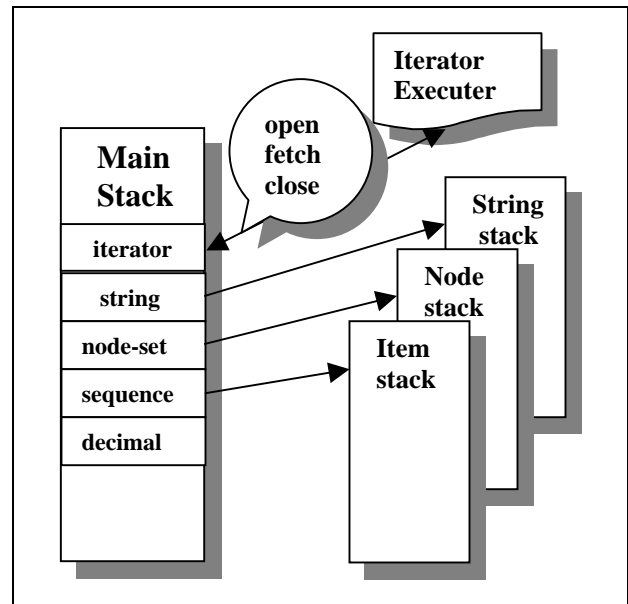


Figure 2 - XVM Stack Layout

The result of execution is a sequence, which XVM provides with an iterator interface APIs to allow embedding XML applications to fetch the result. However, XVM also provides interface to serialize the sequence so that the embedding applications can fetch the result in the form of DOM trees, SAX events or XML text. In such cases of non-sequence output, XVM generates SAX-like events and depending on the current output mode the events are directed to the DOM builder, Streamer or SAX event generator.

## 5.4 Extended DOM interface

XVM is designed to be XML physical representation independent. It works with XML node tree via extended DOM interface. Although standalone XVM has a default implementation of the extended DOM interface, it allows the embedding XML platforms to override the implementation for their specific physical XML node tree presentation. Therefore, XVM holds the DOM node as an opaque pointer. The actual implementation of the DOM interfaces is opaque to the XVM.

XVM path traversals and node constructions instructions are calling corresponding methods in the extended DOM interface. However, since DOM does not handle schema based XML, therefore, we have to extend the DOM interface to add methods that provide PSVI node information. Furthermore, for node order comparison, the extended DOM

also includes methods to compare the order of two DOM nodes. Note DOM is just an API, it does not entail that the implementation of the DOM API has to load the XML document in memory to implement the API that XVM uses. Efficient and scalable DOM implementation exists that do not rely on loading the entire XML document in memory. For example, if an XML document tree is stored on disk, only the nodes that are traversed by the DOM API needs to be bought in memory.

Having discussed all the details of the XCompiler and XVM, we now show performance evaluation of them.

## 6. Performance Evaluation

We use XMark [20] as one evaluation benchmark for our approach. Although XMark is designed as ‘query like’ XQuery instead of procedural oriented XQuery, we use it as currently there is no XQuery and XQueryP benchmark which is designed for testing mixture load of both procedural and declarative query. We expect that future XQuery/XQueryP benchmark should definitely provide such workload with many modules and functions.

These experiments were carried out on a PC with a 3.06 GHz Pentium-4 processor and 2 GB of main memory. The operating system is Linux version 2.6.9-34.0.1.0.11.Elsmp. To show the efficiency of the pure sequential XVM approach, we do the followings when running the benchmark tests to deliberately not to take advantage of any XML indexing techniques, any external iterator based query processors, any lazy evaluation of DOM strategy at all:

- XVM is coded in C and runs standalone, it is not embedded into any XML database server. All XQueries used in XMark are compiled into pure sequential mode by the XCompiler without any external query optimizer that compiles part of the query fragment into iterator model.
- The XML data file used for XMark is directly read from OS file system and is parsed into vanilla DOM tree using standard Oracle XDK C parser [21]. There is NO Oracle XMLDB specific XML indexing or binary encoding of XML [14] applied at all. The XDK C parser we used for experiment builds a vanilla DOM tree without any lazy DOM loading strategy. Table 4 shows the XML parsing time in milliseconds for different XMark XML file size.

XMark XMLFileSize	Parsing Time in millisecond
4MB	155
12MB	480
100MB	3,900
200MB	7,780

Table 4 – XML Parsing Time

### Compile Time:

The XCompiler is able to compile each XMark XQuery within 1 millisecond. Table 5 shows the size of byte code for each XMark XQuery. The byte code is very compact as the maximum size is 2K bytes among all XMark queries.

Query	ByteCode Size
Q1	1240
Q2	1252
Q3	1484
Q4	1424
Q5	1244
Q6	1176
Q7	1248
Q8	1376
Q9	1504
Q10	2056
Q11	1420
Q12	1448
Q13	1268
Q14	1240
Q15	1336
Q16	1428
Q17	1264
Q18	1256
Q19	1317
Q20	1608

Table 5- XMark XQuery Byte Code Size in bytes

### Execution Time:

Table 6 shows query execution time for various XMark XML file sizes: 4MB, 12MB, 100MB, 200MB. Just as BEA XQuery papers [9][10], the shown query execution time does NOT include XML file size parsing time that has been listed in Table 4. Excluding XML parsing time allows us to be able to give an accurate measurement for pure XVM execution time.

We selected 4MB and 12MB because these are the sizes of XMark document that BEA streaming engine [9][10] paper and Saxon [22] site used to report XMark numbers respectively. We also selected 100MB and 200MB as these are the typical sizes showing the effectiveness of XML database approach using XML indexing and iterator based query execution model. Monet DB reports its XMark numbers ranging from 110KB to 11GB at [27]. Compared with these numbers, the execution time for XVM is very competitive excluding XML document parsing time, Even for 200MB size, other than Q10, all other queries finishes within 9 seconds which includes the dominant XML parsing time of 7.78 seconds for 200MB size XML file. The actual query execution time by XVM is only 1.22 seconds. This means if XVM worked on the XML documents that are stored as tree physically on disk, then the XML document parsing time would be eliminated completely. A scalable DOM API implementation can be done on top of the persistent document tree.

For Q11 and Q12 that contain count() function, the sequential model shows its performance advantage because count() merely returns the size of the previously computed result whereas in the iterative model, each item has to be fetched individually via getNext() call and counted.

## 7. Related Works Comparison

The idea of compiling imperative languages into byte codes and having VM execute byte codes is not new. However, processing XQuery, XQueryP, XSLT using VM approach is not very common. In particular, XQuery has been primarily optimised as a database query languages so far. To our knowledge, XSLT VM [13] is the first virtual machine built for processing XSLT. And the new virtual machine we presented in this paper is the first virtual machine that processes all XML languages: XQuery, XQueryP, XSLT in one machine. Paper [15] has shown the approach of compiling XQuery into Java bytecodes so that one can leverage Java virtual machine to execute XQuery. However, the paper does not address the issue of how to combine both iterator and sequential model of execution. Also it is just a compiler generating Java byte codes and rely on Java virtual machine as the execution machine. In our work we have built both the compiler and the virtual machine. We think that virtual machine approach will become common as people develop large scaled XML applications using pure XQuery, XQueryP and XSLT. In these use cases, the main program control flow will be executed in VM fashion and specific query fragments will be executed using various database indexing and iterator model. However, how to demarcate the two efficiently in a challenge as discussed below.

	4MB	12MB	100MB	200MB
Q1	3	3.999	36.995	88.986
Q2	1	2	7.998	13.998
Q3	0	1	10.999	16.997
Q4	0	2.999	8.999	20.997
Q5	1	1	9.999	11.998
Q6	11.998	36.994	312.952	612.907
Q7	25.996	80.998	647.902	1,314.800
Q8	6.999	21.997	173.973	353.946
Q9	5.999	18.997	158.976	309.953
Q10	186.972	1,272.806	73,529.820	304,440.700
Q11	6.999	21.997	180.973	336.949
Q12	6.000	19.997	162.975	329.950
Q13	5.999	16.997	161.975	314.952
Q14	19.997	57.991	483.926	924.859
Q15	1.000	0	4.999	13.998
Q16	0	1.000	4.000	10.998
Q17	5.999	15.998	136.979	215.967
Q18	0.999	0	3.999	12.998
Q19	13.998	42.994	383.942	787.881
Q20	16.998	48.993	417.937	810.877

**Table 6 – XMark Query Execution Time in milliseconds for various size of XMark XML Files using XVM**

## 8. Chalanges and Future Directions

Although the central dogma in database community is declarative query processing, the general data business logic needs both the declarative query logic and the imperative procedural logic integrated into one environment. The *loose integration* is to have two languages, one declarative and one imperative, invoked from each other via API. The typical example is SQL embedded in Java via JDBC. The *middle integration* is to have one language but having explicit demarcation of what is query part and what is procedural part. The typical examples are SQL PSM [17] and Oracle PL/SQL [18][19]. The *deep integration* is to have one language where the demarcation between the two is blurred so that a seamless integration of DBMS logic with conventional programming facilities is all in one language. Back in time when relational database was in active research time, there were proposals to do deep integration of imperative and declarative paradigm by making database objects, such as tables, views and tuples as built-in datatypes of a program language. Paper Pascal/R [24] and RIGEL [23] proposed such ideas. Now in post-relational time, XQuery and XQueryP follow the same principle by positioning XML as a common data model that can be manipulated in one programming language. However, the challenge of intelligently figure out what part should be executed procedurally and what part should be executed declaratively in cost based manner by the language optimizer remains. For example, we have demonstrated that a procedural loop with conditional expression in its loop body can be optimized into a query with “where” condition and thus opens more query optimization strategies [25]. On the other hand, a declarative query might be cost effective to be compiled and executed as a sequential statement. However, teaching optimizer to understand recursive functions and convert them into recursive query may not be a straightforward exercise. Working on this kind of optimization will be in general challenging future work that requires collation from both the language compiler community and the database community.

## 9. Conclusion

In this paper, unlike in the conventional “execution plan” approach, we have shown the sequential model of processing XQuery, XQueryP and XSLT XML programming languages by using a virtual machine approach. We have shown that doing so does not miss the optimization opportunities from the conventional database style of processing XQuery and XQueryP. Furthermore, doing so makes XSLT no longer foreign to XQuery and XQueryP and opens opportunity of integrating XQuery and XSLT into one language. Currently, a piece of application logic typically performs the ‘searching needles in the haystack’ and the other part does ‘transforming found needles into gold’. While the former is best processed by database style of execution using index and iterator model, for the latter is better to be applied the conventional sequential execution model. We see the merits of both and it is the deep integration of the two makes the best implementation strategy of processing XML data. We think as XQueryP becomes widely adopted in the future, compiling and executing XQueryP program using virtual machine

approach will become more common. However, how to optimize the XML languages to take advantage of both processing styles will remain a challenge that requires the cooperation between both database and programming language communities.

## 10. References

- [1] XQuery/XPath: <http://www.w3.org/TR/xquery/>
- [2] XSLT: <http://www.w3.org/TR/xslt>
- [3] D. Chamberlin, M.J. Carey, M. Fernandez, D. Florescu, G. Ghelli, D. Kossmann, J. Robie: XqueryP: AnXML application Development Language  
<http://2006.xmlconference.org/proceedings/38/presentation.pdf>
- [4] D. Chamberlin, M. Carey, D. Florescu, D. Kossmann, J. Robie : XQueryP: Programming with XQuery, XIME-P 2006
- [5] XQuery Scripting Extension:  
<http://www.w3.org/TR/xquery-sx-10-requirements/>
- [6] G. Graefe. Query Evaluation Techniques for Large Databases. ACM Comput. Surv. 25(2):73-170, 1993.
- [7] M. Krishnaprasad, Z. Hua Liu, A. Manikutty, J. Warner, V. Arora, S. Kotsovolos: Query Rewrite for XML in Oracle XML DB,VLDB 2004
- [8] Z. Hua Liu, M. Krishnaprasad, V. Arora: Native XQuery Processing in Oracle XML DB. SIGMOD 2005
- [9] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. Carey, and A. Sundararajan. The BEA streaming XQuery Processor. VLDB Journal, 13(3):294-315, 2004.
- [10] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. Carey, A. Sundararajan, Geeitka Agarwal. The BEA/XQL streaming XQuery Processor. VLDB 2003: 997-1008.
- [11] Z. Hua Liu, A. Novoselsky: Efficient XSLT Processing in Relational Database System. VLDB 2006: 1106 – 1116.
- [12] F. Ozcan, R. Cochrane , H. Pirahesh, J. Kleewein, K. Beyer, V. Josifovski , C. Zhang: System RX: One Part Relational, One Part XML, SIGMDO 2005
- [13] A. Novoselsky, K. Karun: XSLT VM – An XSLT Virtual Machine.  
<http://www.gca.org/papers/xml europe2000/papers/s35-03.html>
- [14] R. Murthy, Z. Hua Liu, M. Krishnaprasad, S. Chandrasekar, A. Tran, E. Sedlar, D. Florescu, S. Kotsovolos, N. Agarwal, V. Arora, V. Krishnamurthy: Towards An Enterprise XML Architecture , SIGMOD 2005
- [15] Per Bothner: Compiling XQuery to Java bytecodes. XIME-P 2004: 31-36.
- [16] I.O. for Standardization (ISO). Information Technology- Database Language SQL-Part 14: XML-Related Specifications (SQL/XML)
- [17] Database language – SQL – Part4: Persistent Stored Modules (SQL/PSM). ANSI/ISO/IEC 9075-4-1999.
- [18] <http://www.oracleplsqlprogramming.com/>
- [19] <http://www.unix.org.ua/oreilly/oracle/prog2/index.htm>
- [20] XMark: <http://www.xml-benchmark.org/>
- [21] Oracle XML Development Kit:  
<http://www.oracle.com/technology/tech/xml/xdkhome.html>
- [22] SAXON XMark Results:  
<http://saxonica.blogharbor.com/blog/archives/2006/11/6/2477675.html>
- [23] L.A. Rowe, K. A. Shoens: Data abstraction, views and updates in RIGEL. In Proceedings of the 1979 ACM SIGMOD, Boston, MA, 1979.
- [24] Joachim W. Schmidt. Some High-Level Language Constructs for Data of Type Relation. ACM Transactions on Database Systems, 247-261, 2(3) 1977
- [25] D. Florescu, Z. Hua Liu, A. Novoselsky: Imperative Programming Languages with Database Optimizers. PLANX-2006:83
- [26] Z. Hua Liu, Muralidhar Krishnaprasad, Hui J. Chang, Vikas Arora: XMLTable Index - An Efficient Way of Indexing and Querying XML Property Data, ICDE 2007
- [27] MonetDB XMark Results:  
<http://monetdb.cwi.nl/XQuery/Benchmark/>
- [28] XBRL: <http://www.xbrl.org/Home/>