XIPE
An XML Integrated Processing Environment

Anguel Novoselsky and Zhen Hua Liu
Oracle Corporation, 400 Oracle Parkway, Redwood Shores, CA 94065, U.S.A.

Keywords: XML, XQuery, XQuery Scripting Extension, XQuery Update Facility, XSLT, XPath, XML DB, Software Engineering.

Abstract: As XML becomes a common flexible data model for data exchange and representation, more and more adoptions of “XML-only” application design paradigm start to appear. This pure XML based approach views XML as a logical data model and uses the high-level XML declarative and imperative programming languages, such as XQuery, XQuery scripting extension, XQuery Update Facility, XQuery Full Text, XPath, XSLT, as the primary languages for application development. The “XML only” paradigm has its merits because it promotes the opportunities of global cross-tier optimisations and eliminates the impedance mismatches between different data models and different programming language styles that exist in the alternative “multi-language” approach. In this paper, we present a “pure” XML Integrated Processing Environment (XIPE) built around an XML Virtual Machine (XVM) and XML Data Repository (XDR). We present the XIPE key components - XCompiler, XVM, XQDOM, XML Tree Index, … etc, that are needed to build such a programming environment. We also show the design rationale and principles we apply to build each one of the components. The goal is to make the XIPE itself open, flexible and scalable. In order to achieve that, we use an interface-based component interaction model and use the so called “light” and “heavy” data repositories, which helps XIPE to scale seamlessly with different sizes, shapes and characteristics of the underlying XML.

1 INTRODUCTION

XML and its associated technologies have been widely used for application development. With the emergence of high-level XML languages, such as XQuery/XPath (Fernandez, 2007), XSLT (Kay, 2007) and with the help of the recent XQuery extensions XQuery/UF (Chamberlin, Florescu, 2008), XQuery/FT (Amer-Yahia, 2008), XQuery/SE (Chamberlin, Engovatov, 2008), the number of “pure XML” applications started to grow. The “pure XML” approach is characterized with using XML as a single data model and using an XML high-level language for application development.

The design of XSLT, XPath and especially XQuery was heavily influenced by SQL and this is why they are considered as declarative languages. Declarative languages let the users to specify what they want instead of how to do it. The XML languages can be used and processed as queries but unlike SQL they are not restricted to be used in SQL like way only. Therefore, there have been considerable efforts from programming language and database communities to find an efficient processor implementation for them. XQuery scripting extension (SE) further adds imperative procedural constructs, such as statements and assignments, which from one hand improves the language expressiveness but on the other hand adds more problems to processors designers.

XML data can range from highly structured data to non-structured one, which affects the way XML is stored. For example, the structured data can be stored in relational DB with SQL language access whereas unstructured data can be stored in a file system in some kind of binary form with file system interface to access it. In addition to that, the XML Schema (Thomson, 2004) defines a wide range of data characteristics, which widens the range of possible XML representations. However, for application builders, XML provides **a** single data interface (DOM) plus type information (PSVI). In this paper we call the combined interface XQDOM.

In a multi-tier application, tiers are usually programmed in different languages using different data models. For example, in a typical Web application, the front web tier is coded in scripting languages, such as Perl, PHP, manipulating XHTML based text. The application server mid-tier is coded in Java or C# using object oriented data model and the database backend is coded in SQL with relational data model. Optimization among tiers is not feasible automatically and usually the decision of what tier runs what code to manipulate what data has to be determined ahead of time. If the same application is build with one XML language (XQuery) and uses XML data model only then a global optimization among tiers is feasible with possibility of applying data shipping or code shipping techniques or both.

Based on our experience with XIPE, we believe that a full-blown “pure” XML application development environment should support both declarative and imperative XML language extensions and it should use a single data interface for accessing the different underlying XML representations. Also, it should allow users to easily create a broad range of XML applications starting from “light” standalone one to “heavy” multi-tier DB based application.

This paper describes our experience of building XIPE and how its design satisfies the above requirements. In order to achieve that we comply with the following design paradigms:

- XIPE components are self-contained and plug-able. They communicate with each other through well-defined abstract interfaces, which hide the implementation details. That allows each component to be swapped when necessary in order to satisfy the new requirements.

- The “light” and “heavy” data repository models allow XIPE to scale seamlessly when deployed on variety of platforms dealing with wide range of XML volume-processing requirements.

The rest of the paper is organized as follows: section 2 gives an architecture overview of various components to support XIPE and discusses the most important XIPE components in details. Section 3 discusses related work. Section 4 concludes the paper.

2 XIPE ARHITECTURE

2.1 Architecture Overview

The XIPE consists of the following components:

- **XCompiler -** it compiles XPath, XSLT, and XQuery/SE/UF/FT into a machine and platform independent *XML byte-code*. Because the byte-code is platform independent it can be compiled, stored, distributed and executed on XIPE on different tiers;
- **XVM**, an XML Virtual Machine component that runs the byte-code produced by the compiler. The virtual machine uses *stacks* for function calls, single-assigned variables, parameters and intermediate results and a *heap* for multi-assigned variables and some intermediate XML data;
- **XML Data Repository** *(XDR)* provides an XML data abstraction on the top of underlying XML data. The data abstractions is implemented as an extended DOM interface (*XQDOM*) that abstracts out the XML tree creation, traversal, modification, node type retrieval and XML document deletion. The XDR provides both “heavy weight” XML Data Base (XDB) and “light weight” in-memory only XQDOM implementations;
- **XML Schema Processor -** it includes schema repository, schema compiler and a validator. The schema compiler compiles XML Schemas into an internal schema object. The XIPE components can access the schema objects via an abstract *XML Schema Interface*. Like XDR, schema objects offer both “heavy weight” (XDB schema) and “light weight” (in-memory only) schema implementations;
- **XML Parser** component that provides fast XML parsing, implementing both SAX, DOM and XQDOM (with XML schema processor) output over textual XML input data;
- **XDebugger** – a symbolic debugger that is able to control the XVM execution and to provide an introspection of the run time stack and variables via JDWP

(Java 2 SDK) protocol to the external debugger agent, such as JDeveloper;

- **XEditor** provides a language sensitive, GUI based, editing for XIPE supported XML languages.

XIPE components are shown in Figure 1.

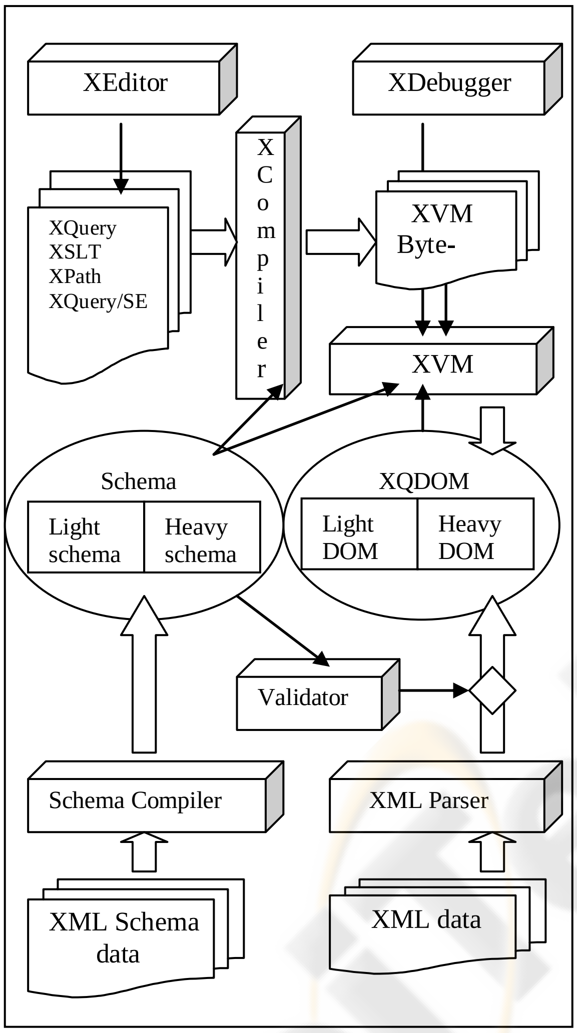

Figure 1: XVEE architecture.

All XML languages supported by XIPE share the same XQuery data model (XQDM) (Liu and, Novoselsky, 2006) and the same function and operator library (Liu and Krishnaprasad, 2005). Therefore, they share a common compiler internal representation and a common byte-code instruction set. The byte-code is platform independent it can be distributed to XIPE running in potentially different tiers and platforms and executed without recompilation. The instruction set is RISC (Reduced Instruction Set Computing) like. However, it may interleave with CISC (Complex Instruction Set Computing) style instructions that can compute certain XQuery expressions more efficiently by delegating the computation to other processors.

The XVM runs byte-code that manipulates data on the main stack where all function calls, XSLT template invocation, local variables and expression results reside. Execution can be stopped with debugger instructions. The debugger component communicates with external GUI debugger agent via JDWP protocol.

As it was mentioned earlier XIPE addresses the scalability requirement by using light and heavy XDR. When the XML document size is small, creating DOM tree in memory is sufficient and the most efficient method to implement XQDM interface. However, when XML document size is large or large document collections are to be processed, then XIPE uses page-able XML Tree Indexing to implement XQDM interface, which can scale with limited memory environment.

Direct XPath expression evaluation or value comparison via XQDOM interface may not be efficient with large XML document collection containing millions of XML documents. In such an environment, XMLIndex (Murthy, 2007) (Liu, 2007) is built on the document collection and for XIPE the leveraging of that index becomes critical. In such a case the XCompiler generates CISC like instructions that use the host-indexing component to efficiently evaluate certain fragments of the XQuery expressions.

Similar to scale with large size XML document, scaling large size XML Schema also needs to be addressed. XML Schema component may have its own XML Schema and index support. However, this is hidden from the rest of the components as the schema information is obtained only through XML Schema Interface.

2.2 XIPE Components

2.2.1 XCompiler

XCompiler is traditionally multi-phased designed where the compilation consists of four processing:

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

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. The second phase uses data-flow analysis in order to perform loop optimizations, let clauses code replacement, constant propagations, type dependent optimizations, … etc. The third phase applies platform-specific iterator-based CISC optimizations, if the corresponding plug-able components are registered with the compiler. The last phase generates the XVM byte-code.

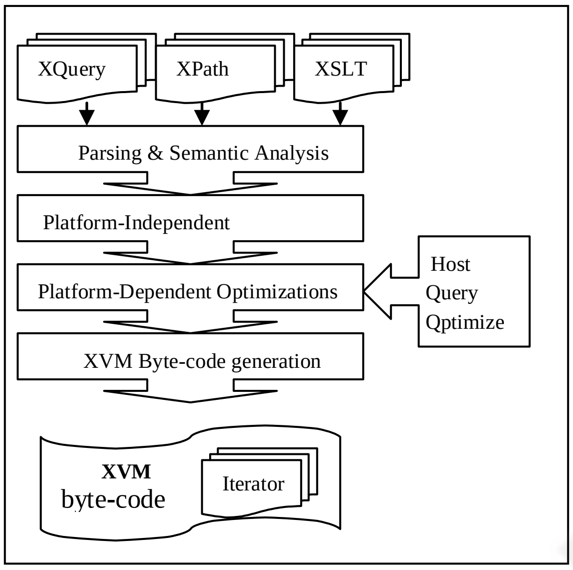

Figure 2: XCompiler Architecture.

Traditional imperative programming languages, such as C/Java, require user to write code as a sequence of computation steps. Side effects from each step are visible to all subsequent steps. The computational state for each step, is not explicitly saved because it is captured by the result and side effects of the step. Therefore, expressions in imperative language are compiled into a set of RISC style instructions that compute the result of an expression from each of the sub-expressions in bottom up fashion. Some processors may use a stack where results from sub-expressions are pushed and popped. We call this *eager evaluation* strategy.

On the other hand, the query execution engine for declarative query languages uses *lazy evaluation* (*iterator*) execution in order to avoid materialization of large intermediate results (Graefe, 1993; Florescu, 2003). That is, results are not fully computed and materialized until when they are absolutely needed for consumption. Side effects are not applied directly. Instead, they are recorded and are not applied until the entire query is finished. This is known as *snapshot semantics*. Consumer of query language typically uses iterator interface to fetch a subset of results at a time instead of the entire result set at once. Therefore, expressions in declarative query language are compiled into an *expression **iterator tree***, which is a tree of nodes, each of which is responsible for computing one expression using an iterator interface with its computational state stored as part of the node. Execution is driven top down. That is, consumer requests the top tree node for the next set of results, then the top tree node passes down the result fetching requests to the rest of the nodes recursively and each node computes next set of results from its previous computational state stored in the node and derives the new computation state stored in the node.

XQuery/SE and XSLT have both imperative procedural and declarative query semantics. XQuery/UF (Chamberlin, Florescu, 2008) explicitly defines snapshot semantics for its updating expressions while XQuery/SE (Chamberlin, 2008) has a pure imperative semantics.

There is a trade-off between processing declarative query construct using lazy evaluation model versus eager evaluation model. The lazy evaluation model scales with large data size but does not scale with large program size because the entire execution tree with all of its intermediate computational have to be tracked. The eager evaluation strategy scales with large program size but not with large data size because intermediate results have to be materialized. Eager evaluation strategy is more efficient than lazy evaluation when all intermediate results are needed to determine an answer. However, eager evaluation strategy is sub-optimal if only partial results are needed. In our compiler design we try to combine both eager and lazy strategies in order to achieve a sub-optimal balance of the two.

XCompiler compiles sequential expressions into sequential instructions so that they are evaluated in eager evaluation manner. In run-time, XVM executes the instructions and immediately consumes the results. On the other hand, XCompiler can compile non-sequential expressions into an iterator CISC instruction, which contains a serialized expression iterator tree, stored as part of the byte code data segments. In run-time, the CISC tree is de-serialized and executed. The result of the execution is an **iterator data object,** whose elements are consumed in an iterator manner.

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, total lengths and offsets to each body section (Novoselsky, 2008). The body contains sections for the byte-code itself (which are referred as XVM virtual 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. Each virtual instruction has op-code, operands and flags. The flags carry information about the operand type, XPath step modes, sequential type occurrence, … etc.

XCompiler implements both static and dynamic (default) modules linking. If the static linking mode is set, all dependent modules are compiled together with the main module, all external references are resolved and one composite executable byte-code module is generated. The result byte-code contains all dependent modules with no demarcation boundary between them. If dynamic mode is set, all modules are compiled separately and their byte-code has an extended header containing tables for imported and exported entities. The exported entities are top-level function and variables while the imported ones are all external references that refer to entities in other modules plus type references. When compiler compiles an import statement, it reads the imported module header and adds all module export entities to the symbol-table. If the compiler encounters a reference to imported entity, it adds it to the current module import-table. All external references are resolved by name and module id, quite like references in Java classes. In run-time, when XVM executes an instruction that refers to unresolved imported entity, it checks if the corresponding module is loaded. If the module is not loaded, the XVM loads it and allocates a table for the module external references. As it was said earlier, the external references are resolved lazily on demand. The XVM dynamic module linking is better suited for larger applications that use module libraries. Also, it has a smaller run time memory footprint. On the other hand the static linking has a minor performance advantage over the dynamic one.

When XIPE works with a “heavy” XDR (usually an XML DB) some of CISC instruction can be executed directly by XDR host processor. To accomplish this, XCompiler provides a *Query Push-Down Interface,* which allows the host XDR to plug in a *host query optimizer*. The host query optimizer has a detailed knowledge of the underlying XML data indexing so it can decides what part of the input program will be compiled into CISC instructions. In run-time the corresponding host *iterator-executer* will be invoked by the XVM as an external function and will perform the iterator tree execution. The returned iterator data object does not materialize the result sequence. Instead, it provides an iterator *interface* (open-fetch-close) to its consumer.

2.2.2 XML Virtual Machine (XVM)

The data model all XML languages referred here use the XQMD (Fernandez, 2007), which treats data objects as *sequence of items*. The item type can be a basic built-in type, such as number, string, dates etc, or an XML node reference. The size of the sequence is dynamic and so is the size of strings. XQuery/UF/FT, XSLT and XPath are functional languages. They don’t allow side effects and their variables are single-assigned only. That means that a single stack can be used to hold their intermediate results during execution. This is why XVM uses stacks to hold XQDM instances in order to minimize dynamic memory allocations. Intermediate results that have a fixed data size, such as number, dates etc, are loaded into the system main-stack. The content of the intermediate results with a dynamic data size, such as strings and sequences, is stored into complimentary content stacks with an object descriptor in the main-stack. That way, since intermediate results are transient, the majority of the intermediate computation results do not require additional memory allocation. Only when the stack is full, XVM dynamically grows that stack with an additional segment. Also, all single-assigned variables, function and template parameters, function and template call-frames reside in the main stack. Logically, XVM doesn’t need more than one stack but because run-time objects have a dynamic size the loop variables can’t be hold in pre-allocated slots in the main stack. This is why XVM uses a second stack (context-stack) for loop and context variables. Both, the main-stack and the context-stack have the corresponding item, string and node stacks for dynamically sized objects.

However, XQuery/UF/SE introduces some sequential (non-functional) construct such multi-assigned variables and DOM updates. That means, that dynamically sized multi-assigned variable values can’t be stored in the complimentary stacks anymore. To hold such variable values XVM uses a non-stack based dynamic memory (heap). Garbage collector techniques are applied to free the memory when the results are no longer needed.

XVM execution architecture is quite simple. There is a set of functions, one for each instruction, implementing the instruction semantics. The XVM 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 main-stack and pushes back the result.

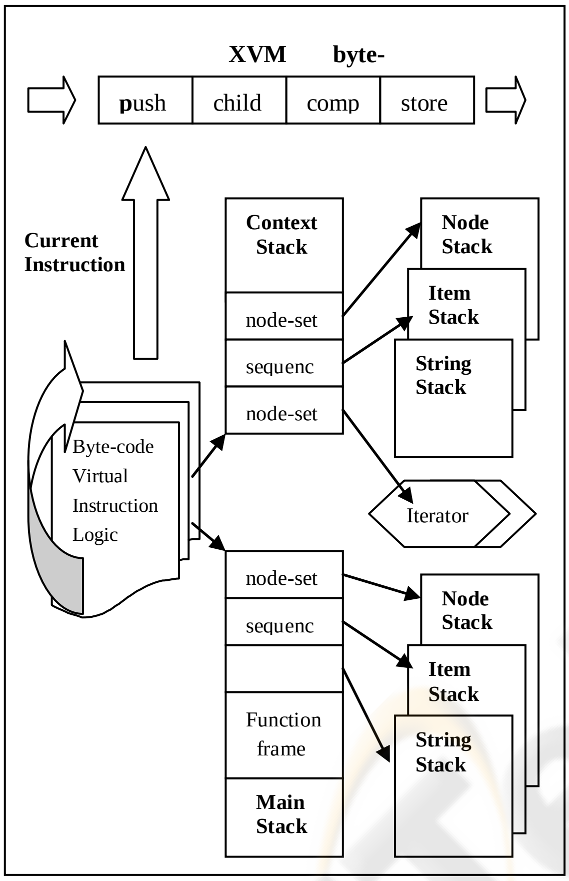

Figure 3: XVM Architecture.

When a function is called or an XSLT 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. The result of an execution is a sequence, which XVM provides with an iterator interface to allow XML applications to fetch the result. However, XVM also provides alternative interfaces to serialize the output so the embedding applications can fetch the result in the form of DOM trees, SAX events or XML text. In such a case 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.

2.2.3 XML Data Repository (XDR)

XVM interacts with XML data layer via *XQDOM interface*. XQDOM interface provides full navigation capabilities of the DOM tree in order to implement the XPath child-parent-sibling axis traversal semantics and provides a virtual XML storage layer for XVM.

When XML input size is small, building a "light" in-memory DOM tree that implements XQDOM interface is efficient. However, when the input XML is large or in a case of "heavy" XML DB, a scalable XQDOM implementation is required. To accomplish this, an intermediate *XML Tree Index (XTI)* layer is used. XTI works as a mediator between XML users and the actual XML representation. It provides XQDOM interface for XML users and it accesses the physical XML storage via a simple *XML Content Interface (XCI).* XCI has the classical file system methods such as open, read, write, seek and close plus few other methods for data storage and retrieval. XTI manages a set of fixed size pages, where each one of them contains a constant number of XTI nodes. Since nodes have a constant size, they can be addressed by an offset (number of nodes) from the beginning of the document. Nodes contain their parent, child and sibling node addresses (offsets), making that way the full axis traversal more efficient. For element or attribute nodes, the encoded qualified-name (Bray, 2006) is stored in the node in order to speed up name comparisons during XPath evaluation. For text, comment and processing-instruction nodes, only a locator to the content of the nodes is stored in the node. This approach efficiently separates the XQDOM navigational fixed size component (XTI) from the physical variable size XML data representations. This separation allows applying significantly more efficient memory management algorithms for the fixed size XTI nodes. Also, XTI can be viewed as an *index* that provides efficient node navigational access over the physical XML content, which remains opaque to the XTI layer. Not all XTI pages are needed to be loaded into memory at a time. Instead, they are cached as an in memory page cache, which keeps frequently accessed pages. This "heavy" DOM scales very well with any XML document size compared to the "light" in-memory DOM approach where all DOM nodes have to be loaded in memory.

2.2.4 XML Schema Component

Unlike relational data schema where schema is typically much smaller than actual data, XML Schema can be large and complex. In fact, XML Schema is served as data validation constraint instead of playing the traditional meta-data role. Although some XML schemas are small, others can be large. Therefore, the design of building DOM tree for XML schema and providing XML schema access through in memory data structures backed up by DOM tree may not be scalable solution with large XML schema. XIPE handles XML schemas similar to the way it handles XML data. There is a "light-weight" in-memory schemas and "heavy-weight" scalable schemas stored in schema repository. XIPE schemas are accessed by XIPE components via an *XML Schema Interface*. The interface contains all the methods needed for schema navigation and validation. When XML schema is initially loaded into the system, it goes through a **schema registration** process where the schema compiled into an internal format and stored as a set of dependable modules. Each module can be separately loaded into memory. The module has logical pointers to imported modules, which are loaded into memory only when needed. The idea is similar to the one of XTI page cache so that not all schema information needed to be loaded in memory at once.

3 RELATED WORKS COMPARISON

There are lots of research effort of processing XML programming languages from both SIGPLAN and SIGMOD communities. There are also both commercial and open source efforts of providing XML language processing in variety of environments. Our XIPE approach is different from others in the way that we combine both declarative and imperative language processing paradigms in one place with the help of XCompiler and XVM. The idea of handling imperative languages by using a byte-code and a virtual machine is not new. However, processing XQuery/SE/UF/FT and XSLT using an imperative virtual machine is not common mainly because these languages have been primarily studied as an XML database query languages. To our knowledge, XSLT VM (Novoselsky, 2000) is the first virtual machine built for processing XSLT. XVM is the first single virtual machine capable to process all XQuery/SE/UF/FT, XPath and XSLT (Novoselsky, 2008). This paper is the first to show the advantages of combining both declarative and imperative approaches of handling XML programming languages. Furthermore, this paper addresses XML Tree Indexing as a scalable way of handling large XML documents. Brothner (Brothner, 2004) proposes compiling XQuery into a Java byte-code so that JVM can be used to execute XQuery. However, the idea of combing both declarative and imperative language processing technique is not mentioned plus JVM is not designed to work with XQuery Data Model.

Our second unique approach is that we design XIPE with different size of XML data and XML schemas in mind. We have articulated the key concepts of XML Tree Indexing component and XML Schema Indexing component that are essential to scale large size XML document and schema. However, they are all abstracted using XQDOM interface and XML Schema Interface from the rest of components. This is particularly different from other approaches where decisions of XML storage are typically hardwired with the XQuery/XSLT processors.

4 CONCLUSIONS

In this paper, we propose the concept of XIPE with various key components and the component architecture paradigms. We follow the interface based design approach so that implementations for these interfaces can be open and flexible. Then when XIPE is embedded into different host system, platform-specific native components can be plugged. In fact, since XML data and schema size can vary within a broad range, these components are designed with "*light-weight*" vs. "*heavy-weight*" XML data design paradigm in mind.

For XML languages processor architecture we follow the classical programming language compiler and virtual machine design paradigms as the basis. We anticipate that for pure XML application environment, using XQuery/SE/UF/FT languages to write business application logic will be the main stream. Imperative XQuery/SE language constructs will be used heavily and many XQuery modules will be independently developed and shared to build large-scale XML application programs. This is the perfect use case for XVM imperative or "eager" type of processing. Meanwhile, the XML language compiler and virtual machine allow declarative XQuery and XSLT constructs to be efficiently processed by using lazy evaluation technique whenever needed. These declarative constructs processed in a "lazy" non byte-code execution manner are like CISC instructions embedded within a stream of RISC instruction and are executed on a native co-processor. This so called "mixed RISC/CISC" way of processing XML languages achieves the best balance between eager and lazy evaluation and generally yields better performance compared with systems that use one technique only.

REFERENCES

Boag, S., …, 2007. XQuery 1.0: An XML Query Language. In *http://www.w3.org/TR/xquery/*

Amer-Yahia, S., …, 2008. XQuery and XPath Full Text 1.0. In *http://www.w3.org/TR/xquery-full-text/*

Bray, T., …, 2006. Namespaces in XML 1.0. In *http://www.w3.org/TR/REC-xml-names/#ns-using*

Bothner, P., 2004. Compiling XQuery to Java bytecodes. In *XIME-P 2004: 31-36.*

Chamberlin, D., Engovatov, D., …, 2008. XQuery Scripting Extension 1.0. In *http://www.w3.org/TR/xquery-sx-10/*

Chamberlin, D., Florescu, D., …, 2008. XQuery Update Facility. In *http://www.w3.org/TR/xquery-update-10/*

Chamberlin, D., Carey, M., …, 2006. XqueryP: An XML Application Development Language. In *http://2006.xmlconference.org/proceedings/38/ presentation.pdf*

Fernandez, M., …, 2007. XQuery 1.0 XPath 2.0 Data Model. In *http://www.w3.org/TR/xpath-datamodel/*

Florescu, D., Hillery, C., …, 2003. The BEA/XQL streaming XQuery Processor. In *VLDB 2003: 997-1008.*

Fourny, G., Kossmann, D., …, 2008. XQuery in the browser. In *SIGMOD Conference 2008, 1337-1340.*

Graefe, G., 1993. Query Evaluation Techniques for Large Databases. In *ACM Computing Surveys 25(2):73-170.*

Liu, Z., Krishnaprasad, M., 2005. Native XQuery Processing in Oracle XML DB. In *SIGMOD 2005.*

Liu, Z., Novoselsky, A., 2006. Efficient XSLT Processing in Relational Database System. In *VLDB 2006: 1106 – 1116.*

Liu, Z., Krishnaprasad, M., …, 2007. XMLTable Index - An Efficient Way of Indexing and Querying XML Property Data, In *ICDE 2007.*

Java 2 SDK. Java Debug Wire Protocol (JDWP). In http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jdwp/jdwp-protocol.html

Kay, M, 2007. XSL Transformations (XSLT) Version 2.0. In *http://www.w3.org/TR/xslt*

Le Hors, A., …, 2000. Document Object Model (DOM). In *http://www.w3.org/DOM/*

Malhotra, A., …, 2007. XQuery 1.0 XPath 2.0 Functions and Operators. In *http://www.w3.org/TR/xquery-operators/*

Murthy, R., Liu, Z., …, 2005. Towards An Enterprise XML Architecture , In *SIGMOD 2005.*

Novoselsky, A., Karun, A., 2000. XSLT VM – An XSLT Virtual Machine. In *http://www.gca.org/papers/xmleurope2000/papers/s35-03.html*

Novoselsky, A., Liu, Z., 2008. XVM – A Hybrid Sequential-Query Virtual Machine for Processing XML Languages. In *PLAN-X 2008.*

Thomson, H., …, 2004 XML Schema Part 1. In *http://www.w3.org/XML/Schema*