A High-Performance Interpretive Approach to Schema-Directed Parsing

Morris Matsa, Eric Perkins, Abraham Heifets, Margaret Gaitatzes Kostoulas,
Daniel Silva, Noah Mendelsohn, Michelle Leger
{mmatsa,perkinse,aheifets,mgg,dsilva,noah_mendelsohn}@us.ibm.com
maleger@gmail.com

IBM Corporation
One Rogers Street
Cambridge, MA 02142 USA

ABSTRACT

XML delivers key advantages in interoperability due to its flexibility, expressiveness, and platform-neutrality. As XML has become a performance-critical aspect of the next generation of business computing infrastructure, however, it has become increasingly clear that XML parsing often carries a heavy performance penalty, and that current, widely-used parsing technologies are unable to meet the performance demands of an XML-based computing infrastructure. Several efforts have been made to address this performance gap through the use of grammar-based parser generation. While the performance of generated parsers has been significantly improved, adoption of the technology has been hindered by the complexity of compiling and deploying the generated parsers. Through careful analysis of the operations required for parsing and validation, we have devised a set of specialized bytecodes, designed for the task of XML parsing and validation. These bytecodes are designed to engender the benefits of fine-grained composition of parsing and validation that make existing compiled parsers fast, while being coarse-grained enough to minimize interpreter overhead. This technique of using an interpretive, validating parser balances the need for performance against the requirements of simple tooling and robust scalable infrastructure. Our approach is demonstrated with a specialized schema compiler, used to generate bytecodes which in turn drive an interpretive parser. With almost as little tooling and deployment complexity as a traditional interpretive parser, the bytecode-driven parser usually demonstrates performance within 20% of the fastest fully compiled solutions.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors - code generation, compilers, optimization, parsing, retargetable compilers. ; D.2.8 [Software Engineering]: Metrics — Performance measures.

General Terms: Performance, Experimentation, Standardization, Languages

Keywords: parsing, XML, schema, interpreter, compiler, performance

1. INTRODUCTION

XML delivers key advantages in interoperability due to its flexibility, expressiveness, and platform-neutrality. The broad range of applications and growing user base for XML technologies have driven the development of common tooling, providing a consistent, robust infrastructure on which to build applications. These advantages have spurred widespread adoption of SOAP and XML-based web services, as key components of the next generation of business computing infrastructure. It is increasingly clear, however, that with these advantages XML also carries a heavy performance penalty, and that the parsing technologies currently in use are unable to meet the performance demands of an XML-based computing infrastructure. Traditionally, this performance gap is especially acute in scenarios where validation is required to ensure security and integrity of loosely coupled systems. Thus, validation performance is critical to the viability of the infrastructure of the web.

Some progress has been demonstrated in recent years in improving XML validation performance through the use of grammar-based parser generation [3, 7, 9]. Compiled schema-specific parsers have been shown to significantly increase performance and virtually eliminate parsing as a performance bottleneck in the overall processing stack. Unfortunately, this increased performance comes at a cost; parser generation and compilation increase tooling and deployment complexity. By requiring a source code compiler, and burdening the user with the management of multiple compiled artifacts, source code compilation undermines the usability of XML technologies.

While schema-driven, compiled parsers demonstrate dramatic performance advantages over existing interpretive approaches, traditional compilation is not required to achieve these benefits. Indeed, the performance gains of compiled parsers have been attributed to a combination of layer-breaking optimizations and grammar-directed scanning. [7] While these optimizations are particularly well suited to expression through code generation, where otherwise distinct layers are integrated by being compiled together, this is not the only way to achieve similar results. In this paper we draw on the conclusions of our work with code generation to develop an interpretive solution for parsing and validation which captures the key benefits of well-tuned code generation, while minimizing the overhead of runtime interpretation.

Runtime interpretation of the rules of the grammar, the approach used in traditional XML validation, does not allow sufficient opportunity for layer-breaking and scanning optimizations. Interpretation of low-level instructions, as generated by a full-compilation approach, would incur significant overhead, overwhelming any performance benefit. Thus, it is clear that an approach that balances these two designs is required. Using a domain-specialized, custom instruction set it is possible, by controlling the granularity of the instructions, to control the overhead imposed by the interpreter. If the specialized instructions are further able to capture the full breadth of optimization strategies used in the code generation approach, then performance is not compromised.

In small or *ad hoc* deployments, such bytecodes can be generated on-the-fly, posing no management overhead to the user. When materialized on disk, the execution plan provides a compact artifact that can be efficiently stored and managed in large-scale deployments. The stability and robustness of an interpretive parser is also improved with respect to fully-generated parsers, as the execution is tightly constrained by the limited instruction set, and mediated by a single trusted runtime.

In short, our approach of novel, special-purpose bytecodes delivers the performance advantages of compiled systems, while maintaining the convenience of use associated with traditional parsers and validators. Indeed, the interpretive approach described here proved essential to the adoption of high-performance validation technologies in a large scale, commercial, Web-enabled database.[1] This real-world application demonstrates the feasibility of this approach to enable high-performance validation throughout the web.

1.1 Organization of the Paper

In this paper, we discuss in depth the analysis and reasoning underlying the design and selection of a bytecode instruction set for XML processing. The instruction set incorporates the fine-grained composition techniques required for high performance, while using instructions that are themselves coarse grained, minimizing interpretive overhead.

After describing a complete set of bytecodes for XML parsing and validation, we demonstrate the value of the approach using a functional, bytecode-interpreting, validating parser that balances the need for performance against the requirements of simple tooling and robust scalable infrastructure. This prototype leverages componentry from previous compilation work for XML Screamer [7,9], such as the content-model compilation engine and the optimized scanning infrastructure, thus supporting the full breadth of XML and Schema, including full support for namespaces and dynamic typing. Rather than generating source code, however, it compiles a schema grammar into specialized bytecode instructions, which may be efficiently interpreted. Together, the bytecode compilation engine and the interpretive parser are called *iScreamer*, indicating both the compilation heritage of the system and its interpretive nature.

[1]DB2 9 for z/OS

2. BYTECODE SCHEMA COMPILATION

The bytecodes used in iScreamer are designed specifically for the task of XML parsing and validation. Leveraging this specialization, the instructions can be coarse-grained and compact. This design minimizes the overhead of instruction interpretation relative to the actual work of parsing and validation because each instruction may represent tens or even hundreds of lines of code. The time to load and dispatch a given instruction thus remains small relative to the time required to execute the work of the instructions themselves. In addition to minimizing the overhead of the interpreter, the specialized design of the instruction set also simplifies the interpreter itself; since the range of supported operations is small, the implementation of the interpreter is accordingly simple. This is in contrast to the generic bytecodes used by bytecode compilers for a general purpose language such as Java [13], Perl [16], and Tcl [8], where instruction granularity is fine and interpretive overhead is therefore high. In the following section we motivate the selection of the bytecodes used in iScreamer with an analysis of the process of XML parsing and validation. Careful attention is paid to the theoretical limits of grammar-driven optimizations, and these limits are used to delineate the bounds of the instruction primitives.

2.1 Design of the Instruction Set

Schema-based parser generation technologies achieve performance by combining parsing and validation through compilation. This allows a compiled validating parser to avoid reprocessing the parsed input to verify validity, and significantly speeds the parse. In some cases, the schema information can also speed the scanning itself. When exactly one element is expected, scanning for the particular tag, rather than the generic element production, can be significantly more efficient. Similarly, when only an end tag is expected, it may be compared verbatim against its balancing start tag. In a parser generator, these optimizations are a natural result of the translation of the grammar into compiled code. In an interpreter, the granularity of that approach is not practical, and so more care must be taken to abstract the idioms of generated parser into a set of high-level primitives that can be interpreted efficiently.

Since the overhead of interpretation is proportional to the number of instructions evaluated, it is essential that the instructions be very coarse-grained, each representing a relatively large piece of required work. On the other hand, the grammar structure, which is embodied in the composition of the instructions, must be expressed at a sufficiently fine granularity to allow the scanner to take full advantage of the grammar-sensitive scanning techniques that drive generated parser performance. [7] The challenge is therefore to identify the minimum set of points of interaction between the grammar automaton and the scanner.

2.1.1 Scanning

XML Schema constrains character data in the instance either with a simple type, or the content-type of a complex type. The content type of a complex type may be simple, mixed, element-only, or empty. The restrictions on character content for simple types and complex types with simple content are equivalent. Thus at any given point in the document, character data is validated according to one of the four content-types. The content-type, and thus the validation rule, for any particular sequence of character data is determined by the type definition for its enclosing element. This means that from the end of one tag, to the start of another, the validation rule is fixed.

Because XML Schema specifies a grammar over qualified elements, the namespace of a tag must, in general, be resolved before the element can be validated. Since namespace declarations may occur among any of the attributes that follow the tag name, it is impossible to unconditionally validate a tag name until the whole tag is read. The same problem applies to the attributes and their values, with an additional complication; dynamic type assertions in the form of *xsi:type* may also appear among the attributes. The effect of an xsi:type declaration is to stipulate, in the instance, the type used for validation of the element. Because attribute occurrence and type constraints are specified on the type definition (and not the element declaration), neither the occurrence nor the values of attributes may be conclusively validated until the whole tag is read. In short, the validation rule for the tag name and the attribute names and values of a particular tag are as dependent on the instance as they are on the grammar, at the granularity of intra-tag structures. It is in general impossible, having read in a tag-name, for a grammar automaton to further constrain the validation rules for subsequent attributes. Similarly, attributes, once read, do not further clarify the validation of their subsequent siblings. We can therefore conclude that the grammar automaton has no substantive interaction with intra-tag scanning.[2]

Noting, finally, that XML Schema does not provide a mechanism for character data to affect the validation rule for subsequent tags, we may conclude that the only point in the scan where a substantive interaction between the grammar and the scanner occurs, is at the end of a tag (or the start of the document). This observation forms the basis for the processing model of the scanning instructions. Each advances the scanner forward through the end of the next start or end tag, validating character data according to one of the four content-types. The various flavors of scanning instructions, and their contributions to grammar-directed scanning, are discussed in detail in Section 2.2.

Due to the limitations above, validity cannot be completely assessed within the scanner. In particular, once a tag is read, it must be checked against the allowable elements specified by the grammar automaton. Once the type of an element is established, its attributes must be assessed for occurrence (are all of the attributes allowed, are any required attributes missing?), and each attribute value must be validated by the appropriate simple type validator. Again, the goal of the instruction set design is compact, coarse-grained operations. To this end, validity assertions are made through task-specific instructions that interact with the scanner data structures. Where possible, constraints are evaluated in bulk, through the use of bit vectors. This simplifies constraint evaluation, while at the same time minimizing instruction space, and thus load time.

[2]In simple cases, it may be possible to provisionally identify element and attribute names, and in the absence of xsi:type declarations, to guess the types and occurrence constraints on the attributes of a tag. Such heuristics provide diminishing returns however, as they become more elaborate. For simple heuristics, the important grammar information can be established before scanning of the tag begins.

2.1.2 Grammar

With the scanning and validity assessment features of the instruction set specified, it remains to represent the grammar automaton in the instruction sequence. This requires at least basic control flow primitives. With interpretation overhead in mind, however, these primitives must be specialized to the transitions used by the grammar automaton. XML Schema provides a guarantee of content-model determinism in the form of the *Unique Particle Attribution Constraint.* This specifies that when an element is read, it must be possible to uniquely determine the part of the content model (particle) that validates it. Furthermore, no support for co-occurrence constraints is provided, so any transitions in the grammar automaton will be made based solely on the most recently read tag name. Thus we can save instructions by replacing generic control flow primitives like `JNE` (jump if not equal to zero) with branches that activate based on the name of the current tag.

Two XML Schema content model constructs create transitions in the grammar automaton that are not well-represented by the simple tag-based transitions described above. Arbitrary, finite occurrence constraints allow schema authors to specify that any part of the content model may be repeated a specific, arbitrary number of times. In the simple model of tag-based transitions, this would lead to an explosion in the number of states in the instruction representation of the grammar. The logical solution is to supplement the instruction set with counters and counter-based branches, to allow the repeated evaluation of a given part of the content model, according to its occurrence constraint.

In addition to occurrence constraints, *all groups* require further attention. In XML Schema, all groups duplicate for elements the restrictions normally placed on attributes. An unordered set of elements is specified, where each is either required or optional. Validation of such a constraint is best achieved, as with attributes, in bulk. Accordingly, we use a stack of bit vectors to evaluate the constraint. With the addition of the bit vector check, the all group content model may be treated as a repeated choice, which is easily handled by the rest of the control flow constructs.

Many schema-based parser generators represent each type in the schema with a function. This is a natural division point for the grammar, and makes support of recursive schemas trivial. Accordingly, the overall execution plan is organized into type handlers which are analogous to subroutines. Some specialization is required, however, to limit the complexity of the handler structure. XML Schema provides two mechanisms (xsi:type, and xsi:nil) that alter the mapping from element to type. Both are driven by attributes on the element itself, and can be handled completely within the interpreter. Since the subroutine call is used to redirect control from the instructions that validate an element to the handler for the actual type of that element, these special conditions are best handled by a call instruction. As such, a call instruction is required to validate any xsi:type and xsi:nil attributes, in addition to redirecting control to the appropriate handler.

2.2 Bytecodes for XML Processing

As discussed above, the interpreter derives significant performance savings through the use of specialized, high-level XML processing instructions. In this section, we present the full set of instructions in detail, highlighting the balance in granularity between context sensitivity and efficient interpretation.

Many of the bytecodes presented make use of the scanner's symbol table to reference QNames by an integer identifier. At compile time, all of the QNames in the schema are assigned identifiers, which are populated into this symbol table. Additionally, special handles are set aside for end tags and the end-of-file marker. All end tags, regardless of their name, are represented by the same special identifier. When an end tag is scanned, it is implicitly checked against its matching start tag for well-formedness.

Bytecodes and all of their arguments are simple integers. In these descriptions, all arguments are shown in italics, and repeated arguments are shown with ellipses. If any validation or assertion in a given bytecode fails, then all processing halts immediately with a validation failure.

Several instructions make use of bit vectors. These are expressed as a variable length list of integers, each of which is used to represent 32 bits of the vector. The length of the list is a separate, explicit argument to the instruction. Bit vectors are used for processing attributes, elements, and all groups.

2.2.1 Scanning

The fundamental process of XML parsing is the traversal of the input byte-stream by the scanner. This is controlled with specialized `READ` instructions, which represent the basic primitives of XML scanning. As set out in Section 2.1, these instructions are designed to maximize opportunities for grammar-driven optimization, while remaining as coarse-grained as possible. Performance gains are realized in two ways. Several of the instructions can be used to invoke more efficient, specialized scanning, based on the grammar (`READ_TAG_WITH_QNAME` is an example of this). Other instructions combine scanning and validity assertions, avoiding additional, possibly costly checks.

In the following discussion, we use the general term *tag* to mean any start or end tag, including all of its attributes. Empty element tags are treated as if they had been expressed in the equivalent syntax using separate start and end tags, with no intervening content. This is handled automatically by the scanner. After scanning an empty element tag, the scanner will appear to be positioned immediately after a normal start tag. An additional call to the scanner will be required to advance the state beyond the virtual end tag.

The processing model used by the scanner, as discussed in Section 2.1, operates on the input tag by tag. At every point in the execution plan, the scanner is logically positioned just after a given tag (or at the start of the document), and before any subsequent character data. This tag (i.e. the one most recently read) is called the *current tag*. Information about the current tag, its attributes, and any comments, processing instructions, and character data that directly preceded it is available to all instructions until the next tag is read. Every `READ` instruction advances the scanner through the end of the next tag (except for `READ_EOF`, which reads the trailing comments, processing instructions, and whitespace at the end of the document), or fails to complete.

- `READ_TAG` *`IS_MIXED`* :
  Read forward to the end of the next tag in either mixed or element-only mode. The tag to be consumed may be any start or end tag. End tags are implicitly checked for well-formedness against their matching start tags.

- `READ_TAG_WITH_QNAME` *`IS_MIXED QNAME_ID`* :
  Read forward to the end of the next tag in either mixed or element-only mode. While scanning, validate that the tag's name matches the given QName. Here tag scanning is optimized, because the scanner can directly compare the input stream against a fixed tag name. The symbol table lookup is also avoided since the identifier is already known.

- `READ_END_TAG` *`IS_MIXED`* :
  Read forward to the end of the next tag in either mixed or element-only mode. While scanning, verify that the tag is an end tag, and match it lexically against its balancing start tag. Like `READ_TAG_WITH_QNAME`, this instruction optimizes tag scanning by comparing the input buffer against a known string, in this case the balancing start tag.

- `READ_EMPTY` :
  Read the next tag, which must be an end tag, and must match its balancing start tag, and validate that there is no intervening character content. This instruction works much like `READ_END_TAG`, except that it also validates that no character content was read.

- `READ_SIMPLE_CONTENT` *`TYPE_ID`* :
  Read character data forward to the end of the next tag, which must be an end tag, and must match its balancing start tag. Validate the intervening content using the built-in simple type handler given by the type ID. The content is handled by a type-specific scanner that will validate the content as it scans. The end tag is handled as in `READ_END_TAG`.

- `READ_EOF` :
  Read forward to the end of the file, matching the XML production for `TrailingMisc`.

2.2.2 Content Model Assertions

Following a given `READ` instruction, the schema may require one or more assertions on the state of the current tag. These assertions are accomplished with a family of `ASSERT` instructions that operate on the current tag.

- `ASSERT_TAG_QNAME` *`LIST_LENGTH QNAME_ID`* `...` :
  Assert that the current tag matches one of the given QNames.

- `ASSERT_TAG_QNAME_BV` *`BV_SIZE BV`* `...` :
  Assert that the current tag matches one of QNames in the set defined by the given bit vector. This instruction duplicates the functionality of the one above, but is used for larger QName sets, where a bit vector comparison is more efficient.

- `ASSERT_ATTRS` *`BV_SIZE EXCL_BV`*`...` *`REQD_BV`* `...` :
  Assert that the current tag's attributes obey the collective attribute occurrence constraint as represented with two bit vectors, one for excluded attributes, and one for required attributes.

- `ASSERT_ATTR_CONTENT` *QNAME_ID TYPE_ID* :
  Check the current tag's attributes for the given attribute QName, and if present, validate its content using the built-in simple type handler given by the type ID.

- `FAIL` :
  Unconditional assertion failure.

2.2.3 Bit Vectors

In addition to the assertions related to the current tag, we also introduce bit vector assertions, which are used to check *all group* occurrence. These instructions manipulate a stack of bit vectors.

- `PUSH_BV` *BV_SIZE* :
  Push a new, empty, bit vector onto the bit vector stack.

- `TEST_AND_SET_BIT` *BIT* :
  Set the given bit in the current bit vector, asserting that it was not already set.

- `POP_ASSERT_BV` *BV_SIZE REQD_BV ...* :
  Pop the current bit vector off of the bit vector stack, and assert that it is a superset of the given required bits.

2.2.4 Control flow

Control flow in the parser is governed largely by the state of the current tag. As suggested in Section 2.1, this is implemented with a family of tag-based control flow instructions. In each of the `JUMP` instructions, an offset is used to refer to the target instruction, and is measured from the start of the `JUMP` instruction.

- `JUMP_TAG_NOT_EQUAL` *QNAME_ID OFFSET* :
  Branch to the instruction at the given offset if the tag does not match the given QName.

- `JUMP_TAG_TABLE` *TABLE_SIZE TABLE_PART ...* :
  Branch to one of several locations using the current tag as a key in a jump table.

- `JUMP_UNCONDITIONAL` *OFFSET* :
  Branch unconditionally to the given offset.

2.2.5 Counters

In order to verify occurrence constraints on element content, we support a simple stack of counters, manipulated by a family of counter instructions. These instructions supplement the tag-based control flow above.

- `PUSH_COUNTER` :
  Push a new counter onto the stack, and initialize it to zero.

- `INCREMENT_COUNTER` :
  Increment the top counter on the stack.

- `POP_COUNTER` :
  Discard the top counter on the stack.

- `JUMP_COUNTER_LESS` *VALUE OFFSET* :
  Branch to the given offset if the top counter is less than the given value

- `ASSERT_COUNTER_GREATER_OR_EQUAL` *VALUE* :
  Assert that the top counter is greater or equal to the given minimum value.

2.2.6 Call and Return

The basic scanning, validation and control-flow primitives above are used to parse and validate the content of an XML Schema type. Each type in the schema has a handler, analogous to a function, which is composed of the basic instructions. The handlers are executed, like functions, in a recursive descent manner. The call and return idioms are, however, specialized to the task of processing XML. In particular, the call instruction (`CALL_TYPE`) is made against an element declaration. At runtime however, the virtual machine jumps to the correct type handler corresponding to the element's type, including instance type stipulations (*xsi:type*), and handling of *xsi:nil*. [14]

- `CALL_TYPE` *NIL DEF_TYPE BV_SIZE EXCL_TYPES* :
  Dispatch to the content model for the current (start) tag. The content model is determined either by the instance value of *xsi:type*, or by the supplied default type ID. Either way, the actual type ID is checked against the type exclusions bit vector (*EXCL_TYPES*). The excluded types bit vector aggregates the various constraints on the runtime type of the elements (as specified by the element declaration and the type definition for its default type). If the resolved type is complex, the interpreter will dispatch to the handler for that complex type, and resume interpretation. If the resolved type is simple, the corresponding built-in simple type handler is used. The *NIL* argument tells the complex type which processing is allowed for *xsi:nil* in this context.

- `RETURN` :
  Return control from the current complex type handler to its caller.

- `RETURN_IF_NIL` :
  Check the current tag's attributes for the *xsi:nil* attribute. If present and bearing the value true, read the next tag as in `READ_EMPTY`, and return control from the current complex type handler to its caller. In all other cases, continue execution.

2.3 Example Type Handler

As an example of how the bytecodes above are used to validate the content model for a complex type, consider the following schema fragment. The type `ExampleType` is defined to have one required attribute, `attr1`, and two optional attributes, `attr2` and `attr3`. The element content is constrained to be either `Name1` or `Name2`, followed by a mandatory `Name3`.

```
<xsd:complexType name="ExampleType">
  <xsd:sequence>
    <xsd:choice>
      <xsd:element name="Name1"
                   type="Type1"/>
      <xsd:element name="Name2"
                   type="Type2"/>
    </xsd:choice>
    <xsd:element name="Name3"
                 type="Type3"/>
  </xsd:sequence>
  <xsd:attribute name="attr1"
                 type="Type3"
                 use="required"/>
  <xsd:attribute name="attr2"
                 type="Type4"/>
  <xsd:attribute name="attr3"
                 type="Type5"/>
</xsd:complexType>
```

The validation plan for this complex type can be expressed in the bytecode sequence in figure 1. For convenience, in the figure, the integer ID's for the types, attributes, and elements are assumed to be the same as the numbers in their names (i.e. `attr1` has attribute ID 1, element `Name2` has element ID 2, and so forth). ID 0 is reserved. The bit vector arguments to the instructions are shown as hexadecimal values, to facilitate bitwise reading.

```
OFFSET  BYTECODE INSTRUCTION
------  -----------
    00  ASSERT_ATTRS 1 0xFFFFFFF1 0x00000002
    04  ASSERT_ATTR_CONTENT 1 3
    07  ASSERT_ATTR_CONTENT 2 4
    10  ASSERT_ATTR_CONTENT 3 5
    13  READ_TAG 0
    15  ASSERT_TAG_QNAME 2 1 2
    19  JUMP_TAG_NOT_EQUAL 1 10
    22  CALL_TYPE 0 1 1 0xFFFFFFFD
    27  JUMP_UNCONDITIONAL 7
    29  CALL_TYPE 0 2 1 0xFFFFFFFB
    34  READ_TAG_WITH_QNAME 0 3
    37  CALL_TYPE 0 3 1 0xFFFFFFF7
    42  READ_END_TAG
    43  RETURN
```

Figure 1: Bytecode example

In this code sequence, the first step is to check the attribute occurrence constraints. In this case, the attribute bit vector size is one, indicating that the highest known attribute ID is less than 32. In order to validate the attribute occurrence, the current tag's attribute bit vector (which has a bit turned on for each of the attributes that were scanned), is compared against the occurrence constraint bit vectors. The first bit vector excludes all attributes except `attr1`, `attr2` and `attr3`. The second requires `attr1`. After occurrence checking, the content of each possible attribute is validated, against its type, in this case `attr1`, `attr2` and `attr3`, against `Type3`, `Type4` and `Type5`, respectively.

Following attribute validation, the element content is validated. In this case, we expect either `Name1` or `Name2`, so we execute a plain read (with the mixed argument set to false). After the tag is read, the QName is validated against the two possibilities, using `ASSERT_TAG_QNAME`. Finally, a `JUMP_TAG_NOT_EQUAL` is executed to either allow control to continue at the next instruction at offset 22, or to divert control to the instruction at offset $19 + 10 = 29$.

Both branches immediately execute the `CALL_TYPE` instruction, which evaluates any *xsi:type* or *xsi:nillable* attributes on the tag, and dispatches to the correct type handler. In the above example, none of the types allow substitution, so their type exclusion bit vectors all have only one non-zero bit.

After reading `Name1`, control is diverted with a `JUMP_UNCONDITIONAL` to the code for `Name3` at offset $27 + 7 = 34$, which otherwise follows directly from the code to read `Name2`. Following `Name3`, we read an end tag, and return control to the caller.

2.4 Content Model Compilation

The content model compilation engine which iScreamer shares with the XML Screamer compiler uses an an approach in which the parser code is generated directly from the abstract schema components [9]. In contrast with methods where the schema is first translated to a more traditional grammar structure, such as a finite state machine, this approach allows the compilation engine to support the full range of XML Schema content model constructs without suffering from the growth in space requirements (in either the compiler or the runtime) associated with the translation of the schema components. Direct reference to the schema components also allows the compilation engine to leverage the determinism guarantees of XML Schema to simplify the generated code.

The core concept of the compilation technique is the *component template*. The validation logic is produced directly from the schema component model, using component-specific code templates for the various components in the schema. The templates are combined together mirroring the composition model of XML Schema content model components (e.g. the template for a repeated particle emits code to repeatedly evaluate the code produced by the term's template). This method was shown to produce simple, efficient code for a wide variety of schemas, with a predictable, stable relation between schema complexity and generated code size. [9]

In iScreamer the standard source-code templates for the schema components are replaced with equivalent bytecode templates. The composition model described above, and the scope of support for schema constructs remains unchanged, with only the target language being changed to a bytecode form. Because the instruction set of the interpreter is significantly simpler than a fully general source code language, such as C or Java, and because the instructions themselves are tailored to the purpose of XML processing, the infrastructure required to generate the bytecode form is simpler. The bytecode form itself is also terser, and therefore more efficient to produce, and more compact when materialized on disk.

2.5 Bytecodes Discussion

Because of the constrained nature of XML validation, the entire execution plan can be built using these few, coarse grained primitives, most corresponding to hundreds of instructions in a traditional general purpose language such as Java. The scanning and validation steps embodied by the instructions of the execution plan can be seen to be identical to those required of any XML validator, regardless of the embodiment of the steps themselves. By ensuring that the same opportunities for optimized, context-sensitive scanning (i.e. scan for exactly one tag, or scan a given simple type) are available to the interpreter, we may further assume that the actual work carried out by these primitives will be equivalent to the work required for a source-code compiled XML validator. The remaining overhead, then, can be credited to instruction dispatch overhead, which is proportional to the number of instructions. While a source-code compiled parser avoids this added runtime cost, it forfeits the many benefits of the bytecode-compiled approach. The coarse granularity of our instruction set keeps this overhead to a minimum, making the tradeoff more attractive. In the next section we discuss the costs and benefits of this tradeoff.

3. EVALUATION

Grammar-specific parser generation techniques have been shown by several authors [3, 7, 9, 10, 15] to significantly improve XML parsing performance. It is our belief, however, that for most real-world use-cases, from individual adhoc scenarios to enterprise-scale business computing environments, the increased tooling and deployment complexity of the compilation model undermines the ease of use and reliability of XML-based technology. The iScreamer system addresses this problem with a high-performance interpretive parser that takes advantage of many of the optimizations of the best source-code compiled solutions, without the tooling and management overhead of code generation and source compilation. In the following two sections we explore the effectiveness of the iScreamer system at overcoming these obstacles to usability, and its success in meeting the performance challenge set by solutions that generate source code or native code.

3.1 Usability

In actual deployments, standard parser-generators pose several usability problems. The most obvious obstacle to the usability of source-code generated parsers is the need for a source-language compiler to compile down to native code. In environments hosted within another application, such as a database or a web server, access to system compilers may be impractical. In many cases, users will not have or be familiar with a source-language compiler for their system. Even in cases where compilers are available, the performance of the generated parser is dependent on the quality of the available compiler, and the care with which it is configured. This tends to incur the kinds of support difficulties normally encountered with source-code delivered products, and not normally acceptable to everyday users and vendors.

Code generation also presents a trust and reliability issue. With XML technologies becoming increasingly more central to business computing, the difficulty of verification and quality control of the generated code becomes a serious problem. Through use of an extremely limited instruction set, iScreamer makes it possible to shift the issue of quality control back to the product itself, rather than the code that it generates, greatly simplifying the task of ensuring reliability. Additionally, the risks of malicious or accidental errors are greatly reduced in a system with limited capability, when compared to running an arbitrary piece of code as a parser.

Finally, source-code generation burdens the user with the management of compilation artifacts. For *ad hoc* deployments, this is likely to undermine the very reasons for adopting XML technologies in the first place. In environments with significant management infrastructure (such as a web server, or database), this management role must be fulfilled by the hosting application. This introduces dependencies between the hosting program and the schema compiler, which complicate interoperability. Furthermore, in very large-scale deployments, the artifacts themselves may become too numerous or too large to manage practically.[3]

One way to mitigate some of these concerns is to have the compiler directly generate native code. Unfortunately, such a strategy is quite complex to implement and does not relieve most of the concerns mentioned above. Verification, reliability and security issues remain, as well as most management issues. The compilation engine is necessarily significantly more complex because it must generate a lower level language, and because it must handle all of the optimizations that would otherwise be performed by the source language compiler, with one backend per runtime platform. Furthermore, direct generation of native code makes the task of maintenance and debugging much more difficult.

Furthermore, XML is increasingly being viewed as a feature of the network infrastructure, rather than an application technology. An interpretive parser provides the experience that users expect from XML. The iScreamer interpretive schema compiler effectively straddles the divide between source-code generation, and classic interpretation, yielding a balance of high-performance, and high usability.

3.2 Performance

In this section we detail benchmarks of iScreamer, and analyze performance in the context of real-world deployments. Our measurements show that iScreamer achieves parsing performance in the same regime as source-code compiled solutions, while maintaining the simplicity and cross-platform convenience in tooling that has made XML the success it is today. The iScreamer system delivers a mixture of performance and maintainability that is appropriate across the whole spectrum of performance-sensitive deployment scenarios.

Our benchmarks are intended to model, in spirit, production quality Web Service deployments of XML, with each test instance consisting of a single UTF-8 XML entity stored in contiguous buffer memory. Filesystem or network overhead is not measured. The tests reported here were run on an IBM eServer xSeries Model 235 with a 3.2 GHz Intel Xeon Processor, and 2GB of main memory, using Microsoft Windows Server 2003 Service Pack 1. XML Screamer parsers, and the iScreamer interpreter were compiled with Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077.

3.2.1 Systems compared

For the performance measurements below, we compare the iScreamer system to four baseline systems. As a measurement of industry standard validation performance, we use the Xerces [1] parser. Xerces is a standard interpretive validator, with schema caching capabilities. In terms of deployment complexity, it represents the simplest validating parser presented, as it has no runtime artifacts of any kind, and conforms to widely used standard APIs. For comparison, we also measure performance of Xerces without validation. Both measurements of Xerces use the SAX [12]

[3]In practice, all of these issues prevented IBM's mainframe database product from implementing compiled validation technologies. The approach outlined in this paper was shown to successfully mitigate the complexity issues, while significantly improving validation performance, and the technology was adopted for use in DB2 9 for z/OS.

Test Case	Schema Filename	Schema Size (Bytes)	Instance Filename	Instance Size (Bytes)	Number of Bytecodes
1	po.xsd	2094	po.xml	990	87
2	ipo.xsd	4240	ipo.xml	1406	154
3	MI_AUS_RESPONSE2_1.xsd	6912	mismoResponse1.xml	1572	126
4	po.xsd	2094	8kpo.xml	8062	87
5	ipo.xsd	4240	8kipo.xml	8077	154
6	bibteXML.xsd	29121	bibtex1.xml	8609	3375
7	po.xsd	2094	64kpo.xml	63754	87
8	ipo.xsd	4240	64kipo.xml	64233	154
9	periodic_table.xsd	4564	periodic.xml	116506	284
10	play.xsd	2375	much_ado.xml	202031	326

Table 1: Test Cases, their sizes, and sizes of the generated instruction set

API. We used Xerces 2.6 for Windows with performance enhancements targeted for inclusion in version 2.7.

As a benchmark of non-validating performance, we use Expat [4]. Expat is a high-performance parser that checks only XML well-formedness of the input. Expat uses its own specialized API, and is significantly faster than both Xerces modes, with and without validation. While it does not make use of a standard API, Expat is widely used, and well-known for its performance capabilities. We used Expat 1.95.8 for Windows.

For a representative benchmark of source-code compilation, we use the XML Screamer [7, 9] parser generator, on which iScreamer is partly based. XML Screamer is designed for extremely high performance, and supports the same range of schema features as iScreamer. For each input schema, it generates a fully custom native-code parser, which is then compiled with a native language compiler. XML Screamer demonstrates the highest performance of any of the parsers measured, and is considered the performance target for this work. Furthermore, it is worth noting that XML Screamer and iScreamer share many aspects including many scanning primitives, and many features of the compilation engine. As such, comparisons with XML Screamer are particularly apt, in that they may be considered to highlight the performance tradeoff inherent to the iScreamer interpreter.

Both XML Screamer and iScreamer were tested in two configurations, one which renders SAX events, and the other producing only a validity result. In XML Screamer the SAX API is chosen at compile time, and compiled directly into the generated parser source code. In iScreamer, generation of SAX events is a runtime option on the interpreter that alters the semantics of the parsing bytecodes to throw the relevant SAX events as a side effect of parsing the document.

The SAX measurement is included for comparison against the other parsers, and Xerces in particular, which is measured using the same API. The *NoAPI* result is included to highlight the raw validation performance, in contrast to the overhead of the standard, but less efficient, SAX API.

3.2.2 Test Cases

To test performance, we use a range of schemas and instances which, combined, form a ten-piece suite of test cases. The *PO* schema is taken directly from the XML Schema Primer [5], and is used in three of the test cases, with varying instance sizes. The *IPO* schema is based on the same purchase-order schema from the XML Schema Primer, and inspired by the sections describing an international purchase-order schema. The IPO schema includes use of namespaces, dynamic typing and element substitution-groups, and may thus be considered a more rigorous benchmark than the standard purchase-order. Instances of varying length for both schemas are generated by expanding repeated sections with duplicate copies. The remaining schemas and instances are taken from the Sarvega XML Validation Benchmark [11]. The details of the test cases are given in table 1, where each case is assigned a number.

3.2.3 Measurements

Performance measurements were taken for each of the parsers detailed above, on each of the test-cases in table 1. The parsers were warmed up before taking measurements, although priming them seemed to have no sizeable effect when measuring large numbers of iterations. Indeed, our results demonstrate that throughput is mostly independent of instance or schema size, except for very small documents and documents with very little markup. The results for all of the test cases and configurations are given in Table 2.

Figure 2 shows a comparison of throughput across all of the test cases, for each of the validating parsers that were measured (Xerces, XML Screamer and iScreamer). For the purposes of this broad comparison, we show SAX performance, as the best point of comparison with Xerces. As expected, XML Screamer shows the best performance, several times faster than Xerces. The performance of iScreamer, however, can be seen to be quite comparable to that of XML Screamer, across the whole range of test cases. This result demonstrates that interpreter overhead is indeed slight, with respect to the performance advantage that compilation yields.

In Figure 3, we present a more detailed comparison of one test case, namely the moderately sized IPO instance (8kipo.xml), test case 3. Here, we show the performance of all of the measured configurations. As in Figure 2, XML Screamer and iScreamer are seen to be significantly faster than Xerces. They are also much faster than either Expat or Xerces in non-validating mode.

In the *No API* case, iScreamer exhibits some slowdown with respect to XML Screamer, indicating that the overhead of interpretation is not completely negligible with respect to the execution time of the instructions themselves. With the cost of rendering SAX events included, however, it becomes clear that this overhead is minimal compared to the cost of rendering the SAX API. In essence, both XML

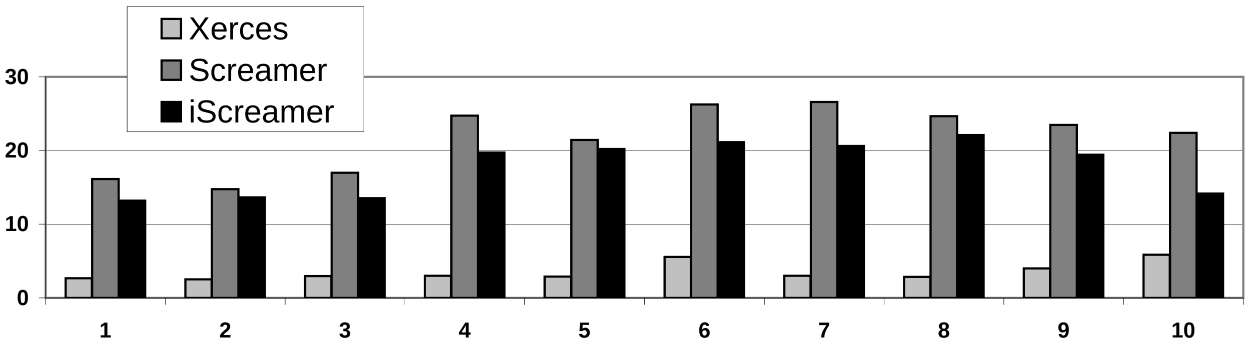

Figure 2: Comparison of Validation Throughput (SAX) in $MB/GHz \cdot sec$

Screamer and iScreamer are operating in a regime in which other computations such as UTF-16 transcoding dominate. This observation in part validates the use of XML Screamer as a performance target, and demonstrates that the real-world performance of iScreamer is likely to be almost indistinguishable from that of the best source code compiled solutions.

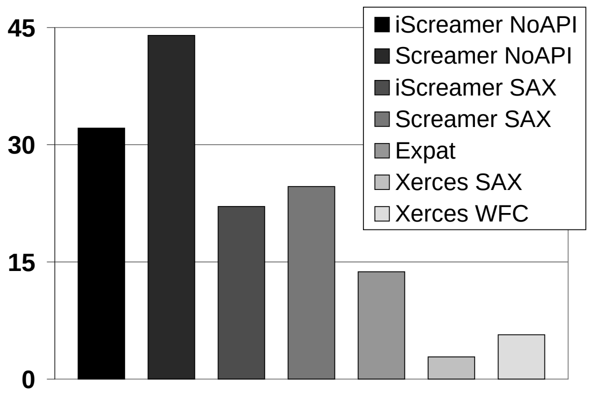

Figure 3: Test Case 3: Throughput Comparison of Validating & Non-Validating Parsers in $MB/GHz{\cdot}sec$

4. RELATED WORK

There have been many attempts to improve the performance of XML validation. Chiu et al. [3] extend DFAs to nondeterministic generalized automata and describe a technique for translating these into deterministic generalized automata, from which a parser can be generated. These parsers operate on a byte-level, performing well-formedness checking and validation concurrently. As a tradeoff for the unification of content-model validation and byte-level scanning, however, the technique is limited to small occurrence constraints because the construction of deterministic generalized automata from nondeterministic generalized automata causes a multiplicative blowup in the number of states. Further, the DFA model has difficulty supporting xsi:type. The global optimizations made possible by this and related methods seem insufficient to mitigate the lack of support for commonly used features of XML Schema.

While generalized automata could be used to drive an XML-specific bytecode interpreter much like iScreamer, the majority of the predicates and actions used in Chiu et al. are specified at a much finer granularity than the primitives used by iScreamer. As a result, more instructions would be required to lay out the execution plan. As instruction dispatch overhead is proportional to the number of instructions, we expect that interpreting the actions and predicates directly would lead to a large performance penalty.

Several authors [10, 15] avoid limitations of the generalized automata technique through the use of a two-level approach. Cardinality-constraint automata [10] extend deterministic finite automata with cardinality constraints on state transitions. These automata easily handle the encoding of occurrence constraints. Unfortunately, the CCA does not perform well-formedness checking but runs as a separate layer on top of a separate SAX parser, with the associated performance penalty.

Van Engelen [15] uses a two-level schema in which a lower-level FLEX scanner drives a DFA validation layer. The two-level approach resembles the separation in iScreamer between scanning and content-model validation, but does not allow the validation engine to drive the scanner with schema knowledge, either in scanning tags, or in scanning simple data and whitespace. Instead, the same Flex [6] scanner is used for every byte of the input, regardless of context.

XML Screamer, on which our work partly builds, demonstrates a parser generator that translates XML Schemas into a parser in C source code. The architecture of the schema compiler is discussed in detail in one paper [9], and the architecture of the generated parser code and how it achieves performance is explained in a second paper [7]. The differences between our work and XML Screamer are discussed in more detail in Section 2 of this paper.

Many other attempts to improve XML performance have focused on changing the form of the XML. Examples include the various proposals for binary XML forms. Unfortunately, binary XML has a disruptive effect on the user in terms of usability, interoperability, and standardization, the core strengths of XML. For a thorough analysis of binary solutions see [2]. As the other efforts cited increase the performance of XML solutions, they may correspondingly reduce the number of cases in which a binary XML is required.

	Throughput ($MB/GHz \cdot sec$)							SAX Comparisons with iScreamer (Percentage Difference)			
Test	Xerces		Screamer		iScreamer		Expat	Screamer	Expat	Xerces	
Case	WFC	Val	No API	SAX	No API	SAX	WFC			WFC	Val
1	4.41	2.65	35.08	16.12	21.2	13.22	6.85	-18	+93	+200	+398
2	4.24	2.51	23.23	14.76	20.55	13.64	6.81	-8	+100	+222	+444
3	3.21	2.98	25.95	17.00	19.14	13.55	5.21	-20	+160	+322	+355
4	6.79	3.01	48.00	24.73	32.52	19.73	13.68	-20	+44	+191	+556
5	6.08	2.90	38.40	21.44	30.31	20.22	10.31	-6	+96	+232	+597
6	8.28	5.58	47.49	26.25	29.65	21.14	15.54	-19	+36	+155	+279
7	6.88	3.02	49.87	26.58	33.53	20.63	15.65	-22	+32	+200	+583
8	5.68	2.85	44.00	24.65	32.13	22.09	13.75	-10	+61	+289	+675
9	6.03	3.99	34.61	23.47	28.18	19.44	15.25	-17	+27	+222	+387
10	8.20	5.85	42.41	22.41	24.69	14.17	19.10	-37	-26	+73	+142

Table 2: Performance results

For an analysis of more work related to XML performance and compilation of XML Schema for increasing XML parsing performance, see the section on related work in our XML Screamer paper. [7]

5. CONCLUSION

Through the use of a carefully tuned set of special-purpose bytecode instructions, we have shown that the performance advantages of schema-based XML parser generation can be achieved in an interpreter. The technology is demonstrated with iScreamer, a schema compiler and interpretive parser that make use of these bytecodes. In a series of performance tests, iScreamer is shown to perform within 20% of high-performance generated parsers, and significantly faster than any widely available parsers, both validating and non-validating.

The bytecode instruction set, specialized to the task of XML processing, is designed to incorporate the optimization techniques used in source code compilation while maximizing the amount of work done by any given instruction. The interpretive dispatch overhead is thus minimized, while the opportunity for grammar directed optimization remains high.

While maintaining the benefits of compilation, this eliminates the need for deployment-time source code management and compilation. By using a compact bytecode stream, it further eliminates the large space requirements typical of generated artifacts. The management complexity is thus reduced to a level similar to conventional parsers that have no compile step at all. Using this approach, the combined benefits of high performance and simplified management provide a viable solution for XML validation throughout the web.

6. ACKNOWLEDGMENTS

Joe Latone managed this project and contributed to it greatly. We would also like to thank Joe Kesselman, David Marston, Chet Murthy, Dennis Quan, and many others.

7. REFERENCES

[1] The Apache Foundation. *Xerces.* http://xml.apache.org.

[2] R. J. Bayardo, D. Gruhl, V. Josifovski, and J. Myllymaki. An evaluation of binary XML encoding optimizations for fast stream based XML processing. In *World Wide Web Conference*, May 2004.

[3] K. Chiu and W. Lu. A compiler-based approach to schema-specific XML parsing. In *First International Workshop on High Performance XML Processing*, May 2004.

[4] J. Clark. *Expat XML parser.* http://expat.sourceforge.net/.

[5] D. C. Fallside and P. Walmsley, editors. *XML Schema Part 0: Primer Second Edition.* W3C, second edition, Oct 2004. http://www.w3.org/TR/xmlschema-0.

[6] The GNU Project. *Flex.* http://www.gnu.org/software/flex/.

[7] M. Kostoulas, M. Matsa, N. Mendelsohn, E. Perkins, A. Heifets, and M. Mercaldi. XML screamer: An integrated approach to high performance XML parsing, validation and deserialization. In *15th International Conference on World Wide Web (Edinburgh, Scotland, May 23 - 26, 2006)*, pages 93–102, New York, NY, May 2006. ACM Press.

[8] J. K. Ousterhout. *Tool Command Language.* http://www.tcl.tk/.

[9] E. Perkins, M. Matsa, M. Kostoulas, A. Heifets, and N. Mendelsohn. Generation of efficient parsers through direct compilation of XML schema. *IBM Systems Journal*, 45(2):225–244, 2006.

[10] F. Reuter and N. Luttenberger. Cardinality constraint automata: A core technology for efficient XML schema-aware parsers. http://www.swarms.de/publications/cca.pdf, 2003.

[11] Sarvega, Inc. *XML Validation Benchmark.* http://www.sarvega.com/xml-validation-benchmark.html.

[12] saxproject.org. *SAX: Simple API For XML.* http://www.saxproject.org/.

[13] Sun Microsystems, Inc. *Java Technology.* http://java.sun.com/.

[14] H. Thomson, D. Beech, M. Maloney, and N. Mendelsohn, editors. *XML Schema Part 1: Structures.* W3C, second edition, Oct 2004. http://www.w3.org/TR/REC-xmlschema.

[15] R. van Engelen. Constructing finite state automata for high-performance XML web services. In *International Conference on Internet Computing*, 2004.

[16] L. Wall. *Practical Extraction and Report Language.* http://www.perl.org/.