XML Screamer: An Integrated Approach to High Performance XML Parsing, Validation and Deserialization

Margaret G. Kostoulas, Morris Matsa, Noah Mendelsohn, Eric Perkins, Abraham Heifets
IBM Corporation, One Rogers Street, Cambridge, MA 02142 USA
{mgg,mmatsa,noah_mendelsohn,perkinse, aheifets}@us.ibm.com

Martha Mercaldi
University of Washington
mercaldi@cs.washington.edu

ABSTRACT
This paper describes an experimental system in which customized high performance XML parsers are prepared using parser generation and compilation techniques. Parsing is integrated with Schema-based validation and deserialization, and the resulting validating processors are shown to be as fast as or in many cases significantly faster than traditional nonvalidating parsers. High performance is achieved by integration across layers of software that are traditionally separate, by avoiding unnecessary data copying and transformation, and by careful attention to detail in the generated code. The effect of API design on XML performance is also briefly discussed.

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.* I.7.2 [**XML]**

General Terms
Performance, Experimentation, Standardization, Languages.

Keywords
XML, XML Schema, performance, validation, parsing, schema compilation, JAX-RPC, SAX.

1. INTRODUCTION
XML [21] is widely accepted as a means of exchanging structured information on the Web and in other software systems. By explicitly tagging information with named *elements* and *attributes*, XML enables the creation of documents that are to a significant degree self-describing, offering the promise of more robust information sharing between loosely coupled organizations and systems. An application processing an XML document can use such element and attribute markup to identify particular information items and to detect some classes of errors in document content.

Although the performance of XML has been adequate for many important purposes, processing speed unfortunately remains a problem in more demanding applications. Some limitations are inherent in core features of XML: it is text based, flexible in format, and carries redundant information. A key goal of the work described here is to show that with careful attention to processor implementation, API design, and application integration, XML can in fact be processed much more rapidly than common practice would suggest.

XML Schema validation [23] provides a degree of automated error checking for XML applications. Due to slow performance, validation is typically applied during debugging and testing if at all, and is often disabled in production systems. We seek to show that validation can be achieved with negligible overhead, even relative to the faster processing promised above; with such performance, XML Schema will indeed become practical for improving the robustness of the loosely coupled systems that XML was designed to enable.

In this paper we analyze a variety of architectural considerations relating to the design of high performance XML systems, and we report on the implementation and performance of an experimental prototype implementation. Known as XML Screamer, our prototype compiles customized validating XML parsers from an XML Schema. The generated parsers, which can be in either C or Java, leverage optimizations that are integrated across processing tasks that in many traditional systems are separate, i.e., scanning, parsing, validation, and deserialization.

2. HARDWARE PERFORMANCE
No parser can process input faster than its supporting hardware accesses data, but the additional cost of parsing and validation should be minimized. On a 1 GHz Pentium processor a simple character-scanning loop runs at about 100 Mbytes/second, which is 10 cycles/byte. As shown in Section 8.2, traditional validating parsers perform in the range of 2.5–6 Mbytes of input per second or 160–400 cycles/byte, a penalty of between 16x and 40x — intuition suggests that this cost can be greatly reduced.

Achieving better performance obviously requires lowering the number of bytes manipulated and/or reducing the work done per byte. Parsers operate not just on their input, but also on output structures, and notably on extra copies of data resulting from format conversions, such as UTF-8 to UTF-16. Any need to repeatedly process or scan the same data adds overhead proportionally. Accordingly, our strategy is to minimize unnecessary data copying and transformation, and to ensure that most input and output data is accessed just once. To do this, we optimize across software layers that are traditionally separate.

WWW 2006, May 23–26, 2006, Edinburgh, Scotland.
ACM 1-59593-323-9/06/0005.

3. OPTIMIZING ACROSS LAYERS

A key inspiration for XML Screamer was the 1987 work of Watson and Mamrak [20], who report on the optimized implementation of layered network protocols. Among many other important insights, they offer one that is particularly pertinent for XML processing: "a common mistake is to take a layered design as a requirement for a correspondingly layered implementation.".

Consider the use of a SAX-based validating parser to construct business objects such as those provided by gSOAP [18], JAX-RPC [6], etc. A deserializer, customized to the particular XML documents to be processed, receives SAX events from the parser (Figure 1). In this example, deserialization requires that a single binary integer field be computed from the information in a UTF-8 encoded XML instance:

Schema:

```
<xsd:element name="inventoryItem">
  <xsd:sequence>
    <xsd:element name="quantity">
      <xsd:simpleType
           base="xsd:integer">
         <xsd:maxInclusive="10000"/>
         <xsd:minInclusive="0"/>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:element>
```

Instance (in Unicode UTF-8 encoding):

```
<inventoryItem>
   <quantity>10</quantity>
</inventoryItem>
```

Output business object structure:

```
class inventoryItem {
  int quantity;
};
```

The schema requires a root element `<inventoryItem>` and a child element `<quantity>`. The content of the latter is constrained to be an integer in the range from 0 to 10,000. The XML input results in a single business object of class `inventoryItem` , which in turn contains a single `int` with the quantity. Note that the quantity is conveyed in character form in the XML, but is stored in the output as a binary integer.

A conventional parser and deserializer would likely perform the steps listed below. Steps labeled "P" are performed by the parser, and those labeled "D" by the deserializer. Note that in typical SAX implementations, both XML element names and character data are presented to applications as UTF-16 strings:

1. (P) Convert start tag string "`inventoryItem`" to UTF-16.
2. (P) Validate the resulting UTF-16 string against the expected element name "`inventoryItem`". (Note that in this case, which is common in Western cultures, the UTF-16 strings are longer than the corresponding UTF-8 forms, and take longer to compare.)
3. (P) Throw a SAX event from parser to deserializer, to signal the element start.

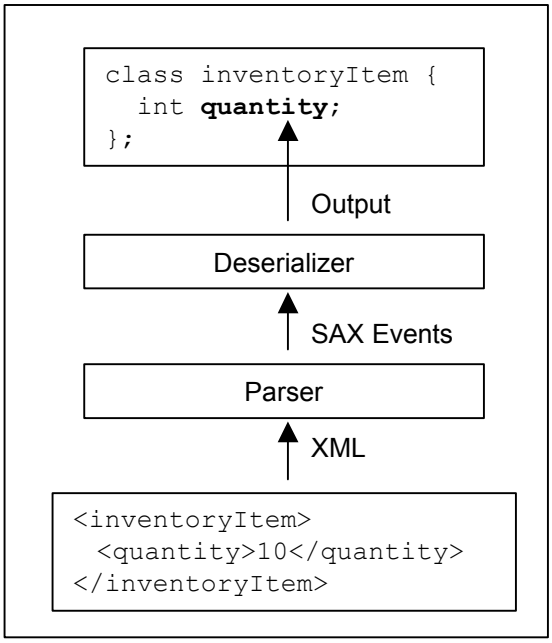

Figure 1: Conventional processor layering

4. (D) Verify that the element name in the SAX event is "`inventoryItem`" (Note that the validator already checked this in step 2.)
5. (D) Do a "new" for the output structure.
6. (D) Discard the SAX event.
7. (P) Convert start tag string "`quantity`" to UTF-16.
8. (P) Validate the resulting UTF-16 string against the expected element name "`quantity`".
9. (P) Throw a SAX event signaling the element start.
10. (D) Verify that the element name in the SAX event is "`quantity`" (already checked in step 8.)
11. (D) Save state to indicate that the next value received is to be stored as the quantity.
12. (D) Discard the SAX event.
13. (P) Convert the two UTF-8 characters "`10`" to UTF-16, resulting in 4 bytes.
14. (P) Convert the same two characters to a binary integer (Many implementations do the conversion using as input the 4 byte UTF-16 form computed in the step above, rather than the smaller UTF-8 two byte sequence in the input.)
15. (P) Verify that the integer is between the expected bounds of 0 and 10,000.
16. (P) Throw a SAX event with the UTF-16 representation of the two characters "`10`".
17. (D) Convert the two character sequence to a binary integer (the 2nd time this conversion has been done…see step 14.)
18. (D) Copy the integer to the quantity field of the output.
19. (D) Discard the SAX event.

The above illustrates why traditional parsers and deserializers are significantly slower than one might expect given the capabilities of the underlying hardware.

Leveraging Watson and Mamrak's guidance to optimize across layers, XML Screamer generates a processor specialized to the expected form of the input and to the required output API (see Figure 2). The code for low-level character scanning, parsing, validation, and deserialization is completely integrated, so it is no longer meaningful to label steps as (P) or (D). Indeed, use of SAX events is in this example eliminated entirely, since they require data transformations and are unnecessary for the generation of the desired output. As in the list above, we concentrate on the processing of element names and data content, skipping details of other checking (such as looking for "<" or ">" element delimiters), and again we omit steps related to end tag checking.

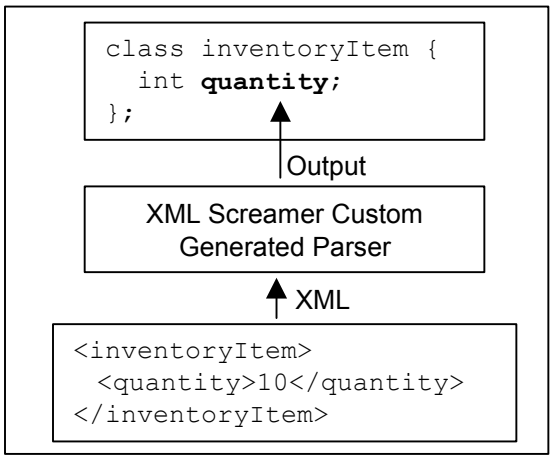

Figure 2: XML Screamer layering

The generated parser performs the following steps:

1. Compare the UTF-8 start tag "`inventoryItem`" to the "`inventoryItem`" name required by the schema. The comparison is performed directly against the instance input buffer, so no data copying is required.
2. Do a "new" for the output structure.
3. Compare the UTF-8 start tag "`quantity`" to the "`quantity`" name required by the schema. Again, no data copying is required. The parser is recursive descent, so the "state" corresponding to step 11 in the previous list is implicit in the program counter.
4. Verify that the following two UTF-8 characters are a legal lexical form for an xsd:integer, and during that same check compute the binary integer value 10. I.e. each character is retrieved, verified to be a digit, and applied to the polynomial computation of the resulting binary integer.
5. Check the resulting integer against the bounds 0 to 10,000.[1]
6. Copy the integer to the output structure.

Far fewer steps are involved, and those that remain are in general less expensive than their counterparts. At least in this ASCII-based example, the UTF-8 string comparisons involve half the number of bytes of their UTF-16 equivalents. Much less data is copied or transformed, so locality is improved, and the processor cache is likely to be used more effectively. Object or memory buffer allocations that were necessary for the UTF-16 strings are also avoided. Indeed, in XML Screamer, string pools with expected element names are typically prepopulated at compile time, thereby minimizing object creations and increasing locality. The overhead of SAX event creation and related object management is entirely eliminated.

4. CONFIGURABLE API SUPPORT

Although the above example highlights some drawbacks of using SAX as an intermediate form, SAX is popular and can be an excellent choice for interacting with a broad range of SAX-based tools. XML Screamer includes an API generation framework which is configurable to support a variety of output APIs, including JAX-RPC for Java and similar business object APIs for the C language, SAX, as well as others specific to particular applications.

Indeed, XML Screamer provides compiler-based optimizations for SAX. Advance knowledge of the schema allows XML Screamer to precompute data for SAX events that are invariant from one instance to another. For example, using the schema just shown, every instance invariably results in startElement and endElement events for both `inventoryItem` and `quantity` tags. XML Screamer can do much of the work to prepare such events at compile time. In summary, the selection, layering and optimization of processor APIs has a crucial impact on performance. XML Screamer is designed to optimize performance of SAX, of business objects, and of other specialized APIs.

5. OTHER CUSTOMIZATIONS

Other parameters in addition to the XML Schema have proven useful as input to compilation. For example, our users often have advance knowledge that a particular Unicode encoding such as UTF-8 will be used for input documents. We provide to the compiler information about encodings to be supported, and generate code that is tailored accordingly. As shown in Section 3, low level scanning, tag checking, and simple type validation can often be done directly in the input encoding; when the schema and the instance are provided in different encodings, XML Screamer converts tag names from the schema to the expected instance encoding at compile time. The resulting converted strings are stored in string pools, or directly in the generated tag name comparison code.

6. DESIGN OF THE COMPILER

This paper is concerned primarily with performance, and details of the XML Screamer compiler have been documented elsewhere [11]. Accordingly, we give here a very brief overview of the compiler and then discuss in more detail the design and performance of the generated parsers.

The XML Screamer compiler is written in Java, and is capable of producing parsers in both Java and C. Schemas are read into the compiler using the `org.apache.xerces.xs` package [1], which supports the XML Schema API [24], and which does the work of composing one or more schema documents. This Xerces-based tool resolves XML Schema `xsd:import`, `xsd:include`, and `xsd:redefine` constructs, so these are fully supported by XML Screamer. The result is a connected graph of Java objects, each of which corresponds to a *component* of the schema to be compiled. Components are formally defined

[1] See prototype limitation discussed in footnote to Section 9.

in the W3C XML Schema Recommendation [23]; informally, there is a component, and thus a compile-time Java object, for each simple or complex type, each element declaration, each attribute declaration, and so on.

The code generation phase of the compiler consists of visiting the Java object corresponding to each schema component, and generating the corresponding validation code. Each such compiled component is invoked repeatedly at runtime if, for example, more than one element or attribute has the same simple or complex type. Code to populate the required output API, such as SAX or JAX-RPC, is generated along with the code for validation, and wherever possible duplicate work is avoided. In the example from Section 3, the binary form of the `quantity` value was needed both in validation and for output in the generated JAX-RPC object. By generating validation code and output code together, such sharing becomes straightforward.

Often, certain information that will be required at runtime is known statically from a schema component or other compile-time information. In the same example, the `<inventory>` and `<quantity>` tags are known to appear in all valid instances. Code generation logic has the option to precompute output data structures at compile time, to store in the generated code UTF-16 or other converted forms of strings, to prepopulate string pools, and so on; the precomputed SAX event optimizations discussed in Section 4 are achieved in this manner.

Unlike many retargetable compilers for traditional programming languages, XML Screamer does not generate a register- or stack-based pseudo-machine code. Indeed, early versions of Screamer had such an intermediate representation and we did not find it helpful in generating the low-level optimizations required for parsing and validation. Instead, we have created language-specific templates and code generators separately for Java and for C. Similarly, the compiler provides common hooks useful to support code generation for a wide range of runtime APIs, but the code for each such API is hand crafted; the Screamer compiler itself is modified whenever a new parser API is to be supported.

7. DESIGN OF CUSTOMIZED PARSERS

The performance of XML Screamer is determined primarily by the techniques used in the generated parsers. The sections below discuss the design of these parsers and some optimizations used to speed XML processing.

7.1 Recursive Descent Parsing

XML Screamer generates recursive descent parsers in which a subroutine is invoked for each complex type (i.e., each instance element) to be validated. Overall, we have found this to be a reasonable tradeoff: the generated code has a clean structure which is isomorphic to the compiled schema, and at runtime the state of the parse is efficiently captured by the program counter and invocation stack.

In earlier versions of XML Screamer we experimented with LALR-based parsers. LALR is fast and easily captures the element structure of most complex types, but it does not deal well with the complexities of conformant namespace and `xsi:type` [23] processing (see Section 7.5). Löwe et al. [8] describe a system based on LALR(1) and LL(1) parser generators, but it apparently does not support those XML and schema features. On balance, we believe that recursive descent for elements coupled with carefully optimized start tag handling (Section 7.5), is an effective compromise for XML parsing and schema-validation.

7.2 Inline vs. Subroutine Code Generation

Having settled on a recursive descent design, we experimented with explicitly inlining some of the smaller validation routines, and found in most cases that the overhead of an out of line call is negligible relative to other work to be done; unlike low level character scanning, element validation is a relatively coarse grained operation. Low-level character scanning and testing is indeed more performance critical and must be inlined for maximum speed.

7.3 Minimizing Backtracking

As discussed in Section 2, parsing and validation of XML can be optimized by minimizing redundant scanning of input data. Accordingly we adopt a design principle for our scanners that, in the typical case, each input character is "visited" just once. If several tests are needed on a particular character, for example to determine whether it is an angle bracket `">"` or an expected element name character, then all such tests are performed before the next character is inspected. If the character contributes to the output or to computation of some binary value, then to the extent possible such side effects are handled at the same time the character is inspected for validity. A similar philosophy applies to output generation: to the extent practical, information is precomputed at compile time ("zero" visits), and in the remaining situations we attempt to update the value of each output byte exactly once.

String pools for declared element and attribute names are precomputed at compile time, using small integer string pool handles that are directly applicable as array indices. During validation, the local name of an element in a start tag is checked against or added to a string pool, and the resulting handle is used to index a table of element declarations. Similar pools are used at both compile time and runtime to manage namespace prefixes and namespace URIs, and to index types named by `xsi:type`. String pools help to minimize repeated scanning of string data.

Although the ideal of visiting each character just once cannot always be achieved, we have found it to be a useful yardstick against which to evaluate proposed optimizations: those that greatly reduce the number of bytes manipulated are likely to be significant. Furthermore, we believe that the "single visit" design results in excellent locality of reference, and thus tends to use processor memory hierarchies efficiently. Even if multiple tests on a single character are needed, that character tends to be loaded into a processor register just once, and manipulated there. Nearby portions of the input buffer are likely to be in first-level processor caches, so access to successive characters is usually fast. String pools and other compact data structures used to facilitate sharing have high utilization rates, and are also likely to be cached effectively.

7.4 Optimized Simple Type Validators

In XML Screamer, simple type validators such as those for `xsd:integer` are integrated with the parsing framework, and operate directly on the input buffer. Accordingly, separately optimized validators are provided for UTF-8, UTF-16, or for any other required encodings. Additional validators are available to optimize particular common cases. Consider for example the parsing of the `xsd:integer` "10" in Section 3. Both the API and a bounds check in the schema require the binary form of the number 10. Accordingly, XML Screamer generates an integer validator that computes the binary value while scanning, evaluating the necessary polynomial as each character is

retrieved. Also available is a slightly faster integer scanner that is used to validate unconstrained integers, and in other cases where the converted value is not required. The XML Screamer compiler chooses the appropriate form to use in each case.

In some cases, optimistic simple type validation algorithms are used. Consider, for example, the parsing of comments within an `xsd:integer`:

```
<e>123456</e>
<e>123<!-- comment -->456</e>
```

Although both of these represent the same integer, the second form is very uncommon. Accordingly, XML Screamer does most integer validation with a very fast scanner that accepts only digits, falling back to a more general validator if the initial parse fails. Such reparsing violates the guideline that each input character be visited just once, but in practice such revalidation is rare, and usually only a few characters are involved.

7.5 Optimized XML Start Tag Processing

XML Namespaces [22] and `xsi:type` are significant challenges for a high performance XML validator. Consider the following XML instance:

```
<n1:e1 xmlns:n1="http://example.org/ns1"
       xmlns:n2="http://example.org/ns1">
   <-- Following element has a name and
       a type that are known only after
       27 attributes and a namespace
       declaration are parsed -->
  <n2:e1 xsi:type="n2:type"
         a='1' b='2'  ...   z='26"
         xmlns:n2="http://example.org/ns2">
  </n2:e1>
</n1:e1>
```

This example is very difficult to validate efficiently. As the second tag is encountered, prefix `n2` is still bound to `http://example.org/ns1`. Accordingly, the child element appears to have the name {`http://example.org/ns1`, `e1`}, and the `xsi:type` appears to specify {`http://example.org/ns1`, `type`}. Only after scanning through 26 more attributes and reaching the namespace declaration can the parser discover the true values for either name or type, and only once the type is known can the 26 attributes be validated (because the complex type named by `xsi:type` may specify simple types for some attributes.) Thus, the simplest approaches to start tag validation involve at least two passes, one to determine namespace prefix associations and `xsi:type` attributions, and a second to validate the element name and attribute content.

XML Screamer handles even such difficult start tags correctly and efficiently. Indeed, the occurrence of problematic constructions cannot usually be predicted in advance, so the algorithm described here is used for all start tag validation. XML Screamer prepares during its initial scan data structures that support efficient validity checking. The possible local names of all attributes usable in an XML vocabulary are for the most part known from the schema at compile time, and each such name is assigned an offset in a bit vector. During scanning, the corresponding bit is turned on as each attribute of a given name is encountered. Two corresponding bit masks are also precompiled for each complex type, representing respectively the required and allowed attributes for that type. At the end of start tag scanning, after the type of the element has been reliably determined, simple bitmap comparisons are used to ensure that all required attributes are present, and that no disallowed attributes have been provided.

Some backtracking may be required to perform attribute validations, but XML Screamer notes during its initial start tag scan the offset and length of each attribute value, and also the positions of any attribute content whitespace. Once the element's type is determined, only attributes requiring validation are revisited. Note that integration of such optimizations into a single pass using LALR would at best be tricky, perhaps impossible, because the grammar to be used for checking attributes is not known during the initial scan.

7.6 Summary of Generated Parser Features

In summary, the efficiency of generated XML Screamer parsers results primarily from:

- Optimizing across what would traditionally be separate layers of scanning, parsing, validation and deserialization.
- Validating and deserializing directly from the input buffer, in the native encoding of the input document.
- To the extent possible, visiting each input and output character exactly once, or where necessary generating efficient data structures to minimize backtracking.
- Compile-time precomputation of invariant output.
- Carefully optimizing low level character scanning and validation routines, and providing tailored versions for common performance-critical cases.

8. XML SCREAMER PERFORMANCE

The principal measure of success for XML Screamer is the efficiency of the generated parsers. As previously noted, the compiler is capable of generating parsers in both C and Java, but the C parsers are much more carefully optimized, and are a more realistic indicator of the performance achievable using our approach. Accordingly, all results reported here are for C-language parsers.

8.1 Benchmark Methodology

Our benchmarks are intended specifically to model 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, and streaming of large documents is not considered. Modeling the assumption that successive Web Service messages would not in general occupy the same memory buffer, multiple instances are parsed from separate buffers, and the average time is reported. We believe that repeated parsing of the same buffer might give unrealistically high cache hit rates and thus an overoptimistic estimate of performance.

We run each set of tests on several different machine models, and report only results that can be stably reproduced. Although the numbers reported here are for one particular machine, we have thus ensured that they are broadly representative. At one point during our development work we saw an anomalous result on just one model used for testing: a 30% discrepancy in one test was traced by hardware monitoring to differences in processor cache layout. While such problems are rare, they can obscure important results, and tend to be invisible unless multiple machines are benchmarked. The tests reported here were run on an IBM eServer xSeries Model 235 with a 3.2 GHz Intel Xeon Processor,

Table 1: Measurements

Test case			Throughput (Mytes/Sec/ProcessorGHz)							Comparisons			
			Xerces - SAX		Expat	XML Screamer							
						Validating			any Type				
ID	Schema Filename	Size	Non val	Val	Non val	NOAPI	Business Object API	SAX	SAX	Screamer SAX vs Expat	Screamer Business Object vs. Expat	Screamer SAX vs. Xerces Val SAX	Screamer: Schema vs anyType Sax
1	po	990.00	4.41	2.65	6.85	35.08	33.88	16.12	12.75	**2.4x**	**4.9x**	**6.1x**	**1.3x**
2	ipo	1,406.00	4.24	2.51	6.81	23.23	23.56	14.76	14.25	**2.2x**	**3.5x**	**5.9x**	**1.0x**
3	MI_AUS_RESPONSE2_1	1,572.00	3.21	2.98	5.21	25.95	23.21	17.00	7.51	**3.3x**	**4.5x**	**5.7x**	**2.3x**
4	po	8,062.00	6.79	3.01	13.68	48.00	44.67	24.73	16.08	**1.8x**	**3.3x**	**8.2x**	**1.5x**
5	ipo	8,077.00	6.08	2.90	10.31	38.40	34.21	21.44	16.46	**2.1x**	**3.3x**	**7.4x**	**1.3x**
6	bibteXML	8,609.00	8.28	5.58	15.54	47.49	NA	26.25	19.77	**1.7x**	**NA**	**4.7x**	**1.3x**
7	MI_AUS_REQUEST2_1	9,429.00	4.06	3.16	6.74	23.15	NA	17.79	8.52	**2.6x**	**NA**	**5.6x**	**2.1x**
8	po	63,754.00	6.88	3.02	15.65	49.87	46.63	26.58	16.37	**1.7x**	**3.0x**	**8.8x**	**1.6x**
9	ipo	64,233.00	5.68	2.85	13.75	44.00	36.15	24.65	16.67	**1.8x**	**2.6x**	**8.6x**	**1.5x**
10	periodic_table	116,506.00	6.03	3.99	15.25	34.61	35.28	23.47	14.16	**1.5x**	**2.3x**	**5.9x**	**1.7x**

and 2 GB of main memory, using Microsoft Windows Server 2003 Service Pack 1. XML Screamer parsers were compiled with Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077.

8.2 Measurements

Table 1 provides benchmark data for XML Screamer, as well as for two other widely used parsers, Xerces Version 2.6 for Windows[2] and Expat version 1.95.8 for Windows. Xerces was tested in nonvalidating and validating modes using SAX; Expat is not capable of validating, and was tested with its native interface which is similar to SAX, but which reports strings in UTF-8.

For each schema, three separate parsers were compiled by XML Screamer: one that parses and validates but reports no data (NOAPI), a second that populates a business object API as earlier described[3], and a third that supports SAX. The SAX parser includes optimizations for precomputation of SAX events, as discussed in Sections 4 and 6. For each instance, a fourth XML Screamer validation was performed using a schema in which root content is validated using XML Schema wildcards (i.e. `xsd:anyType`). This is the closest approximation to a schema-less mode for XML Screamer, and it is used in Section 8.3.4 to quantify the benefits of SAX precomputation.

Each row in the table represents an XML instance; different instances validated by the same schema are distinguished by size and appear in separate rows. `po.xsd` is the Purchase Order sample schema from the XML Schema Primer [23]. `ipo.xsd` is a similar schema modified to use namespaces and `xsi:type`. Longer instances are constructed by duplicating multiple copies of the same data, wrapping in a container element, and adapting the schema accordingly. Other schemas and instances are taken from the Sarvega XML Validation Benchmark [13].

Performance is quoted in Mbytes/sec/ProcessorGHz, in other words, Mbytes/sec normalized to a 1 GHz Intel processor. Thus for our 3.2 GHz machine, a measured throughput of 3.2 Mbytes/sec is quoted in the table as 1 Mbyte/sec/GHz. We have found this normalization to be both useful and stable in comparing benchmarks run on different model Intel machines.[4] All charts in the text below are based on data from Table 1.

8.3 Performance Discussion

The useful performance achievable with XML Screamer can best be seen in the Business Object API column in Table 1. Throughput ranges from 23.21 to 46.63 Mbyte/sec/GHz, including all overhead for parsing, validation and deserialization. On our 3.2 GHz machine this is an absolute performance of roughly 75 to 150 Mbytes/sec.

A character scan loop on the test processor measured at 106 Mbytes/sec/GHz, consistent with the estimate in Section 2, so XML Screamer is parsing, validating and deserializing at between 22% and 44% of the maximum character scanning speed of the machine.

8.3.1 Comparison with nonvalidating parsers

XML Screamer exceeds the speeds of the fastest parsers in common use today. We first examine SAX parsing, which is the slowest mode for XML Screamer, and therefore the most conservative comparison.

[2] The Xerces parser tested is a version 2.6 code base plus selected performance enhancements targeted for eventual inclusion in version 2.7; the enhancements create a somewhat more challenging comparison for XML Screamer, and in any case are believed to affect results by 10% or less compared to version 2.6. Xerces schema caching is enabled in all validating tests.

[3] Due to limitations of the business object test framework, business object parsers were not generated for test cases 6 & 7. The corresponding measurements are shown as "NA" for not available. As can be seen from other table rows, business object performance is typically close to that of NOAPI, which is shown.

[4] Note, however, that Intel Centrino processors typically exhibit higher throughput per cycle than other Pentiums or Xeons.

Figure 3 illustrates the throughput of XML Screamer relative to the non-validating parsers tested. Both XML Screamer and Xerces are tested using SAX, i.e. with UTF-16 converted strings; Expat is measured through its native UTF-8 interface. XML Screamer is between 1.5x and 3.3x faster than Expat, and between 3.2x and 5.3x the speed of nonvalidating Xerces.

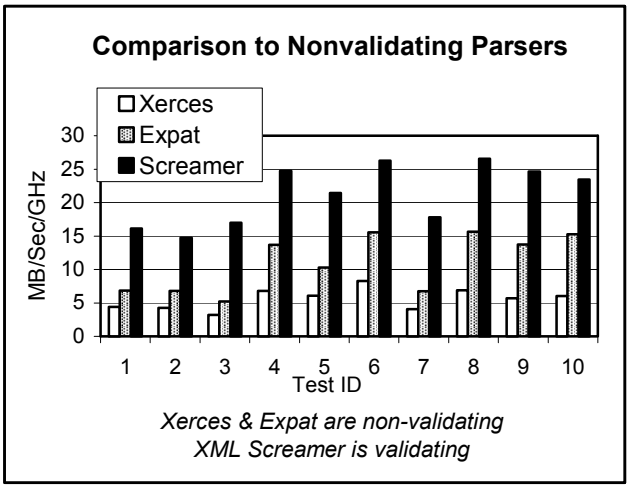

Figure 3: Comparison to nonvalidating parsers (SAX)

8.3.2 Business Object Performance

XML Screamer's performance advantage over the non-validating parsers is even greater using the business object APIs that are common to many Web Services applications. For the reasons described in Section 3, business object APIs in XML Screamer (but not in the other parsers) are significantly faster than SAX.

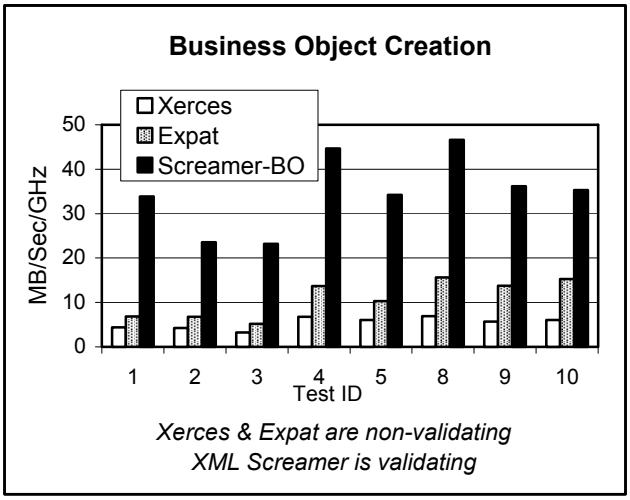

Figure 4: Comparison to nonvalidating parsers (Bus.Obj.)

As shown in Figure 4, XML Screamer builds business objects at between 2.3x and 4.9x the speed of Expat, the faster of the two nonvalidating alternatives, and XML Schema achieves this speed while performing XML Schema validation.

Note that Expat and Xerces are each using their best available (i.e. SAX-like) interfaces; when used to construct business objects, some *additional* deserialization logic would be needed with these parsers. Thus, the above is a conservative estimate of the relative performance of XML Screamer for building business objects.

8.3.3 Validation Performance Compared

Xerces and most other parsers incur a heavy performance penalty for validation. Since XML Screamer significantly exceeds the speed of nonvalidating Xerces, the comparison when both parsers validate is correspondingly more favorable. Figure 5 shows XML Screamer in both SAX and business object modes.

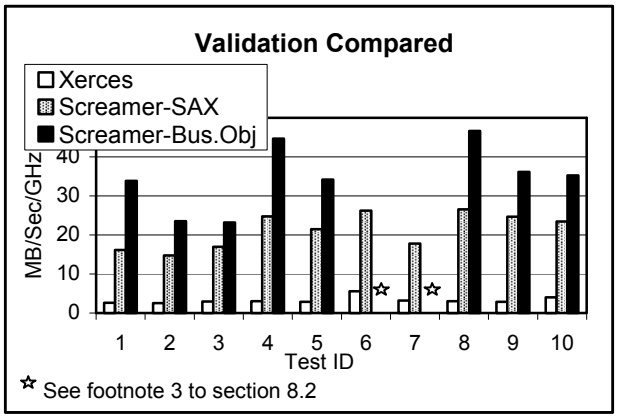

Figure: 5 Validation Performance Comparison

Using its gSOAP-like business object API, XML Screamer delivers from 7.8x to 15.4x the throughput of Xerces when both parsers are validating. Even using SAX, the ratios vary between 4.7x and 8.8x.

8.3.4 SAX Optimization

The results in Figure 6 bear out the speculation that schema-based precomputation of SAX output can yield significant performance gains. XML Screamer performs such optimizations when a useful schema is available, but not when validating with a wildcard (`xsd:anyType`), as the latter provides no advance information about instance structure.

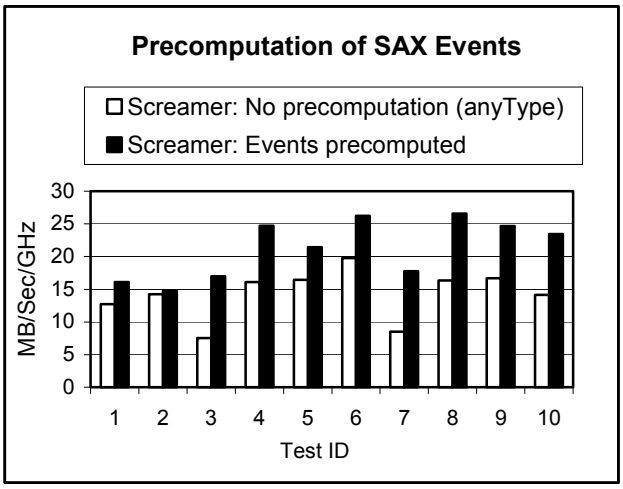

Figure 6: Effect of compile-time SAX precomputation

In all cases, processing with a schema is faster, and we have also done tests verifying that the gains are indeed due primarily to precomputation. Gains of 30% or more are frequently observed. XML Screamer thus demonstrates that, at least sometimes, using a schema and performing validation can increase performance.

8.3.5 Performance Summary

On the tests reported in this paper, using the business object API typical of Web Services applications, XML Screamer parses and schema-validates XML at between 23 and 46 Mbytes/sec/GHz; XML Screamer can thus process XML at speeds of roughly 100–200 Mbytes/sec on the 4 GHz processors now becoming available. Making the assumption that typical systems should devote at most 10–20% of CPU capacity to XML processing, we estimate that applications sustaining throughput of 10–40 Mbytes/sec are practical on a single CPU. Stated differently, one commodity processor is estimated to be capable of parsing, validating and deserializing XML from two saturated 100 Mbit/sec network links, with 80–90% of CPU time left for application-level work.

9. XML FEATURES NOT SUPPORTED

The compiler and generated parsers support most but not all XML and XML Schema features. Our purpose was not to build a fully conforming XML or Schema implementation, but rather to come close enough that our performance results would be representative of a complete implementation. With a few important exceptions detailed below, we believe that the prototype achieves this goal. Many of the more complex XML and Schema features, such as namespaces, `xsi:type`, `redefine`, `import`, `include`, and substitution groups are fully supported. The following are the significant areas of non-conformance in the XML Screamer prototype.

XML Features Not Supported

- DTD external or internal subsets. Note that external subsets are optional in the XML Recommendation, but internal subsets are required [21]. In this respect XML Screamer is non-conforming.
- Support for encodings other than UTF-8 or UTF-16. Our architecture is in principle capable of supporting other encodings, but because our parsers are hand crafted to optimize for the characteristics of particular encodings, the work involved is significant.
- Very large instance documents, i.e. those too large to fit in a contiguous memory buffer.

XML Schema Features Not Supported

- Facets on simple types (these are accepted but not checked; among the facets not checked is the pattern facet.)[5]
- Non-deterministic content models (such as certain models with nested numeric occurrence constraints.)[6]
- Identity constraints (accepted but not checked)
- Validity checking of types other than `anySimpleType`, `date`, `integer`, `decimal`, `nonnegativeInteger`, `boolean`, `positiveInteger`, `negativeInteger`, `nonpositiveInteger`, and `string` (all types are accepted, but validation of lexical forms and conversion to binary is available only for the listed types.)

Most of the above limitations are presumed to have a negligible impact on performance of features that are supported, but a few are probably significant. In particular, because instances are in a single contiguous buffer, the inner character scanning loops most critical to performance do not need to account for chunking of information. Element tag names are always contiguous, and can always be compared with string comparison instructions or simple comparison loops.

The lack of support for DTDs provides a more subtle performance gain, also related to instance buffering. Without DTDs, it is impossible to define XML general entities. Thus, our prototype parsers are never called upon to do the generalized string substitutions necessitated by references to such entities, and so the need for dynamic memory management is reduced.

Quantifying the performance gains from having a single input buffer and no DTDs would have required us to completely reimplement XML Screamer with a more generalized input buffering and scanning scheme. We have not made that considerable effort. Accordingly we can only speculate that the performance benefits may be significant, and warn those comparing our work with other approaches to account for the possible differences in capabilities. We note that there are many scenarios, most notably SOAP-based Web Services, in which DTDs and entities are never used, and we believe our performance is fully representative of what is achievable in those environments.

We have not studied the performance of identity constraints (`xsd:key`, `xsd:keyref`, `xsd:unique`), but our intuition is that these can, with care, be implemented at speeds comparable to other schema features. The optimization of regular expressions has been well studied, and we believe that with careful optimization, the overhead from pattern facets would be a few instructions per character parsed. At worst, support for these features would slow processing of those schemas in which they were used.

10. RELATED WORK

Many parsers now support validation with XML Schema. These include Xerces [1], MSXML [10], Saxon [15], XSV [17], etc. Although each is different in its implementation and in the APIs supported, these are for the most part "traditional" parsers, in which validation is largely separate from parsing, and which do not directly support "business object" or other customized APIs. Some have been well optimized. Few if any attempt the degree of low level performance tuning provided in XML Screamer, and none has the ability to optimize scanning, parsing, validation and custom API generation in a single parser.

There are also a wide range of tools such as Castor [2], gSOAP [18], and Liquid XML [9] that support automatic deserialization into business objects, with mappings derived from XML Schema. Some of these such as Castor sit on top of other SAX parsers, such as Xerces, and therefore run no faster than the underlying parser. Tools such as gSOAP provide more efficient parsing and deserialization, but do not do full Schema validation, and support only a single output API.

---

[5] A consequence of this limitation is that current versions of the XML Screamer prototype do not automatically generate the integer bounds check implied by Step 5 in the list of XML-Screamer generated code in Section 3. We have inserted such tests manually in some generated parsers and have verified that their runtime overhead is negligible, as would be expected.

[6] A proposed design for efficiently handling non-deterministic models is outlined in [11], but is not implemented in the code described here.

Several research teams have discussed compilation of DTDs or schemas, and we have earlier (Section 7.1) commented on the LALR-based work of Löwe [8]. Another high performance compiler is described by Chiu [3]. Like XML Screamer, Chiu compiles XML Schemas, integrating validation with scanning and parsing to achieve high performance. Chiu reports on the application of generalized automata as an intermediate code representation, and applies sophisticated transforms to automatically optimize low level scanning and validation. We would expect his system to be efficient at what it does, but with the important caveat that XML Screamer optimizes not just parsing and validation, but also deserialization and precomputation of invariant output structures. Indeed, Chiu advocates the simplifications that result from *not* including compiled support for deserialization. We believe that the analysis in Section 3, coupled with the business object performance of XML Screamer, demonstrates the advantages of integration across all layers.

Reuter [12] uses a two level approach, in which validators are generated automatically, but run on top of conventional SAX parsers; the performance is presumably limited to that of the chosen parser. van Engelen [19] also describes a two-level system, but uses a high-performance FLEX-based layer [5] to check XML conformance, and a separate DFA for validation. The separation of scanning and validation prevents optimization across the two layers.

Takase [16] describes a system in which invariant output is discovered heuristically at runtime, achieving optimizations in some ways similar to those we report for SAX. As multiple similar input documents are parsed, common substrings are noted and used to build a DFA against which subsequent input is validated. When successful, this system can automatically discover at runtime some of the invariants that XML Screamer determines from a schema. Takase's system seems to have advantages particularly in situations where no schema is available, since it works on any series of structurally similar XML documents. In cases where a schema is available, XML Screamer avoids the runtime overhead and complexity of dynamically comparing instance documents.

Hardware-based systems such as Datapower [4] and Sarvega [14] benefit from running parser/validators in isolated systems where memory management, thread dispatching, etc. can be optimized. Some of these systems also include ASICs customized for XML processing, and most integrate support for security and other features not provided by XML Screamer. Datapower has also reported on the use of JIT-like virtual machine technology to optimize XML processing [7]. Whatever their other advantages, hardware-based systems physically separate parsing and validation from the consuming application, and cannot directly optimize low level parsing with API integration. The availability of additional hardware processing capability may result in improved throughput, but in comparison to XML Screamer the total computation performed is likely to be greater.

11. DISCUSSION

XML Screamer is faster than most available processors, and it demonstrates that XML Schema validation can be done at similarly high speeds. Indeed, schemas can sometimes establish compile-time invariants that make validation a net gain in performance.

Using its business object APIs, XML Screamer scans, parses, validates and deserializes at between 22% and 44% of the tested processor's raw character scanning speed. Except insofar as ways can be found to use such processors more efficiently, e.g. by exploiting hardware string test instructions or on chip SIMD accelerators, gains from further tuning or alternative approaches are likely to be modest. XML Screamer's performance is probably not far from the maximum achievable.

Although it is not the first XML Schema compiler and it is not the first system to automatically generate business object deserializers, XML Screamer demonstrates the importance of integrating deserialization with scanning, parsing and validation, and of providing compiled optimizations particular to each XML API. The XML Screamer prototype supports a few test APIs, including SAX, but it could in principle be adapted to a wide range of others, including some that might be specific to particular environments or applications. For example, one could tailor the parser to directly populate an editor's in-memory structures and indices. Support of such APIs does involve at least one important complication: the XML Screamer compiler must be hand-customized to exploit each new one. Automatic adaptation to additional APIs, perhaps using an API specification compiler, would be an interesting direction for future research. Having raised this concern, we note that XML Screamer provides important optimizations for SAX and similar general purpose APIs. Insofar as these are adequate, there is no need to further customize the compiler.

Our work again makes clear, as others have observed before, that careful API design is crucial to good XML performance. A particular API may necessitate excessive string conversions, object creations or buffer manipulations that even a careful implementation cannot avoid, and as XML processors become faster, the relative impact of even small inefficiencies grows. The performance of the best XML implementations will ultimately be limited by the designs of their chosen APIs.

Any schema compiler has the drawback that compiled artifacts must be deployed with each application, and we have found this to be a significant challenge for adapting XML Screamer technology to production systems. Each generated parser must be appropriate to the operating system, compiler, supporting libraries and hardware on which applications will be run. In some cases, e.g. for multi-platform products or applications, parsers must be deployed in multiple versions for different environments, and all such versions must updated as schemas change.

The system described here is a prototype. Although many of the most challenging XML and XML Schema features are fully supported, some others are not, and components such as the business object generator are incomplete. We believe that, with a few exceptions already discussed, the performance of the prototype is fully representative of what would be achievable in a production quality implementation. The one significant exception is generalized support for entity substitution, which we believe might somewhat impact performance, but which in any case is never required for Web Services.

XML and Web Services are designed in part to enable communication among loosely coupled organizations, precisely the applications in which careful input checking is essential. The technologies demonstrated in XML Screamer should be usable to automate XML instance validation in production systems, and thus to significantly improve the robustness of XML-based communication.

12. ACKNOWLEDGMENTS

We wish to thank the many people who helped to implement or otherwise supported our work on XML Screamer. They include: Sharon Adler, Tom Bridgman, Philippe Charles, Emily Farmer, Wylie Garvin, Daniel Glasser, Ted Habeck, Joe Latone, Michelle Leger, David Marston, Mel Martinez, Alex Morrow, Chet Murthy, Paul Reed, Daniel Silva, Alfred Spector, John Turek, Max Van Kleek, and everyone at Extreme Blue.

13. REFERENCES

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

[2] The Castor Project. http://www.castor.org/index.html.

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

[4] Datapower, Inc. XA35 XML Accelerator. http://www.datapower.com/products/xa35.html.

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

[6] Java API for XML-Based RPC (JAX-RPC). http://java.sun.com/webservices/jaxrpc/index.jsp.

[7] Kuznetsov, E. Method and Apparatus of Data Exchange Using Runtime Code Generator and Translator. US Patent 6,772,413 B2, (August, 2004) .

[8] Löwe, W.M., Noga, M.L., and Gaul, T.S. Foundations of Fast Communication via XML. Annals of Software Engineering, 13 (June 2002), 357-359.

[9] Liquid Technologies Liquid XML 2005. http://www.liquid-technologies.com/.

[10] Microsoft Corp. MSXML Parser. http://msdn.microsoft.com/xml/.

[11] Perkins, E., Matsa, M., Kostoulas, M., Heifets, A., Mendelsohn, N. Generation of Efficient Parsers through Direct Compilation of XML Schema. IBM Systems Journal, 45, No. 2, (May 2006).

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

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

[14] Sarvega, Inc. The Sarvega Speedway™ XSLT Accelerator. http://www.sarvega.com/xml-speedway-accelerator.html.

[15] Saxonica, Ltd. Saxon. http://www.saxonica.com/.

[16] Takase, T., Miyashita, H., Suzumura, T. and Tatsubori, M. An Adaptive, Fast, and Safe XML Parser Based on Byte Sequence Memorization. World Wide Web Conference (May 2005).

[17] Thompson, H. S. and Tobin, R. Current status of XSV. (22 April 2005) http://www.ltg.ed.ac.uk/~ht/xsv-status.html.

[18] van Engelen, R., and Gallivan, K. The gSOAP Toolkit for Web Services and Peer-to-Peer Computing Networks. In 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid — CCGRID'02 (May 2002).

[19] van Engelen, R. Constructing Finite State Automata for High-Performance XML Web Services. International Conference on Internet Computing (2004): 975-981.

[20] Watson, R.W., and Mamrak, S.A. Gaining efficiency in transport services by appropriate design and implementation choices. ACM Transactions on Computer Systems (TOCS), v.5 n.2, p.97-120, May 1987.

[21] W3C. Extensible Markup Language, Version 1.0. 4 February 2004. http://www.w3.org/TR/REC-xml/.

[22] W3C. Namespaces in XML. 14 January 1999. http://www.w3.org/TR/ REC-xml-names/.

[23] W3C. XML Schema, W3C Recommendation, 28 October 2004. Part 1: http://www.w3.org/TR/ xmlschema-1/, Part 2: http://www.w3.org/TR/xmlschema-2/, Primer: http://www.w3.org/TR/xmlschema-0/.

[24] W3C. XML Schema API, W3C Member Submission, March 2004, http://xml.apache.org/xerces2-j/api.html.