icXML: ACCELERATING XERCES-C 3.1.1 USING THE PARABIX FRAMEWORK

by

Nigel Medforth

B.Tech., Kwantlen Polytechnic University, 2009

A PROJECT SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

APPROVAL

Name:	Nigel Medforth
Degree:	Master of Science
Title of Project:	icXML: Accelerating Xerces-C 3.1.1 using the Parabix Framework
Examining Committee:	Dr. Kay Wiese Chair
	Dr. Robert Cameron, Senior Supervisor
	Dr. Ted Kirkpatrick, Supervisor
	Dr. Thomas Shermer, Supervisor
	Dr. Arrvindh Shriraman, SFU Examiner
Date Approved:	August 23, 2013

Abstract

Traditional XML parsers process XML documents sequentially, one byte-at-a-time. Parabix-XML, however, parses documents 128-bytes-at-a-time, through the use of `Pablo`-generated parallel bit stream operations. Prior research on accelerating XML processing using the Parabix Framework lead to a number of interesting yet feature-light research prototypes. This project investigates the integration of Parabix into an existing widely-used XML parser, Xerces-C 3.1.1 of the Apache Software Foundation. Xerces was systematically restructured into nine independent layers that leverage parallel transcoding, deletion and bit stream operations yet still adhere to the existing programmer API. icXML supports all features provided by Xerces with the exception of object serialization and its layered structure supports future multicore acceleration using pipeline parallelism. Evaluation of icXML in a single-core setting demonstrates a speedup of 50% to 100% in a wide range of workloads.

Keywords: XML, Parabix, Parallel bit stream technology, SIMD

Acknowledgments

Building icXML would not have been possible were it not for the help of those around me. Although no single statement could suffice, I would like to thank my parents whose support made it possible for me to dedicate my time to this project. And I would like to express my gratitude to my supervisor Dr. Robert Cameron for his guidance and continual engagement throughout this project, and to my fellow lab mates, Kenneth Herdy and Dan Lin especially, for their assistance and their foundational work with parallel bit streams. Additionally, I'd like to thank my friends for their understanding and copious encouragement and to International Characters Inc. for their suggestions and internship opportunities, both of which were invaluable. Finally, I'd like to thank my examining committee whose feedback was greatly appreciated and helped finalize this report and to SFU as a whole for providing me with the equipment and knowledge necessary to complete my work.

Contents

List of Tables

List of Figures

Preface

Excerpts of this report were published earlier at the Balisage 2013 conference under the title “icXML: Accelerating a Commercial XML Parser Using SIMD and Multicore Technologies” [20]. As lead author of that paper, I can state that it presented an earlier view of icXML that does not completely represent its current state as of submittal of this report.

Chapter 1

Introduction

icXML is a proof-of-concept re-implementation of the Xerces-C 3.1.1 XML parser. The goal was to adapt Xerces to leverage the advantages inherent in the Parabix Framework [18] yet maintain its existing API and functionality. The challenge is that Xerces, like all traditional XML parsers, is an "embarrassingly sequential" byte-at-a-time parser [1] whereas the Parabix Framework uses wide SIMD registers to process input in a parallel block-at-a-time fashion. Given the diametrically-opposed ideologies between the two concepts, blending them required extensive re-engineering.

To leverage SIMD parallelism via bit stream processing and support the eventual inclusion of multicore acceleration, Xerces was conceptually divided into nine separable task-critical layers. In icXML, each layer is an independent module.

Organization of this report is as follows: Ch. 2 provides a basic introduction to XML, Xerces-C 3.1.1 and the Parabix Framework. Ch. 3 discusses the icXML's overall architecture and the key differentiating features between it and Xerces. Ch. 4 to 12 discuss each layer individually, highlighting the core functionality, how they support subsequent modules (both directly through data manipulation and indirectly through a priori knowledge), and their validation contributions. Ch. 13 compares icXML and Xerces and presents a comparative evaluation of their performance over three key benchmarks. Ch. 14 concludes the paper with my closing comments.

Chapter 2

Background

This chapter provides a brief introduction to XML (including standard XML terminology and production rules), XML Processors, the Xerces-C 3.1.1 Parser and the Parabix Framework. It does not contain any material regarding the icXML project itself.

2.1 XML Overview

Extensible Markup Language (XML) [3,4] is a core technology standard of the World Wide Web Consortium (W3C), providing a common framework for encoding and communicating structured and semi-structured information. XML plays a ubiquitous role in data interoperability in `applications` ranging from Office Open XML in Microsoft Office to NDFD XML of the NOAA National Weather Service, from KML in Google Earth to Castor XML in the Martian Rovers, from ebXML for e-commerce data interchange to RSS for news feeds.

In XML, documents consist of two logical components: (1) `markup`, which encodes a description of an XML document's storage layout and logical structure, and (2) `content`, which contains the textual values that is associated with each item of data in the XML instance. Fig. 2.1 provides a standard product list encapsulated within an XML document. All content is highlighted in bold. Anything that is not content is considered markup.

XML documents are typically divided into two categories: (1) data-oriented and (2) document-oriented XML instances. The `markup density` — the percentage of bytes used to express the markup tags over the size of the document — is a key metric used to distinguish between the two. Data-oriented instances typically have a higher magnitude of

```
<Product ID="1931W">
  <ProductName Lang="English">Widget</ProductName>
  <ProductName Lang="French">Bitoniau</ProductName>
  <Company ID="ACME ">ACME Corporation</Company>
  <Price Regular="$19.95" Clearance="$11.95"/>
  <!-- Not For Sale to Actual Customers -->
</Product>
```

Figure 2.1: Sample XML Document

markup compared to content than document-oriented XML instances; they are are typically used for system-to-system communications, such as XHTML, SOAP, GML, KML and VML amongst many others. Document-oriented instances often intended to be human-readable; the markup tags are often quite verbose and sparsely laid out throughout the document.

Virtually any type of information can be represented in XML but for an XML document to be considered well-formed, it must adhere to the following production rules:

```
    Document  :=  Prolog Element Misc*
-----------------------------------------------------------------------------
      Prolog  :=  XmlDecl Misc* (DocTypeDecl Misc*)?
        Misc  :=  Comment | PI | Space
     XmlDecl  :=  '<?xml' VersionInfo EncodingDecl? SDDecl? Space? '?>'
 DocTypeDecl  :=  See Extensible Markup Language (XML) specification (2.8)
-----------------------------------------------------------------------------
     Content  :=  CharData? ((Element | CDATA | Comment | PI ) CharData?)*
     Element  :=  StartTag Content EndTag | EmptyTag
    StartTag  :=  '<' Name (Space Attribute)* Space? '>'
    EmptyTag  :=  '<' Name (Space Attribute)* Space? '/>'
   Attribute  :=  Name Eq AttValue
      EndTag  :=  '</' Name Space? '>'
    AttValue  :=  '"' ([^<&"] | Reference)* '"' | "'" ([^<&'] | Reference)* "'"
     Comment  :=  '<!--' ((Char - '-') | ('-' (Char - '-')))* '-->'
       CDATA  :=  '<![CDATA[' (Char* - (Char* ']]>' Char*)) ']]>'
          PI  :=  '<?' PITarget (Space (Char* - (Char* '?>' Char*)))? '?>'
    PITarget  :=  Name - (('X' | 'x') ('M' | 'm') ('L' | 'l'))
   VersionInfo  :=  Space 'version' Eq ("'" VersionNum "'" | '"' VersionNum '"')
    VersionNum  :=  '1.0' | '1.1'
  EncodingDecl  :=  Space 'encoding' Eq ('"' EncName '"' | "'" EncName "'" )
       EncName  :=  [A-Za-z] ([A-Za-z0-9._] | '-')*
        SDDecl  :=  Space 'standalone' Eq (("'" ('yes' | 'no') "'") |
                    ('"' ('yes' | 'no') '"'))
-------------------------------------------------------------------------------
      CharData  :=  [^<&]* - ([^<&]* ']]>' [^<&]*)
         Space  :=  (#x20 | #x9 | #xD | #xA)+
          Char  :=  Space | [#x21-#xD7FF] | [#xE000-#xFFFD] | [#x10000-#x10FFFF]
            Eq  :=  Space '=' Space
          Name  :=  NameStartChar (NameChar)*
 NameStartChar  :=  ":" | [A-Z] | "_" | [a-z] | [#xC0-#xD6] | [#xD8-#xF6] |
                    [#xF8-#x2FF] | [#x370-#x37D] | [#x37F-#x1FFF] |
                    [#x200C-#x200D] | [#x2070-#x218F] | [#x2C00-#x2FEF] |
                    [#x3001-#xD7FF] |[#xF900-#xFDCF] | [#xFDF0-#xFFFD] |
                    [#x10000-#xEFFFF]
      NameChar  :=  NameStartChar | "-" | "." | [0-9] | #xB7 | [#x0300-#x036F] |
                    [#x203F-#x2040]
```

2.2 XML Processor

An `XML Processor` (or `XML parser`) is a software module that is capable of reading an XML document and distinguishing between the markup and content held within it. It feeds the the resultant XML data to the `application`, typically through an event-based interface.

Traditional XML processors parse an XML document from the first to the last character in the source text. Each character is examined to distinguish between the markup identifiers, such as a left angle bracket '`<`', and the content held within the document. A logical `cursor` position denotes the current character under interpretation. As the parser moves its cursor through the document, it alternates between markup scanning, validation, and content processing operations. In other words, XML parsers are finite-state machines (FSMs) that use byte-space comparisons to transition between data and metadata states. Each state transition indicates the context for interpreting any subsequent characters. Unfortunately, markup and content tends to consist of variable-length strings sequenced in unpredictable patterns; thus any character could be a state transition until deemed otherwise. As an application class, FSMs are considered to be "embarrassingly sequential" [1] and therefore very difficult to parallelize.

2.3 Xerces-C++ 3.1.1

Xerces-C++ 3.1.1 is a widely-used standards-conformant open-source stream-oriented validating XML Parser and is a key component of the Apache XML project. It features comprehensive support for a variety of character-encodings both commonplace (e.g., UTF-8, UTF-16) and rarely used (e.g., EBCDIC), support for multiple XML vocabularies through the XML namespace mechanism, as well as complete implementations of structure and data validation through multiple grammars declared using either legacy DTDs (document type definitions) or modern XML schema facilities. Xerces also supports several APIs for accessing parser services, including event-based parsing using either pull parsing or SAX/SAX2 push-style parsing as well as a DOM tree-based parsing interface.

2.4 Parabix Framework

The Parabix (parallel bit stream) framework is a transformative approach to text processing. The key idea is to exploit the availability of wide SIMD registers in commodity processors to represent data from long blocks of input data by using one register bit per single input byte. Parabix programs can be generally broken into four logical stages: (1) transposition, (2) character classification, (3) lexical analysis, and (4) post-processing operations.

2.4.1 Transposition (from Byte-Space to Bit-Space)

Given a code-unit[i] of n bits, the input data is transformed into n `basis-bit streams` by mapping the i^{th} bit of the j^{th} byte of input data to the j^{th} bit of the i^{th} basis-bit stream.

> Just as forward and inverse Fourier transforms are used to transform between the time and frequency domains in signal processing, bit stream transposition

[i]**Code Unit:** the minimum number of bits required to represent any code point in a specific character encoding format. Some formats (e.g., `UTF-8`, `UTF-16`, etc) use variable-width encoding schemes (i.e., multi-code-unit sequences) to represent some of the code points in the format's character set.

> and inverse transposition provides *bit-space* and *byte-space* views of the source text. The goal of the Parabix Framework is to support efficient text processing using these two equivalent representations in the same way that efficient signal processing benefits from the use of the frequency domain in some cases and the time domain in others. [18]

The Parabix Framework works with blocks whose size in bytes is equal to the bit-width of wide SIMD registers (e.g., 128 for SSE, 256 for AVX). Blocks are transposed into *n basis bit streams* in parallel with an amortized cost of approximately 1 cycle/byte using SSE [18]. For example, in Fig. 2.2, the UTF-8 string "`10<Ppl/>`" was transposed into 8 basis-bit streams, $\mathtt{b}_{0...7}$.

String	1	0	<	P	p	l	/	>
ASCII	00110001	00110000	00111100	01010000	01110000	01101100	00101111	00111110

$\mathtt{b}_7$	$\mathtt{b}_6$	$\mathtt{b}_5$	$\mathtt{b}_4$	$\mathtt{b}_3$	$\mathtt{b}_2$	$\mathtt{b}_1$	$\mathtt{b}_0$
0	0	1	1	0	0	0	1
0	0	1	1	0	0	0	0
0	0	1	1	1	1	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
0	1	1	0	1	1	0	0
0	0	1	0	1	1	1	1
0	0	1	1	1	1	1	0

Figure 2.2: Byte-space to Bit-space Transposition

2.4.2 Character Classification

Typically, `XML Processors` parse byte-space data to locate specific characters to determine if and when to transition between data and metadata parsing. For example, in XML, a left angle bracket character `<` may indicate that we are starting a new markup tag—provided that `<` is not within a `Comment` or `CDATA` section. Traditional `XML Processors` find these characters by comparing the value of each with a set of known significant characters and branching appropriately when one is found, typically using an if or switch statement. Using this method to perform multiple transitions in parallel is non-trivial and may require fairly sophisticated (and often speculative) algorithms to do so correctly.

With the transposed bit-space representation of the input data, classification of these characters can be performed in parallel using the Parabix Framework, which allows characters to be expressed mathematically, using an extended set of Boolean-logic operations. For example, one of the fundamental characters in XML is a left-angle bracket. A character is an '`<`' if and only if $\neg(b_0 \vee b_1) \wedge (b_2 \wedge b_3) \wedge (b_4 \wedge b_5) \wedge \neg(b_6 \vee b_7) = 1$. Similarly, a character is numeric `[0-9]` if and only if $\neg(b_0 \vee b_1) \wedge (b_2 \wedge b_3) \wedge \neg(b_4 \wedge (b_5 \vee b_6))$. An important observation here is that ranges of characters may require fewer operations than individual characters and multiple classes can share the classification cost.

2.4.3 Lexical Analysis

To perform lexical analysis on the input data, Parabix computes lexical and error bit streams from the character-class bit streams using a mixture of both boolean logic and arithmetic operations. Lexical bit streams typically mark multiple current parsing positions. Unlike the single-cursor approach of traditional text parsers, these allow Parabix to process multiple cursors in parallel. Error bit streams are often the byproduct or derivative of computing lexical bit streams and can be used to identify any well-formedness issues found during the parsing process. The presence of a `1` in an error stream indicates that the lexical stream cannot be trusted to be completely accurate and it may be necessary to perform some sequential parsing on that section to determine the cause and severity of the error.

source text			`<a><valid> <string>  <>ignored><error]`
C_0	$=$	`[a-zA-Z]`	`.1..11111...111111.....1111111..11111.`
C_1	$=$	`[>]`	`..1......1........1...1.......1.......`
C_2	$=$	`[<]`	`1..1.......1.........1.........1......`
L_0	$=$	`Advance`(C_2)	`.1..1.......1.........1.........1.....`
E_0	$=$	$L_0 \wedge \neg C_0$	`......................1...............`
L_1	$=$	`ScanThru`(L_0, C_0)	`..1......1........1...1..............1`
E_1	$=$	$L_1 \wedge \neg C_1$	`.....................................1`

Figure 2.3: Lexical Parsing in Parabix

To form lexical bit streams, Parabix provides several parallel bit movement operations in addition to bitwise logic, including, in particular, `Advance` and `ScanThru`. (`ScanThru` is the foundation of Parabix-2 Framework [10]) The `Advance` operator accepts one input parameter, c, which is typically viewed as a bit stream containing multiple cursor bits, and advances each cursor one position forward. On little-endian architectures, shifting forward means shifting to the right. `ScanThru` accepts two input parameters, c and m; any bit that is in both c and m is moved to first subsequent `0`-bit in m by calculating $(c+m) \wedge \neg m$. For example, in Figure 2.3 suppose we have the regular expression `<[a-zA-Z]+>` and wish to find all instances of it in the source text. We begin by constructing the character classes C_0, which consists of all letters, C_1, which contains all '`>`'s, and C_2, which marks all '`<`'s. In L_0 the position of every '`<`' is advanced by one to locate the first character of each token. By computing E_0, the parser notes that "`<>`" does not match the expected pattern. To find the end positions of each token, the parser calculates L_1 by moving the cursors in L_0 through the letter bits in C_0. L_1 is then validated to ensure that each token ends with a '`>`' and discovers that "`<error]`" too fails to match the expected pattern. The erroneous cursors in L_0 and L_1 are handled in the post-processing phase.

2.4.4 Post-Processing

When using the Parabix Framework, the post-processing stage encompasses all byte-space logic that follows the bit-space processing. Certain operations, such as the matching of a start and end tag element names, are too complex to handle efficiently in bit-space.

2.4.5 Pablo Compiler

All Parabix applications are written in `Pablo`, a subset of the `Python` programming language. The `Pablo` compiler transforms the `Pablo` program into its `C/C++` equivalent. The key difference between the two is that `Pablo` views bit streams as unbounded streams whereas the `C/C++` version takes into account details such as finite SIMD register widths and application buffer sizes. This abstracts the details and simplifies proving the correctness of the `Pablo` code by separating the implementation from the mathematical representation of the algorithm.

Unfortunately, SIMD registers were not designed for this form of use. As such, the block-to-block processing model must be simulated through what are referred to as carry queues (`carryQs`). Extended Boolean operations, such as `Advance` and `ScanThru`, are converted into their block-by-block equivalent by integrating carry-out and carry-in functionality. Each call to one of these functions generates a single carry bit that is "queued" for insertion into the subsequent block, effectively simulating an unbounded model. Further explanation of this process can be found in "Parabix: Boosting the efficiency of text processing on commodity processors" [18].

2.4.6 BitStream Iterators

Occasionally it is necessary to correlate positions within the bit streams with the input data. The Parabix Framework uses Bit Stream Iterators to sequentially scan through and extract the position of each 1-bit in a bit stream in either a forward or backward direction, effectively allowing the processor to scan up to word-size's worth of bytes using simple processor intrinsics and mathematical operations. The positions found by these iterators have a one-to-one correspondence with the code-unit-aligned positions in the source data.

Forward iterators transform a bit stream into a sequence of numbers $\{\mathtt{i}_1, \mathtt{i}_2, \ldots, \mathtt{i}_n\}$, where $\mathtt{i}_j$ is the position of the j-th 1-bit in the bit stream. Backwards iterators produce a similar sequence, except $\mathtt{i}_j$ is the position of the $(n-j)$-th 1-bit in the bit stream.

Chapter 3

icXML Parser Architecture

icXML is more than an optimized version of Xerces: most of Xerces's XML parsing logic was entirely replaced with Parabix-style bit stream equations and what was not removed was completely restructured into independent modules. This chapter presents an overview of the icXML parser and how the various components interact with each other. Detailed descriptions of individual modules can be found in later chapters.

3.1 icXML Overview

icXML is divided into three logical groups: the `Prolog Parser`, the `Parabix Subsystem` and the `Markup Processor`. The Prolog Parser is a byte-at-a-time parser that handles the detection and evaluation of the `XMLDecl` and DTD subset (Ch. 2.1). This occurs prior to the instantiation of the Parabix Subsystem or Markup Processor because the `XMLDecl` informs icXML as to which version of XML the document is specified in, the true character-encoding format and whether the document is standalone or not. The DTD subset can be complex and the full resolution of the DTD grammar may require loading an external DTD subset. However since icXML was intended to parse large documents, schema parsing is often the least-expensive component of the overall system.

Every module within the Parabix Subsystem either directly uses the Parabix Framework [18] or processes one or more sets of parallel bit streams (Ch. 2.4). The algorithms used by any module in this group are relatively stateless with respect to traditional XML parsers. On the other hand, the modules within the Markup Processor do not process any bit streams and are almost entirely state-driven. The overall design, organization and the inputs and

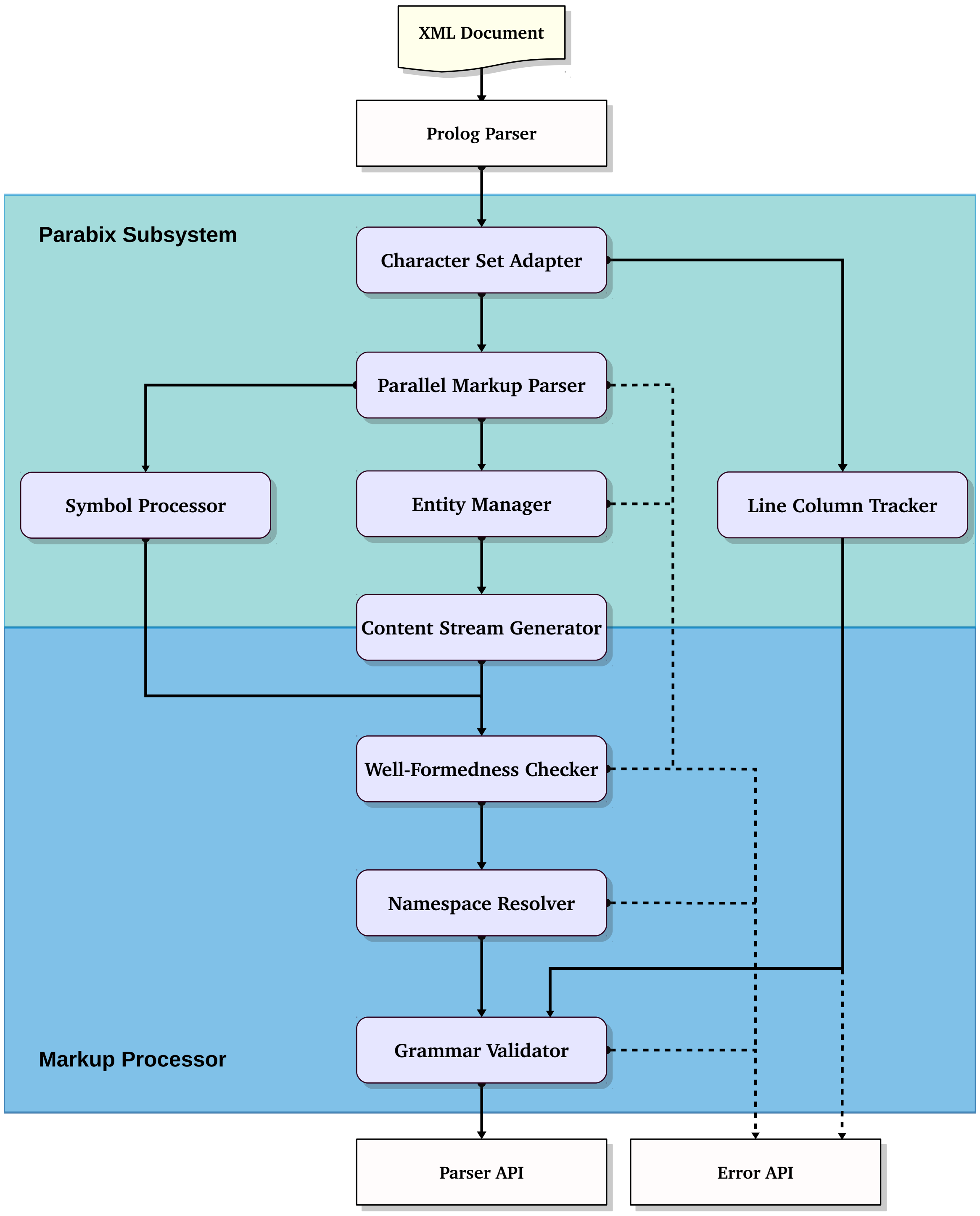

Figure 3.1: icXML Architecture Diagram

outputs of these modules was greatly influenced by the concepts presented by Thies et. al, in StreamIt: A Language for Streaming Applications [25].

3.1.1 The Parabix Subsystem

One of the early insights into XML processing was that the performance of traditional `XML Processors` is negatively correlated with the markup-density of an XML document [6, 10,18]. The Parabix Subsystem was designed with this in mind. The Character Set Adapter (Ch. 4) mirrors Xerces's Transcoder duties; however instead of producing UTF-16 directly, it tranposes the source text and produces a set of character-class and lexical bit streams, which includes u8-indexed bit streams for the UTF-16 representation [9] that are later compressed and inverse transposed to UTF-16 by the Content Stream Generator (Ch. 8).

The primary goal of the Content Stream Generator is to produce the *content stream*, which is a reduced representation of the source text. Almost all markup is filtered from the content stream to reduce the future workload of the Markup Processor. For this to work, the Parallel Markup Parser (Ch. 5) must first perform syntactical analysis on the lexical bit streams. In doing so, it generates a set of marker and callout bit streams in which each 1-bit identifies the position of some critical piece of XML text, such as the beginning and ending of element tags, element names, attribute names, attribute values, entity references, and content, etc. Of all markup text, the names of elements and attributes are the most prevalent and recurrent components. Prior to deletion, the Parallel Markup Parser identifies them and provides the Symbol Processor (Ch. 6) with the information necessary to produce the symbol stream from the source text. The *symbol stream* is a document-order sequence of global identifiers (`gids`) of each name token. The modules within the Markup Processor understand the contractual relationship between the content and symbol streams and parse the document accordingly (discussed in Ch. 8).

Almost all intra-element syntax validation is handled by the Parallel Markup Parser and any violation is considered a fatal XML error. Like Xerces, whenever any error occurs, icXML reports the error type and line-column number through the `Error API`. The Line Column Tracker (Ch. 9) uses the lexical bit streams to calculate the key "cursor" locations through the use of an optimized population-count algorithm. Its output allows icXML to report the position of any well-formedness or schema violation, or, in the case of Xalan and other similar `applications`, disclose the source location of every `Element` tag and `Content` string in the XML Document.

3.1.2 The Markup Processor

Given the streams produced by the Parabix Subsystem, the Markup Processor continues the validation and data-transformation process. The Well-Formedness Checker (Ch. 10) is primarily responsible for ensuring the document meets the requirements of the XML 1.0/1.1 Specifications by validating any aspect of the document that was not handled within the Parabix Subsystem. Once validated, the Namespace Resolver (Ch. 11) — presuming that namespace processing is enabled — rescans the document to determine the namespace bindings and contexts of every element and attribute in the document. This information is fed directly to the Grammar Validator (Ch. 12), which apart from ensuring that the document meets its own schema[i], accumulates the data that will eventually be provided to the `application`.

Although the Parabix Subsystem is almost entirely data-parallel, the modules in the Markup Processor rely on state-dependant logic and, for the most part, must be performed sequentially. Unlike Xerces, icXML is organized in a producer/consumer model with the minimal amount of data being passed between modules. This increases the overall instruction count but opens the opportunity for pipeline parallelism in the future.

3.2 Key Features

3.2.1 Segment-Based Parsing Model

Dividing XML documents into "chunks" is a common technique for parsing very large files whilst preserving system resources. icXML employs this concept to processes documents in what are herein referred to as `segments`. A segment of data in is defined as a 16KB chunk of data from the input document and/or the derivatives of that data from a preceding module. (Other sizes were explored but the difference was minimal.) Each module is optimized for a full segment and stack-allocates its internal memory accordingly.

Unfortunately, the XML specification does not impose any limits on the length of any markup tag or content string. Consequently, no matter what segment size is chosen, each segment will almost surely end with either a partial markup tag or content string. Each partial item must be prepended to the subsequent segment to complete it in the following iteration. Therefore, prior to parsing the next segment, any incomplete data from the

[i]Note: in icXML, the absence of schema is equivalent to having a schema without any validation rules.

`content`, `symbol`, `reference` and `namespace streams` are copied back to the beginning of that stream's buffer. Before doing so, the Markup Processor tests whether there is any possibility that parsing the next segment could result in a overflow exception and if there is, doubles the size of the data stream to accommodate it. This allows icXML to handle partial tags and strings and any markup or content that exceeds the segment size without modifying the segment size itself.

3.2.2 Loosely-coupled Recursion

The occurrence of an general entity reference in an XML document marks the position that a textual replacement will eventually occur. The name of the reference is an alias to the replacement text that will eventually replace it (Ch. 7). In Xerces, each general entity is parsed with an independent `XMLReader` class. The `XMLScanner` — the class responsible for lexing the source text and producing the callback events for the `application` — relies on the `XMLReader` to provide it with data needed to construct each lexeme. The `XMLScanner` interacts with each `XMLReader` through the `ReaderMgr`, which is effectively a stack of `XMLReader` objects. Although separate instances of the `XMLScanner` class are instantiated for the source document and each of its external grammars, entity expansion occurs within the same `XMLScanner`.

In icXML, the concept of a `ReaderMgr` stack was replaced with a loosely-coupled recursive architecture, thereby eliminating the constant cost of two memory indirections that Xerces imposes on every character. Conceptually, each module within the Markup Processor is capable of recursively calling its parsing method, passing in all of the necessary information[ii]. The icXML `XMLReader` mirrors the functionality of the `XMLReader` in Xerces for `prolog` and external DTD schema parsing but operates solely as an input stream manager for the Parabix Subsystem and Markup Processor.

3.2.3 Optimized Progressive Scanning Modes

Xerces supports both push-style document parsing and pull-style progressive scanning modes. Document scanning mode begins when the `application` calls the `parse(...)` method. In this mode, Xerces continuously parses the source text, pausing only to emit markup and content callbacks to the `application`. However, when either `parseFirst(...)` or

[ii]Note: recursion is required only when icXML encounters a general entity containing markup.

`parseNext(...)` are called, icXML provides the `application` with a single piece of markup or content and waits for the `application` to call `parseNext(...)` (again) before scanning any further. Having both modes gives the `application` programmer the means to build either a callback (push) oriented or sequential (pull) oriented system. However, a sufficiently complex `application` will consume a sizable portion of the CPU cache, which will increase the probability of a cache miss when switching between Xerces and the `application`.

icXML supports both modes as well but the Document Accumulator (Ch. 12.2) buffers the sequence of callbacks and related data, eliminating any notion of progressive scanning from the internal modules. The layering approach has an additional side effect of reducing the instruction and data cache contention, which reduces the possibility of a cache miss. With further study, it may be possible to prevent cache misses altogether in icXML by constraining the cache requirements of each module to only what is available within (or easily prefetchable by) the processor.

Chapter 4

Character Set Adapter

In Xerces, all input is transcoded into UTF-16 to simplify the parsing logic of Xerces itself and provide the `application` with a standard output format regardless of the input character encoding. However, this is a "relatively expensive operation that can contribute significantly to the cost of text-oriented APIs" [23]. In icXML, the Character Set Adapter is an XML-aware parallel transcoding system that replaces the functionality of Xerces's `XMLTranscoders` using techniques first explored in the u8u16 transcoder [9].

Given a specified input encoding, it is responsible for checking that input code units represent valid characters, mapping the characters of the encoding into the appropriate bit stream for the Parallel Markup Parser (Ch. 5), and supporting the ultimate transcoding operations of the Content Stream Generator (Ch. 8).

4.1 Transcoding the Source Text

Based on Perkins, E. et. al. observations that transcoding XML documents can be expensive [23], icXML's initial design adopted a piecemeal transcoding strategy for `AttVal` and `Content` strings. Preliminary analysis found that this was not an improvement over Xerces's design because it forced icXML to jump between transcoding and non-transcoding states.

Instead, the Character Set Adapter parses the entire segment of XML text — but does not directly transcode it into UTF-16. When given the input text, it produces a set of `lexical bit streams` and `U16 bit streams`, which are used by the Parallel Markup Parser and Content Stream Generator to parse the input document and produce the content stream, respectively. To do so, the Character Set Adapter first transposes the input document (Ch. 2.4.1) and then decomposes it into a set of `character-class bit streams` (Ch. 2.4.2). The exact process differs depending on the input character-encoding scheme.

4.1.1 UTF-8

Given its 8-bit code-unit size, UTF-8 is one of the most popular character encoding formats for XML Documents. However, as Table 4.1 shows, transcoding UTF-8 to UTF-16 can be complex because of the need to decode and classify each byte of input and map variable-length UTF-8 byte sequences into 16-bit UTF-16 code units.

Unicode Range	UTF-8 Pattern	UTF-16 Pattern
0000-007F	0tuvwxyz	00000000 0tuvwxyz
0080-07FF	110pqrst 10uvwxyz	00000pqr stuvwxyz
0800-FFFF	1110jklm 10npqrst 10uvwxyz	jklmnpqr stuvwxyz
10000-10FFFF	11110efg 10hijklm 10npqrst 10uvwxyz	* 110110ab cdjklmnp 110111qr stuvwxyz

(* where abcd = efghi − 1)

Table 4.1: UTF-8 vs. UTF-16 Character Encoding Format. Adapted from "A Case Study in SIMD Text Processing with Parallel Bit Streams UTF-8 to UTF-16 Transcoding" [9]

Since both UTF-8 and UTF-16 are extensions to the legacy 7-bit ASCII, transcoding from UTF-8 to UTF-16 is trivial whenever the source text for a particular block of text is parsing XML are confined to the ASCII repertoire (`00-7F`). For all other ranges, the corresponding `U16 bit streams` can still be calculated in parallel from the UTF-8 data but this results in series of UTF-8-indexed UTF-16 code points [7]. Transforming the UTF-8-indexed bit streams to UTF-16-indexed bit streams requires the deletion of some of the extraneous bit positions. Specifially, all but the final byte of each multi-byte sequence must be marked in the deletion mask stream. This stream is a bitwise filter, which can be combined using bitwise-`OR`s with other filters so that the deletion algorithm needs to be applied only once per block. Deletion itself is deferred to the Content Stream Generator.

4.1.2 UTF-16

UTF-16 to UTF-16 transcoding is obviously a simpler problem than UTF-8 to UTF-16 but since UTF-16 code points are two-byte units, it introduces endianness-related problems.

Xerces automatically converts UTF-16B into UTF-16L within the `UTF16Transcoder` on little-endian architectures, such as Intel, and UTF-16L to UTF-16B on big-endian machines. icXML employs a similar technique within the Character Set Adapter but only as much as necessary to perform character classification and support the other modules.

4.1.3 Other Character Set Encodings

Currently, icXML natively handles only UTF-8 and UTF-16. These are the two most common encoding formats for XML documents and are required by all XML processors [3, 4]. All other encoding formats use Xerces's Transcoder architecture to first convert the source text to UTF-16 prior to processing it with the UTF-16 Character Set Adapter. This ensures that icXML is fully compatible with any encoding method supported by Xerces, including those supported via `ICU`, `ICONV` and other external transcoding libraries — but at an increased cost when parsing non-standard encoding formats.

4.2 End-of-Line Handling

In XML, end-of-line sequences (EOLs) are uniformly reported with a single LF `#xA` character unless they are specifically included in the source text as a character reference (Ch. 7). Section 2.11 of the XML 1.0 specification dictates that any occurrence of a CR-LF {`#xD,#xA`} or a CR `#xD` not followed by LF must be replaced with a LF [3]. Section 2.11 of the XML 1.1 specification extends the above rule to handle LS `#2028` and NEL `#x85` UNICODE characters. Every occurrence of a LS, NEL or a CR-NEL is also subsituted with a LF.

Line-break normalization occurs prior to XML processing. The Character Set Adapter assists the Parallel Markup Parser by locating and transforming any EOL into a LF character in parallel and `OR`s the second code-point of any two code-point EOL into the deletion mask stream. Fig. 4.1 shows the `Pablo` source code for this transformation[i]. Whilst doing so, the Parallel Markup Parser calculates the `lex.LF` bit stream, which is a sequence of bits in which each 1-bit marks the position of a LF. These are stored within the `LineFeedStream`, which the Line Column Tracker uses to calculate the line-column position of any byte within the input file.

[i]This is a modified version of the original Line Column Tracking code in Parabix-XML.

```
# Apply XML 1.0 line-break normalization rules.
if lex.CR:
  # Modify CR (#x0D) to LF (#x0A)
  u16lo.bit_5 ^= lex.CR
  u16lo.bit_6 ^= lex.CR
  u16lo.bit_7 ^= lex.CR
  CRLF = pablo.Advance(lex.CR) & lex.LF
  callouts.delmask |= CRLF
  # Adjust LF streams for newline/column tracking
  lex.LF |= lex.CR
  lex.LF ^= CRLF
  callouts.skipmask |= CRLF
# Apply additional XML 1.1 line-break normalization rules.
if parameters.XML_11:
  if lex.NEL:
    # Modify NEL (#x85) to LF (#x0A)
    u16lo.bit_0 ^= lex.NEL
    u16lo.bit_5 ^= lex.NEL
    u16lo.bit_6 ^= lex.NEL
    u16lo.bit_7 ^= lex.NEL
    lex.LF |= lex.NEL
    if lex.CR:
      CR_scope1 = pablo.Advance(lex.CR)
      CR_scope2 = pablo.Advance(CR_scope1)
      CRNEL = CR_scope2 & lex.NEL
      callouts.delmask |= CRNEL
      lex.LF ^= CRNEL
      callouts.skipmask |= CRNEL
    # Modify LS (#x2028) to LF (#x0A)
    if lex.LS:
      u16hi.bit_2 ^= lex.LS
      u16lo.bit_2 ^= lex.LS
      u16lo.bit_6 ^= lex.LS
      lex.LF |= lex.LS
```

Figure 4.1: Line-feed Handling in icXML

4.3 Validation Responsibilities

The Character Set Adapter ensures the all input code-units represent valid characters with respect to their given encoding format. As the rules for this validation process are complex and dependent on the input character set, they exceed the scope of this report.

Chapter 5

Parallel Markup Parser

Introducing the Parabix Framework parallel-processing model into an inherently serial system like an `XML Processor` is a complex process. Certain tasks, such has the detection of legal markup tags, can be performed in parallel using `Pablo` [6,10,18] — but leveraging them in Xerces requires extensive modifications. This chapter discusses how icXML incorporates the Parabix Framework and how it exploits data-parallel techniques to identify, validate, and "tokenize" the markup and content found within the XML Document.

5.1 Identifying Markup Tags

Every XML document contains a single root `Element` node, which can either be a start and end tag pair or an empty tag. A start or empty tag can contain any number of attributes but `Content` and child `Elements` can appear only between start and end tags. This structure is complicated by the existence of comment, CDATA and processing instruction tags, all of which can appear between any two pieces of markup or content and can contain characters that could otherwise appear to be the start of some other markup tag. A typical example is the use of a comment tag to hide one or more `Elements` from the `XML Processor` without actually deleting them from the source text.

When the Parallel Markup Parser parses an XML document, it actually parses the lexical bit streams generated by the Character Set Adapter and ignores the source text itself. With them, the Parallel Markup Parser generates a series of `marker` and `callout bit streams`, and in so doing, helps infer the syntactical meaning of the document, such as shown in Fig. 5.1. Marker bit streams are used internally but the callout bit streams are passed out to subsequent modules. The callouts bit streams include: (1) markers indicating the start and end of symols and references, (2) delimiters stating the beginning and end of each tag, (3) the span of each `AttVal` string, (4) the terminal position of every `Content` and `AttVal` string, (5) the penultimate deletion mask stream (Ch. 4.1.1), and (6) any information necessary to calculate the line-column position of each markup tag and `Content` string (Ch. 9).

		Source Text
		`<doc>fee<elem a1='fie' a2 = "foe"/> fum </doc>`
Lexical	Name Chars	`_111_111_1111_11__111__11____111____111___111_`
	Whitespace	`_____________1________1__1_1_______1___1______`
	Left Angle	`1_______1_______________________________1_____`
	Right Angle	`____1_____________________________1__________1`
	Slash	`_________________________________1_______1____`
	Equals	`________________1_________1___________________`
	Single Quote	`_________________1___1________________________`
	Double Quote	`____________________________1___1_____________`
		...
Callout	Start Tag Marks	`_1_______1____________________________________`
	End Tag Marks	`_________________________________________1____`
	Empty Tag Marks	`__________________________________1___________`
	Markup Delimiters	`1___1___1_________________________1_____1____1`
	String Terminal	`________1____________1__________1_______1______`
	AttVal Span	`__________________1111_______1111_____________`
	Deletion Mask	`_1111____111111111____1111111____11______11111`
		...

Figure 5.1: XML Source Data and Derived Bitstreams

Lexing and evaluation is divided into three logical stages: (1) `parseCtCDPI`, (2) `parseTags` and (3) `parseRefs`. The first detects any comments, CDATA or processing instructions and filters what the latter two views as actual markup tags and entity references, respectively. Although these stages are performed sequentially, earlier work has shown that "a [pipeline] parallelism strategy that requires neither speculation nor pre-parsing" [18] can be adopted for further acceleration.

Parsing start and empty tags in parallel is the most complex problem within the Parallel Markup Parser. icXML must contend with variable-length element names, attribute names, and attribute values, and an arbitrary number of attributes themselves. Fig. 5.2 provides a detailed example of this process but omits the detection of erroneous start tags and the existence of any other form of markup tag.

Using the Boolean operators supplied by the Parabix Framework (Ch. 2.4.3), the Parallel Markup Parser first detects the left angle brackets '`<`' denoting the start of some markup tag in E_0 (i.e., those not masked out by `parseCtCDPI`). Advancing E_0 sets each cursor in E_1 to the first character of the element name. Scanning through the `NameChars` moves the cursors in E_1 to the character immediately after the last character of the element name. By scanning from E_1 through any potential whitespace, the algorithm determines which start tags contain any attribute by masking the result with $\neg$`[>]`. The result, $A_{1,1}$, denotes that element `x` has no attributes but both `e` and `e12` have at least one with the presence of a 1-bit at the appropriate position.

$A_{1,1}$ to $A_{1,7}$ handle the identification of the first attribute name and value string within each element. The markers in $A_{1,1}$ and $A_{1,2}$ indicates the start and end position of each attribute name, respectively. By scanning from $A_{1,2}$ through any whitespace, $A_{1,3}$ must mark the position of a '`=`' in order for the attribute production to be valid. This process continues by locating leading and trailing quote of each attribute value ($A_{1,5}$, $A_{1,7}$). However, by scanning through the whitespace from $A_{1,7}$, the Parallel Markup Parser discovers that element `e12` contains a second potential attribute-value pair. Thus the identification process repeats from $A_{2,1}$ to $A_{2,7}$ to extract the second attribute. Because $A_{3,1} = 0$, the algorithm knows that all attribute-value pairs have been identified and stops accordingly.

The identification of other markup tags proceeds similarly but becomes considerably more complicated when the detection of illegal productions is introduced. For more information, please refer to "High performance XML parsing using parallel bit stream technology" [6] and "Parallel Scanning with Bitstream Addition: An XML Case Study" [10].

5.2 Filtering Markup Text

Once the Parallel Markup Parser generates the callout bit streams indicating the key positions within the source text, the later modules can safely ignore the markup identifiers themselves. Identifiers includes any character used in the production of some markup tag that is not an attribute value or the text within a comment, CDATA or processing instruction. icXML generates spans in which every 1-bit denotes a fully-parsed and unnecessary

Source Text	`<e a= "137">---<el2 a ="17" a2="3379">---<x>-`
$N =$ Name Chars	`.1.1...111..111.111.1...11..11..1111..111.1.1`
$W =$ Whitespace	`..1..1.............1.1.....1.................`
$Q = \neg\texttt{["<]}$	`.11111.111.1111.11111.1.11.1111.1111.1111.111`
$E_0 = \texttt{[<]}$	`1..............1.........................1...`
$E_1 = \texttt{Advance}(E_0)$	`.1..............1.........................1..`
$E_2 = \texttt{ScanThru}(E_1, N)$	`..1................1.......................1.`
$A_{1,1} = \texttt{ScanThru}(E_2, W) \wedge \neg\texttt{[>]}$	`...1................1........................`
$A_{1,2} = \texttt{ScanThru}(A_{1,1}, N)$	`....1................1.......................`
$A_{1,3} = \texttt{ScanThru}(A_{1,2}, W) \wedge \texttt{[=]}$	`....1.................1......................`
$A_{1,4} = \texttt{Advance}(A_{1,3})$	`.....1.................1.....................`
$A_{1,5} = \texttt{ScanThru}(A_{1,4}, W) \wedge \texttt{["]}$	`......1................1.....................`
$A_{1,6} = \texttt{Advance}(A_{1,5})$	`.......1................1....................`
$A_{1,7} = \texttt{ScanThru}(A_{1,6}, Q) \wedge \texttt{["]}$	`..........1...............1..................`
$A_{2,1} = \texttt{ScanThru}(A_{1,7}, W) \wedge \neg\texttt{[>]}$	`............................1................`
$A_{2,2} = \texttt{ScanThru}(A_{2,1}, N)$	`..............................1..............`
$A_{2,3} = \texttt{ScanThru}(A_{2,2}, W) \wedge \texttt{[=]}$	`..............................1..............`
$A_{2,4} = \texttt{Advance}(A_{2,3})$	`...............................1.............`
$A_{2,5} = \texttt{ScanThru}(A_{2,4}, W) \wedge \texttt{["]}$	`...............................1.............`
$A_{2,6} = \texttt{Advance}(A_{2,5})$	`................................1............`
$A_{2,7} = \texttt{ScanThru}(A_{2,6}, Q) \wedge \texttt{["]}$	`....................................1........`
$A_{3,1} = \texttt{ScanThru}(A_{2,7}, W) \wedge \neg\texttt{[>]}$	`.............................................`

Figure 5.2: Example of Start Tag Parsing. Adapted From "Parallel Scanning with Bitstream Addition: An XML Case Study" by Cameron, R. et. al. [10]

markup character. These spans are OR-ed into the deletion mask stream, which can reduce the length of the content stream output by as much as 50% (Ch. 8.1).

5.3 Parallel Symbol Tokenization

The Symbol Processor extracts every element and attribute name as a symbol token but on its own it has no concept of XML nor what constitutes a legal XML document beyond the characters allowed within a NCName or QName (Ch. 6.6). Section 5.1 showed how the Parallel Markup Parser identifies the components of an start tag but in so doing, also alluded to how it also identifies the symbol tokens within each. By OR-ing together the start and end position of every element and attribute name (i.e., $E_1 \vee E_2 \vee (A_{1,1} \vee A_{1,2}) \vee (A_{2,1} \vee A_{2,2}) \vee \cdots \vee (A_{n,1} \vee A_{n,2})$), the Parallel Markup Parser produces a symbol marker stream in which every 1-bit indicates the start or end position of some element or attribute token[i].

5.4 Attribute-Value Normalization

One of the more vexing issues for any `XML Processor` is whitespace normalization. The XML specification (3.3.3) requires that any whitespace character within attribute values must be replaced with space (`#x20`) [3, 4]. Xerces handles this by testing every `CharData` character read by the `XMLReader` and substituting each whitespace character with its appropriate replacement, despite the fact that normalization is typically not required. Although schema-related whitespace normalization rules are impossible to apply in pure bit-space without the development of a just-in-time (JIT) schema parser for the Parabix Framework, icXML performs all non-schema-related whitespace normalization in parallel within the Parallel Markup Parser by simply altering the appropriate bits in the `u16lo` bit streams, as depicted with the `Pablo` code in Fig. 5.3.

Both the DTD and XML Schema grammars provide mechanisms to remove any leading or trailing whitespace from attribute values and replace any contiguous string of whitespace characters with exactly one space character. An unexplored enhancement to this process could be used to detect whether any attribute value could be modified by this process and either "collapse" the whitespace in the Content Stream Generator (in the case of DTD stipulation) or inform the Grammar Validator whether it would be necessary.

[i]Note: by virtue of the XML specification, it is impossible for any two symbol tokens to overlap [3, 4].

```
callouts.AttValSpan = pablo.ExclusiveSpan(OpenQuotes, CloseQuotes)
...
# Normalize Whitespace in Attributes (replace LF, HT with SP)
WS_in_AttVal = lex.WS & callouts.AttValSpan
if WS_in_AttVal:
   u16lo.bit_2 |= WS_in_AttVal
   u16lo.bit_4 = u16lo.bit_4 &~ WS_in_AttVal
   u16lo.bit_5 = u16lo.bit_5 &~ WS_in_AttVal
   u16lo.bit_6 = u16lo.bit_6 &~ WS_in_AttVal
   u16lo.bit_7 = u16lo.bit_7 &~ WS_in_AttVal
```

Figure 5.3: Attribute Value Whitespace Normalization

5.5 Supporting Entity Expansion

In XML, a character or entity reference is an alias to some replacement text that ought to be substituted into the document at the point of insertion — but they are too costly to resolve efficiently in bit space. Even character references, which could be converted in parallel, are too costly to be computed given their expected density per bit-block. Instead when the Parallel Markup Parser detects a reference, it marks all of its characters in the deletion mask stream except the trailing semi-colon '`;`', which it marks the position of in the `reference marker stream`. The rationale behind this is threefold:

1. Single-character replacements are the most common case: all pre-defined entities are exactly one character in length and the characters `[#x0000,#xFFFD]` all correspond to code-points in the `Basic Multilingual Plane`, "which contains the vast majority of common-use characters for all modern scripts of the world" [11].

2. A general entity could be any number of characters in length and the validity of the substitution is dependent on whether it is parsed or unparsed, an internal or external entity, and whether reference is within content or an attribute value. Additionally, unparsed external entities could be resolved by the `application` via callbacks. As such, its difficult to know upfront how much space is required but even if it were, all general entities must be reported to the `application` at the point of expansion.

3. The Parallel Markup Parser parses the XML text in 16KB segments, which means that it is possible for a reference to begin in one segment yet end in a subsequent one.

Thus by using the terminal '`;`', all later phases have the a priori knowledge that any reference marked in the `reference marker stream` has been fully seen.

Further discussion on this topic can be found in Ch. 7.

5.6 Syntax Validation

In Xerces, syntax validation is an inherit part of the `XMLScanner` and `XMLReader` architecture. However, the nested conditionals used to perform this verification makes the underlying assumption that the input is incorrect until proven otherwise but, in practice, an XML document will almost surely be well-formed and fully adhere to its internal schema.

A novel feature of the Parabix Framework is that even when it is given invalid data, it is impossible for any subsequent fault to cause a hardware exception, segmentation fault, or any other form of system crash during parallel bit stream operations. Such failures can only potentially occur during postprocessing, depending on context. However, so long as a proper mechanism for computing and testing for error bits prior to postprocessing, the Parallel Markup Parser can operate under the assumption that the input is valid even when given an illegal document.

In icXML, each type of well-formedness error is represented with an `error bit stream` and the determination of whether an error of that type occurred is calculated using Boolean logic. For example, Fig. 5.4 illustrates how icXML determines whether an attribute production is valid. Currently, icXML tests whether any error occurred at a block-at-a-time granularity (e.g., every 128 code-units using SSE) and then determines the type of error that occurred if one exists. The ideal method would test only once at the end of each 16KB segment but this would require additional compiler support to properly implement. Namely, the ability to store and recall an arbitrary `carryQ` state (Ch. 2.4) and the ability to compile two separate versions of the Parabix code: one that simply uses a single error bit stream throughout the entire segment but tests it only once at the end of the segment and another version that tests each error bit streams upon derivation with the knowledge that an error has occurred somewhere in the document. While possible to develop manually, this was deemed to be outside of the current project scope.

```
...
errors.ExpectedAttrName |= (AttNameStart &~ lex.NameScan)
AttNameFollow = pablo.ScanThru(AttNameStart, lex.NameScan)

# Scan through WS to the expected '=' delimiter.
EqExpected = pablo.ScanThru(AttNameFollow, lex.WS)

# Check if any '='s are missing and report them as errors
errors.ExpectedEqSign |= EqExpected &~ lex.Equals

# Scan through whitespace to the expected quote delimiter
AttValPos = pablo.AdvanceThenScanThru(EqExpected, lex.WS)
DQuoteAttVal = AttValPos & lex.DQuote
SQuoteAttVal = AttValPos & lex.SQuote
errors.ExpectedAttrValue |= (AttValPos &~ (lex.DQuote | lex.SQuote))

# Test whether the closing quote delimiter matches the opening quote
DQuoteAttEnd = pablo.AdvanceThenScanTo(DQuoteAttVal, lex.DQuote)
SQuoteAttEnd = pablo.AdvanceThenScanTo(SQuoteAttVal, lex.SQuote)
errors.UnterminatedStartTag |=
    ((DQuoteAttEnd &~ lex.DQuote) | (SQuoteAttEnd &~ lex.SQuote))
...
```

Figure 5.4: Attribute Production Validation

5.7 Validation Responsibilities

The Parallel Markup Parser handles all intra-element well-formedess validation, excluding: (1) the verification of element and attribute names, (2) entity-reference expansions, and (3) scope-dependent production rules (e.g., matching a start tag name with an end tag name). These are handled by the Symbol Processor (Ch. 6), Entity Manager (Ch. 7) and Well-Formedness Checker (Ch. 10), respectively.

Chapter 6

Symbol Processor

Apart from the Parabix Framework, the largest divergence between Xerces and icXML is in handling element and attribute names. When Xerces encounters a name, it buffers the string into a `QName` object, performs namespace resolution (Ch. 11), and then locates or creates the appropriate grammar-specific element declaration or attribute definition. This is repeated for every occurrence of a QName in the XML Document — greatly impacting the performance of the entire system. In fact, $\approx 7\%$ of the total running time in Xerces can be directly attributed to `memcpy` operations of which this was the greatest contributor. icXML eliminates this entirely by viewing all QNames as `XMLSymbol` objects. This chapter discusses how icXML is capable of doing this and the Symbol Processor assists future work.

6.1 Locating Symbols in XML Documents

One artifact of the `Parallel Markup Parser` is an amalgamated symbol marker stream in which every 1-bit indicates the start or end position of some element or attribute token (Ch. 5.3). This stream can be scanned through using a `Bit Stream Iterator` (Ch. 2.4.6) to produce a sequence of start and end positions for each token[i].

Although the iterators enable icXML to cheaply locate the symbol tokens, mapping them to the appropriate data structures requires a hash table. The `XMLSymbolTable` is effectively a `XMLSymbol` factory; it relies on the `BaseSymbolTable` to map the `symbol tokens` to their unique global identifier (`gid`). All `XMLSymbol`s are retrieved from the `XMLSymbolTable` using

[i]The code-unit size is known by the Character Set Adapter at compile time.

their `gid`. This allows memory to be moved within the table without interfering with any other module.

6.1.1 Base Symbol Table

To map the `symbol tokens` to their unique global identifier (`gid`), the `BaseSymbolTable` employs a Cuckoo hashing scheme (*see below*). Cuckoo hashing trades a potentially higher insertion cost for an $O(2)$ worst-case look-up cost. Given that a typical XML Document contains relatively few unique element and attribute names, insertions are rare. However, each name will be looked up repeatedly throughout the document. Thus Cuckoo hashing seems like a natural fit for icXML but any algorithm that can map variable-length byte sequences to a `gid` is a viable alternative.

Cuckoo Hashing

When developing the Cuckoo hashing technique, Pagh and Rodler noted that most hashing schemes assume that "the hash function values were uniformly random and independent" and sought to develop one that did not [22]. Conceptually, Cuckoo hashing is relatively simple: it uses two hash functions instead of one. New entries are hashed using the primary function and greedily inserted into a hash table, kicking out any existing key occupying the chosen slot. Any displaced entry is rehashed using the secondary function and re-inserted into the table, potentially evicting yet another key. This process repeats until all entries are placed. Generalizations of this scheme use $m \geq 2$ hash functions and $n \geq 1$ hash tables but their use has not been explored in icXML.

When comparing Cuckoo hashing to regular hashing techniques, it has two major disadvantages: (1) the insertion cost can be higher and (2) infinite hashing chain arise whenever inserting item x causes item y to be evicted more than once. However, Drmota and Kutzelnigg noted there is an $\approx 81.6\%$ probability that the expected insertion cost is at most $O(4)$, subject to the condition that the table is less than half-full [14], and many techniques can mitigate infinite cycles, such as the dynamic selection of new hash functions (as recommended by Pagh and Rodler) or the expansion of the hash table itself.

6.1.2 XML Symbol Table

The function of the `XMLSymbolTable` is to construct a `XMLSymbol` object whenever a new `gid` is created. Each `XMLSymbol` object includes a `QName` object that Xerces uses to contain the qualified name (*QName*) or non-colonized name (*NCName*) of XML element and/or attribute that the `gid` refers to, and the set of grammar-specific references, which Xerces uses for validation and provides to the `application` (See Ch. 12). Because the symbol marker stream is effectively stateless, the `XMLSymbolTable` cannot distinguish between element and attribute symbols during construction. However, the syntax of an element or attribute `Name` is identical and can be safely evaluated and validated independent of its context. Although Xerces 3.1.1 conforms to either XML 1.0 4th edition or XML 1.1 2nd edition syntax specification for `Name` fields, icXML adheres to XML 1.0 5th edition and XML 1.1 2nd edition. This ensures acceptance of documents according to the latest editions of the standards and simplifies processing because of the convergence of `Name` syntax.

6.1.3 Maximum Symbol Length

icXML does have one major limitation. In Xerces, element and attribute names can be arbitrarily long. However, since icXML maps the symbol tokens to `gids` using the input data, the maximum length of any symbol is bounded to segment size $-$ 1. In practice, this limitation is a security feature since absurdly-long element and attribute `Names` are almost surely an attempt to instigate a buffer-overflow (denial-of-service) attack and with recompilation the segment size can be adjusted to any multiple of the SIMD-register width.

6.2 Assisting the Namespace Resolver

The Namespace Resolver effectively maps each QName to the appropriate Uniform Resource Identifier (URI) [RFC3986] in accordance with the namespace declarations and the scoping rules of the `Element` structure (Ch. 11). A QName is either a `PrefixedName` or an `UnprefixedName`, depending on whether it contains a colon '`:`'. Xerces parses each QName and determines its `PrefixId` of any `PrefixedName` byway of an internal string pool. Each `PrefixId` is mapped to a `URIId` by the element stack and grammar resolver.

To support namespace binding, the Symbol Processor contains a set of `XMLPrefix` objects. Each `XMLSymbol` contains an immutable `prefixId`, which refers to exactly one of them.

The `prefixId` of an `UnprefixedName` is always 0 but the `prefixId` of a `PrefixedName` is unique to each distinct prefix and is dependent on the occurrence order of the prefixes in the XML Document. Using `prefixIds` eliminates a major parsing cost of the Namespace Resolver, which in turn improves the performance of icXML.

Although it is possible to store prefixes in a hash table, they are stored in a simple unordered list. Typically, unique prefixes are rare with respect to the `XMLSymbols` themselves. As such, the cost of the hash function and the memory overhead necessary to maintain a hash table could easily exceed the cost of a linear probe. It could be beneficial to reorder the set of `XMLPrefixes` and use a binary search to locate them but value of such an optimization has not been explored at this time.

6.3 Assisting the Grammar Validator

Xerces relies on string comparisons to validate NCNames and QNames against their grammar-counterparts, which incurs a substantial performance penalty when validating an XML document. To allow for pointer-based comparisons within the Grammar Validator, icXML repurposes the `StringPool` within Xerces's `GrammarResolver` to intern each NCName and QName when constructing each `XMLSymbol` and when parsing the DTD or XML Schema grammars.

6.4 Identifying Default Attributes

Default attributes are problematic for `XML Processors`. Conceptually, they allow a document author to append attributes to an `Element` whenever the attribute is not explicitly provided for that `Element` in the document. Speciously, this does not seem to be a challenge—until one considers the possibility of default namespace-binding (`xmlns`) attributes. These attributes must be processed by the Namespace Resolver (Ch. 11) to correctly identify the Uniform Resource Identifiers (URIs) [RFC3986] of the elements and attributes in the document and potentially which grammar must be used by the Grammar Validator (Ch. 12) when validating the `Element`—but default attributes are not in the `content` or `symbol streams`: they are stored in the DTD or Schema grammar.

The DTD `ATTLIST` grammar definition provides both default and `#FIXED` attribute declarations that must be processed by the `XML Processor` as if they were part of the document itself. Internal and external DTDs are defined in the `prolog` of the XML Document, which is parsed by a modified version of Xerces's `DTDScanner`. Since the `prolog` precedes the actual data portion of the XML document, the `XMLSymbolTable` has a priori knowledge of any default DTD attributes. Identifying default attributes derived from an XML Schema, however, is vastly more complex because the correct set of default attributes is dependent on the namespace binding of the `Element` itself. However, section 3.2.6 of the XML Schema specification explicitly excludes `xmlns` as a default attribute name [5]. Thus these attributes can be safely handled by the Grammar Validator.

During `XMLSymbol` creation, any potential `DTDElementDecl` is located for that symbol. If one exists and it contains default attributes, the Symbol Processor stores the set of default-attribute `gids` and default-value string pairs in the `XMLSymbol`, creating new symbols for each as necessary. Entity resolution and whitespace normalization are performed by the `DTDScanner`. Although some work is wasted attempting to locate non-existent `XMLElementDecl`s for attributes, this occurs only once per symbol when the symbol is first encountered in the document and greatly simplifies the logic in the Markup Processor.

6.5 Handling Disparate Character Encodings

When Xerces parses a general entity from an internal or external DTD subsets, it transcodes it to UTF-16 for storage. Since icXML does not modify Xerces's schema parsing functionality, the `XMLSymbolTable` contains a secondary table to map the UTF-16 symbol tokens to `gids` of symbol tokens recorded in the primary document character set. Even if icXML were to handle DTD parsing in the future, similar functionality would still be necessary to handle external DTDs encoded with a different character set.

6.6 Validation Responsibilities

Whenever a new `XMLSymbol` is created, it entails that icXML has encountered a previously unseen symbol `Name`. The Parallel Markup Parser identifies the start and end position of each token by scanning through the `lex.NameScan` character class (Ch. 2.4.2) but that class is a proper super-set of all `NameChars`, as shown in Table 6.1. Since the cost of fully validating each `Name` in the Parallel Markup Parser is considerable, icXML fully validates each unique `Name` once, during symbol creation.

`Name`	`:=`	`NameStartChar (NameChar)*`
`NameStartChar`	`:=`	`":" | [A-Z] | "_" | [a-z] | [#xC0-#xD6] | [#xD8-#xF6] | [#xF8-#x2FF] | [#x370-#x37D] | [#x37F-#x1FFF] | [#x200C-#x200D] | [#x2070-#x218F] | [#x2C00-#x2FEF] | [#x3001-#xD7FF] |[#xF900-#xFDCF] | [#xFDF0-#xFFFD] | [#x10000-#xEFFFF]`
`NameChar`	`:=`	`NameStartChar | "-" | "." | [0-9] | #xB7 | [#x0300-#x036F] |[#x203F-#x2040]`
`lex.NameScan`	`:=`	`NameChar | [#x80-#xFFFF]`

Table 6.1: Legal Name characters vs. Lexical NameScan

Additionally when namespace processing is enabled the Symbol Processor rejects any symbol token that contains more than one colon ‘`:`’ and prohibits the creation of any `xmlns:xmlns` symbol. The XML Namespace specification states that every `Name` must be a prefixed or unprefixed QName (4) and explicitly forbids the declaration of the `xmlns` namespace and disallows the use of the `xmlns` prefix in any element QName (3). All of these errors are reported as fatal errors to the `XMLParser`.

Chapter 7

Entity Manager

The occurance of an entity in an XML document marks the position that a textual replacement will eventually occur. The name of the entity is an alias to the text that will eventually replace it. Character references (e.g., `&#28657;` or `&#x6FF1;`, which are decimal and hexadecimal Unicode values for 濱, respectively) and pre-defined references (i.e., `&lt;`, `&gt;`, `&amp;`, `&apos;`, `&quot;`) are directly substituted for their replacement text. General entities refer to user-defined strings, which can be declared either internally or externally with respect to the source document. If the replacement text of a general entity contains general entity references, they are recursively expanded.

Xerces automatically converts any character or predefined references into the appropriate character sequence but it expands general entities (both internal and external) by using a recursive reader architecture. Essentially, the `ReaderMgr` contains a stack of `XMLReader` instances, each of which can parse external DTD subsets and general entity replacement text. When a general entity reference is encountered, an `XMLReader` is instantiated, directed to the entity's replacement text and pushed onto the `ReaderMgr` stack. The `XMLScanner` then views the first character of the replacement text as the next character in the XML Document and parses accordingly. This tight integration allows the recursive instance to seamlessly work with the registered callback methods of the document instance and its associated internal data structures. Although logically simple, the imposition of a `ReaderMgr` overseeing every character of data read incurs a considerable cost on parsing in the normal (entity-less) case. Any alternative design must deal with the issue of recursive parsing implied by the occurrence of general entity references or references to DTD-defined internal or external entities). This chapter discusses the design and techniques used in icXML to provide the necessary functionality without needlessly penalizing performance.

7.1 Locating Entity References

In icXML, the entity references themselves are simply another type of symbol in the source text. Like element and attribute symbols, reference symbols can be detected with `bit scans` (Ch. 6.1) on the `Reference Opener` and `Closer` like those shown in Fig. 7.1.

```
Source Data         <doc>fee&fie;<elem attr='&foe;'></elem>fum</doc>
Reference Opener    ________1________________1______________________
Reference Closer    ____________1________________1__________________
```

Figure 7.1: Entity Reference Identification

The `Reference Openers` and `Closers` are produced by the Parallel Markup Parser (Ch. 5). Given the similarity between the Symbol Processor and Entity Manager, the `XMLEntityTable` extends the `BaseSymbolTable` class (Ch. 6.1.1). Apart from the production and use of the actual "symbol" data structures, they behave identically.

7.2 Producing Replacement Text (Objects)

The concept of replacement text is well defined in the XML specification. After an expansion is performed, the text is processed as if it were actually incorporated into the original document at the point of insertion. However, icXML represents an XML document as a content stream and symbol stream. Since the replacement text is written in XML and must be effectively inserted into an XML Document, the Entity Manager splits it into content and symbol streams during the construction of an `XMLReplacementText` object.

Intuitively, storing only fully-expanded replacement text in the `XMLEntityTable` ought to improve parsing performance. However, the Xerces API provides two entity-specific callbacks: `startEntityReference(const XMLEntityDecl&)` and `endEntityReference (const XMLEntityDecl&)`, which effectively trigger when any entity-instantiated `XMLReader` is pushed onto or popped off of the `ReaderMgr` stack, respectively. Generally these callbacks are superfluous but they are required for the proper construction and annotation of `DOM` `trees` [16]. Additionally, all line-column numbers are reported at their point of origin in the source text. This means the reported position within a general entity will be that of its replacement text in the DTD and not the location of the entity replacement in the XML document. Further, the `application` could restrict — or outright prohibit — the expansion of general entities. Given these stipulations, icXML records entities in their unmodified state.

A future enhancement to icXML could be to map any identical expanded reference to the same internal data structure (e.g., `&lt;` and `&#60;` both map to '`<`' in any context). The value of such an optimization has not been investigated at this time.

7.2.1 Entity Contexts

To distinguish between entities found within `Content` and `AttValue` spans, the Entity Manager uses the `AttValueSpanStream`, produced by the Parallel Markup Parser (Ch. 5). The n-th bit of the `AttValueSpanStream` indicates whether the n-th code-unit of the input data is within an attribute value. By extracting the bit corresponding to the reference closer '`;`', the Entity Manager can determine the context without checking the source text itself.

7.2.2 Whitespace Normalization

Character references and pre-defined entities are not subject to the normal whitespace normalization rules. General entities, however, are — but the rules differ depending on whether they are found within `Content` or `AttValue` spans. Typically general entities will not be referred to in both contexts so icXML stores a separate `XMLReplacementText` object for each case upon detecting at least one incidence of it in the particular context.

Entities are loaded in two stages. First, when icXML parses the prolog section it uses a modified version of Xerces's `DTDScanner` to load the DTD into memory. The `DTDScanner` is a character-at-a-time parser that relies on the `XMLReader` to supply it with transcoded input. The `XMLReader` automatically performs line-break normalization (Ch. 4.2) when reading the source text. Attribute-value normalization is performed within the `XMLReferenceTable` after reading the entity extracted from the DTD grammar.

7.3 Handling Entity Expansion

Superficially, whenever an entity expansion occurs, the replacement text is inserted in place of the reference marker '`;`' in the content stream — but there is room for only one code-unit in it. To overcome this limitation, icXML uses loosely-coupled recursion. I.e., any module within the Markup Processor is capable of recursively instantiating its parsing function. For simplicity, this new instance will be referred to as the `EntityParser`. When the `XMLParser` detects an entity expansion, the anterior and posterior content surrounding the '`;`' is buffered around the replacement text and a pointer to the `XMLReplacementText` object is given to the `EntityParser` in place of the content stream (Fig. 7.2).

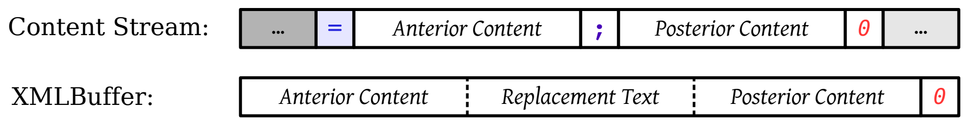

Figure 7.2: Content Stream Entity Expansion

Entity references can occur in one of two places: within attribute values and within content — but what replacement text is admissible in those cases differs considerably: in attribute values, no opening '`<`' markup delimiters may occur and any quote within the replacement text is always interpreted as a literal value. In other words, replacement text in attribute values will never alter the structure of the XML Document. In content, however, markup is permissible within the replacement text but any markup must be fully well-formed and completed within the replacement text. (A recursive `EntityParser` is only instantiated in the latter case.) These constraints are enforced during the creation of the `XMLReplacementText` object but introduce two hidden complexities:

- A start tag or empty tag could contain many attributes and each attribute may contain many references. Consequently, the `XMLBuffer` must persist for the life-time of the tag currently being parsed with all of the attribute values with entity expansions for that tag contained within it. Additionally, all whitespace characters within general entities are replaced with space characters `#x20`; whitespace character references are not modified, however.

- Since replacement text in content can contain markup, the `XMLBuffer` must be safely reusable by the `EntityParser`. The `XMLParser` will first buffer the anterior content along with `CharData` that precedes the first markup item in the replacement text. The `EntityParser` will emit it as content, reset the `XMLBuffer`, parse its own markup, and then buffer the very final portion of `CharData` within the replacement text, allowing the `XMLParser` to append the posterior text and output the resulting content accordingly.

7.3.1 Line-Column Position Calculation

The `Locator` object provides the logical XML cursor location to the `application`. In Xerces, it's constantly updated as the top-most `XMLReader` parses the XML Document, entity or grammar. In icXML, each general-entity `XMLReplacementText` object encapsulates an `XMLEntityDecl`, which contains the source position of the entity's replacement text. When a `Locator` is requested whilst parsing these entities, the `EntityParser` uses the `XMLReplacementText`'s `LineColumnDiffStream` to report the position (Ch. 9.3).

7.3.2 Reader Number Removal

To ensure that every expanded entity is well-formed, Xerces assigns an unique `ReaderNum` to each `XMLReader` upon instantiation. Whenever a start tag is added to the `ElemStack`, the `ReaderNum` of the top-most `XMLReader` is pushed along with it. Upon reaching an end tag, the current `ReaderNum` is compared against the one on the top of the `ElemStack` and a `PartialTagMarkupError` is thrown if they do not match. Prior to removing them from the system, recording and testing these `ReaderNums` accounted for up to 0.6% of the total instructions fetched[i]. Instead, when icXML parses the replacement text, it verifies the well-formedness of it; thereby eliminating the need to test it in the parser itself.

7.4 Validation Responsibilities

When icXML constructs a `XMLReplacementText` object, it knows whether it will be placed within an attribute value span or a content string. At the point of creation, the Entity Manager tests the validity of the replacement text to ensure that it meets the well-formedness constraints. That is, no entity can expand into a '`<`' in an attribute value and no external entity can be referred to within an attribute value (3.1). Additionally, every general entity that contains markup (i.e., contains at least one '`<`') is transformed and validated with the

[i]Based on before and after removal measurements obtained from `callgrind` using roads-2.gml.

Parabix Subsystem and Well-Formedness Checker to ensure that it is both syntactically legal and fully well-formed (4.3.2). Finally, the Entity Manager must ensure that no general entity can recursively call itself by any number of intermediate expansions and may have to consult with Xerces's security manager to ensure that no expansion exceeds the expansion depth or maximum size stipulations of the `application`.

Chapter 8

Content Stream Generator

When Xerces parses an XML Document, it parses every character sequentially — but the Parallel Markup Parser (Ch. 5) and Symbol Processor (Ch. 6) performs all of the syntactical analysis and extract all of the element and attribute tokens up front, eliminating the need to parse markup text in the Markup Processor.

The Content Stream Generator transforms the generally-unpredictable XML input into a near-optimal *content stream* model, which the Markup Processor processes along with the *symbol stream*. This chapter discusses the design of the Content Stream Generator and how the content stream model differs from the original XML text.

8.1 Content Stream Model

The objective of the Content Stream Generator is to produce a minimal representation of the XML document for the Markup Processor to parse sequentially. Several factors influence the overall design and requirements of the content stream.

1. After each `application` callback, Xerces reuses any memory dedicated to attributes, `AttValues` and `Content` strings.

2. In Xerces, all strings are encoded in UTF-16 and are null-terminated.

3. General entity references must be announced to the `application` through the API.

Once the Parallel Markup Parser generates the bit streams indicating the key positions within the source text, icXML can safely ignore the markup identifiers themselves. Similarly, after the Symbol Processor (Ch. 6) and Entity Manager (Ch. 7) resolves the `gids` of the element, attribute and entity tokens, the Markup Processor can bypass them. Consequently, they are filtered from the content stream by `OR`-ing their spans into the deletion mask stream. Deletion of the bits within the `U16 bit streams` is a complicated process. Currently, icXML employs Steele's parallel prefix compression algorithm [26]. It and many of its alternatives are explained in detailed in "A case study in SIMD text processing with parallel bit streams: UTF-8 to UTF-16 transcoding" [7]. The key advantage of this algorithm is that performance is independent of the number of positions deleted and the instructions required are well suited for commodity vector processors, such as SSE and NEON. Future versions of icXML are expected to take advantage of the parallel-extract operation [15] that Intel is now providing in its `Haswell` architecture.

Inverse transposition of the "compressed" `U16 bit streams` to the content stream follows deletion. It uses "SIMD merge instructions... [to] interleave fields from parallel registers to generate merged results." [7] A full description of this process can be found in "u8u16 – A High-Speed UTF-8 to UTF-16 Transcoder Using Parallel Bit Streams" [9].

8.2 Markup Identifiers

Unfortunately, the modules in Markup Processor still requires some form of identifier for every attribute value and markup tag. As shown in Fig. 8.1, seven identifiers are required in total, `=`, `>`, `/>`, `</`, `?`, `-`, and `[`. These are converted in parallel to hex-numeric codes 01, 02, 12, 03, 04, 05, and 07[i], respectively, to facilitate the construction of a jump table within each module. By virtue of the XML specification, a (potentially-empty) `Content` string follows every markup tag; ergo, no identifier is required for them.

The rationale behind the choice of these identifiers is that while a `StartTag` or `EmptyTag` can both contain an arbitrary number of `Attributes`, every `Attribute` must contain an '`=`'. Similarly, it is only possible to differentiate between a `StartTag` and an `EmptyTag` by making sure there is no '`/`' before the markup closer '`>`'. Additionally, whenever a `StartTag` or `EmptyTag` with n `Attributes` is parsed, it generates exactly $n+1$ symbol `gids` — and

[i]Note: 06 is reserved for entity references but is not set by the Content Stream Generator.

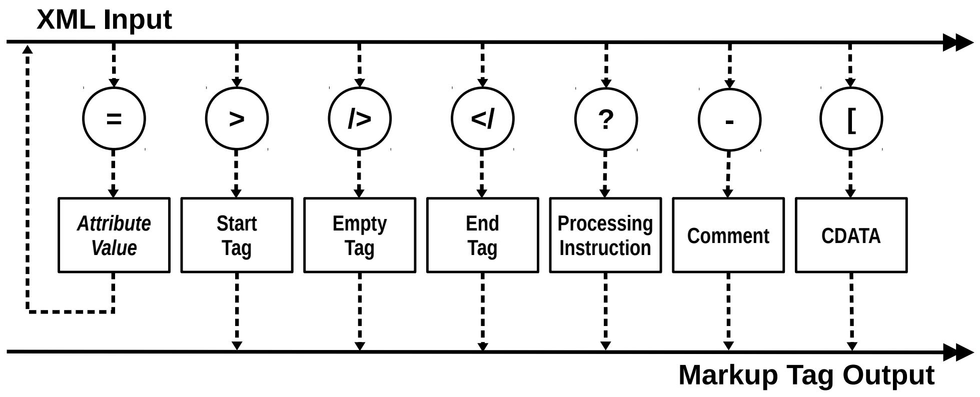

Figure 8.1: Content Stream Model

$n + 1$ identifiers. Although the `gids` are out of order with respect to the identifiers, this interrelation between the streams simplifies the consumption of `gids`[ii]. The choice of the other markup identifiers follows a similar logic.

8.3 Parallel String Construction

In Xerces, every `Content` and `AttVal` string is stored in an `XMLBuffer` and implicitly null-terminated prior to being reported to the `application`. To output text directly from the content stream, icXML null terminates these strings in place.

In any legal XML Document the `CharData` in a `Content` or `AttValue` span is always immediately followed by either a left-angle bracket `<` or closing quote {`'`,`"`}, respectively. By `OR`-ing together the bit streams marking the end of each comment, CDATA, processing instruction, closing attribute quote, and the beginning of every markup tag, the Content Stream Generator produces the delimiter marker stream. With it, the Content Stream Generator can null-terminate every string in parallel with a series of `ANDC` operations.

To bypass searching for string terminals in the Markup Processor, the Content Stream Generator uses a Bit Stream Iterator (Ch. 2.4.6) to locate the end position of each. Since general entities must be reported to the `application` and can alter the produced string, their markers are `OR`-ed into the delimiter marker stream prior to scanning it. These positions are converted into pointers referring to the delimiters within the content stream. The use of

[ii]Note: rearrangement of `gids` was explored but was deemed to have little value in terms of performance.

pointers over string lengths simplifies the segment-to-segment processing model by allowing the Content Stream Generator to ignore the existence of partial strings from a previous segment.

8.4 Expanding Entities in the Content Stream

In XML, a character or an entity reference is an alias to some replacement text that ought to be substituted into the document at the point of insertion — but they are too complex to resolve efficiently in bit space. In this regard, the Content Stream Generator acts as a filter for the Parallel Markup Parser and Entity Manager. When the Parallel Markup Parser detects a reference, it marks all of its characters for deletion except for the trailing semi-colon '`;`', which it marks the position of in the `reference marker stream` (Ch. 5.5). The Entity Manager, on the other hand, constructs `XMLReplacementText` objects for each reference as reported to it by the Parallel Markup Parser (Ch. 7.2). It also generates the `ReferenceStream`, which is a sequence of `gids` indicating the appropriate `XMLReplacementText` object for each marker in the `reference marker stream`. There are two types of reference replacements: direct and recursive substitutions.

- **Direct Substitution:** these occur when icXML encounters a character reference or a predefined-entity reference. All predefined entities and character references within the basic multilingual plane (`[0x0000,0xFFFD]`) occupy exactly one code-unit of space but references to characters in the supplementary planes (`[0x10000,0x10FFFF]`), herein known as *surrogate character references*, require two code units of space.

  Recall that the Parallel Markup Parser marks every character within a reference for deletion except for the closing '`;`' (Ch. 5.5). To ensure that there is room for a surrogate character reference, the Entity Manager must modify the deletion mask stream by inverting the bit immediately prior to the bit indicating the position of the '`;`'. This effectively expands the allotted space within the content stream by treating the expansion as a non-deletion operation.

  Although the shortest character reference that can result in a surrogate character reference is eight characters in length (e.g., `&#65536;`), icXML parses the source text in 16KB segments. As such, it is possible that the trailing '`;`' could be the first byte of the current segment. To handle this eventuality, the deletion mask stream actually contains $n + 1$ bit-blocks where n blocks is sufficient to represent the entire segment. The 0th deletion mask is initialized with all 0s. The Parallel Markup Parser begins writing the deletion mask stream on the 1st block. The Content Stream Generator effectively ignores the 0th block except to test whether the Entity Manager has modified it; in which case it knows that the very first byte of the segment terminated a surrogate character reference and allots the necessary space.

- **Recursive Substitution:** when icXML encounters a general entity reference, it knows that an n-length string will be inserted into some `Content` or `AttValue` string, effectively substituting the '`;`' with the appropriate replacement text. All general entities are reported to the `application` prior to their substitution, effectively giving the `application` an opportunity to dynamically resolve any such entities. General entities could be bypassed or restricted by the `application`.

  Note: icXML will guarantee only that the first dynamic resolution will be stored as the replacement text. Currently, it is possible for an `application` to modify the replacement text whenever a callback occurs under Xerces. Such a system would drastically complicate the Entity Manager, is illegal according to the XML specifications [3, 4] and is only indirectly supported by Xerces as a consequence of their API structure.

8.5 Validation Responsibilities

The Content Stream Generator does not validate the input. It relies on the previous modules to provide it with valid data.

Chapter 9

Line Column Tracker

Line-column tracking is a ubiquitous problem in Xerces: as the cursor moves through the transcoded data, each character is tested to see whether it's the start of an end-of-line sequence (EOL), the set of which differs depending on the XML version of the Document. Xerces solves this problem using two hard-coded character-flag tables, one for each supported XML version (1.0 and 1.1). Each table contains an 8-bit flag for every character within the basic plane (`[0x0000,0xFFFD]`). Xerces accesses the appropriate table accessed via a pointer stored in the `XMLReader` class to determine whether the EOL flag is set. This means every character of XML data has at least one conditional comparison and two memory indirections simply to test whether it is an EOL. This chapter discusses how icXML handles line-column tracking without any per-character tests.

9.1 Use Cases for Line-Column Tracking

As a standalone application, Xerces only needs the line-column position when it encounters an error within an XML Document. Xalan, however, annotates every `Element` and `Content` string with its source position so that if any errors occur during transformation, Xalan can report its original placement in the source document.

Generalizing these two cases encapsulates every possible use of line-column tracking. In the first case, it is important to note that outside of the XML conformance suite test cases, errors are exceptionally rare in XML Documents. This is because the cost of detecting and handling any error is detrimental to the performance of any validating `XML Processor` and a potential fault source for the `application`. In the second case, Xerces must provide the location of every tag, general entity reference, and content string. However, Xerces only provides the line-column number of the terminal character for each of these locations, greatly reducing the set of possible positions.

Unfortunately, XML errors can occur due to a variety of reasons, i.e., violations of the character-encoding specifications, well-formedness constraints (including those imposed by standalone documents), namespace-processing rules, and schema-related stipulations. In Xerces, character-encoding errors are caught by the `XMLTranscoder` prior to lexical analysis by the `XMLReader`. Well-formedness errors are detected by the `XMLScanner` when it "tokenizes" the input into markup-identifiers, `QNames`, attribute-value and content strings and performing syntactic analysis on the resulting tags. These types of errors are reported on the character in which they are found. Errors found during namespace resolution and grammar validation are reported on the terminal character of the tag in which they occur.

9.2 Calculating Positions in the Parabix Subsystem

Reporting the line-column position within the Parabix Subsystem implies that there is a fatal error in the source text. Upon encountering an error, the Line Column Tracker uses the `LineFeedStream`, which is a bit stream that marks the position of each EOL in the source data (Ch. 4.2), to calculate the actual position.

The Line Column Tracker uses an optimized partial-sum population count to tally the number of encountered EOLs, thereby giving the line position. When the column number of some arbitrary byte-offset is required, a combination of bit masks and bit scans is used to calculate the actual position with respect to the input data. When an error is detected within the Parabix Subsystem, the appropriate module directs the Line Column Tracker to compute the position of the error, which pushes the cost of detection to the occurrence.

9.3 Calculating Positions in the Markup Processor

Reporting line-column numbers within the Markup Processor is a more nuanced problem than within the Parabix Subsystem. The need for a line-column position within the Parabix Subsystem indicates the presence of a fatal error; ergo, the cost of the function is inconsequential as it will never occur in a well-formed document. However, when the line-column position is reported by the Markup Processor, it could be at the bequest of the `application`.

To support a future multi-threaded design, the Markup Processor is prevented from accessing the input data. Unfortunately, when the Content Stream Generator transforms the input data into the content stream, it strips out the majority of the markup text and performs whitespace normalization on the attribute value and content strings (Ch. 8.1). To solve this problem, the Line Column Tracker pre-calculates the line-column of the terminal character of every content string and markup tag, which are indicated by 1-bits in the markup delimiter stream. Using a Bit Stream Iterator (Ch. 2.4.6), the Line Column Tracker scans through the markup delimiter stream to locate the code-unit-offset of each terminal character. The offset is then converted into a line-column pair using the population count algorithm similar to the one used for the Parabix Subsystem but optimized for the case that there is at least one delimiter in every 32/64 characters (i.e., the CPU word-size).

Unfortunately, the maximum line or column position is bounded only by the file size of the document. Since documents could be extremely large, safely storing the line and column position requires either a sequence of very large integers or a complex variable-length encoding scheme to be used. Neither option is particularly effective — but storing the actual position of each terminal is not the only option.

Even though line numbers are guaranteed to be monotonically non-decreasing, column numbers are potentially random (with respect to the Kolmogorov complexity of the number sequence [12]) — but the difference between any two positions within a segment is bounded by the segment size / code-unit size. Given a segment size of $\leq$ 32KB, it is possible to represent the difference between any two positions in the same segment with a signed 16-bit integer. As such, icXML generates a sequence S of n signed 16-bit integer pairs such that $L_i = L_{i-1} + \sum_{j=0}^{n} S_j[0]$ and $C_i = C_{i-1} + \sum_{j=0}^{n} S_j[1]$, where $\{L_i, C_i\}$ is the initial line-column position of segment i and $\{L_0, C_0\}$ is the position immediately following the `Prolog` section. Since the line-column position of every markup tag, general entity and content string must be provided to the `application`, no additional work must be performed to calculate L_i and C_i at the end of every segment to keep a running tally for the subsequent one.

9.3.1 Caveat Regarding Standalone Documents

One functional deviation from Xerces is in how icXML reports the position of some standalone errors. A standalone XML document can have an internal or external schema associated with it but an external schema cannot modify the structure of the XML document.

In Xerces, standalone errors are detected and reported on the character in which they occur. Although icXML reports all standalone errors, it reports them at the subsequent line-column indicated by the markup delimiter stream. For example, icXML can report the position of an external general entity but cannot report the exact line-column of an attribute-value normalization error. The ability to provide the exact location of every possible standalone error would dramatically increase the cost and complexity of every module within the Markup Processor. Since XML documents are almost-surely error free, this deviation was deemed a non-issue.

9.4 Handling Multi-Code-Unit Characters

Every character within a character set is referred to as a code point within the code space. Some code points require multiple code units to represent a particular code point. Additionally `XML Processors` consider all end-of-line sequences, such as CR-LF, to be a single character. To calculate the correct column number, the Character Set Adapter generates a `ColumnSkipMaskStream` in which every 1-bit indicates the existence of a code unit that should be "skipped" when calculating it. The Line Column Tracker uses a population count on the relevant portion of the `ColumnSkipMaskStream` to calculate how many positions to subtract the actual column position before reporting it.

Note: in a typical XML document, skipped positions will either be non-existent or abundant. The algorithm discussed in Section 9.3 is optimized for both cases.

9.5 Validation Responsibilities

The Line Column Tracker does not validate any input. It relies on the Parallel Markup Parser and Content Stream Generator to provide it with correct data.

Chapter 10

Well-Formedness Checker

In Xerces, well-formedness checking is an integral part of the scanning process. As the cursor moves through the document, every character of text is tested to determine whether it changes the processor's markup-state. This often occurs by way of a switch table, nested if statements, or look up tables depending on the current markup context. On average, Xerces requires 6 - 13 branches per byte of schema-less XML Data [18] — the vast majority of which is dominated by this logic. This chapter discusses how icXML parses the input streams to prove the document is well-formed and — more importantly — what a priori knowledge it provides to the later stages of the Markup Processor.

10.1 Prescanning and Insertion of Default Attributes

Apart from validating the document, the Well-Formedness Checker has two major responsibilities: (1) it calculates the space needed for any memory structure within the Namespace Resolver, Grammar Validator and Document Accumulator; and (2) it tests whether any an element contains a default attribute derived from a DTD schema. Default attributes are rare but must be treated as if they occurred in the document itself. The only difference being they are not but can be are superseded by any like-named attribute that is. Unfortunately, they can be used to declare namespace bindings and may affect the correctness of a document when validated against an XML Schema. Consequently, the Namespace Resolver and Grammar Validator must be aware of them and process the document accordingly.

Since DTDs are rare and the use of default attributes rarer still, icXML is optimized for the case in which they do not occur. However, whenever they arise, they are likely to manifest many times throughout the document. To handle this case without penalizing the performance of the Namespace Resolver and Grammar Validator, the Well-Formedness Checker reconstructs the content stream, symbol stream, and string end stream so that the default attributes and their values may be effectively inserted (Ch. 6.4).

Augmented streams are built in-place as the Well-Formedness Checker assesses the document and are copied back over the original streams. As a result, the Namespace Resolver and Grammar Validator cannot differentiate between the two. The cost of constructing augmented streams is non-negligible, but paid only in the presence of default attributes.

10.2 Validation Responsibilities

The primary objective of the Well-Formedness Checker is to ensure that the XML Document is fully well-formed with respect to the XML Specification [3, 4]. Although the Markup Processor is responsible for the majority of the well-formedness and syntax validation, some tests are simply too costly to perform in bit-space.

The Well-Formedness Checker ensures that the `gid` of any `StartTag Name` matches the `gid` of its accompanying `EndTag Name`. Additionally, it tests whether the `gid` of any attribute is duplicated within any `StartTag` or `EmptyTag`. It also handles any of the post-processing logic for the Parabix Framework, such as verifying that any `<![CDATA[...]]>` tag contains the exact word `CDATA` rather than `<![` followed by some number of `NameChars`.

Chapter 11

Namespace Resolver

Namespace resolution is a fundamental requirement of any `XML Processor` [2–4]. An XML document may contain element and attribute names from multiple XML vocabularies. Namespaces help prevent naming conflicts in the `application` when two or more of those vocabularies share the same local names or when a vocabulary refers to application-dependent functions, e.g., when XML or SVG documents are embedded within XHTML files.

Each namespace refers to an Uniform Resource Identifier (URI) [RFC3986], which is a string used to identify specific abstract or physical resource, such as an XML Schema URL. Each unique URI is given an distinct `uriId`. Xerces maintains a stack of in-scope namespace bindings that are pushed (popped) every time a start tag (end tag) occurs in the document. This is a costly function considering that a typical namespaced XML document only comes in one of two forms: (1) those that declare a set of namespaces upfront and never change them, and (2) those that repeatedly modify the namespaces in highly-predictable patterns. For that reason, icXML contains an independent namespace stack that uses bit vectors to cheaply perform data-parallel speculation and scope-resolution operations. This chapter discusses how namespace binding is transformed into a bitwise problem and how icXML exploits it to improve performance.

11.1 Declaring Namespaces

Namespaces are declared by way of an `xmlns` pseudo-attribute declaration. Any unprefixed element is mapped to the default (empty) namespace, which can be bound to a specific URI using `xmlns="URI"`. Similarly, any element or attribute whose name starts with `prefix:`

can be bound to a specific namespace name via `xmlns:prefix="URI"`. For example, in Fig. 11.1, the XHTML Processor is instructed to render the string "`Rotated Text`" rotated 180°. On Line 2, the default namespace is bound to `"http://www.w3.org/2000/svg"`. This instructs the parser to interpret everything between Line 2-4 using the SVG Processor, which allows the parser to correctly interpret the text transformation. Were the parser to ignore the binding, the XHTML Processor would discard the `text` element as a superfluous HTML tag and simply display the "Rotated Text" as plain content string.

```
1. <html>
2.   <svg xmlns="http://www.w3.org/2000/svg" version="1.1">
3.     <text transform="rotate(180)">Rotated Text</text>
4.   </svg>
5. </html>
```

Rotated Text
Rotated Text

(a) With namespacing (b) Without namespacing

Figure 11.1: XML Namespace Example 1

Every namespace binding is scope dependent. Whenever an `XML Processor` encounters a `Element` with an `xmlns` declaration, that `Element` and any of its children inherit the same namespace binding. A declaration will override any existing binding but the namespace will revert back to its original state once the matching end tag is reached. For example, on Line 1 of Fig. 11.2, the default namespace is bound to `"books.org"`. To the `XML Processor`, the QName of elements, `title`, `author` and `publication` all expand to `"books.org":title`, `"books.org":author` and `"books.org":publication` respectively. On line 4, the namespace URI `"publisher.net"` is bound to the prefix `pub` and remains in-scope between the lines $4-7$. Consequently, `pub:name` is read as `"publisher.net":name` by the `XML Processor`. Line 9, however, illegally uses the prefix `pub`; this would be considered a fatal error by any `XML Processor` that supports namespaces. To correct this issue, a namespace would have to be bound to `pub` within the element on line 1, 8 or 9 to ensure it's in-scope on line 9. Note that if `pub` was bound on line 1 to something other than `"publisher.net"`, it would still be bound to `"publisher.net"` on line $4-7$.

```
<book xmlns="books.org">
  <title>Rendezvous with Rama</title>
  <author>Arthur C. Clarke</author>
  <publication year="1973" xmlns:pub="publisher.net">
    <pub:name>Gollancz (UK)</pub:name>
    <pub:name>Harcourt Brace Jovanovich (US)</pub:name>
  </publication>
  <publication year="1990">
    <pub:name>Spectra</pub:name>
  </publication>
</book>
```

Figure 11.2: XML Namespace Example 2

11.2 Handling Namespace Resolution in icXML

XML documents often contain many namespace declarations and may refer to multiple grammars, which themselves may also contain additional namespace declarations and their own grammars. In Xerces, URIs are bijectively mapped to `uriIds`. To properly validate a document, this mapping persists for the lifetime of the `XML Processor`. Since icXML relies on the Xerces's infrastructure for grammar validation, icXML also uses `uriIds` — but how it generates and resolves them is vastly different.

When the Symbol Processor encounters a new `XMLSymbol`, its Prefix and LocalPart NCNames are extracted from the `QName` and its `prefixId` and `localPartId` are determined (Ch. 6.2). The `prefixId` of any non-`xmlns` symbol is equivalent to the `localPartId` of a QName that begins with "`xmlns:`", or 0 in the case of a NCName. If `prefixId` has not been encountered before it's added to the `XMLPrefixTable`. This process builds the `XMLPrefixTable` prior to parsing any URIs.

The Namespace Resolver parses the content and symbol streams, which provide the scoping-context, URIs and the element and attribute `XMLSymbol` objects. Like Xerces, icXML maintains a global string pool to map URIs to `uriIds`. However, whenever icXML encounters an `xmlns` attribute, the mapping between the `prefixId` and `uriId` is permanently stored in the `Namespace Binding Table` (with respect to the lifetime of the `XML Processor`) even after the namespace goes out of scope. Namespace resolution is performed using a series of `Namespace Binding Set`, using simple intrinsic functions and boolean operators to determine what bindings are still in-scope.

11.2.1 Namespace Binding Table

A namespace binding is a scope-dependent mapping between a prefix and an URI. Every entry of the namespace binding table consists of a {`prefixId`, `uriId`} pair. These are either pre-defined (e.g., `xml`, `xmlns`, `xsi` and the default namespace) or encountered during processing. The index of each entry is known as the `namespace binding id` (`nsId`), which serves as an unique identifier for a particular namespace binding. For example, Fig. 11.3 provides a sample GML document and the resulting `Namespace Binding Table`. The first four entries are predefined XML namespace mappings but only $0-2$ are visible by default. (The `xsi` namespace binding is crucial for any `XML Processor` [5] but the discussion of it goes beyond the scope of this chapter.)

```
<wfs:FeatureCollection
    xmlns:gml="http://www.opengis.net/gml"
    xmlns:wfs="http://www.opengis.net/wfs"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:van="http://www.galdosinc.com/vancouver"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  ...
  <gml:featureMember xmlns:gml="http://www.opengis.net/gml">
    <van:RP2U gml:id="RP2U11742">
      <gml:description xlink:type="simple">Roads rP2U</gml:description>
      <gml:name xmlns:xlink=""/>
      ...
    </van:RP2U>
  </gml:boundedBy>
</wfs:FeatureCollection>
```

`nsId`	`Prefix`	`URI`	`prefixId`	`uriId`
0	*none*		0	0
1	`xml`	`http://www.w3.org/XML/1998/namespace`	1	1
2	`xmlns`	`http://www.w3.org/2000/xmlns/`	2	2
3	`xsi`	`http://www.w3.org/2001/XMLSchema-instance`	3	3
4	`wfs`	`http://www.opengis.net/wfs`	4	4
5	`gml`	`http://www.opengis.net/gml`	5	5
6	`xlink`	`http://www.w3.org/1999/xlink`	6	6
7	`van`	`http://www.galdosinc.com/vancouver`	7	7
8	`xlink`		6	$0/-1$

Figure 11.3: Namespace Binding Table Example

Upon seeing `xmlns:gml="http://www.opengis.net/gml"` on Line 2, the <u>sixth</u> entry is added (`nsId:5`). This is because the Symbol Processor first encounters the `wfs` prefix on Line 1 and assigns it `prefixId:4`. Normally, each prefix is bijectively bound to exactly one URI. Consequently, the Namespace Resolver reserves `nsId:4` and `uriId:4` for it, leaving a temporary gap in the table. Line 3 fills the gap for `wfs`, establishing the correct prefix-URI mapping. Line 4 and 5 add the entries for `xlink` and `van`, respectively. Line 6 formally declares the `xsi` binding but since it's already in the table, it's simply marked as being in-scope. Line 7 redeclares the `gml` namespace but does not change the mapping so no modification is made to the table. Line 10 rebinds the `xlink` namespace, which adds the 9th binding to the table (`nsId:8`). Unlike the earlier ones, however, this one is a non-bijective binding because it maps `prefixId:6` to `uriId:0`, in the case of an XML 1.0 document or `uriId:-1` in the case of an XML 1.1 document. `uriId:-1` is a special `uriId` within the Namespace Resolver because it effectively marks the namespace as unknown or unusable.

The `Namespace Binding Table` can be viewed as a relational database, consisting of two vectors, a hash table and a string pool, organized as follows:

- `PrefixToNamespaceBindingTable`: a vector of `Namespace Binding Set`s, indexed by `prefixId`, giving for each prefix the set of namespace bindings having that prefix.
- `NamespaceToUriBindingTable`: a `gid` vector, indexed by `nsId`, giving the `uriId` for each namespace binding.
- `XMLNamespaceTable`: a hash table, keyed by URI, giving the `uriId` of each predefined or encountered URI string.
- `XMLPrefixList`: an unordered list of prefix strings, for which the index in the list is also the `prefixId` of that prefix.

11.2.2 Namespace Binding Sets

A `Namespace Binding Set` is a bit vector in which each bit corresponds to a `nsId` in the `Namespace Binding Table`. The $n^{\mathtt{th}}$ bit of any `Namespace Binding Set` marks whether the $n^{\mathtt{th}}$ `nsId` is in that binding set. For example, in Fig. 11.3, the set of all bindings for the prefix "`xlink`" is MSB `000000101`.

Internally, the `Namespace Binding Set` class is implemented using an expandable set of integers whose initial length is equal to the wordsize of the target machine[i]. Boolean operations, such as `OR`, `XOR`, `AND`, and `ANDC` are all included as overloaded operators. Additional namespace-related functions, such as `mask_and_extract` and `scan_to_first`, also exist to simplify the code within the Namespace Resolver.

11.3 Assessing Namespace Visibility

During namespace resolution, icXML maintains a `CurrentlyVisibleNamespaces` variable, a `Namespace Binding Set` identifying all of the in-scope namespace bindings. As each (sequential) element is processed, any changes in namespace state associated with that element are identified in the `LocallyModifiedNamespaces` array of `Namespace Binding Sets`. These binding sets have the property that upon entry or exit of an `Element`, the set of visible namespaces can be updated as follows:

```
CurrentlyVisibleNamespaces ^= LocallyModifiedNamespaces[CurrentScope]
```

This `XOR` operation eliminates the need for maintaining an independent stack for each namespace attribute and the cost of resolving each attribute individually. When entering a start tag the `LocallyModifiedNamespaces[CurrentScope]` is initialized to 0, indicating that no namespace modification occured. If an `xmlns` attribute is found, the corresponding `nsId` is resolved through the `Namespace Binding Table` and the namespace-scope state is updated as follows:

```
if (!CurrentlyVisibleNamespaces[nsId]) {
  XMLNamespaceBindingSet hidden =
     CurrentlyVisibleNamespaces & PrefixToNamespaceBindingTable[nsId];
  XMLNamespaceBindingSet modified =
     hidden | (1 << nsId);
  LocallyModifiedNamespaces[CurrentScope] |= modified;
  CurrentlyVisibleNamespaces ^= modified;
}
```

[i]In the future, SIMD registers could be used to store the bit vectors but the additional functions are currently too costly to compute with the given SSE intrinsics.

11.3.1 Speculative URI Resolution

Before all of the `xmlns` attributes for an element are processed, it's impossible to fully resolve which `nsIds` will be in-scope and what `uriIds` to apply to each prefix. However, icXML provides fast speculative resolution for most "normal case" types of XML Documents, i.e., when each prefix uniquely denotes a distinct URI. This technique works as follows: the Namespace Resolver retrieves the `prefixId` of each non-`xmlns` attribute. If the `Namespace Binding Table` does not have an entry for that `prefixId`, the Namespace Resolver assumes that it's canonical, i.e., its `prefixId = uriId`. Otherwise, it assumes that the `uriId` will remain unmodified with respect to its last known state. By definition, any canonical set will always be correctly speculated.

A `CanonicalBindingSet` marks any canonical bindings in the `Namespace Binding Table` with a 1. For example, the `CanonicalBindingSet` for Fig. 11.3 is MSB `111111110`.

11.3.2 Full URI Resolution

If speculation fails, then the Namespace Resolver can rely on the a priori knowledge given by the Well-Formedness Checker that no two `xmlns` attributes are identical for the same element. Therefore, there must be a nonbijective relationship between the namespace mappings, i.e., some `prefixId` and/or `uriId` is associated with more than one `nsId`. To determine which mapping is the correct (in-scope) mapping, the following resolution process is required for each attribute.

1. Using the `prefixId` of the attribute, retrieve the set B of all namespace bindings that may apply:

   ```
   B = PrefixToNamespaceBindingTable[prefixID];
   ```

2. Determine which one, if any is visible.

   ```
   V = B & CurrentlyVisibleNamespaces;
   ```

3. Determine the `nsId`. At most, one namespace can be visible at any given time.

   ```
   nsId = scan_to_first(V);
   ```

4. Lookup the canonical `uriId`.

   ```
   uriId = NamespaceToUriBindingTable[nsId].uriId;
   ```

11.4 Supporting the Grammar Validator

Although the Namespace Resolver generates the `uriIds` for the elements and attributes, the Grammar Validator sometimes requires namespace mappings of entities that are not present at that point in the document. For instance, the value of an `xsi:type` attribute is a QName whose prefix must be resolved to the correct URI. To handle any scenario in which namespace resolution must be performed outside of the Namespace Resolver, it generates a `context id stream`, which is a sequence of `gids` in which each refers to a particular set of visible namespaces. Each unique set of namespace bindings is given a `context id`.

11.5 Processing XML Schemas in icXML

In Xerces, XML Schemas are loaded the moment that the `XMLScanner` encounters an `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation` attribute; the value of which indicates the location of the schema document.

Each XML Schema could contain its own namespace bindings and may import (or include) its own schemas. Because icXML requires that all {URI, `uriId`} bindings remain consistent throughout the lifetime of the `XML Processor`, the `XMLNamespaceTable` must be shared by every instance of a Namespace Resolver of any sub-document. Conceptually, this does not pose any problems. In practice, however, it does degrade performance when trying to establish maximum number of canonical binding sets.

Since the URIs declared within schemas are typically not used by the parent document, this can lead to very large gaps in the `Namespace Binding Set` space, which reduces the effective value of each bit in the binding sets. icXML mitigates this in two ways:

1. The Namespace Resolver makes special note of any attribute bound to the `xsi` namespace and stores the value of any `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation` attribute. Following namespace resolution (of the 16KB segment), icXML loads any previously unparsed schema documents indicated by any of the values. Indirectly, this seperation greatly increases the probability that the namespaces within the main XML document will be maximally canonical because typical XML documents will contain all possible namespace bindings within the first 16KB of raw text.

2. Although icXML uses a global `XMLNamespaceTable`, any encountered namespace is stored within a local `XMLNamespaceTable` with respect to the document being parsed.

By using the `XMLNamespaceTable`, the Namespace Resolver effectively discards any namespace bindings found within a schema, which results in a greater number of canonical sets.

11.6 Validation Responsibilities

Apart from reporting the illegal use of `xmlns` as an element name, the Namespace Resolver is responsible for reporting all namespace related errors that are not covered by well-formedness rules:

- **Reserved Prefixes and Namespace Names:** The `xml` and `xmlns` namespaces are bound to "http://www.w3.org/XML/1998/namespace" and "http://www.w3.org/2000/xmlns/" by default. `xml` can be declared but can only be bound to its default namespace. Neither the `xmlns` nor the xmlns namespace can be declared or bound any other prefix respectively.
- **Prefix Declaration:** Excluding `xml` and `xmlns`, every namespace prefix must be declared within the element or be in-scope prior to being used. Only XML 1.1 allows prefixes to be undeclared with `xmlns:prefix=""`.
- **Uniqueness of Attributes:** Although the well-formedness constraint requires that an element cannot contain more than one attribute with the same name, namespacing further requires that no two attributes have the same expanded name. In other words, if two or more attributes are qualified names with the same `LocalPart`, the `Prefix`es cannot be bound to identical URIs [2]. Since DTDs have no concept of namespacing, this restriction is applied after appending any default attributes that are a result of a DTD `ATTLIST` grammar declaration.

Chapter 12

Grammar Validator

In Xerces, an instance of one of the `XMLScanner` classes drives the parsing of an XML Document. Each piece of markup is extracted sequentially, potentially validated, and immediately given to the `application`. This constant switching between markup parsing and `application` code was shown to degrade cache utilization, prefetcher and branch-predictor behaviour [20]. icXML uses heavily-modified versions of the `XMLScanner` classes, which are templated into the Grammar Validator framework. Any parsing-related functionality was removed and the interface was completely restructured to take advantage of the streams built by and the validation performed by the previous modules. Instead of providing the output directly, all data is buffered in the Document Accumulator.

12.1 Fundamental Differences between icXML and Xerces

In Xerces, the raw text for a markup tag or `Content` string is accumulated in an `XMLBuffer`. The text is then validated and transformed into the appropriate output for the `application`. In the case of `Element` tags, Xerces first determines which `Grammar` is applicable to the `Element` then uses a hash table associated with that `Grammar` to look up the element `Name` and extract an `XMLElemDecl`. Any attributes (in the case of a start or empty tag), are converted into `XMLAttDefs` by way of a hash table stored within the `XMLElemDecl`. Validation is then performed on the resulting objects and normalized attribute values. This system works well when handling XML documents with schemas that are associated with multiple namespace bindings or vice versa — but this type of document is exceptionally rare.

Instead, in icXML each `XMLSymbol` (Ch. 6.1.2) contains a dynamic list of `XMLElemDecls` and `XMLAttDefs` that have ever been associated with that `XMLSymbol`. The list is annotated with the information necessary to determine which object to use, such as the namespace binding and element scope. For typical documents, each list will contain exactly one `XMLElemDecl` or `XMLAttDef`. As such, icXML uses a simple unordered list to store them, thus discarding the overhead of a hash table in the normal case. When the `XMLSymbol` does not contain the desired object, icXML uses the `Grammar`'s hash table to locate it and updates the list accordingly. Validation is then performed in a similar manner to Xerces but the `DTDValidator` and `SchemaValidator` were optimized to take advantage of interned `Names` (Ch. 6.3).

12.2 Document Accumulator

The Document Accumulator is a new feature in icXML with respect to the version presented in "icXML: Accelerating a Commercial XML Parser Using SIMD and Multicore Technologies". Conceptually, all data that would have been provided to the `application` by the Grammar Validator is instead buffered in a `DocumentObject` and `DocumentContent` stream. The former is a sequence of `XMLElemDecl` and `XMLAttDef` objects (as well as any others required by the Post-Schema-Validation Infoset) and the latter is essentially a schema-normalized content stream. These are then parsed by the Document Disseminator, which is a templated class that wraps the `XMLParser` object that was instantiated by the `application` to parse the XML Document.

Currently, only the Grammar Validator is capable of transforming the input streams into the data required by the Document Accumulator. This is far from ideal, however, since not all XML documents have schemas nor is validation always enforced. Even though standard output of the SAX, SAX2 and DOM `XMLParser` classes consists of UTF-16 strings, efforts to make the Grammar Validator an optional module were hampered by existence of so called `advanced handlers`. In Xerces, an advanced handler is a user-defined class that is installed in the `XMLParser` at runtime. An `XMLParser` can have any number of advanced handlers. Each are provided with the schema-derived (or faulted in) objects that internal to Xerces immediately after Xerces provides the standard output to the `application`. Even when validation is disabled, Xerces locates the appropriate grammar object in memory to provide the advanced handlers with the correct instances but bypasses the actual validation test.

Additionally, the concept of the accumulator and disseminator was based around the notion that the `application` would perform a substantial amount of work — but for extremely simple applications, the buffering cost could exceed the benefit. A better method would require that the output of any module within the Markup Processor could be directly fed to the Document Disseminator, making the Document Accumulator an optional feature but that has not been explored at this time.

12.3 Identity Constraints

The XML DTD and XML Schema specification both provide mechanisms for ensuring the uniqueness of certain identifiers within the XML Document. Whereas `ID` fields, as defined in the XML specification (3.3.1), require that an identifier must be unique throughout the document instance, identity constraints (XML Schema specification 3.11) use `XPath` queries to enforce the uniqueness of identifier relative to a specific `Element`. The latter provides a flexible mechanism for asserting uniqueness — but in so doing, requires a vastly more complex system to ensure it.

In Xerces, processing these constraints is an integral part of the `XMLScanner` processing logic. Each constraint is effectively processed by an internal advanced handler that receives start and end element events from the `XMLScanner` as it processes the file. Because this parsing logic is optional and complex, in icXML the Grammar Validator performs all of the standard validation first and then if and only if it observed (or is already processing) an identity constraint, does it rescan the document to validate them.

12.4 Validation Responsibilities

The Grammar Validator handles all DTD and/or XML Schema-based validation of an XML Document to the extent that Xerces was capable of supporting them. The full description of the DTD and XML Schema grammar goes beyond the scope of this report. For more information, please refer to the XML specification [3, 4] and the XML Schema specification [5].

Chapter 13

Performance Evaluation

This chapter evaluates icXML and Xerces against three benchmarks: (1) raw XML parsing using `SAXCount` and `WFXMLScanner`; (2) grammar parsing and validation using `SAX2Count` and `IGXMLScanner`, and (3) a real world GML to SVG transformation `application` using the `IGXMLScanner` and SAX2 interface. Both icXML and Xerces were instrumented using PAPI Version 5.1.1.0. The performance measurement counters (PMCs) were obtained from an Intel Core i7 quad-core (Sandy Bridge) processor (3.40GHz, 4 physical cores, 8 threads (2 per core), 32+32 KB (per core) L1 cache, 256 KB (per core) L2 cache, 8 MB L3 cache) running the 64-bit version of Ubuntu 12.04 (Linux). Due to the high volatility of some of the PMCs, each benchmark consisted of $1,000 + 1$ trials. The first trial was discarded and the median value of the remaining runs was recorded for each PMC.

13.1 Evaluation Datasets

Table 13.1 shows the document characteristics of the XML instances selected for this performance study, including both document-oriented and data-oriented XML files. “arw.xml”, “jaw.xml” and “dew.xml” are document-oriented XML instances of Wikimedia books, written in Arabic, Japanese and German, respectively. The remaining files are data-oriented. “roads-2.gml” file is an instance of Geography Markup Language (GML), a modelling language for geographic information systems as well as an open interchange format for geographic transactions on the Internet. “po.xml” is an example of a standard purchase order and “soap.xml” contains a large SOAP message. Markup density is the ratio of all markup (including attribute values and entity references within `Content`) vs. the total file size.

File Name	*Document Workloads*			*Data Workloads*		
	arw.xml	dew.xml	jaw.xml	roads-2.gml	po.xml	soap.xml
File Size (KB)	113,082	66,240	7,343	11,584	74,658	2,653
Elements	684,641	203,396	37,441	140,362	2,317,055	90,002
Attributes	62,747	18,808	3,529	160,418	463,397	30,001
References	555,377	796,851	89,949	0	0	0
Markup Count	1,346,143	400,978	73,979	260,673	4,634,110	170,004
Markup Density	10.6%	5.4%	9.0%	70.9%	57.2%	85.7%

Table 13.1: XML Document Characteristics

13.2 XML Parsing Performance

The first test evaluates icXML and Xerces-C 3.1.1 using the `SAXCount application` and the `WFXMLScanner` scanner module. This `application` reports the total number of elements and attributes and the total length of all `Content` within a given XML document. The `WFXMLScanner` is the simplest `XMLScanner` for both icXML and Xerces: its sole purpose is to test whether a document is well-formed and report the structure of the document to the `application`. While the `WFXMLScanner` is capable of namespace processing (Ch. 11), it can neither parse nor validate a document against any schema.

The purpose of this test is to measure the raw parsing speed of both icXML and Xerces but since icXML was designed with the assumption that the `application` would perform a significant amount of work, this test favours Xerces in terms of relative speedup. As Fig. 13.1 shows, icXML out-performs Xerces with respect to throughput — but interestingly, Table 13.2 shows it behaves worse in terms of data cache (*D-Cache*) and instruction cache (*I-Cache*) utilization, which starkly contrasts my findings presented in "icXML: Accelerating a Commercial XML Parser Using SIMD and Multicore Technologies" [20]. (Note: *DCM*, *ICM*, and *TCM* stands for D-Cache, I-Cache, and Total Cache Misses, respectively.)

However, since that paper was submitted, icXML underwent extensive modifications in preparation for a scalable multicore implementation. The multicore version presented in that paper only split the Parabix Subsystem and Markup Processor into two separate threads and did not use the Document Accumulator concept. The resulting increase in cache accesses may explain the discrepancy between the current version of icXML and the one presented in the paper – especially in the case of "po.xml", which exhibited the greatest increase in total cache accesses, cache misses and the worst overall speedup (≈ 40%).

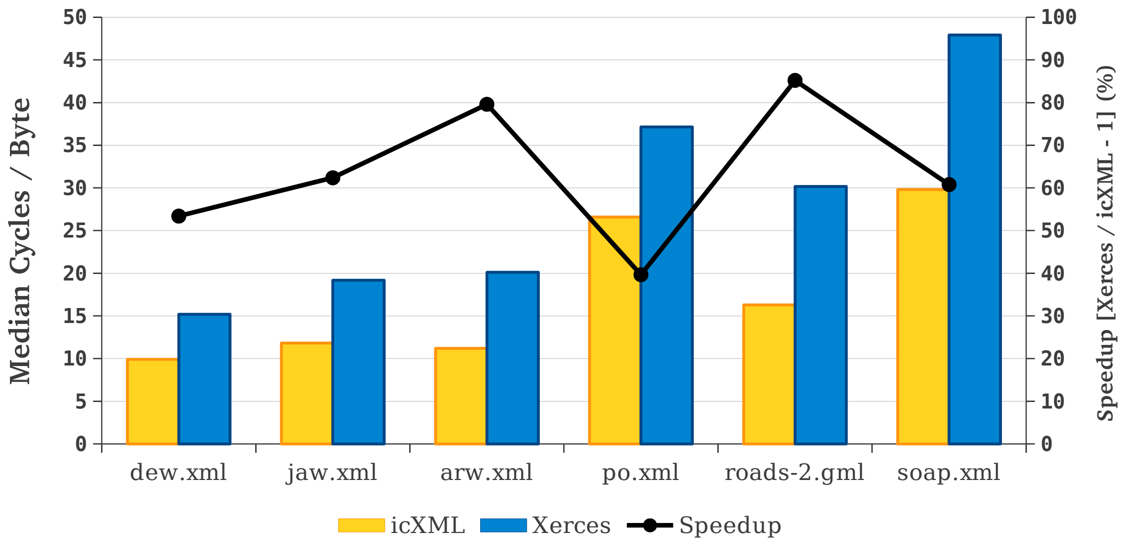

Figure 13.1: Performance of icXML and Xerces using WFXMLScanner and SAXCount (Sorted by Markup Density)

Data	Library	L1 DCM	L1 ICM	L2 DCM	L2 ICM	L3 TCM
dew.xml	icXML	219.3	14.3	13.5	4.8	0.1
	Xerces	121.9	7.7	3.7	2.5	0.1
jaw.xml	icXML	233.5	16.4	11.2	5.1	0.3
	Xerces	93.4	8.9	4.9	3.3	0.2
arw.xml	icXML	221.1	15.3	10.3	4.0	0.1
	Xerces	96.5	9.1	3.4	2.6	0.0
po.xml	icXML	378.8	12.6	19.9	5.2	0.0
	Xerces	110.0	24.3	2.9	1.2	0.0
roads-2.gml	icXML	287.9	13.6	13.8	4.3	0.2
	Xerces	124.2	16.3	4.0	2.1	0.1
soap.xml	icXML	405.9	19.4	28.4	8.6	0.6
	Xerces	121.0	40.6	5.2	2.7	0.3

Table 13.2: Median Cache Utilization with Well-Formedness Checking per KB input

To determine whether this problem was isolated to a few modules, the above experiment was re-run on icXML with each module individually instrumented. The PMCs were normalized against the total observed in Table 13.1. The results are displayed Table 13.3 and Table 13.4, detailing the average cache utilization in Document and Data workloads respectively[i]. (Note: the *Miss Rate* column gives the misses to accesses ratio.)

Interestingly, the Character Set Adapter and Parallel Markup Parser layer (CSA+PMP) exhibits the worst cache utilization for both types of workload but it also performs the greatest amount of work (with respect to total cycles). Surprisingly, it appears that $18-21\%$ of the L1 instruction cache misses are L2 misses as well. This implies that its I-Cache footprint is causing the processor to evict the cached instructions, which is likely introducing cache stalls — exactly what the layered design was intended to avoid.

The decision to fuse the CSA and PMP to reduce the D-Cache footprint of the modules likely hurt I-Cache utilization but given that there were less than 20 L1 I-Cache misses per KB of input data in all cases, its unlikely that this is significantly affecting performance. However, Table 13.2 shows that L1 and L2 D-Cache misses could be a concern. Unfortunately, every byte of data read by the CSA is a new byte of input from the input stream. Optimizing icXML's I/O architecture ought to reduce this — but the degree of improvement is limited by the file stream.

Optimizing the D-Cache usage of the Content Stream Generator (CSG) is a difficult challenge: generating the content stream is inherently a sequential process — but only because the cost of deletion is significant enough that it is counterproductive to isolate the inverse transposition, entity replacement and string delimiter calculation stages within it. However, with some careful reorganization it may be possible to lower the D-Cache miss rate by rearranging the input to work more efficiently with the prefetchers.

For data-based workloads, many of the modules appear to be fairly well balanced but the L3 misses of all the modules account for less than 50% of all L3 misses. Given the simplicity of the SAXCount application, the greatest contributor to the remaining L3 misses is likely the file I/O and segment-to-segment buffer copy mechanism. However with ≤ 0.6 L3 misses per KB, even a substantial reduction is unlikely to improve performance.

[i]Note that none of the PMC columns total to 100% as each data point represents only the portion the particular module contributes to the observed measurement over the lifetime of the library and application.

Module	L1		L2				L3		Total
	DCM %	**ICM %**	**DCM %**	**Miss Rate**	**ICM %**	**Miss Rate**	**TCM %**	**Miss Rate**	**Cycles**
CSA+PMP	25.9	25.6	11.3	3.3	19.6	17.9	4.6	0.2	31.8
SP	2.7	4.3	2.8	7.8	4.9	22.0	5.8	1.1	6.1
EM	2.5	4.0	1.9	5.6	2.6	14.8	1.9	0.5	7.4
CSG	23.3	13.6	9.7	3.1	9.6	16.4	19.4	0.6	22.9
LCT	3.2	1.7	1.1	2.6	1.7	17.7	0.2	0.1	4.4
WF	3.7	8.4	4.0	8.2	8.8	21.2	19.8	1.2	3.3
NR	1.7	7.9	2.7	11.8	6.6	16.6	4.0	0.6	5.6
GV	5.7	8.0	6.8	9.2	6.3	17.8	10.6	0.6	6.9
DD+API	1.9	3.4	1.3	5.8	6.8	37.8	1.1	0.2	5.3
	70.6	77	41.5	–	66.9	–	67.4	–	93.5

Table 13.3: Average Per Module Ratio of Median Cache Misses when Parsing Document Workloads with icXML

Module	L1		L2				L3		Total
	DCM %	**ICM %**	**DCM %**	**Miss Rate**	**ICM %**	**Miss Rate**	**TCM %**	**Miss Rate**	**Cycles**
CSA+PMP	15.9	22.6	10.0	4.7	14.7	21.4	8.2	0.4	16.6
SP	5.6	5.1	3.1	4.0	4.5	25.9	10.5	1.6	14.2
EM	0.6	1.9	0.3	3.4	1.9	26.4	0.4	0.4	0.3
CSG	14.0	11.8	3.6	1.9	8.5	25.4	2.8	0.3	14.8
LCT	3.3	1.9	1.1	2.4	1.9	26.8	0.3	0.1	4.8
WF	7.6	9.1	4.7	4.6	11.7	36.5	13.5	0.9	4.6
NR	6.3	10.5	3.8	4.6	10.5	27.6	6.3	0.5	12.8
GV	18.7	9.0	9.3	3.7	8.5	28.2	5.4	0.3	15.0
DD+API	10.3	4.2	1.6	1.2	7.1	45.6	2.2	0.4	13.2
	82.3	76.1	37.6	–	69.3	–	49.4	–	96.3

Table 13.4: Average Per Module Ratio of Median Cache Misses when Parsing Data Workloads with icXML

Data	Branches		Mispredictions		Mispredictions %	
	icXML	Xerces	icXML	Xerces	icXML	Xerces
arw.xml	1896.9	9841.1	55.0	183.0	2.9	1.9
dew.xml	1592.3	7666.0	47.4	113.7	3.0	1.5
jaw.xml	1825.1	9151.5	54.6	151.0	3.0	1.6
po.xml	6008.6	17952.9	120.8	129.5	2.0	0.7
roads-2.gml	3291.8	14961.1	59.4	137.8	1.8	0.9
soap.xml	6792.0	23568.1	99.0	172.1	1.5	0.7

Table 13.5: Branch Behaviour in icXML and Xerces per KB input

Finally, as shown in Table 13.5, branch behaviour in icXML is significantly better than Xerces. Proportionately, icXML has a higher misprediction rate but also a $67 - 81\%$ reduction in branch instructions. Interestingly, "po.xml" again exhibits the worst behaviour. This file warrants additional study to determine what properties are causing the degradation.

13.3 Grammar Validation Performance

This experiment reevaluates icXML and Xerces against the document workloads, "arw.xml", "dew.xml" and "jaw.xml", using the `IGXMLScanner` and `SAX2Count application`. The `IGXMLScanner` is the most feature-rich `XMLScanner` provided by Xerces. It is capable of parsing and validating a document against a DTD and/or any number of XML Schemas, which necessitates the ability to perform general entity expansions and namespace processing. `SAX2Count` is functionally identical to the `SAXCount application` except that it adheres to the SAX2 interface. Apart from enforcing a slightly different API structure, its major functional difference is that reports any change to the namespace state through the `startPrefixMapping` and `endPrefixMapping` callbacks and suppresses all `xmlns` namespace declarations from the standard document API. The documents are dependant on the "export-0.3.xsd" XML Schema, which imports "xml.xsd", and is transitively dependant on "XMLSchema.dtd". These schemas were modified to point to local copies of the grammar to eliminate network transfer costs from the PMCs. Grammar caching was disabled.

The purpose of this test is to determine the performance of icXML and Xerces when processing document-based workloads. Typically, data-based workloads have complex schemas and are validated only during production. Because icXML and Xerces uses the same internal data structures to construct the `Grammar` object, icXML's DTD and XML Schema parsers

File Name	**export-0.3.xsd**	**xml.xsd**	**XMLSchema.dtd**
Schema Type	XML Schema	XML Schema	DTD
File Size (KB)	4.8	4.8	16.1

Table 13.6: Schema Characteristics

are functionally identical to Xerces's. The only major difference between is that icXML's versions took advantage of its own string interning and namespace processing capabilities. However, both libraries share a similar grammar construction overhead.

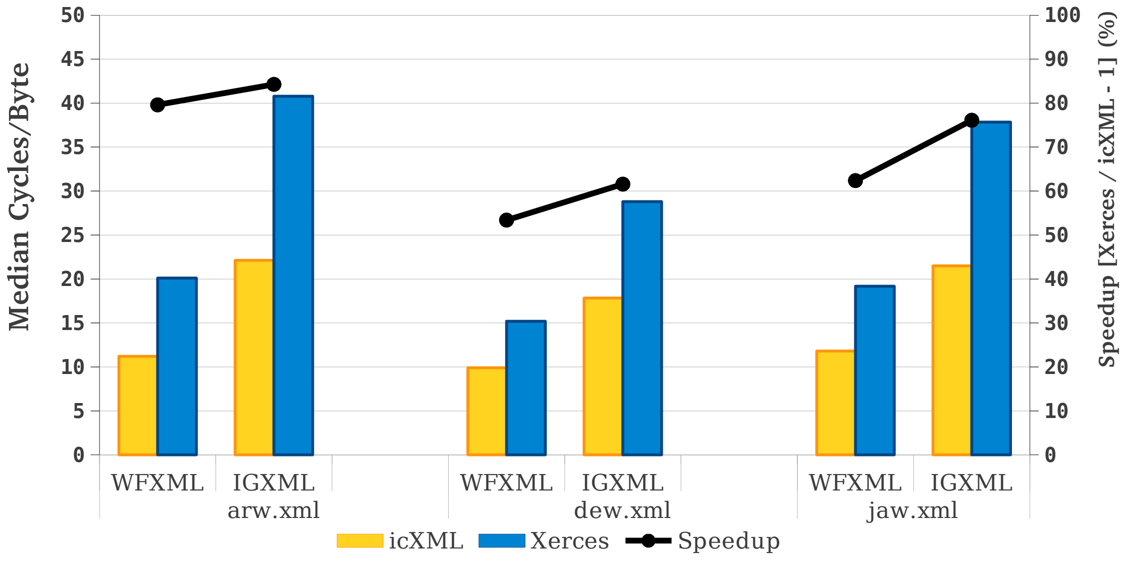

Figure 13.2: Performance of icXML and Xerces using IGXMLScanner and SAX2Count

As expected, schema validation significantly increases the cost of XML parsing for both icXML and Xerces, nearly doubling it in most cases (Fig. 13.2). icXML showed an improvement in the relative performance when comparing its speedup of the datasets in the previous benchmark to this one. Interestingly, while "arw.xml" still exhibited the best speedup when comparing icXML to Xerces, its relative speedup when comparing the (WFXML) results from the last dataset to the (IGXML) figures of this experiment was the worst. If markup density was the key factor, "jaw.xml" should have displayed a similar behaviour and

Data	**Library**	**L1**		**L2**		**L3**
		DCM	**ICM**	**DCM**	**ICM**	**TCM**
arw.xml	icXML	407.6	185.5	66.9	39.6	0.2
	Xerces	466.7	758.8	43.2	40.7	0.2
dew.xml	icXML	386.7	118.9	76.9	36.7	0.4
	Xerces	368.2	400.0	45.7	34.9	0.4
jaw.xml	icXML	391.9	170.3	64.8	39.7	1.0
	Xerces	417.9	649.4	46.5	41.6	0.8

Table 13.7: Median Cache Misses with XML Schema Validation per KB input

"dew.xml" the greatest increase. Since each workload is validated against "export-0.3.xsd", its difficult to ascertain why this occurred — but it may be related to cache utilization.

Cache behaviour in this benchmark, presented in Table 13.7, is strikingly different than the previous experiment. As expected, D-Cache and I-Cache access and miss rates increased considerably but unlike that test, icXML is comparable to Xerces in terms of L1 D-Cache and L2 I-Cache utilization but $\approx$ 35% worse than Xerces with respect to L2 D-Cache behaviour and $\approx$ 6% worse in L3 miss rate. L1 I-Cache usage, however, shows a substantial improvement: $\approx$ 275%. These results are in-line with but not quite as good as the results shown in "icXML: Accelerating a Commercial XML Parser Using SIMD and Multicore Technologies". Again, the additional layering is the likely culprit.

When comparing the results shown in Table 13.2 and Table 13.7, the "arw.xml" workload showed the greatest relative increase in L1 and L2 cache misses but the lowest for L3 cache misses. This file warrants additional study to determine whether this is an effect of the file I/O and segment-to-segment buffer copy system or if it has any underlying properties that could be affecting performance.

13.4 GML2SVG Performance

For the final benchmark the GML-to-SVG (GML2SVG) application was chosen to evaluate icXML and Xerces using a substantial application that was designed for Xerces. This application transforms geospatially encoded data represented using an XML representation in the form of Geography Markup Language (GML) [17] into a format suitable for displayable maps: Scalable Vector Graphics (SVG) format [19]. In the GML2SVG benchmark, GML feature elements and GML geometry elements tags are matched. GML coordinate data are then extracted and transformed to the corresponding SVG path data encodings. Equivalent SVG path elements are generated and output to the destination SVG document. The GML2SVG application is thus considered typical of a broad class of XML applications that parse and extract information from a known XML format for the purpose of analysis and restructuring to meet the requirements of an alternative format. The SVN source code repository for this application is located at “http://parabix.costar.sfu.ca/svn/proto/gml2svg”.

GML	**File Size (KB)**	**Markup Density**	**Max Content Length (B)**	**Cycles/Byte icXML**	**Cycles/Byte Xerces**	**Speedup (%)**
rp2u	11583.8	57.2	2468	37.3	55.9	49.6
rl1u	4175.2	54.7	3776	37.7	54.2	43.8
rp4u	1220.4	58.3	644	37.5	56.8	51.5
singletrack	296.8	54.8	1283	37.3	52.8	41.3
rl2u	274.4	53.3	1661	38.8	53.9	39.0
school	253.7	47.9	1728	37.4	48.1	28.8
rp4d	245.3	56.8	1382	38.1	56.3	47.6
bridge	218.1	63.5	508	36.1	59.9	65.9
rrough	214.3	39.4	4658	40.2	43.9	9.3
church	207.0	56.9	441	35.5	53.8	51.5
transmissiontower	185.8	45.2	407	36.5	44.2	20.9
rp2u1w	165.7	51.7	1050	39.6	52.7	33.3
ocean	**135.1**	**4.4**	**45595**	**58.7**	**18.4**	**-68.6**
transmissionline	118.1	63.9	439	35.9	57.0	58.6
lake	**102.6**	**24.9**	**20979**	**45.5**	**33.0**	**-27.4**
lightrailtransit	44.6	63.9	237	38.9	58.3	49.8
hospital	43.5	41.8	1425	41.8	45.1	8.1
spur	43.4	50.4	832	42.3	53.3	26.2
riverb	24.1	53.0	1248	44.5	54.9	23.5
footbridge	22.3	64.2	237	42.0	62.1	47.7
college	21.7	49.6	1522	46.0	53.0	15.3
firestation	18.2	58.1	371	45.0	58.1	29.2

GML	File Size (KB)	Markup Density	Max Content Length (B)	Cycles/Byte icXML	Cycles/Byte Xerces	Speedup (%)
unspecifiedbuilding	16.9	41.7	976	48.4	46.9	-3.2
multipletrack	16.8	49.4	946	49.7	54.0	8.7
postoffice	13.1	57.2	305	49.7	59.8	20.3
trestle	12.5	68.3	67	48.1	68.5	42.3
policestation	11.0	56.8	406	52.5	60.4	15.1
cutearthwork	10.7	40.6	1011	55.3	50.8	-8.1
fillembankment	9.7	53.1	643	55.5	59.6	7.4
doubletrack	8.3	64.8	265	57.4	69.4	21.0
pipeline	8.0	61.4	270	60.0	68.6	14.4
reservoir	6.8	51.9	574	65.0	63.2	-2.8
rp3u	6.5	43.0	1144	66.4	59.4	-10.6
retainingwall	6.3	63.7	203	62.7	70.9	13.2
greenhouse	6.0	59.3	330	67.0	69.8	4.2
cityhall	4.7	56.7	441	75.4	72.2	-4.3
buildup	4.3	21.4	3190	86.2	49.9	-42.1
communications	3.6	63.0	169	90.6	81.1	-10.6
ferryroute	3.3	56.1	742	90.1	77.5	-14.0
tunnel	3.2	67.6	101	91.5	88.7	-3.0
courthouse	2.2	61.8	304	125.4	96.1	-23.3
university	2.0	45.1	909	127.6	86.0	-32.6
abandonedtrack	1.9	63.9	305	140.4	102.7	-26.9
ferryterminal	1.6	59.5	405	154.8	108.0	-30.2
rp6u	1.6	73.1	101	152.0	118.8	-21.8
tailingpond	1.3	69.5	168	186.4	125.3	-32.8

Table 13.8: GML Datasets (in Decreasing Order of File Size)

Internally, the GML2SVG application uses the (default) `IGXMLScanner` but the gml files themselves are not associated with any schema — which is standard for GML files given the complexity of the grammar. As such, both icXML and Xerces dynamically construct

`DTDElemDecls` and `DTDAttDefs` in memory to represent the "faulted in" elements and attributes. The GML to SVG data translations were executed on GML source data modelling the city of Vancouver, British Columbia, Canada. The GML source document set consists of 46 distinct GML feature layers ranging in size from approximately 1.3 KB to 11.3 MB and with an average document size of 130 KB. Markup density ranges from approximately 4.4% to 73.1% and with an average markup density of 55.7%. The 19.3 MB of source GML data generates 9.1 MB of SVG data. These workloads are listed in Table 13.8 in descending order of document size.

	L1		**L2**		**L3**	**Total**	
	DCM	**ICM**	**DCM**	**ICM**	**TCM**	**Instructions**	**Cycles**
icXML	388.8	79.4	73.6	32.2	2.0	81,479.1	38,838.4
Xerces	410.6	725.6	42.3	24.8	0.1	103,484.3	56,166.9
Change (%)	-5.6	-813.9	42.5	23.2	93.3	-27.0	-44.6

Table 13.9: Average GML2SVG Performance (per KB input)

Overall, icXML provided an average speedup of 44.6% over Xerces yet reduced the total instructions retired by only 27.0% (Table 13.9). Given the significant reduction in L1 I-Cache misses, it is possible that icXML's segment-based parsing model is improving the performance of the GML2SVG `application` by making more conservative use of the I-Cache space. More investigation is necessary to make a definitive claim.

However, in many cases involving small files, GML2SVG was faster with Xerces than icXML. Initially, as shown in Table 13.8, I observed that icXML tended to perform worse than Xerces when parsing gml files that were under 6KB, which I believe is an effect of the 16KB segment size window. However, this does not explain "ocean.gml" or "lake.gml", which are 135KB and 104 KB in size, respectively. Both files have a relatively low markup density — yet markup density does not appear to be a reliable indicator of performance in GML2SVG ($R < 0.2$). This is not surprising: it is just as difficult to transform a few long sequences of coordinates as it is to convert many short sequences.

A unique property of both "ocean.gml" and "lake.gml" is that they both have the longest `Content` spans of any of the tested datasets. More importantly, they are the only files in which icXML would be forced to expand the content stream (within the segment-to-segment buffer copy mechanism) to fit the entire `Content` string within it. Although Xerces is faced with a similar challenge, it has to contend with only a single string whereas icXML must heap allocate enough to handle a full segment's worth of data. This is likely causing havoc with the D-Cache and TLB yet neither file is large enough that the cache behaviour can stabilize sufficiently. These files warrant additional study to determine whether this is correctable or simply an unfortunate side-effect of icXML's segment-based processing model when handling extremely long `Content` strings.

Chapter 14

Conclusions

icXML is a large project. Initially, it began as a research prototype to determine whether it was feasible to introduce parallel bit streams into a commercial-grade XML processing library. Xerces-C 3.1.1 was chosen due to its wide-spread use in programming community.

Modifying the `WFXMLScanner` to use the Parabix Framework was a relatively easy problem. Unfortunately Parabix cannot handle stateful validation unless the all of the data needed to perform it is present in the source. As such, adapting the `IGXMLScanner` was a difficult process but doing so begat the content stream model, which united the parallel transcoding and deferred deletion techniques. Were I to begin this project again, my first goal would have been to redesign Xerces's internal API between `XMLReader` and `XMLScanner`s and divorce the parsing and buffering logic from each of the scanning routines. By separating the parsing routines from the (stateful) validation functionality, it was possible to isolate what parallel bit streams work well with within the Parabix Subsystem and what they do not handle well in the (sequential) Markup Processor.

The performance improvements in icXML appear to be substantial: a speedup of $50 - 100\%$ was noticed in many cases but D-Cache utilization is a significant issue. An earlier attempt at pipeline parallelism was reported on in "icXML: Accelerating a Commercial XML Parser Using SIMD and Multicore Technologies" [20]. icXML-p, the pipelined version of icXML presented in that paper, was $1.5 - 2\times$ faster than icXML. Although icXML's current design ought to support a greater degree of parallelism, implementation of icXML using the StreamIt model [25] was deemed to be beyond the scope of this project.

In conclusion, the icXML project was successful: it adheres to the Xerces-C 3.1.1 API and shows an improvement in parsing performance for all XML documents over 6KB and supports all features of Xerces (with the exception of object serialization) whilst providing a solid foundation for future enhancements.

Bibliography

[1] Krste Asanovic et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] Tim Bray, Dave Hollander, Andrew. Layman, Richard Tobin, and Henry S. Thompson. Namespaces in XML 1.0 (third edition). W3C Recommendation, 2009.

[3] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML) 1.0 (fifth edition). W3C Recommendation, 2008.

[4] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible markup language (XML) 1.1 (second edition). W3C Recommendation, 2006.

[5] Tim Bray, Henry S. Thompson, David. Beech, Murray Maloney, and Noah. Mendelsohn. XML schema part 1: Structures (third edition). W3C Recommendation, 2004.

[6] R.D. Cameron, K.S. Herdy, and D. Lin. High performance XML parsing using parallel bit stream technology. In Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds, page 17. ACM, 2008.

[7] Rob Cameron. A case study in simd text processing with parallel bit streams: Utf-8 to utf-16 transcoding. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 91–98. ACM, 2008.

[8] Rob Cameron, Ken Herdy, and Ehsan Amiri. Parallel bit stream technology as a foundation for XML parsing performance. In International Symposium on Processing XML Efficiently: Overcoming Limits on Space, Time, or Bandwidth, volume 8, 2009.

[9] Robert D Cameron. u8u16–a high-speed utf-8 to utf-16 transcoder using parallel bit streams. Technical report, Technical Report TR 2007-18, Simon Fraser University, Burnaby, BC, Canada, 2007.

[10] Robert D. Cameron, Ehsan Amiri, Kenneth S. Herdy, Dan Lin, Thomas C. Shermer, and Fred P. Popowich. Parallel scanning with bitstream addition: An XML case study. In Euro-Par 2011, LNCS 6853, Part II, Lecture Notes in Computer Science, pages 2–13, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] Unicode Consortium. The Unicode Standard. Core Specification. Unicode Consortium, http://www.unicode.org/versions/Unicode6.2.0/ch02.pdf, 6.2 edition, 2012.

[12] Thomas M. Cover and Joy A. Thomas. Elements of information theory. Wiley-Interscience, New York, NY, USA, 1991.

[13] John Cowan and Richard. Tobin. XML information set (second edition). W3C Recommendation, 2004.

[14] M. Drmota and R. Kutzelnigg. A precise analysis of Cuckoo hashing. ACM Transactions on Algorithms (TALG), 8(2):11, 2012.

[15] Yedidya Hilewitz and Ruby B Lee. Fast bit compression and expansion with parallel extract and parallel deposit instructions. In Application-specific Systems, Architectures and Processors, 2006. ASAP'06. International Conference on, pages 65–72. IEEE, 2006.

[16] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Jonathan Robie, Mike Champion, and Steve. Byrne. Document object model level 3 core. W3C Recommendation, 2004.

[17] R. Lake, D.S. Burggraf, M. Trninic, and L. Rae. Geography mark-up language (GML) [foundation for the geo-web]. Wiley, Chichester, 2004.

[18] Dan Lin, Nigel Medforth, Kenneth S. Herdy, Arrvindh Shriraman, and Rob Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In International Symposium on High-Performance Computer Architecture, pages 1–12, Los Alamitos, CA, USA, 2012.

[19] C.T. Lu, R.F. Dos Santos, L.N. Sripada, and Y. Kou. Advances in gml for geospatial applications. Geoinformatica, 11(1):131–157, 2007.

[20] Nigel Medforth, Dan Lin, Kenneth Herdy, Rob Cameron, and Arrvindh Shriraman. icXML: Accelerating a commercial XML parser using simd and multicore technologies. In Balisage: The Markup Conference, 2013.

[21] M. Mitzenmacher. Some open questions related to Cuckoo hashing. Algorithms-ESA 2009, pages 1–10, 2009.

[22] R. Pagh and F. Rodler. Cuckoo hashing. Algorithms—ESA 2001, pages 121–133, 2001.

[23] E. Perkins, M. Kostoulas, A. Heifets, M. Matsa, and N. Mendelsohn. Performance analysis of XML APIs. XML, 2005.

[24] Jie Tang, Shaoshan Liu, Chen Liu, Zhimin Gu, and Jean-Luc Gaudiot. Acceleration of XML parsing through prefetching. Computers, IEEE Transactions on, 62(8):1616–1628, 2013.

[25] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A language for streaming applications. In Proceedings of the 11th International Conference on Compiler Construction, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag.

[26] H.S. Warren. Hacker's delight. Addison-Wesley Professional, 2003.